FIELDPLACER

A FLEXIBLE, FAST AND UNCONSTRAINED FORCE-DIRECTED PLACEMENT METHOD FOR HETEROGENEOUS RECONFIGURABLE LOGIC ARCHITECTURES

Inaugural-Dissertation
zur
Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität zu Köln

vorgelegt von
Dustin Feld
aus Kevelaer

Köln 2016

Berichterstatter (Gutachter):

PROF. DR. RER. NAT. MICHAEL JÜNGER
Institut für Informatik
Universität zu Köln
Deutschland

PROF. DR.-ING. ANDRÉ BRINKMANN
Zentrum für Datenverarbeitung
Johannes Gutenberg-Universität Mainz
Deutschland

Tag der mündlichen Prüfung: 24. Oktober 2016

Ohana means family.
Family means nobody gets left behind - or forgotten.

— Lilo & Stitch —

Dedicated to my family.
In loving memory of Cincinnati Kid.

Abstract

The field of placement methods for components of integrated circuits, especially in the domain of reconfigurable chip architectures, is mainly dominated by a handful of concepts. While some of these are easy to apply but difficult to adapt to new situations, others are more flexible but rather complex to realize.

This work presents the FieldPlacer framework, a flexible, fast and unconstrained force-directed placement method for heterogeneous reconfigurable logic architectures, in particular for the ever important heterogeneous FPGAs. In contrast to many other force-directed placers, this approach is called 'unconstrained' as it does not require a priori fixed logic elements in order to calculate a force equilibrium as the solution to a system of equations. Instead, it is based on a free spring embedder simulation of a graph representation which includes all logic block types of a design simultaneously. The FieldPlacer framework offers a huge amount of flexibility in applying different distance norms (e. g., the Manhattan distance) for the force-directed layout and aims at creating adapted layouts for various objective functions, e. g., highest performance or improved routability. Depending on the individual situation, a runtime-quality trade-off can be considered to either produce a decent placement in a very short time or to generate an exceptionally good placement, which takes longer.

An extensive comparison with the latest simulated annealing placement method from the well-known Versatile Place and Route (VPR) framework shows that the FieldPlacer approach can create placements of comparable quality much faster than VPR or, alternatively, generate better placements in the same time. The flexibility in defining arbitrary objective functions and the intuitive adaptability of the method, which, among others, includes different concepts from the field of graph drawing, should facilitate further developments with this framework, e. g., for new upcoming optimization targets like the energy consumption of an implemented design.

Zusammenfassung

Das Gebiet der Platzierungsverfahren für Komponenten integrierter Schaltkreise wird, insbesondere im Sektor der rekonfigurierbaren Chiparchitekturen, im wesentlichen von einer Handvoll Konzepten dominiert. Während einige davon einfach anzuwenden aber schwer an neue Situationen anzupassen sind, sind andere flexibler aber relativ komplex zu realisieren.

Diese Arbeit präsentiert das FieldPlacer Framework, eine flexible, schnelle und uneingeschränkte kräftebasierte Platzierungsmethode für heterogene rekonfigurierbare Logikarchitekturen, insbesondere für die immer wichtiger werdenden heterogenen FPGAs. Im Gegensatz zu den meisten anderen kräftebasierten Platzierungsverfahren wird dieser Ansatz hier 'uneingeschränkt' genannt, da er keine a priori fixierten Logikelemente erfordert um ein Kräftegleichgewicht als Lösung eines Gleichungssystems zu bestimmen. Stattdessen basiert der Ansatz auf einer freien Spring-Embedder Simulation einer Graphrepräsentation des Designs, welche alle Logikblocktypen simultan einschließt. Das FieldPlacer Framework bietet große Flexibilität in der Anwendung verschiedener Distanz-Normen (z. B. der Manhattan-Distanz) für das kräftebasierte Layout mit dem Ziel, angepasste Layouts für verschiedene Zielfunktionen zu erstellen, beispielsweise höchste Performanz oder verbesserte Verdrahtbarkeit. Abhängig von der individuellen Situation kann ein Laufzeit-Qualität Kompromiss gewählt werden, um entweder eine ordentliche Platzierung in sehr kurzer Zeit oder eine außerordentlich gute Platzierung in längerer Zeit zu produzieren.

Ein umfangreicher Vergleich zum aktuellen Simulated Annealing Platzierungsverfahren aus dem bekannten Versatile Place and Route (VPR) Framework zeigt, dass der FieldPlacer Ansatz Lösungen vergleichbarer Qualität deutlich schneller erstellen kann als VPR oder, alternativ, eine bessere Platzierung in ähnlicher Zeit erzeugen kann. Die Flexibilität in der Definition beliebiger Zielfunktionen und die intuitive Anpassungsfähigkeit des Frameworks, welches unter anderem auf verschiedenen Konzepten aus dem Graphenzeichnen basiert, soll weitere Entwicklungen mit jenem ermöglichen, z. B. für neuartige Optimierungsziele wie den Energieverbrauch eines implementierten Designs.

Publications & Talks

During the time of my PhD research, I have dealt with different aspects of optimizing parallel programming for 'ordinary' multi-core CPUs and different types of accelerators, also automatically within compiler-optimization passes. My diploma thesis about an efficient vectorization technique based on the polyhedral model was kind of the starting point for these works. In this context, I published papers in collaboration with other researchers from the *Fraunhofer Institute for Algorithms and Scientific Computing*, the *University of Cologne*, the *University of Mainz*, the *University of Bonn*, the *London Metropolitan University*, the *Chinese Academy of Science* and the *Universities of Bielefeld, Heidelberg and Gießen*.

I sincerely want to thank everybody involved for the fruitful collaboration and the valuable influences you had on me.

Conference talks I have been giving in that time are listed below. Some of these have been supported by travel grants or other support, my heartfelt thanks to the generous donors. In addition, I want to thank the *MINO Initial Training Network of the European Union* and the *ICT COST action TD1207* for a travel grant to participate the *MINO/ COST Spring School on Optimization* in march 2015. Last, but definitely not least, I want to thank *Fraunhofer SCAI* for all the many opportunities to visit conferences during my PhD due to SCAI's (personal and financial) support.

In 2015, I also had the pleasure to be part of the program committee of the 'SPIE High-Performance Computing in Remote Sensing' conference and reviewing papers for the *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*. Thanks for these opportunities that enhanced my knowledge and interest in the transdisciplinary field of parallel (multi-core, MIC and GPU) computing and remote sensing.

No part of this work has been published in advance.

CONFERENCE PUBLICATIONS

Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation - *Dustin Feld, Sven Mallach, Thomas Soddemann and Michael Jünger*, Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (pp. 45–54), Jan 2013, Berlin, Germany

Hardware-Aware Automatic Code-Transformation to Support Compilers in Exploiting the Multi-Level Parallel Potential of Modern CPUs - *Dustin Feld, Sven Mallach, Thomas Soddemann and Michael Jünger*, Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores (pp. 2:1–2:10), ACM, Feb 2015, San Francisco, USA

Impact of the Scheduling Strategy in Heterogeneous Systems That Provide Co-Scheduling - *Tim Süß, Dustin Feld, Nils Döring, Lars Nagel, Eric Schricker, Ramy Gad, André Brinkmann and Thomas Soddemann*, Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications (pp. 37ff.), Jan 2016, Prague, Czech Republic

VarySched: A Framework for Variable Scheduling in Heterogeneous Environments - *Tim Süß, Dustin Feld, Nils Döring, Lars Nagel, Stefan Lankes, Ramy Gad, André Brinkmann and Thomas Soddemann*, Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER 2016), Sep 2016, Taipei, Taiwan

JOURNAL ARTICLES

Multicore Processors and Graphics Processing Unit Accelerators for Parallel Retrieval of Aerosol Optical Depth From Satellite Data: Implementation, Performance, and Energy Efficiency - *Jia Liu, Dustin Feld, Yong Xue, Jochen Garcke and Thomas Soddemann*, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing (vol. 8, nb. 5, pp. 2306-2317), May 2015

Comparison of acceleration techniques for selected low-level bioinformatics operations - *Daniel Langenkaemper, Tobias Jakobi, Dustin Feld, Lukas Jelonek, Alexander Goesmann and Tim W. Nattkemper*, Frontiers in Genetics (vol. 7), Feb 2016

An efficient geosciences workflow on multi-core processors and GPUs: a case study for Aerosol Optical Depth retrieval from MODIS satellite data - *Jia Liu, Dustin Feld, Yong Xue, Jochen Garcke, Thomas Soddemann and Peiyuan Pan*, International Journal of Digital Earth, Feb 2016

Energy-Efficiency and Performance Comparison of Aerosol Optical Depth (AOD) retrieval on distributed Embedded SoC architectures with Nvidia GPUs - *Dustin Feld, Jia Liu, Eric Schricker, Yong Xue, Jochen Garcke, Thomas Soddemann*, Scientific Computing and Algorithms in Industrial Simulations - Projects and Products of Fraunhofer SCAI, Springer Verlag [accepted, to be published]

CONFERENCE TALKS

Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation - *European Network on High Performance and Embedded Architecture and Compilation (HiPEAC)*, Berlin, Germany, Jan 2013

Hardware-Aware Automatic Code-Transformation to Support Compilers in Exploiting the Multi-Level Parallel Potential of Modern CPUs - *International Symposium on Code Generation and Optimization (CGO)*, San Francisco Bay Area, USA, Feb 2015 [ACM SIGPLAN Professional Activities Grant]

Energy-Efficiency and Performance Comparison of Aerosol Optical Depth (AOD) retrieval on distributed Embedded SoC architectures with Nvidia GPUs - *GPU Technology Conference (GTC)*, Silicon Valley (San José), USA, April 2016 [NVIDIA Full Conference Fee Grant]

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [113]

Acknowledgments

It is said that you should surround yourself with people who inspire you, who build you up and believe in you, people who are good for you - and with music. I am infinitely thankful that I had the pleasure to meet many such people that supported and fostered me.

First of all, I want to express my gratitude to my doctoral supervisor Prof. Dr. Michael Jünger from the University of Cologne. You not only supervised me but you often inspired and always supported me in many developments of this work through discussions and suggestions and, very fundamentally, by your university courses which initially opened up this field of research to me. In addition, a very great thank you to Prof. Dr. André Brinkmann from the University of Mainz who also mentored me. Working with you in several projects in the field of high performance computing has always been interesting and instructively.

In the last eight years and, therefore, during most of my time at the university, I had the pleasure to research at the Fraunhofer Institute for Algorithms and Scientific Computing SCAI. This not only gave me the opportunity to write my diploma thesis and this dissertation in the context of applied research, I have also been able to participate in many projects and to visit several conferences to present my work. I personally want to thank Dr. Thomas Soddemann and Dr. Johannes Linden for their confidence and support.

Further, I would like to thank my colleagues at SCAI and at the University of Cologne for their fellowship and feedback during these years. A comprehensive listing of everyone would span pages, just to name some: Lauri, Bea, Eric, Ottmar, ThoSo, ThoBra, Johannes, Christiane, Martin, Sven, Daniel, Andi, Francesco ... Thank you! My most sincere apologies to all those that have not been named here.

Due to the fact that I actually worked in two departments (Fraunhofer SCAI and University of Cologne), a considerable amount of administration effort has been carried out. Thank you Dorit and Göntje for your generous support!

To my friends and my bandmates ($bandmates \subset friends$) - Spending time with you has and will always be one of the most important things in my life - more frequently from now on again.

Finally, I want to thank my family, especially my mum and my dad (a. k. a. 'Chef'). Thank you all for everything!

Most importantly and most particularly, I want to thank my girlfriend Katharina for all the backing - throughout the last fourteen years and in particular during these last few years of my PhD. I cannot thank you enough!

Regarding the typography, further thanks go to André Miede[1] for his public available LaTeXthesis style 'A Classic Thesis Style - An Homage to The Elements of Typographic Style' which I used as the base for my thesis. We even share a common passion for one of the best Disney movies ever. Your postcard is on the way!

P.S.: Thanks to Dr. Guy Lonsdale for giving helpful advice regarding the language!

1 http://www.miede.de

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

ASIC	Application-Specific Integrated Circuit
BB	BoundingBox
BLE	Basic Logic Element
BPU	Basic Processing Unit
CB	Compute Block
CFV	Cost Function Value
CG bounds	Christofides and Gerrard bounds
CLB	Configurable Logic Block
CPD	CriticalPathDelay
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
FF	Flip-Flop
FPGA	Field Programmable Gate Array
GL bounds	Gilmore and Lawler bounds
GML	Graph Modelling Language
GPGPU	General Purpose Computation on Graphics Processing Unit

GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
IC	Integrated Circuit
I/O	Input & Output
IP	Integer Programming
ITS	Iterated Tabu Search
LAP	Linear Assignment Problem
LP	Linear Programming
LQP	Linearly Constrained Quadratic Programming Problem
LS	Local Search
LSI	Large-scale integration
LUT	LookUp Table
MCNC	Microelectronics Center of North Carolina
MILP	Mixed Integer Linear Programming
MUX	Multiplexer
OGDF	Open Graph Drawing Framework
OU	OverUse
PLD	Programmable logic device
QP	Quadratic Programming
QAP	Quadratic Assignment Problem
SA	Simulated Annealing
TS	Tabu Search
TSP	Traveling Salesman Problem
ULSI	Ultra-large-scale integration
VLSI	Very-large-scale integration
VPR	Versatile Place and Route
VPR SA	VPR's simulated annealing approach
VTR	Verilog-to-Routing (Project)
WL	WIreLength
WSPD	Well Separated Pair Decomposition

Symbols

MATH SYMBOLS

$\mathbb{B}$	Set of binary numbers $\rightarrow \{0, 1\}$
$\mathbb{Z}$	Set of integer numbers $\rightarrow \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{N}$	Set of natural numbers $\rightarrow \{1, 2, \ldots\}$
$\mathbb{N}_0$	Set of natural numbers including $0 \rightarrow \{0, 1, 2, \ldots\}$
$\|.\|_1$	1-norm (*Manhattan distance*, L_1-norm)
$\|.\|_2$	2-norm (*Euclidean distance*, L_2-norm)
$\|.\|_{max}$ or $\|.\|_\infty$	∞-norm (*Chebychev distance*, L_{max}- or L_∞-norm)

OTHER SYMBOLS

$\mathcal{N}$	Neighborhood
$\overline{\mathcal{N}}$	Full neighborhood
$\mathcal{G}$	A general graph
$\mathcal{G}_D$	General graph representation of an FPGA design
$\mathcal{G}_D^{layout}$	Graph layout of an FPGA design
$\mathcal{G}_D^{arch}$	Graph representation of an FPGA design embedded on the chip architecture
$\mathcal{G}_D^{2ndlayout}$	2nd energy phase graph layout of an FPGA design
$\mathcal{G}_D^{2ndarch}$	2nd energy phase graph representation of an FPGA design embedded on the chip architecture

Part I

What is the background of this approach?

This chapter briefly introduces the background and the challenge of the presented approach. In addition, the organization of the work is outlined and the benchmark system is described. This chapter is kept rather short as each individual part of this work will also be preceded by an 'informal' introduction.

1

Introduction

*"A **graph** is worth a thousand words."*
— common sense —

Contents

1.1 BACKGROUND & IDEA

Compilation flows are often accepted as black boxes. Based on the input code, their task is to produce an executable version of the process description. This translation is necessary for any compiled programming language and execution environment. Therefore, compilers generally accomplish the task of translating abstract descriptions into concrete machine instructions.

Whereas this procedure is frequently used by programmers for all different kinds of compiled languages, influencing the result is often limited to common compiler options, e. g., the specification of an optimization level. However, compiler development itself is a very interesting discipline of computer science. The availability of different possibilities to influence the translation process can improve not only the quality of a resulting implementation but also the process of development itself. For x86 processors, compilers usually offer a huge number of options while only very few are regularly taken into account (like the typically used '-O3' option in, e. g., Intel's or GNU's compiler collections).

When dealing with more 'exotic' architectures, the situation is similar or even more pronounced. Different situations and different goals demand flexible configurability regarding 'which direction a compiler should take' to translate an input description into executable instructions or even synthesized hardware. This is especially the case if hard restrictions (like indispensable timing constraints) have to be respected in order to guarantee the correct functionality of a system. Under such circumstances, FPGAs or other reconfigurable hardware devices are often the architecture of choice. The straight implementation in hardware, paired with the option to renew this hardware implementation without the need to replace any hardware parts, make such devices more and more important, especially if frequent changes of the hardware design are anticipated or if only a small number of the hardware should be produced. The manufacturing of an ASIC design in small quantities is often much more expensive than integrating an appropriate reconfigurable equivalent (which has been produced in very large amounts). In the end, this relatively generic chip only has to be configured with the appropriate functionality. In this context, 'placement' is the part of the compile flow that assigns synthesized logic units to suitable positions on the architecture.

For such situations, this work introduces the FieldPlacer framework, a flexible, fast and unconstrained force-directed placement method for heterogeneous reconfigurable logic architectures.

The FieldPlacer framework aims at providing an intuitive entry point to the field of FPGA placement and facilitates the consideration of various optimization goals. In general, the appropriate placement of logic blocks onto an underlying FPGA architecture is an important part of the FPGA compile

flow. Furthermore, the FieldPlacer method is able to facilitate the subsequent routing in the flow. In contrast to many available placers, especially to others from the field of analytical placement, no a priori fixation of nodes in the design is necessary. The general idea of the FieldPlacer approach is to create a graph representation of a chip design including *all* types of logic blocks (including I/O) and create a layout for the entire graph with a spring embedder simulation in a force-directed manner which imitates a *system of springs and magnets*. It will be pointed out that the resulting layout has many favorable properties to be used as a basic sketch for an FPGA design and that the frequently applied and necessary fixation of I/O nodes in other methods can be detrimental. Based on this generated basic layout description, many different strategies can be applied in the FieldPlacer in order to create a placement that matches the specific demands of the designer. To perform all this in very short times and with high resulting quality, several recent techniques, particularly from the field of graph drawing, are included in the framework. Finally, universal interfaces are provided to be able to modify and extend the method in the future.

This work describes all main parts of the FieldPlacer framework. In addition, a large number of benchmarks are included to demonstrate the achieved quality on the one hand and to give suggestions about an appropriate usage of the different possibilities in the application of the framework on the other hand. Before addressing the developed placement method, the theoretical foundations of the general placement problem are presented. This should elucidate the problem's complexity and outline available solution strategies to, finally, explain and justify the chosen strategy for this placer.

1.2 WHO ARE THE ADDRESSEES OF THIS WORK?

This is not an 'FPGA fundamentals' work. Even though the basic concepts that are necessary to explain the developed method are described, details about these should be found in other works. I personally recommend the book '*Architecture and CAD for Deep-Submicron FPGAs*' of Betz et al. [21] for that purpose. It is also not intended to be a 'graph drawing fundamentals' work. Again, the fundamental ideas and concepts are of course described. However, more details about graph drawing algorithms can be found in many referenced works. For the applied force-directed approach, the dissertation '*A potential field based multilevel algorithm for drawing large graphs*' of Hachul [81] is *the* reference for further research. Finally, apart from the actual development of the FieldPlacer framework, this work aims at bridging gaps between all mentioned domains in order to solve a practically occurring problem from an up-to-date field of computer science.

1.3 ORGANIZATION OF THIS WORK

The work is organized in seven main parts. This first part contains the introduction of the work. After that, Part II provides the domain of FPGAs in general while Chapters 3 and 4 in Part III outline the theoretical backgrounds of the quadratic assignment problem and force-directed graph layouting, respectively. Part IV presents the general FieldPlacer approach and Part V adds the surrounding statistical framework for repeated FieldPlacer runs. Finally, Part VI concludes the work and Part VII contains supplementary material.

1.4 TEST ENVIRONMENT

Unless otherwise mentioned, each measurement is repeated 10 times (with 10 different seeds for randomized decisions). The maximal and minimal value is neglected to disregard outliers (especially important for runtime measurements) and the remaining results are averaged.

In order to avoid any misunderstanding, the *wall-clock time* that is needed to run an algorithm (*running time*, *time spent*, *time span*, ...) is denoted by the term '***runtime***' in this work.

A detailed description of the benchmark environment is given in Table 1. Even though the processor has four physical cores, all measurements are performed single-threaded.

Architecture:	x86_64
Model name:	Intel(R) Core(TM) i7 — 4790K CPU @ 4.00GHz
CPU MHz:	4400.000
L1i/d / L2 / L3 cache:	32K / 256K / 8192K
Memory (RAM)	32GB
Harddisk type	SSD
OS	Scientific Linux release 7.1 (Nitrogen)
Kernel	Linux version 3.10.0 — 229.14.1.el7.x86_64 (mockbuild@sl7-uefisign.fnal.gov) gcc version 4.8.3 20140911 (Red Hat 4.8.3 — 9) (GCC)
Compiler	gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5 — 4)

Table 1: System configuration

Part II

What is the domain of this approach?

*This chapter covers the basics of **Field Programmable Gate Arrays**. It is explained what they are, from where they arose, how they work, what they are used for, how they are programmed, how the entire compilation workflow for those look like and what these workflows have to accomplish. In addition, the heterogeneous architectural 'baseline' model of an FPGA for this work is explained. Throughout the chapter, diverse details are given about techniques that are important for this work and that are, for example, developed and implemented within the FPGA framework that is used for the later implementations. In the end, this should provide a basic technical grounding for the subsequently presented idea of a chip placement based on force-directed graph drawings.*

2

Field Programmable Gate Arrays

"The measure of intelligence is the ability to change."
— probably Albert Einstein —

Contents

2.1 HISTORY OF PLDS

Apart from the large variety of 'common' processor alternatives in today's computers like x86 or ARM multi-core CPUs with 32- or 64-bit characteristics, GPUs or (for pure computing purposes) so called GPGPUs, *FPGAs* are becoming more and more popular in different fields of computing.

In the early 1970s, *programmable logic devices* (PLDs) entered the market and extended chip designs (and consequently circuit boards) towards flexibility with dynamically configurable elements instead of using solely combinations of fixed logic-gates. Unlike the usual *application-specific integrated circuits* (ASICs) or *integrated circuits* (ICs) in general which contain a predefined variety of logical functions, PLDs provide the possibility (*and necessity*) to define (*and redefine*) a chip's behavior within the system after its fabrication. A simple PLD can be seen as a programmable box that creates a user-defined output for every input-combination in a specified and reconfigurable way. The 'size' of the box along with the number of connections in and out the chip constrain the amount of possible functions. As an alternative to logic based implementations, such a functionality can (*and has*) statically also been realized by much slower ROM-based approaches before. Due to the fact that the final definition of the chip takes place outside of the factory in a productive environment, such devices introduced the so called '*fabless*' semiconductor industry. Thus, it is, for example, possible to use a PLD within a system for different demands at different times instead of integrating all necessary logics in separate chips in the system at fabrication time.

Computing systems (or logic gate systems in general) of today still are, in simple terms, machines that produce binary outputs based on binary inputs. Hence, they are physical devices implementing and combining Boolean functions. The most basic logic elements are gates with one or two binary *inputs* and one binary *output*. The 'standard' logic gates are namely the *NOT*, *AND*, *OR* and *XOR* gates (as well as their logical combinations *NAND*, *NOR* and *XNOR*). By connecting these gates to a system of gates, more complex logical functions can be implemented in hardware. The process of combining *hundreds*, *thousands* or even *hundred thousands* of logic gates is covered by the terms *large-*, *very-large-* or *ultra-large-scale integration* (LSI, VLSI, ULSI). Analogously, transistor counts of *thousands*, *hundred thousands* or *millions/billions* per chip refer to the same terms.

PLDs are able to carry out specialized tasks (that may vary over time) while still implementing them in hardware. Thus, they can be more '*efficient*' (with more *predictable* behavior) than very *general* processing units (like CPUs) in various ways but they are likewise, in general, less efficient than extremely *specialized* hardwired elements like ASICs. This holds true at least as long as the desired task is 'relatively simple'. *Efficient* in this context can,

for example, mean that a minimum amount of hardware is involved on the chip to process a requested operation or that as little power as possible is consumed. On the one hand, this generally comes (for PLDs) at the price of a lower circuit speed compared to fixed fabricated logic (like an ASIC) as such reconfigurable logic elements cannot be packed as densely and work as fast as their static counterparts. On the other hand, a function in a PLD can be realized and changed *without fabrication* and *within the actual system*. Early works like the one of Brown et al. [26] already examined the advantages and drawbacks of *Field Programmable Gate Arrays* (FPGAs, see Section 2.2), a special class of PLDs which is in the focus of this work. Later works like, for example, the one of Kuon and Rose [117] from 2007 compared ASIC designs' efficiencies in many details to the one of such FPGAs. The authors compared both hardware types from a 90-nm production generation in terms of logic density, circuit speed and power consumption. Their experimental results showed that the gap between ASICs and FPGAs was still very significant, for example, concerning needed area for the logic (a factor of 35 was reported to give an idea of the magnitude) or concerning speed (with a summarizing factor of 4). This gap is nowadays more and more closing with heterogeneous FPGAs (see Section 2.2.5) and with upcoming FPGA frequencies of up to 500 MHz (cp. Lim [134]) enabled by 28-nm production and other technological progress.

Besides the great potential due to the reconfigurability in productive systems, PLDs also offer the possibility of easy and cheap hardware *prototyping* of chips instead of only simulating their behavior in the construction phase or fabricating actual hardware prototypes. The decision whether to eventually produce ASICs with the functionality of the final PLD prototype or to use a PLD even in the resulting product, potentially the same PLD that was used for prototyping (as it is a common practice for different PLD-driven developments like network routers, modems, DVD players or automotive navigation systems) is not only governed by the option of later reconfigurability of the system. It is often also a question of production costs. As ASIC designs have to be fabricated with very high initial and fixed costs (a factory has to be consulted), it is only worth the effort (and the money) if a very large number of chips is finally produced. For specialized products, this is often not the case and an ASIC design could therefore drastically increase the production costs. Instead, a PLD chip can be used for many different applications and can therefore be fabricated and sold in large quantities what can consequently make them cheaper (per unit).

One further advantage of PLD technologies compared to ASICs is a drastically reduced *time-to-market*. A chip that has finally been prototyped can simply be cloned to prefabricated PLDs in a very short time compared to the time that would be needed to produce the respective ASICs. Furthermore,

the possibility to *update* a chip's design *after delivery* is an advantage for certain applications, even when being incorporated in ordinary products like those mentioned before.

Except for the already mentioned FPGAs (which will be further investigated in the remainder of this work), *complex programmable logic devices* (CPLDs) are the second market dominant type of *larger reconfigurable logic* today. Smaller units (with only several hundreds of logic gates) are, for instance, *programmable array logics* (PALs) which are, in general, *one-time-programmable* or only very difficultly reprogrammable. Their reconfigurable equivalents are *generic array logics* (GALs) that are, hence, often used for PAL prototyping.

2.1.1 *CPLDs and FPGAs*

CPLDs are, compared to FPGAs, made of a relatively simple and homogeneous structure consisting mainly of a configurable matrix of *AND- and OR-gates* combined with a very small number of *flip-flops* to *store* states. The matrix is accessible through a large number of *in- and output pins* from the outside of the CPLD whereas these pins are often the only elements 'equipped' with a (single) flip-flop. The large amount of in- and outputs predestinates CPLDs to be used in a highly parallel manner. Instead of describing the logic directly by connecting logic gates 'manually', higher level languages are often used to describe the functionality in an abstract way. The translation from such an abstraction level into the actual *netlist* (logic description as a *circuit diagram*) is called *synthesis* and is, in general, the initial step of a compilation workflow (cp. Section 2.4) from a more or less abstract description language into an actual hardware description. Prior to the actual translation into hardware, the behavior of the later chip can be simulated in the chip-specific software environment. However, the more complex a chip is and the more degrees of freedom for the actual implementation exist, the more difficult is any prediction. Due to the rather homogeneous structure and also due to a very simple routing architecture, the timing of a CPLD is relatively easy to predict. Inter-logic delays are small and the overall timing is quite consistent for several *compilation runs* of the same functionality from a higher description language.

A further difference between FPGAs and CPLDs is that the latter use *electrically erasable programmable* ***read only memory*** (EEP**ROM** memory) to store the configuration *of* the chip (and *on* the chip) while the former ones often use *static* ***random-access memory*** (S**RAM**). One core advantage of using EEPROM (e. g., flash memories are a subdivision of EEPROMs) is that a CPLD is ready to use just after powering it up. An SRAM-based FPGA con-

figuration instead is *volatile* and needs to be loaded from an external memory (sometimes also an EEPROM) to the FPGA's SRAM-cells in the booting process. EEPROMs generally have the disadvantage that the number of erase/write-cycles is rather limited as erasing degrades the oxide barrier on the silicon which at some point may lead to failures. This is, for example, described in the work of Buitenkamp [40]. The article primarily presents a software technique to extend the operational life of EEPROMs. However, hardware advancements could accomplish the same in the future. This technical difference between common CPLDs and FPGAs has already been overcome by some manufacturers providing flash based FPGAs. Such developments only became possible by decreasing the size of flash cells to maintain the relatively high logical density of FPGAs.

Remark 1. *Apart from **electrically erasable** programmable read only memories, there are also EPROM memories that can be erased by **ultra-violet (UV) light**. However, such EPROMs are not really practicable when a reconfiguration is frequently required.*

Due to these characteristics, CPLDs are the right choice for relatively simple use cases like critical control applications or generally simple pure combinatorial designs like *glue logics* to basically combine/connect other resources of the chip, see Greaves and Nam [78]. This is especially true if the functionality will probably not change too often while the system is frequently rebooted and a processing of the chip is desired directly after power up. Due to their simple structure, CPLDs also require only extremely low amounts of power, what is especially important in battery-operated systems. Finally, CPLDs are relatively cheap.

FPGAs (in contrast to CPLDs) base on *lookup tables* (LUTs) as their principal building blocks instead of simple logic gates. A LUT with k binary inputs consequently has 2^k possible input constellations while the output for each of these can be specified by the SRAM table. Just like a CPLD, an FPGA needs *in- and outputs* to communicate with the rest of the system. In addition to these *basic* elements, FPGAs contain a relatively large number of *flip-flops* and modern architectures provide more and more heterogeneous on-chip elements such as *hardwired processor cores*, dedicated *random-access memory*, *digital signal processing elements (DSPs)* including *multipliers*, various *clock management systems* and support for advanced device-to-device *signaling technologies*.

Even though the processing time of signals in a CPLD *before the actual implementation in hardware* (resp. in a simulation) is easier to predict than for the more complexly structured FPGAs, final hardware implementations of *both* systems have great advantages in terms of *predictability* compared to ordinary computing architectures like CPUs as these include many mechanisms

that are difficult to predict, like, for example, caches with hard- and software prefetchers.

Due to their mentioned characteristics, FPGAs are applied in a wide variety of applications like networking hardware, data processing and storage, general instrumentation, telecommunication systems or even as hardware-configured digital signal processors.

Today, FPGAs are not any more only available as expensive niched special-purpose hardware. Embedded in so called *Systems-on-a-Chip* (SoCs), several FPGA manufacturers offer 'all-in-one' solutions which are dedicated especially towards *early development* and *research*. These boards often contain a central processing unit (e. g., an ARM CPU), potentially along with other specialized processing elements, memory regions, periphery, graphic processors, audio and further interfaces. The two most dominant manufacturers of FPGAs over the last years have been *Xilinx* and *Altera*[1][2][3] (part of *Intel* since 2015), both providing such relatively cheap development boards especially for researchers in addition to their 'professional' products. Other vendors of PLDs in general are *Lattice Semiconductor* (*Vantis* (*AMD*)), *Microsemi* (*Actel*), *Quicklogic*, *Lucent*, *Cypress* or *Atmel*. Altogether, the market of PLDs is constantly growing and the role of FPGAs has become even more important in the recent past[4].

2.2 FIELD PROGRAMMABLE GATE ARRAYS

As already stated in the previous section, FPGAs form a special class of PLDs which is moving more and more into the field of 'mainstream' accelerators due to their wide applicability and improved programmability. An FPGA's main feature is its *reconfigurability*. This can be achieved by the use of different *programmable* hardware elements. Many popular FPGA architectures are configurable by *SRAM*-cells as against *EEPROMs* (*flash* memories) on regular CPLDs. However, some manufacturers base their FPGAs on flash memory or on antifuses even though the latter class only allows for a *onetime configuration* of the system. They could therefore more precisely be called *configurable* instead of *reconfigurable*. Flash-based FPGAs are technically difficult to realize compared to SRAM-based architectures because SRAM-based FPGAs can achieve a much higher density on the chip. Still, as mentioned before, flash-based (as well as antifuse-based) chips are ready to operate *directly* af-

1 http://sourcetech411.com/2013/04/top-fpga-companies-for-2013/ (accessed 19 April 2016)

2 http://hackaday.com/2015/08/24/two-new-fpga-families-designed-in-china/ (accessed 19 April 2016)

3 http://www.fpgadeveloper.com/2011/07/list-and-comparison-of-fpga-companies.html (accessed 19 April 2016)

4 http://www.dailytech.com/Why+Intels+Massive+167B+USD+Plan+to+Purchase+Altera+Makes+Perfect+Sense/article37380.htm (accessed 19 April 2016)

ter powering up and have an exceptionally low power consumption. SRAM-based FPGAs are volatile and therefore need separate memory to load the configuration from on start up. This memory can in turn be a flash memory in- or outside the chip or even an external configuration memory such as a harddisk drive or the like. It depends on the circumstances of the use case what kind of an architecture is the best choice.

Remark 2. *For the approach presented in this work, it does not matter **how** the reconfigurability is achieved. Even though the explanations in the following sections assume **SRAM**, this does not play a role for the introduced model at all. Performance indicators on which it would have an influence (e. g., in the timing prediction model) are provided by external pieces of software, for example, **VTR/VPR** (see Section 2.3) in the later benchmark sections.*

2.2.1 *Operating principle*

Remark 3. *The figures in the following section are partially based on schematics from the book '**Architecture and CAD for Deep-Submicron FPGAs**' by Betz et al. [21] to preserve good comparability for any reader who wants to dive deeper into FPGA details with the cited book. The equations are also partially related to the book as the described framework in the book has been the basis for the framework in which the methods of the presented work are finally implemented and benchmarked.*

The most basic and important elements of an FPGA are the *lookup tables* (LUTs). A k-input lookup table (k-LUT) is a small memory element with k *(binary) inputs* and *one (binary) output*. The *table* can be programmed arbitrarily by specifying a desired output for each of the 2^k input combinations. Thus, 2^k cells of (e. g., SRAM) memory are required and such a table can be programmed into 2^{2^k} possible states. Figure 3a shows a 2- and a 4-LUT together with *one* possible configuration for each. The 2-LUT in Figure 3a in fact implements the same functionality as an ordinary *OR*-gate. However, a simple 2-LUT can implement even more functions than the *elementary logic gates* which have been named in the previous section (see Figure 1, the symbols are settled in *ANSI/IEEE Std 91-1984* [1]).

LUTs are, in general, *the* dominant operating logic elements of an FPGA architecture.

An important characteristic of FPGAs is that designs implemented into them are generally '*synchronous*'. When the FPGA is operating, a *clock* (or several clocks) are driving the transition from one state to the next. Hence, the processing of a signal from one point of the design to a subsequent one is aligned to the underlying clock by *flip-flops* (FF), see Figure 3b. A flip-flop (or a *latch*) is a bistable multivibrator. Thus, it can hold two stable states. Flip-flops can be *clock-driven* to form a simple 1-bit storage element in sequential

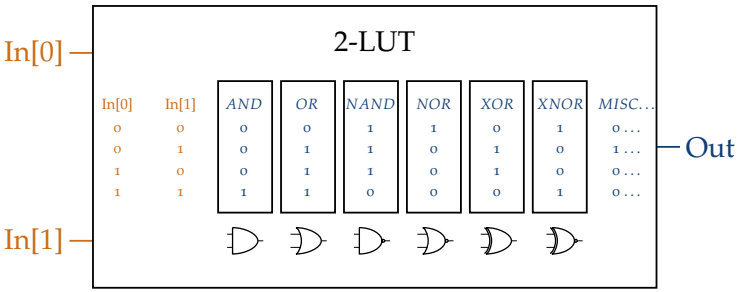


Figure 1: Lookup Table - 2-LUT possibilities

logics. If so, the input data is *latched*, *stored*, and the output data is *refreshed* in each clock-cycle. Due to the insertion of flip-flops into a design, no additional logic is added but the current state is *stored* and the execution of different signals passing the chip are *harmonized*. This also helps to equalize uncertainties (resp. unpredictabilities) or, in general, small variations in the timing. Section 2.2.2 will discuss the use and importance of flip-flops to steer the timing of an FPGA chip in more detail.

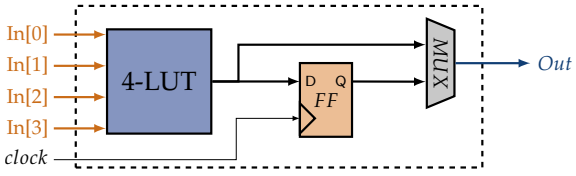
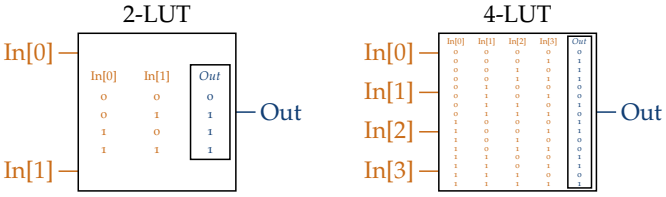


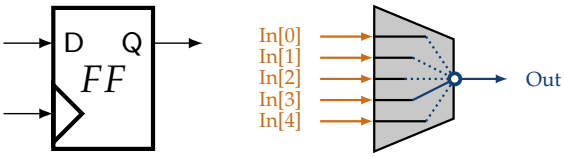
Figure 2: Basic Logic Element (BLE)

Figure 2 illustrates a *basic logic element* (BLE) which is a combination of the mentioned components. To align such a *BLE* with the clock, the flip-flop can be used to store the result from the LUTs output. However, if no synchronization is needed at this point of the logic, the LUTs output (*Out*) can directly be connected bypassing the flip-flop. This option is realized by the use of a (programmable) *multiplexer* (MUX). A $(k:1)$-MUX can dynamically connect any of its k inputs to the output of the multiplexer. The MUX can be programmed (e. g., through an SRAM cell of size $\log_2(k)$) to enable one of the two strategies (*with* or *without* the flip-flop) for each BLE in the circuit. Since a flip-flop is able to store *one* bit, a so called *register* of size r (group of r flip-flops) is able to store r bits. The fact that a flip-flop needs a certain amount of time to 'transport' the input data to the output introduces a delay called the *propagation delay*. In an FPGA design, flip-flops are used for a num-

ber of purposes. The obvious one is to simply preserve a state within the circuit, e. g., for the synchronization (and combination) of multiple signals. In Section 2.2.2 it will be shown how the *activation* (or *bypassing*) of flip-flops by the multiplexers in BLEs can influence the *maximal possible clock-speed* of the circuit, e. g., by *pipelining* the design flow.



(a) Lookup Table (LUT)



(b) Flip-Flop (FF) (c) Multiplexer (MUX)

Figure 3: Main FPGA building blocks for CLBs

Finally, all these basic elements are hierarchically combined to the comprehensive logic element of the FPGA, the *configurable logic block* (CLB). A CLB consists of a number of BLEs sharing I inputs and N outputs, whereas the outputs of the BLEs can again be internally connected to (the same or other) BLEs' inputs in the CLB (see Figure 4). For certain architectures, the content of several BLEs are first combined to a *slice* and then several of these slices are combined to a CLB, a principally technical difference to improve the performance of the logics' execution.

The overall hierarchy of the logic units on an FPGA can be summarized as follows:

$$\{\text{LUTs}, \text{FFs}, \text{MUXs}\} \subset \text{BLEs} \; (\subset \text{slice}) \subset \text{CLBs} \subset \text{FPGA} \; .$$

Instead of only placing *simple* LUTs of a specified size into the CLBs on the FPGA architecture, $(k+1)$-LUTs can, for example, be constructed by combining two k-LUTs sharing the same k inputs whereas their outputs are combined with a $(2:1)$-multiplexer which is consequently switched by the additional '+1' input. Such configurations offer the possibility either to use *fewer and larger* or *more smaller* LUTs. However, such a hierarchy of LUTs needs more area and wiring on the chip. It is a *weighing* of pros and cons to choose

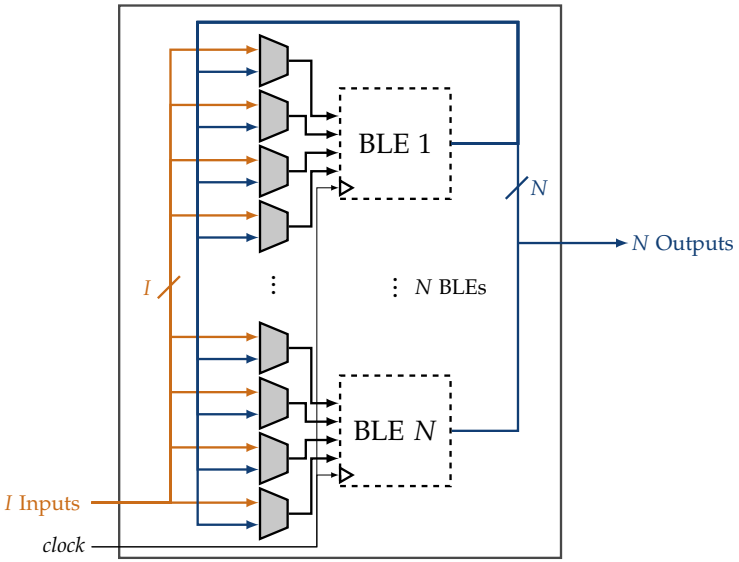


Figure 4: Configurable Logic Block (CLB)

the best LUT size and their best combination within the CLBs. Ahmed and Rose [3] investigated the effect of the LUT sizes and of the number of LUTs per CLB (*logic cluster*) on the needed *area*, the introduced *delay* and on a norm called the *area-delay*. Area-delay is a product of both metrics and therefore models the compromise between two *contradicting* goals. They came to the conclusion that, if this combined area-delay norm is the criterion of choice, clusters made of 3-10 LUTs each with 4-6 inputs produce the best overall results. An earlier work of Betz and Rose [18] therefore experimentally provided a value for the number of inputs for the BLEs to achieve a high ($> 98\%$) *utilization of the clusters* (for a certain set of benchmarks).

To *provide data from outside* the FPGA as *inputs* for the logic and likewise to emit *outputs* of it, the *I/O blocks* (in- or output *pads*) are the second basic type of blocks on FPGAs. Due to bidirectional operability (*tri-state capability*), many FPGAs' pads (e. g., from Xilinx) can be configured to be used either for input or output (the keyword is *IOBUFs*). Thus, I/O resources can often be assigned arbitrarily to such bi-directional pads, what in turn offers maximal flexibility in the design phase. This holds true at least as long as the outer connections are not fixed due to architectural constraints. This depends strongly on the specific demands on a system and its operational status. If the FPGA is embedded in a fixed setup that is possibly not accessible or not

changeable, the position of several or even all I/O pads may be fixed (see Section 5.9.1). However, especially in a the design phase or the prototype stage, the FPGA's overall design can often influence the positioning of the I/Os considerably (see Section 5.5.4). Thus, a good arrangement of the I/O pads can be crucial to achieve a good performance of the chip or/and good routability.

In addition to their pure function of providing access to signals *for* or *from* the FPGA, the I/O pads can also contain flip-flops to store the data right *before the actual output* or just *after the input* as a buffer.

Finally, the interconnection of such elements through wires is realized through the FPGA's *routing architecture* which traverses signals from one element to the next. To do so, *routing paths* are available on the chip. To meet the requirements of the reconfigurability, the wires can be connected dynamically. The description of the interconnectivity of logic elements is called *netlist*. The netlist forms a hypergraph and the hyperedges are accordingly called *nets*.

2.2.2 *Timing the delay*

As already stated before, the flip-flops synchronize the progress of signals in the design on an FPGA. Therefore, the flip-flops are clocked by so called *clock generators*. FPGAs usually contain one or more such clock generators to create sources for the synchronization signals with frequencies defined by either the user or by (*or with support of*) the design tools. For a predictable synchronization, it is obvious that the time of arrival of the clock signal should ideally be identical (or at least very precisely predictable) for all clock driven elements (*here:* flip-flops) on the chip. Thus, the clock signal should be distributed through independent *nets* (*wires*) on the architecture. Such nets are often called *global nets* and the wires on the chip are accordingly called *global lines*. They do not interfere with the 'ordinary' interconnects of logic elements and can be *hardwired* so that their behavior is very stable and predictable (*as desired*).

However, *every transition of a signal takes (some amount of) time*. Thus, even the best clock architecture with fast interconnects imposes delays (resp. differences in the arrival times) between different targets (e. g., flip-flops). These delays of the clock signals are called *clock skew*. A commonly used (in VLSI design) and very efficient distribution network for such global clock signals (in terms of *low* clock skew) is an H-tree like the one shown in Figure 5. An H-tree is a self-similar fractal with Hausdorff dimension 2 (see, for example, Ullman [183] or Browning [27]). In such a global H-tree net, the path from

the central point (the *root*) to any of the tree's endpoints (*leaves*) is equally long so that the theoretical time of arrival is *identical* for all elements.

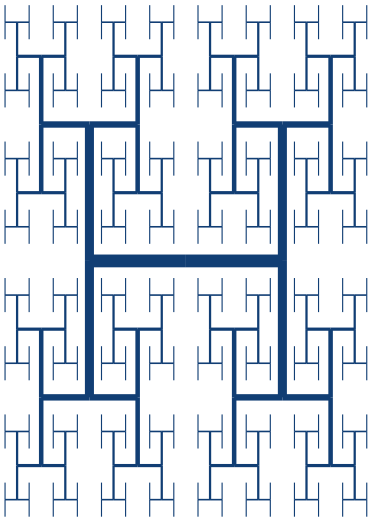


Figure 5: H-tree clock net

Clock skews have to be taken into account when performing a detailed timing analysis, especially for designs that are *time-critical*, which means that they need to operate as fast as possible near the timing bound and therefore with as little *slack* as possible. *Slack* is basically the difference of required and actual arrival time at a point in the design and it occurs due to several reasons, some of which will be discussed in the following paragraphs.

Wire delay (d_w)

The fundamental sources for the necessity of timing analyses in general are different types of delays that are present within a chip layout. Consider the simple succession of flip-flops and logic elements in Figure 7a on page 25. Neglecting the clock skew (by assuming a perfect distribution network for the clock), an obvious delay occurs from the 'normal' wires connecting the different basic logic elements. The signal needs, for example, a certain amount of time to pass from FF1 to FF2 through the wire between the elements. Depending on the length of this connection, the imposed *wire-delay* d_w influences the timing. In simple words, the longer the connection is, the longer the signal takes to traverse from the output of FF1 to the input of FF2. Section 2.3 will

provide some more details about the routing architecture of FPGAs. However, it is generally desirable to keep connections relatively short.

Remark 4. *Section 6.3 will show how detailed information about* ***slack*** *is used to iteratively optimize the layout in cooperation with the graph-layouting approach.*

The delay through a wire segment can, for example, be modeled very easily and still relatively accurately by the Elmore delay [55]. Assuming consistent resistance and consistent capacitance through the wire, the delay is proportional to half of the product of the *resistance* and the *capacitance*. The resistance of a wire segment depends (among other factors) on the *material*, where copper resistance is relatively low and silver resistance is, for example, even lower. Other influencing factors are the cross-sectional area of the wire, its temperature and, finally, its length. Assuming that the first three properties are given and constant throughout the architecture, a wire segment's length determines its resistance.

An accurate approximation of the capacitance of a wire is a difficult task as it depends on several factors like, for example, the environment, the distance to surrounding wires and, finally, its shape (cross-sectional area and length). However, by (again) neglecting several of these effects (which are very difficult to estimate in detail), a linear dependence of the length of a wire segment on the capacitance can be assumed as a simple approximation.

As both properties (*resistance* and *capacitance*) linearly depend on the wire's length and as the delay introduced by the wire is proportional to half of the product of both values, the delay of a wire segment becomes (following all simplifications and *Elmore*'s model) a quadratic function of its length.

Even though minimizing the length of connections between logic elements in the circuit is therefore a reasonable goal, a good routing is in fact way more complex. Available *wires* on a routing architecture may have *different lengths* (see Figure 6) and connections of different wires affect each other. The resulting choice of the *right* wire segments for each *connection* is extremely challenging. It can, for example, be very beneficial in a VLSI design to split long routing paths and insert *amplifiers* (*buffers*) to speed up the traversal of the signals (see, for example, Bartoschek et al. [15]).

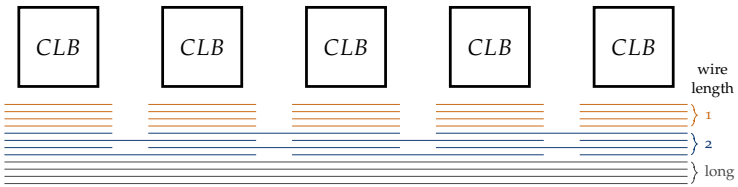


Figure 6: Wires with different lengths on the architecture

In addition, even the available switches on the architecture to connect wires can be of different kind. The benefits of using *pass transistors* or *tri-state buffers* for this task differ from one situation to the next (e. g., influenced by the lengths of wire segments or the number of connected segments). Furthermore, the two types demand for different amounts of area on the chip. Thus, a good mix of these architectural options should be present on the chip's architecture (see, for example, the work of Vaughn and Rose [20] or the book of Betz et al. [21]).

Remark 5. *Wire delay is a major bottleneck in the design of* ***high-speed systems*** *and is therefore an important optimization goal in the design phase of a (high-speed) circuit (see, for example, the work of Zhou et al. [193] or Bartoschek et al. [15]). Due to the trend towards smaller and faster structures (****deep sub-micron****), Betz et al. [21] (resp. Rose and Hill [160]) already stated in the late 90s that the importance of routing delays, rather than delays introduced by the logic, will become more and more dominant. Consequently, a large proportion of the area on many modern FPGAs is used for the routing architecture (cp. Vaughn and Rose [20]).*

Summarizing at this point, it can be stated that routing is difficult and that there are many influencing factors. However, the routing is not primarily considered in this work (except marginally in Section 5.4.2). For now, it is sufficient to note that the connections through wires on the architecture impose a delay that grows with the wirelength. Thus, relatively short connections between logic elements are desired.

An improvement of the delay model for wires in terms of accuracy has been presented by Avci and Yamacli [14]. However, Elmore's model is still often used in practice due to its simplicity in combination with an acceptable accuracy. To use Elmore's estimation on an entire *RC-tree* (Resistor-Capacitor-tree), the capacitances of subtrees must be combined. Details on this technique and links to other techniques like the *Penfield-Rubenstein* model [161] can, for example, be found in Betz et al. [21, Section 2.2.4].

Remark 6. *For this work, we assume that a delay model for both the approximation of all delay types* ***before*** *actual routing and* ***after*** *it is provided by the surrounding architecture specific software (*VPR/VTR *in this case, see Section 2.3). The model in this work takes this (optionally) as an input for further timing-driven improvement.*

In the design phase (e. g., *before the actual routing* or even before the *placement* of the layout, see Section 2.4), the wire delay can only be roughly approximated due to many uncertainties concerning the final wires' pathways. As, for example, the definite *placement* of, e. g., the CLBs onto the architecture is often performed *before* the routing between them, the wire delay should, in general, be conservatively overestimated until the final layout is determined. This is important to ensure correctness of the resulting configuration (see Section 2.2.3).

Logic delay (d_l)

Even though the delay introduced by the connections is today often dominant in high-speed FPGAs (cp. Remark 5), another considerable type of delay is the *logic delay*. This simply occurs due to the time that is needed to perform a logical operation, e. g., the evaluation within a CLB. For sure, this delay also includes wire-delay due to the block-internal interconnections. The logic delay is often quantified by *static* models which rate the delay of a component in a CLB (e. g., a LUT) with a constant delay and combine all these values to a resulting delay for the entire logic element.

As the delay of the logic elements can be extracted from the hardware specifications, the estimation of this delay type is, in general, rather accurate, especially in a design environment that is specifically adapted for the targeted hardware architecture. See Betz et al. [21, Section 6.2.3] for details about the later incorporated model.

Propagation delay (d_p)

A further basic delay type is the *propagation delay* of a general gate, e. g., of a flip-flop. Propagation delay measures the time that a signal needs to traverse *from the input* of a flip-flop (*or gate*) *to its output*. It is therefore the time needed to '*load*' the flip-flop with its new value. This delay must be taken into account in the timing analysis, especially for time-critical situations operating at maximum speed.

Remark 7. *In addition, the connection of very many inputs to a single output in a logic gate can impose a further propagation delay on the circuit.*

2.2.3 *Correctness, slack and clock-speed*

Figure 7a on page 25 shows a very simple circuit with three flip-flops and one logic element. A theoretical *sequence plan* of an input *data* signal traversing the design *without all the mentioned delay types* is depicted in Figure 7b. As stated before in Section 2.2.1, the signal in a synchronous design traverses from one flip-flop to the next within one clock cycle. The clock signal is shown at the top of Figure 7b, periodically switching between the states 0 and 1. To specify a *point* in time, the clock will be considered to 'take place' and therefore '*to actually clock*' at the *rising edge* of the clock signal. This is the point at which the signal switches from 0 to 1. As this switching does also take a small amount of time, the signal is actually *rising* instead of instantaneously switching. These *clock points* in time are indicated with the gray vertical lines in Figure 7. Now, the status of a flip-flop is updated at each of these points. As a signal needs to be 'transported' from one flip-flop to the next within

one clock period, the overall delay in between two such clock events must not exceed the time span of the clock period.

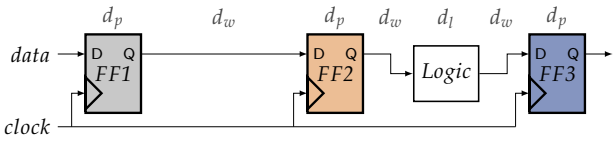
Figure 7c additionally contains delays of all mentioned types so that the traversal through the wires (d_w), the evaluation of the logic (d_l) and the 'loading' of the flip-flops (d_p) is rated with some time consumption. As a result, the signal at the input and output of all flip-flops is distinguished from one another and it becomes visible how the delays influence the actual sequence plan. Consequently, the clock-speed may only be as high that, including all delays, everything within one clock-cycle can be processed. The higher the delays are, the lower must be the clock-speed. If the delays are *too large* and exceed a clock-cycle, the design will not work as expected. On the other hand, a *too low* clock-speed (inducted by the already mentioned clock-generators at the beginning of Section 2.2.1) will result in wasted time at the end of the clock-cycle, so called *slack*.

If a desired *clock-speed* of the circuit *is a priori given*, an analysis of the delays can prove whether the circuit (*as it is*) can run at this speed or not. If not, the insertion of flip-flops (e. g., by *activation* through the multiplexers in the BLEs, see Figure 2) in restricting *critical* paths can facilitate the desired clock-speed. In general, the *pipelining of registers* (flip-flops) is a common technique to increase the clock-speed of a design. Conversely, if the overall delay of two succeeding paths is smaller than the time period of a clock-cycle, these two paths *could* be combined (e. g., by bypassing a flip-flop through the multiplexer in a BLE) to reduce the necessary *number of clock-cycles*. Hence, with a predefined clock-speed, the *slack* on a path can be *negative* in case that the delay exceeds the clock-period.

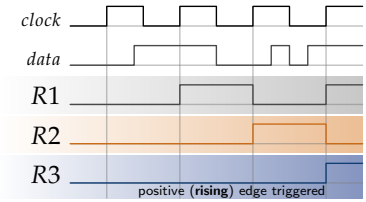
Remark 8. *For the model introduced in Section 6.3, positive slack is assumed in any case by shifting all slack values by the most negative one if necessary.*

In another scenario where only a circuit is given, the timing analysis can determine the (*approximated*) maximal clock-speed for the circuit to run correctly. In this case, the *positive* (but differently sized) slacks at the end of paths quantify how *critical* each path is in the context of the overall design. This slack occurs due to different delay (and *arrival*) times of multiple combined paths and is explained in the following section. Even if the circuit is finally completely established (including the routing), a certain overestimation of the delay is advisable to compensate variations and to guarantee correctness of the circuit.

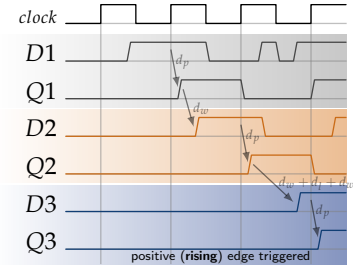
Remark 9. *Timing-constraints can even be defined manually by the designer. This can be necessary for specific paths in the design if these have to meet conceptual timing restrictions.*



(a) A simple circuit



(b) Theoretical sequence plan



(c) Actual sequence plan

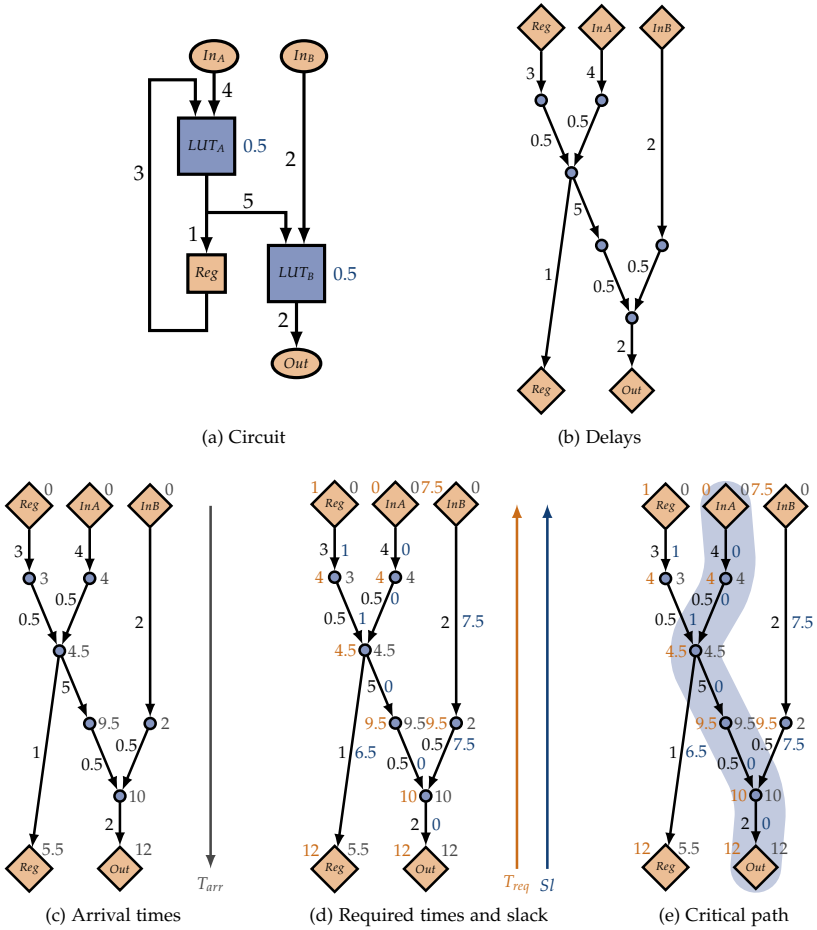
Figure 7: Flip-Flops and delay types

2.2.4 *Slack and critical path(s) calculation*

In the previous section, the sources of delay and the resulting restrictions for the clock-speed of a design were presented. Considering and rating these effects in complex designs is both crucial and difficult. For the model presented in this work, the calculation (or approximation) of *slack* in a design is (more precisely: *can be*) used to influence the placement of logic elements onto the available FPGA architecture to finally improve the circuits timing.

Slack in a simple concatenation of logic elements (like the one shown in Figure 7) appears due to the difference between the clock-period and the logics' and wires' delays. However, even without considering a clock, slack still occurs in more complex designs.

Consider the circuit shown in Figure 8a (this example circuit is taken from the book of Betz et al. [21, Figure 2.12]). The numbers indicate the *delay* introduced by the wires and by the logic elements. For an analysis of the overall timing, the so called *timing graph* (Figure 8b) is created, a directed acyclic graph (DAG). *Registers* and *(primary) in- or outputs* are the 'borders' of a clock-period and are, therefore, the leaves of the graph representing the timing of one such clock-period. They have either *no* or *exclusively incident edges*. Each element that introduces delay (wires, logics) is included with an *edge* between the elements' *contact points* (called *fanin*s and *fanout*s), which themselves become the nodes of the graph. The delay of the elements is then



(a) Circuit (b) Delays

(c) Arrival times (d) Required times and slack (e) Critical path

Figure 8: Circuit (a) and timing graph traversals for slack calculation (b)-(e)

assigned to the edge as an *edge weight* (denoted by $delay(i, j)$ on the edge from node i to node j). The delay of a path between any two points can now be calculated by summing up all delays on this specific path. Starting from the *sources* of the graph (on the top), it can be calculated when the initial signal arrives at any point in the circuit. This time is called the *arrival time* and it is assigned to the *nodes* in Figure 8c (gray) by tracking all paths on a *breadth-first*

traversal from the top to the bottom, summing up the latest arrival times on nodes of the graph (cp. equation (1)).

$$T_{arr}(i) = \max_{\forall j \in fanin(i)} \{T_{arr}(j) + delay(j,i)\} \tag{1}$$

See, for example, the predecessor of *Out*. Even though the incoming path on the right fanin only takes 2 units of time (e. g., *nanoseconds*), the attached logic element LUT_B can calculate its output only after the signal on the left fanin also arrived. As this signal takes 10 units of time, the (maximal) arrival time on this node is $max(2, 10) = 10$.

After the arrival times have been calculated, the highest possible clock-speed of the circuit can be estimated by taking the *maximal* overall delay time of a path (D_{max}), accordingly the latest arrival time on the outputs, as the *minimal* clock-period. In the presented example circuit, this value is $D_{max} = max(12, 5.5) = 12$. When assuming nanoseconds as a unit of time, 12 ns correspond to a *maximal* clock-speed of $\frac{1}{12} \cdot \frac{10^9}{10^6} = 83\frac{1}{3}$ MHz.

As it was already mentioned before, not all paths in the graph limit the clock-speed to this value because, at several nodes in the timing graph, *some* signals have to 'wait' for others what induces *slack* for the 'waiting path'.

To calculate the slack, a backwards *breadth-first traversal* from the bottom to the top of the graph is performed. At each node i, the latest possible arrival time T_{req} (orange) that does not increase the overall time needed to process the circuit (therefore does not increase D_{max}) is calculated as the minimal difference between its successors' latest arrival times T_{req} and the delay between these two nodes (see Figure 8d and equation (2)).

$$T_{req}(i) = \min_{\forall j \in fanout(i)} \{T_{req}(j) - delay(i,j)\} \tag{2}$$

The slack on an edge (blue) between node i and j is now simply the difference between the available time span $(T_{req}(j) - T_{arr}(i))$ on the edge and the actual $delay(i,j)$ on the edge (see equation (3)).

$$slack(i,j) = (T_{req}(j) - T_{arr}(i)) - delay(i,j) \tag{3}$$

Thereby, the slack quantifies *how critical* an edge is for the overall speed of the circuit and is an important criterion for deciding which paths to route as short as possible (or on faster wires if available) and which to relax whenever it becomes necessary due to architectural restrictions. Finally, a *critical path* of the circuit is a path with no slack on its edges (highlighted in Figure 8e). However, it is not necessarily unique. Adding delay to any of the edges on it (e. g., by longer wires) would directly increase D_{max} and consequently decrease the overall maximal possible clock-speed of the circuit. Instead, a longer wire can be used for connections with high slack without

any drawbacks concerning the circuits final speed. This fact will be used in the iterative *slack graph morphing* approach in Section 6.3.

Thus, slack estimations are important for the timing of circuits and are used in general VLSI design (cp. Youssef and Shragowitz [189]) or specifically for FPGAs (e. g., by Frankle [63]).

2.2.5 *Heterogeneous FPGAs*

The FPGA architecture explained so far consists of identical *CLB* and *I/O* blocks combined with a routing architecture. Even though any kind of *Boolean* function could be implemented on such an architecture (as long as there are enough resources on the chip), heterogeneous architectures have become more and more important in the past. A simple form of a heterogeneous FPGA could contain different types of CLBs, for example, with differently sizes LUTs. Thus, CLBs could not abundantly swap positions to improve wirelength on such an architecture.

There are numerous functions that are frequently used in FPGA designs and that can consume a large portion of the CLBs and routing resources. Thus, including such functions as specialized blocks on heterogeneous FPGAs can be advantageous in several ways. One important such function is a *multiplicator*. Multiplicators are on the one hand very often used in designs, on the other hand it is already relatively complicated to realized them with CLBs so that a large amount of area would be used for them. Hardwired (*not* reprogrammable) elements for such operations can be realized more compactly and they operate faster than a reconfigurable logic pendant. Moreover, such specialized blocks simplify the synthesis (see Section 2.4). However, FPGAs are meant to be *initially very general* reconfigurable chips that are *finally* used as *special purpose* hardware. Though, specialized hardwired blocks on the architecture should only be added if they are likely to be useful in *many* designs. Multiplicators, for example, are. *Digital signal processing* (DSP) blocks on the architecture can, for example, contain several multipliers of different sizes and other frequently used functions like *adders*, *subtractors* or *accumulators*. Such DSPs are, for instance, present on *Altera's Stratix Series FPGAs*. A further important type of basic 'hard' blocks on a heterogeneous architecture are *block memories (RAMs)*. They can be used to store data and several of them can be combined if more storage is needed. For example, Kuon and Rose [116, 117] showed the great benefits of heterogeneous FPGA architectures with DSP and memory elements compared to their homogeneous counterparts. In addition, they compared them to ASIC designs in terms of area, speed and power consumption. They came to the conclusion that adding heterogeneous blocks like hardwired multipliers and block

memories leads to a substantial improvement concerning area and power consumption. However, the measured impact on the delay of the circuit was relatively small. The effects on the routing of such a composition of I/O intensive coarse-grained units with fine-grained logic units was, for example, investigated by Yu et al. [190].

Remark 10. *Even extremely complex blocks like 'ordinary' **CPUs** can be part of today's heterogeneous FPGA architectures.*

2.3 THE FPGA 'BASELINE MODEL' (IN VPR)

In the following section, the basic elements of a generic heterogeneous island-style FPGA architecture and their arrangement on the chip are discussed in more details. This includes all relevant components of the architecture that is used in all later benchmarks.

The set of logic blocks

As described before in Section 2.2.5, heterogeneous FPGAs can have great advantages in comparison to homogeneous ones. However, heterogeneity introduces further complexity and challenges for *placement* and *routing*. The heterogeneous architectures that are considered in this work are made up of four basic block types:

- Configurable Logic Blocks (CLBs)
- In- and Output pads (I/Os)
- Multiplicators (MULs) and
- Memory Blocks (MEMs)

However, the method can easily be extended for other types (see Section 5.8). The four mentioned types are supported by the *Versatile Place and Route* (VPR) framework [19] which was initially developed at the *University of Toronto*. The actual implementation of the presented approach in this work is carried out in the VPR framework. *VPR* is the *place-and-route* part (see Figure 11) of VTR, an open source FPGA CAD tool that is widely used by the FPGA research community for testing and extending methods (e. g., for the integration of a power model by Poon et al. [151]) or for the simulation of different possible FPGA architectures in order to gain conclusions about hardware decisions before manufacturing them (see, for example, the architectural conclusions of Betz et al. [21, Chapter 8]). VPR even became part of the *SPEC 2000* [94] suite of computer benchmarks, a standard set of applications which is used to determine the productive and realistic speed of workstations.

The overall FPGA architecture

Some parts of an FPGA architecture were not yet discussed as they are not explicitly considered in the presented approach. However, they should be named at this point to complete the overall picture. The principal FPGA architecture that is targeted in this work is shown in Figure 9.

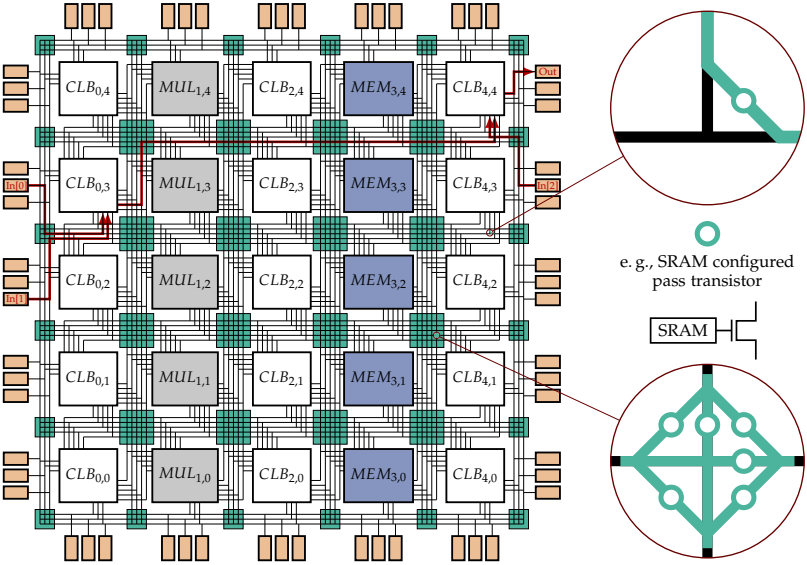


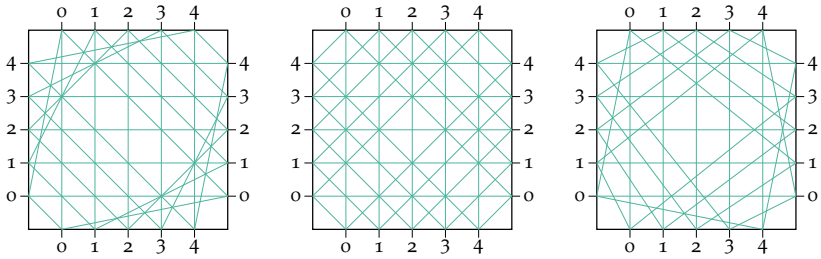
Figure 9: Heterogeneous island-style FPGA architecture

The heterogeneous *island-style* FPGA architecture contains all mentioned block types from the previous section. In the figure, all blocks have the same size. This may not be the case in general but does not play a major role for the presented method as each element is simply represented with its center coordinates. Specialized blocks like multiplicators and memory blocks are often larger than the simple CLBs, thus, they span multiple block positions. However, they are often available in dedicated columns of the island-style architecture. The I/O pads are surrounding all logic elements which is a fact that is important for the presented method and indeed kind of 'natural' as these pads connect the FPGA to the surrounding system, e. g., on an embedded device like a *SoC*.

Remark 11. *In Figure 9 and also in the later introduced graph models, all logic block types are assumed to be of equal size. However, in the future, the* ***actual size of a block*** *could additionally be considered for the graph layout. The layouting*

algorithm used in this work (see Section 4.2) is already able to take such node sizes into account.

Apart from the different wirelengths that were already mentioned before, the routing architecture also contains programmable *connection boxes* that connect the logic blocks to the wires. In addition, programmable *switch boxes* are available to connect two wires in order to 'change the direction' on a path (e. g., from a vertical to a horizontal channel). Both types can, for example, be programmable by *SRAM cells*. The switch boxes do normally *not* allow the configuration of *all possible* connections of wires that reach the box. The choice of the switch box type for an FPGA architecture addresses a trade-off between *routability* and *area efficiency*. Three basic types of such switch boxes are shown in Figure 10.



(a) Disjoint topology (b) Universal topology (c) Wilson topology

Figure 10: Switch box topology types

For comparisons of the different types, see, for example, the work of Rose and Brown [159] from 1991 or more current investigations by Chang and Wong [32] or Fan et al. [59] about *universal* and *disjoint* switch box topologies, respectively. The disjoint switch box topology, for instance, simply allows to switch from horizontal track i to vertical track i and vice versa or to follow the current wire. Today, different vendors include different switch boxes in their FPGA designs.

Even with restrictions concerning the connectivity in the switch boxes (cp. Figure 10), interconnect has been and still is a dominant part of the FPGA in terms of area consumption (see, for example, DeHon [46]).

In addition to all the mentioned options for the design of an FPGA architecture, another degree of freedom is available by assigning different *channel widths* on the routing architecture. Instead of having the same number of routing channels in all rows and columns of wires, it can, for example, be advantageous to have more tracks in the *inner* regions of the chip while reducing the number further out on the chip (as indicated in Figure 9). The channel

width will therefore be incorporated in the presented model in Section 5.4.2. A further improvement included by Xilinx in their architectures consists of extra *I/O-channels* at the I/O-pads to improve accessibility to these elements (cp. Tavana et al. [178]). Another approach is to add additional channels near the center of the FPGA. Even though this idea has been supported by many researchers in the FPGA community to improve routability (cp. Betz et al. [21, Section 5.4.1]), it was not profitable in practice.

Simulation tools like VPR greatly help to test such considerations before manufacturing the FPGA and therefore make it possible for researchers and manufacturers to test and develop new ideas at 'no costs'. In VPR, the architecture for the simulation of the design is 'created' based on a principal input description including the different blocks and their possible arrangement options (see Section 2.4).

Remark 12. *In addition to the architecture model, Figure 9 also contains an example for a 'placed and routed' net (indicated by red lines) from this coarse grained perspective on the FPGA (without visible detailed routing inside the logic blocks).*

Other architecture types

As already stated, modern FPGA architectures contain different types of blocks and their arrangement on the chip is classically following one of the following design principles: *island-style (symmetrical)*, *row-based* or *hierarchical*. The basic layout of *island-style* FPGAs (or PLDs in general) has already been shown in Figure 9. They get their name from the fact that each block (containing different types of logic elements) is surrounded by routing resources, forming structures that look like '*islands of logic in a sea of interconnect*' while everything is surrounded by I/O pads. *Row-based* architectures instead consist of rows of logic blocks interspersed with rows of interconnect and surrounding I/O pads. Finally, another class are hierarchical FPGAs which can, for example, follow a hierarchical layout idea, basically like the H-tree shown in Figure 5.

Even though VPR was initially developed for island-style FPGAs, it is also possible to describe other architecture types. However, this work primarily targets *island-style* FPGAs for the method's development and the benchmarks. Still, many parts of the method could easily or even directly be used to create placements for the other types (see Section 5.8). Another recent development is the use of *three-dimensional* (3D) FPGAs (cp. Chen et al. [34]), Section 5.8 will give a short outlook how the method can directly be extended for them.

Remark 13. *For example, **Xilinx** usually designs its architectures in island-style while **Altera** builds many hierarchical FPGAs [22].*

2.4 COMPILATION FLOW FOR FPGAS

Remark 14. *As this work focuses on a specific part of the FPGA compilation flow, namely the* ***placement*** *of logic blocks onto* ***heterogeneous*** *FPGA architectures, the other main parts will only be explained relatively briefly. A lot of additional details about all following steps in the compilation flow for FPGAs, and also about their specific realization in VPR, can be found in other works, in particular in the book of Betz et al. [21, Figure 2.12], which also explains the VPR framework on which the implementation of this work bases.*

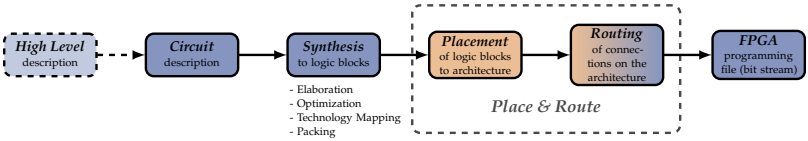


Figure 11: Main steps in an FPGA compile flow

A rough description of the compilation flow for an FPGA architecture is depicted in Figure 11. First of all, it is important to keep in mind that the 'compile process' generates a hardware description of the desired logic. This is a *fundamental difference* to compilers for, e. g., CPUs which create a machine-readable sequence of instructions operating on data in the predefined hardware.

To describe the behavior of a desired piece of hardware in an FPGA, higher level *hardware description languages* (HDLs) like *Verilog* or *VHDL* are commonly used instead of describing the nets of logic gates manually. Parts of the code that should be translated into the FPGAs hardware have to follow the *register-transfer level* (RTL) design abstraction for *synchronous digital circuits*. An RTL description may describe the traversal of digital signals (data) and their transformation in logics between registers. This can, for example, be realized in either of the above mentioned two alternatives.

Even though this work will not deal with actual coding for FPGAs at all, at least a very simple example of a code written in a hardware description language should be given.

Listing 1 contains the implementation of an OR gate in *VHDL* (taken from the EDA playground[5], cp. Figure 1). The `entity` section defines and names the I/O ports of the circuit while the `architecture` declaration describes its logical behavior. In this case, the logic has two inputs (named `a` and `b`) and one output (named `q`). The RTL description of the architecture now simply

5 http://www.edaplayground.com/s/example/615 (accessed 04 May 2016)

determines that the two inputs `a` and `b` should be logically combined by an '`or`' function and the result should be passed to the output `q`.

```
-- Simple OR gate design
library IEEE;
use IEEE.std_logic_1164.all;

entity or_gate is
port(
  a: in std_logic;
  b: in std_logic;
  q: out std_logic);
end or_gate;

architecture rtl of or_gate is
begin
  process(a, b) is
  begin
    q <= a or b;
  end process;
end rtl;
```

Listing 1: OR gate design in VHDL from the EDA playground

Even though it may already look quite complicated to create such simple functions, once they are created, they can be nested by calling each other just like in programming languages for 'normal' CPUs. However, some functionality can be described even more simple than in such *already relatively high level* hardware description languages (at least simple compared to pure descriptions on the gate level). Recently, approaches converting a subset of *ANSI C/C++/SystemC/Matlab* code to RTL descriptions appeared, e. g., *High-Level Synthesis* (HLS) (see, for example, the publication of Martin and Smith [137] or the overview of Meeus et al. [140]).

The *synthesis* step of the compile flow converts the description of the logics behavior into a 'gate-level' netlist. In this step, technology independent optimizations can be performed that are not related to the actual architecture but only base on the evaluation of the logic. This optimization step is comparable to the *machine independent optimization* (e. g., *dead code elimination*) in 'normal', for example *C*, compilers. To run these optimization steps in the synthesis, the *elaboration* phase first has to 'roll-out' and translate the *hierarchical* (e. g., *VHDL* or *Verilog*) code down to the basic blocks of the FPGA, in particular *LUTs*, *flip-flops*, *in- and outputs* etc. In VPR, the tool *Odin* [101] is included to handle this task.

After elaboration, the *technology mapping* converts the general description of logic into one that specifically targets the desired hardware architecture. This, for instance, includes the ascertainment of given LUT sizes. These steps,

along with possible *technology independent* and *technology dependent optimizations*, form the synthesis phase and can, for example, be performed by the 'basic' *ABC* tool [23] (in VTR) or in combination with extended techniques like *WireMap* [102] for improved technology mapping (published by Xilinx).

It has been described in the previous sections that the architecture on an FPGA is somehow hierarchical and that the view on CLBs is coarse-grained in the sense that such logic blocks can contain several BLEs which in turn contain LUTs. Thus, multiple basic blocks like LUTs (or BLEs) can be assigned to the same CLB. This process of *packing* can certainly influence the design's routability and the resulting timing. Successive BLEs that are on a (or on multiple) *critical* path(s) should, for example, be packed into a common CLB to minimize the wire delay between them whereas *uncritical* BLEs can be scattered among different CLBs. Such a *timing-driven* logic block packing is performed by *T-VPack* [136] (timing-driven further development of the basic *VPack* tool) in VTR. Basically, connections between BLEs are rated concerning their criticality compared to the *most uncritical* connection with a maximum amount of slack (slack_{max}) following formula (4) and finally packed into shared CLBs if they are critical (see Marquardt et al. [136] or Betz et al. [21, Chapter 3.1]).

$$\mathrm{criticality}(i,j) = 1 - \frac{\mathrm{slack}(i,j)}{\mathrm{slack}_{max}} \tag{4}$$

After the synthesis step with all its subroutines, the FPGA is described by a *net of logic elements* (CLBs, I/Os, MULs, MEMs in the described heterogeneous model) which has to be *placed* and *routed* on the architecture. As there are, in general, several available slots on the architecture for every single element of each type, this process offers a great further degree of freedom but also demands for techniques to create a 'good' such assignment.

Remark 15. *Due to the complex (heterogeneous) logic and routing structure of FPGAs, both the coding and the synthesis take generally much longer than for simpler reconfigurable logic devices like CPLDs. Optimization techniques like the mentioned* ***register pipelining*** *for FPGAs are on the one hand often not necessary and on the other hand simply not possible due to the very small number of available flip-flops on CPLDs. It should have become obvious that high level description languages like VHDL are, especially for FPGAs,* ***very important*** *to implement more complex projects.*

The assignment of the logic elements onto the architecture is called *placement*. A placement can follow different goals. One can be to optimize the *timing* by short connections on the chip to be able to set a high clock speed for the final layout (*timing-driven placement* - see, for example, Marquardt et al. [135]). Another approach could target a good *routability* of the placed

blocks on the architecture. As routing resources are limited, a pure timing-driven placement can result in a block assignment that makes it impossible or at least very difficult to find enough (or fast enough) connections through wires and switch boxes on the architecture. A placement targeting this goal is called *routability-driven placement* (see, for example, Kim et al. [110]). A timing-driven placement could use the overall wirelength or the length of the critical path as the target to optimize while a routability-driven placement can, for example, take wire densities on the chip into account. The placement algorithm *VPlace* used in VPR can, in a way, consider both goals in the placement. Altogether, it is important to note that the placement *influences* the *timing* and *routability*. Further goals are certainly conceivable. E. g., a combined place-and-route approach called *Independence* with tangible routability-driven placement comparing itself to VPlace has been published by Sharma et al. [167].

VPR Placer

VPlace uses a *simulated annealing* approach to optimize an initial random configuration by pairwise swaps of blocks on the architecture. In this process, swaps are accepted if they improve a certain cost function or, with a decreasing probability throughout the process, even if they worsen it. This peculiarity allows to escape local optima. For details on simulated annealing, see Section 3.2.7. To find pairs of blocks that are considered for swapping, a *first* element is chosen randomly and a *second* one is taken from a frame of size D^{limit} (with $1 \leqslant D^{limit} \leqslant FPGA\_size$) around the first one in the current assignment in order to check whether this swap meets the requirements of the cost function or not. The frame is continuously resized depending on the number of accepted moves in the previous step. The more successful swaps were found (within a number of swaps), the more is the frame kept large or even enlarged. The fewer successful swaps are found at a specific temperature of the annealing, the more is the frame *shrunk* so that in the end only swaps of elements nearby each other are considered. More details about this *adaptive annealing*, about the *temperature update schedule* and other peculiarities of the method can be found in Betz et al. [21, Section 3.2.2] and other referenced works in this section.

As the calculation of the exact total wirelength after swaps of blocks would be too time consuming in the process and as the actual final routing is not known at this point, a norm called the *semi-perimeter metric* of a bounding box surrounding a *net* with *terminals* (connected blocks) is used to approximate the wirelength that will be necessary to route the net (see Figure 12).

The wirelength necessary to connect all the terminals in net i is approximated by the sum of its *vertical span* ($bb_y(i) = y_{max} - y_{min}$) and its *hor-*

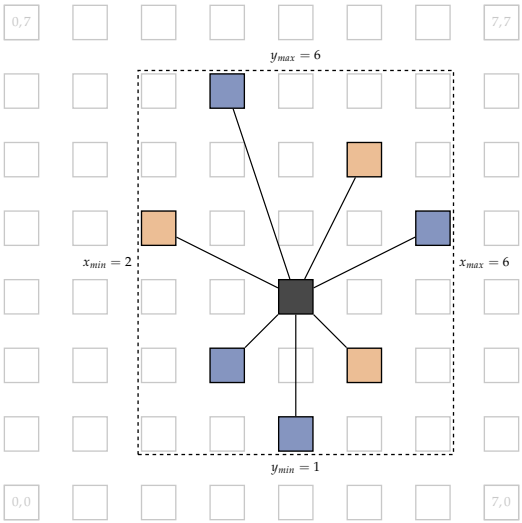


Figure 12: Bounding box of net with 8 terminals

izontal span ($bb_x(i) = x_{max} - x_{min}$). An update of the cost function after two elements have been swapped generally only requires an update of these properties of (at most) two bounding boxes instead of remeasuring all connections that are attached to the moved terminals.

The resulting cost function in VPlace is shown in equation (5). It contains a factor $q(i)$ which grows with the number of terminals in the net following the work of Cheng [125] from the field of ASIC designs.

$$cost_\beta = \sum_{i=1}^{N_{nets}} q(i) \left[\frac{bb_x(i)}{C_x^{avg}(i)^\beta} + \frac{bb_y(i)}{C_y^{avg}(i)^\beta} \right] \tag{5}$$

In addition to the wirelength approximation, the cost function implemented in VPlace contains a *linear congestion model* with $C_x^{avg}(i)^\beta$ and $C_y^{avg}(i)^\beta$ and $\beta = 1$. For the region where the net is placed, these two values contain the average number of wires per horizontal and vertical channel, respectively. Increasing β would penalize higher congestion in certain regions *superlinearly* (as it makes the subsequent routing more difficult or even impossible). However, the authors of VPlace experimentally verified that a linear congestion (thus $\beta = 1$) led to the highest quality of placements in practice (see Betz at al. [21, Section 3.2.3]). With $\beta = 0$, the approach neglects the routing considerations and assumes a 'traditional' pure timing-driven nature.

Remark 16. *A common effect in such simulated annealing approaches is, that (as a rule of thumb)* ***"reducing the number of moves per temperature by a factor of 10, for example, speeds up the placer by a factor of 10 and reduces the final placement quality by less than 10%"***[6]*. Therefore, further improvements come at a high price.*

In contrast to the already mentioned *combined (place & route)* method of Sharma et al. [167], the approach presented in this work is a placement method after which the routing has to follow separately. This is also the case for the compared baseline method in VPR which applies simulated annealing.

Thus, the actual *routing* of connections follows after the placement was performed. Routing is often realized in two steps: *global* and *detailed* routing. However, numerous combined *one-step* global-detailed-routers are also available, e. g., the work of Lee and Wu [123] or the one of Plazcewski [148] whereas only the one of Lee and Wu takes timing considerations into account to assign critical connections to fast wires. *Global* routing (cp. Chang et al. [31]) assigns the *pins* and *channel segments* (threads of multiple parallel wires between the logic blocks) used for the connections between the blocks in a *coarse-grained* manner without specifying the explicit wires that will be used. This is consequently done in the *detailed routing* phase (e. g., Brown et al. [25]). In addition, *local* routing (*within* the logic block clusters) sets the necessary connections *inside* of blocks (*clusters*), e. g., within CLBs.

Remark 17. *Global routing and the effect of a specific placement on the feasibility of the global routing is taken into consideration within the presented approach in this work in Section 5.4.2. Thus, it is not only influencing the placement, but also (partially) the routing. This has been indicated by the orange regions in Figure 11 on page 33.*

Just as there are very many different approaches besides simulated annealing to solve the *placement* problem (which will be discussed in more details in Section 5.1), there are also plenty of attempts to create good *routings* (see Betz et al. [21, Section 2.2.3] for an overview of early fundamental works in this field). In the recent past, some interesting works specifically targeting heterogeneous architectures have been published (see, for example, Deepak and Rajendra [156] from 2005 or Zha and Athanas [191] from 2013). However, this work is not primarily about routing. It only aims at supporting the router with a 'router-friendly' placement. Thus, it does *not* target a *specific* router but makes some general assumptions about how *a router* will principally proceed after the placement.

6 quote from Betz at al. [21, Section 3.2.2]

In general, it is desirable to use the shortest possible connection between all pairs of connected elements. However, routing resources are limited and elements like the switch-boxes on FPGA architectures with topologies which can not contain all possibilities to interconnect meeting wires further restrict routing capabilities. A shortest path between two points on the architecture can, for example, easily be found with Dijkstra's algorithm [48] from 1959 or the related *A*-search* algorithm from 1968 (see Hart et al. [91]) which both generally operate on a graph representation. In case of a regular routing grid in two (or three) dimensions (like it is present on an FPGA), the problem can even be solved with a basic simple technique called *maze-routing* or *wave-propagation* (see the work of Lee [122] from 1961). A more sophisticated method is, for example, the *MaizeRouter* published by Moffitt [141], which is an effective global router using and extending the basic approaches.

VPR's router fundamentally bases on the *PathFinder negotiated congestion-delay algorithm* [138] but includes several enhancements like modeling the delay with *Elmore*'s model (see Section 2.2.2) instead of assuming constant delays in a linear model. With Elmore's approach, even the effect of different buffer types can be taken into account. Another extension is the *local wave-front restart* technique (see Betz at al. [21, Section 4.3.2]) to reduce the time that is spend in the routing phases. The PathFinder algorithm itself takes delays into account to route critical paths through fast tracks, even if the tracks are already *congested*. In a subsequent *rip-up* and *rerouting* step (called a *routing iteration*), non-critical paths (with *slack*) in congested areas can be redirected to a detour.

If each connection would, instead, be strictly routed on the shortest possible path until a wire channel is congested, later routes would have to take greater detours. In general, the *order* of the routes in the routing queue would greatly influence the chance of obtaining better resources in such a simplified approach. However, Section 5.4.2 will explain how such a simple model without any rerouting is used in this work to simulate and *rate* the routability of a *placement*. Anyway, the ability to reroute nets is extremely important for 'real' routers. A different order of the connections in the routing process can possibly already improve the situation, e. g., if an evitable constellation occurs which would significantly deteriorate the timing. For example, if an uncritical connection has been routed early in the process. As a consequence, this connection could be 'blocking' wire segments that would in fact be needed for a more critical connection later in the procedure.

Remark 18. *Finding the **minimum rectilinear Steiner Tree (MRST)** of a net to interconnect it is an $\mathcal{NP}$-hard problem and would, thus, not be applicable for compilation situations in general. Instead, approximation algorithms or heuristics like those mentioned above are often used in practice.*

Part III

What is behind all this?

The first chapter of this part (Chapter 3) presents a model to describe the chip-layout problem as a ***quadratic assignment problem*** *(**QAP**). Solving QAP is NP-hard and even today's algorithms are only capable of solving very small QAP instances to optimality. Nevertheless, promising results can be obtained with several (meta-)heuristics, due to which three main classes of iterative neighborhood search based methods are presented and compared in this chapter. Therefore, an idealized model of a chip with known optimal solution is defined, perturbed and used as the input for the aforementioned algorithms. The results show distinct peculiarities of the methods and lead to an initial justification of the subsequently presented approach.*

Chapter 4 contains a survey on 'the evolution of force-directed graph drawing methods' and explains the force-directed model and implementation used in this work including several improvements that the method contains in comparison to 'traditional' ones. Two fundamentally used and extended concepts for this work are the FM^3 *algorithm and Tutte's layout approach with fixed nodes. To sum up, the chapter aims at providing principal, practical and figurative reasons for the strategy chosen in the presented framework.*

3

The Quadratic Assignment Problem

"Es gibt nichts Praktischeres als eine gute Theorie."
— Kurt Lewin, 1951 —

(Nothing is as practical as a good theory.)

Contents

3.1 MODEL THE PROBLEM OF CHIP-LAYOUTING BY QAP

The problem of chip placement (or *floorplanning*) with equally dimensioned facilities and a priori defined locations can be formalized by the *Quadratic Assignment Problem* (QAP) [8], introduced by Beckman and Koopmans [16] in 1957 in the mathematical field of *operations research*. The QAP is a special case of general floorplanning from the category of 'facility location problems'. It can be formulated as follows:

Definition 1 (Quadratic Assignment Problem (informal))**.** *Given a set of* n ***facilities*** F *and* n *respective* ***locations*** L*, along with a definition of* ***distance*** *between two locations and* ***flow*** *(sometimes* ***weight****) that has to be transported between every pair of facilities.*

Find an assignment of the facilities to the locations that minimizes the sum of costs which are in turn the product of distance and flow.

Definition 2 (Quadratic Assignment Problem)**.**

Given :

$F = \{f_1, \dots f_n\}$	*a set of* n *facilities*
$L = \{l_1, \dots l_n\}$	*a set of* n *locations*
$d : L \times L \to \mathbb{R}$	*a distance function*
$w : F \times F \to \mathbb{R}$	*a flow function*

Find :

$\pi : F \to L$	*an assignment*

minimizing :

$$\sum_{f_i, f_j \in F} w(f_i, f_j) \cdot d(\pi(f_i), \pi(f_j)) \qquad \textit{the total cost.}$$

In the general (non-homogeneous) formulation of the *Koopmans-Beckmann form*, the cost function contains additional linear costs b as addend, representing costs that arise from placing a facility f_i to location $\pi(f_i)$, regardless of the placement of the other facilities:

$$\sum_{f_i, f_j \in F} w(f_i, f_j) \cdot d(\pi(f_i), \pi(f_j)) + \sum_{f_i \in F} b(f_i, \pi(f_i)) \; .$$

Another commonly used notation of the problem in the literature is shown in Definition 3.

Definition 3. *Given two matrices W (flow) and* D *(distance) of size* $n \times n$ *with entries* w_{ij} *and* d_{ij}*. Find a permutation* π *of the underlying set of* n *elements that minimizes (6).*

$$\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}\, d_{\pi(i)\pi(j)} \left(+ \sum_{i=1}^{n} b_{i\pi(i)} \right) \tag{6}$$

Lawler [121] introduced a generalized version of the QAP by representing the overall costs (e. g., the product of *flow* and *distance*) in a combined four-dimensional matrix C with entries c_{ghij} and the target to minimize (7).

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij\pi(i)\pi(j)} \left(+ \sum_{i=1}^{n} b_{i\pi(i)} \right) . \tag{7}$$

3.1.1 *Problem definition*

Assume that a chip contains *slots* for different fundamental types $\{t_1, t_2, \ldots, t_c\}$ of *units*. Let $u_j^{t_i}$ denote the j-th unit which is of type t_i and $s_l^{t_k}$ the l-th slot of type t_k. The units correspond to the aforementioned facilities while the slots in the model correspond to the locations. A *unit* $u_j^{t_i}$ can be assigned to *slot* $s_l^{t_k}$ if they are of the same type, therefore if t_i=t_k.

The following section is based on a idealized architecture model to apply QAP as a study case on an 'FPGA-like' chip. It is assumed that a chip is a square grid of $N \times N$ slots and that all these are of the same type. Due to that fact, slots and locations can be used equivalently in the following. Thus, the task is to place a number of $n = N \cdot N$ units on an $N \times N$ grid.

Remark 19. *The orange coloring of units in figures of this section is applied to mark the elements on the outer frame of the idealized chip as they are typically of another type (**I/O**) than the inner elements (**logic units**). Nevertheless, this distinction is not involved in the assignments or any other part of the idealized model at this point. It is indeed used to present some effects of different placement methods later in this section.*

First of all, a metric to define a measure of distance between two locations on the grid is needed. For that, the available slots on the integer grid are indexed by a simple enumeration as follows:

$$\text{the location with coordinates } (x_i, y_i) \text{ is referred to as } l_{x_i \cdot N + y_i}. \tag{8}$$

The distance between two locations l_i and l_j can then be described by a distance matrix

$$\overline{\mathcal{D}} \in \mathbb{R}^{n \times n} \quad \text{with} \quad \bar{d}_{ij} \widehat{=} \text{distance between } l_i \text{ and } l_j \ . \tag{9}$$

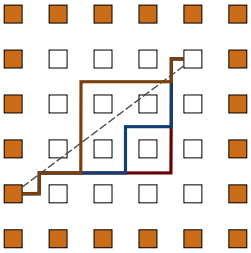


Figure 13: Different Manhattan routes and the direct connection

The distance in the model is measured by the L_1-norm, also known as the Manhattan distance, which is the sum of horizontal and vertical components, formally

$$\|(x, y)\|_1 = |x| + |y| \tag{10}$$

for the two-dimensional case and $\|\vec{p}\|_1 = \sum_{i=1}^{n} |p_i|$ in general for a vector $p \in \mathbb{R}^n$. The Manhattan norm is a good estimate for the distance between two locations in the model, as it measures the length of a shortest possible wire connection following only orthogonal paths on the chip. In general, such a *routing architecture* is present on FPGAs and many other *integrated circuits*. A path with the shortest Manhattan distance between two locations is usually not unique. Figure 13 shows different such shortest L1-connections on a grid and the direct connection in terms of the L2-norm, also referred to as the Euclidean norm with $\|(x, y)\|_2 = \sqrt{x^2 + y^2}$ and $\|\vec{p}\|_2 = \sqrt{\sum_{i=1}^{n} p_i^2}$, respectively. While the Euclidean norm of coordinates on the integer grid maps to *real* values, the Manhattan norm of integer coordinates is always *integer*.

The 2-dimensional coordinates of each location l_i on the integer grid with width N, assigned to the grid as determined in (8), can be recalculated by

$$x(l_i) = i - N \cdot \left\lfloor \frac{i}{N} \right\rfloor \quad , \quad y(l_i) = \left\lfloor \frac{i}{N} \right\rfloor . \tag{11}$$

Combining formulas (10) and (11), the Manhattan distance d_{ij} between locations l_i and l_j is comprised in

$$\begin{aligned} \mathcal{D} &\in \mathbb{N}_0^{n \times n} \text{ with} \\ d_{ij} &= \left| \left\lfloor \frac{i}{N} \right\rfloor - \left\lfloor \frac{j}{N} \right\rfloor \right| + \left| \left(i - N \cdot \left\lfloor \frac{i}{N} \right\rfloor \right) - \left(j - N \cdot \left\lfloor \frac{j}{N} \right\rfloor \right) \right| . \end{aligned} \tag{12}$$

The *distance-matrix* $\mathcal{D}$ is symmetric and has a zero-diagonal.

The units of a chip are interconnected by *nets* to pass electrical signals from one element to another. Together with their interconnections, the units form a graph $\mathcal{G} = (V, E)$ with nodes $V = \{v_1, v_2, \ldots, v_n\}$ ($|V| = n$) which represent the units of the chip description (resp. the *facilities*) and $|E|$ vertices between units to model the interconnections. The graph can be represented by an *adjacency* matrix

$$\begin{aligned} \mathcal{V} &\in \mathbb{B}^{n\times n} \text{ with} \\ v_{ij} &= \begin{cases} 1 & \text{if } v_i \text{ and } v_j \text{ are connected} \\ 0 & \text{else .} \end{cases} \end{aligned} \tag{13}$$

Due to modeling only connectedness in the construction, $\mathcal{G}$ is an undirected graph and the *connection-matrix* $\mathcal{V}$ is, thus, symmetric.

The *primary goal* is to find an assignment for each node v_i to a location l_j, representable by the *assignment-matrix*

$$\begin{aligned} \mathcal{X} &\in \mathbb{B}^{n\times n} \text{ with} \\ x_{ij} &= \begin{cases} 1 & \text{if } v_i \text{ is assigned to location } l_j \\ 0 & \text{else .} \end{cases} \end{aligned} \tag{14}$$

This assignment is a bijective one-to-one mapping between L and V. Each row and each column contains exactly one '1', whereas all other entries are '0'. It can directly be transformed into, or interpreted as, a permutation π of the nodes in the following sense:

$$\begin{aligned} \pi &: \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\} \text{ with} \\ \pi(i) &= j \quad \text{if } x_{ij} = 1 \end{aligned} \tag{15}$$

Hence, matrices like $\mathcal{X}$ with the aforementioned properties are called 'permutation matrices'.

A *common objective* is to find an assignment-matrix that minimizes the sum of distances between all connected nodes. To achieve this, the connection-matrix $\mathcal{V}$, indicating whether two *nodes* are connected, has to be transformed through the assignment-matrix $\mathcal{X}$, describing to which location a node is assigned. The result is a matrix $\mathcal{V}^{loc}$ indicating which *locations* are connected through the assignment of nodes to the locations:

$$\begin{aligned} \mathcal{V}^{loc} &\in \mathbb{B}^{n\times n} \text{ with} \\ \mathcal{V}^{loc} &= \mathcal{X}^T \cdot \mathcal{V} \cdot \mathcal{X} \text{ with} \\ v^{loc}_{ij} &= \begin{cases} 1 & \text{if location } l_i \text{ is connected to location } l_j \\ 0 & \text{else .} \end{cases} \end{aligned} \tag{16}$$

With $\mathcal{V}^{loc}$ it is now possible to compute the overall sum of distances of the given assignment $\mathcal{X}$ by multiplying the locations' connection-matrix $\mathcal{V}^{loc}$ with the locations' distance-matrix $\mathcal{D}$ and summing up the diagonal elements. An explanation why to sum up the diagonal elements is that each *row* i of $\mathcal{V}^{loc}$ contains the information with which other locations l_j the location l_i is connected ($v^{loc}_{ij} = 1$) while the corresponding *column* i of $\mathcal{D}$ contains the distances between location l_i and the others. The inner product of row i of $\mathcal{V}^{loc}$ and column i of $\mathcal{D}$ is the sum of costs that position i 'produces' and these inner products are present as the diagonal elements of the product-matrix of $\mathcal{V}^{loc}$ and $\mathcal{D}$. A diagonal element a_{gg} of a matrix A is denoted by $\mathtt{diag}_g(A)$ in the following.

Remark 20. *In general, matrix multiplication is not commutative. As the distances are modeled to be symmetric ($d_{ij} = d_{ji}$) and as the graph is undirected, the connection-matrix of locations $\mathcal{V}^{loc}$ is accordingly also symmetric ($v^{loc}_{ij} = v^{loc}_{ji}$), the inner product of* ***column*** *i (instead of row i) of $\mathcal{V}^{loc}$ with* ***row*** *i (instead of column i) of $\mathcal{D}$ would lead to the same results, formally:* $\mathtt{diag}_g\left(\mathcal{V}^{loc} \cdot \mathcal{D}\right) = \sum_{i=1}^{n} v^{loc}_{gi} \cdot d_{ig} = \sum_{i=1}^{n} v^{loc}_{ig} \cdot d_{gi} = \sum_{i=1}^{n} d_{gi} \cdot v^{loc}_{ig} = \mathtt{diag}_g\left(\mathcal{D} \cdot \mathcal{V}^{loc}\right)$*. It does therefore not play a role whether to apply $\mathcal{V}^{loc} \cdot \mathcal{D}$ or $\mathcal{D} \cdot \mathcal{V}^{loc}$ for the cost calculation in this model.*

In that way, the distance of each assigned connection $(v_i, v_j) \in E$ between two nodes v_i and v_j, accordingly the distance between two connected locations, is counted twice, once for node v_i and a second time for node v_j. Thus, the 'correct' sum of distances $\tilde{c}$ for such an undirected graph is half the sum of these inner products.

Remark 21. *The sum of the diagonal elements of a matrix is also called the* ***trace*** *of the matrix, denoted with $\mathtt{tr}(A)$. It can be shown that the trace of a matrix is equal to the sum of its eigenvalues.*

Finally, the costs $\tilde{c}$ can be calculated by

$$\begin{aligned} \tilde{c} &= \frac{1}{2} \cdot \sum_{g=1}^{n} \mathtt{diag}_g\left(\mathcal{V}^{loc} \cdot \mathcal{D}\right) \\ &= \frac{1}{2} \cdot \mathtt{tr}\left(\mathcal{V}^{loc} \cdot \mathcal{D}\right) . \end{aligned} \tag{17}$$

Remark 22. *Examples for the construction of the QAP model in this form can be found in Appendix A.1.*

As

$$\begin{aligned}
\tilde{c} &= \frac{1}{2} \cdot \mathrm{tr}\left(\mathcal{V}^{loc} \cdot \mathcal{D}\right) \\
&= \frac{1}{2} \cdot \sum_{g=1}^{n} \mathrm{diag}_g(\mathcal{V}^{loc} \cdot \mathcal{D}) \\
&= \frac{1}{2} * \sum_{g=1}^{n} \mathrm{diag}_g(\mathcal{X}^{\top} \cdot \mathcal{V} \cdot \mathcal{X} \cdot \mathcal{D}) \\
&= \frac{1}{2} * \sum_{g=1}^{n} \sum_{h=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} x_{hg} \cdot v_{hi} \cdot x_{ij} \cdot d_{jg} \qquad (18)
\end{aligned}$$

and as the prerequisites for a legal permutation-matrix (as stated in formula (14)) can be expressed by $2n$ constraints with n^2 binary variables

$$\sum_{i=1}^{n} x_{ij} = 1 \; \forall j \; \wedge \; \sum_{j=1}^{n} x_{ij} = 1 \; \forall i \; \wedge \; x_{ij} \in \{0, 1\}\,, \qquad (19)$$

the system to be solved can be written as shown in equation (20).

$$\begin{aligned}
\min \quad & \frac{1}{2} \cdot \sum_{g=1}^{n} \sum_{h=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} v_{hi} \cdot d_{jg} \cdot x_{hg} \cdot x_{ij} \\
\text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1 \; \forall j \; \wedge \; \sum_{j=1}^{n} x_{ij} = 1 \; \forall i \; \wedge \; x_{ij} \in \{0, 1\}
\end{aligned} \qquad (20)$$

This formulation of QAP is the *trace* formulation, introduced by Edwards [53, 54]. To find an optimal solution, the factor $\frac{1}{2}$ can, of course, be dropped.

Remark 23. *When relaxing the constraints in formulation (20) to general non-negative real numbers x_{ij}, these constraints directly form the 'Birkhoff polytope' $\mathcal{B}_n$ (e. g., in Paffenholz [146]) containing the so called 'doubly stochastic matrices' with real entries and rows and columns summing up to one. The vertices of the Birkhoff polytope form the set of permutation matrices, as already indicated in formula (15). Consequently, the Birkhoff polytope $\mathcal{B}_n$ is the convex hull of the permutation matrices of size $n \times n$ (known as the* ***Birkhoff-von Neumann theorem****). The Birkhoff polytope appears in various different sectors of mathematics like geometry, enumerative combinatorics, optimization theory and statistics [147].*

Finally, (20) forms a generally non-convex quadratic binary optimization problem with linear constraints and quadratic objective function which is solvable, for example, by *Quadratic Programming* (QP), by general *Integer Programming* (IP) techniques after a linearization of the problem or by *heuristical* methods.

Remark 24. *The following explanations in Section 3.1.4 are partially based on the work 'An Analytical Survey for the Quadratic Assignment Problem' of Loiola, de Abreu, Boaventura-Netto, Hahn and Querido [128] and the ones in Section 3.1.3 on the work 'The Quadratic Assignment Problem' [29] of Burkard, Çela, Pardalos and Pitsoulis. In addition, the survey by Pardalos et al. [150] should be named for any reader interested in a deeper insight to QAP. The main intention of these sections is to provide an overview of the subject area and to reference important works to extend it. Details are provided in the just mentioned surveys and in the referenced original works.*

3.1.2 *The problem's complexity*

In 1976, Sahni and Gonzales [162] proved that QAP is $\mathcal{NP}$-hard and that even finding an ϵ-approximation of the solution is not possible in polynomial time, unless $\mathcal{P} = \mathcal{NP}$. Moreover, QAP is said to be one of the hardest problems of combinatorial optimization. Clausen et al. stated that QAP *"belongs to the hard core of $\mathcal{NP}$-hard optimization problems"* [39]. One main reason for the absence of good solution methods is the absence of good lower bounds for the problem (see Section 3.1.4). These are crucial for improved branch-and-bound techniques to skip branches that are inferior to the best obtained solution so far in the process but also for the evaluation of heuristical approaches [128]. For example, '*A Branch and Bound Algorithm for the Quadratic Assignment Problem using a Lower Bound Based on Linear Programming*' has been presented by Ramakrishan et al. [154].

3.1.3 *Linearizations*

A fundamental approach to come to a solution of QAP is to linearize the quadratic objective function and thereby transform the QP into a *mixed integer linear problem* (MILP), solvable by a wide range of MILP solvers. This can be achieved by the introduction of new variables representing the old quadratic coherences and additional constraints. Even though this idea seems very promising due to the strong presence of research and tools in this area, the linearization adds an enormous amount of variables and restrictions. Coupled with the fact that MILP is $\mathcal{NP}$-hard and that the solution time of MILP instances highly depends on the size of the formulation (variables and restric-

tions), a considerable enlargement of the problem's formulation may make the solution time of such an MILP even for small QAP instances inapplicable.

But even if the exact solution of the respective MILP may not be desirable, the MILP formulation quasi incidentally delivers a starting point for computational lower bounds for the problem's solution when neglecting the constraints of variables being integer or binary (e. g., $x_{ij} \in \{0, 1\}$) and instead solving for real variables ($x_{ij} \in [0, 1]$). Such an *LP relaxation* of the problem contains, besides an infinite number of additional solutions, all feasible solutions of the original problem and can be solved by *linear programming* (LP) solvers. Because a solution of the original problem can not be better than the one obtained under these extended circumstances, the optimal solution of the LP relaxation forms a lower bound for the original problem. Furthermore, *each feasible* solution of the dual of the relaxation is also such a (not as *tight*) lower bound.

A number of linearizations of the problem appeared since *Lawler* published his approach in 1963 [121], many of them basing on a substitution of the quadratic terms in the target function by new additional variables $y_{ghij} = x_{gh} \cdot x_{ij}$ and an accordingly adjusted system of constraints. A good overview of four important linearizations of QAP into an MILP is given in Burkard et al. [29]. While the *Lawler linearization* needs $\mathcal{O}(n^4)$ binary variables and $\mathcal{O}(n^4)$ constraints, Frieze and Yadegar [65] form an equivalent MILP with only $\mathcal{O}(n^2)$ binary variables (like in the original formulation) but plus additional $\mathcal{O}(n^4)$ *real* variables and $\mathcal{O}(n^4)$ constraints. By approximation through a *Lagrangian relaxation*, they obtained a lower bound for QAP based on their MILP formulation and proved that it is *tighter* than particular GL based bounds (cp. Burkard et al. [29, Section 4.3]). Adams and Johnson [2] introduced a mixed integer binary program formulation with (likewise) $\mathcal{O}(n^2)$ binary variables, $\mathcal{O}(n^4)$ *real* variables and $\mathcal{O}(n^4)$ constraints.

While all these linearizations base on the outright $y_{ghij} = x_{gh} \cdot x_{ij}$ substitution, others like the Kaufmann and Broeckx [108] linearization follow a different path using the general linearization method introduced by Glover [72], resulting in a system with only $\mathcal{O}(n^2)$ binary and $\mathcal{O}(n^2)$ real variables with $\mathcal{O}(n^2)$ constraints.

Adams and Johnson [2] also stated a more compact formulation but favored the original one due to a good structure for approximations to obtain lower bounds. For example, '*Improved lower bounds for QAP*' based on the linearization of Adams-Johnson are described in Sergeev's work [166].

Especially the two linearizations of *Frieze and Yadegar* and the one of *Adams and Johnson* have been thoroughly used to obtain lower bounds for QAP by relaxing the formulation (e. g., in Burkhard et al. [29, Section 6.2]).

3.1.4 *Lower bounds*

As already stated, lower bounds can diminish the solution time of branch-and-bound methods to solve MILPs by helping to decide whether a branch of the program is still relevant to obtain an improved solution. One of the oldest and best known class of bounds are the *Gilmore and Lawler* bounds (GL bounds) [71, 121] based on solving a *linear* assignment problem (LAP) with relatively low computational costs instead of the original *quadratic* assignment problem. The linear assignment problem (LAP) can, for example, be solved in $\mathcal{O}(n^3)$ time by the application of the *Hungarian method* but the quality of GL bounds is, in general, rather weak, especially when the problem size increases (cp. Li et al. [124]). Further approaches, like the one of Christofides and Gerrard [38] (CG bounds), base on solving a series of (e. g., $\mathcal{O}(n^4)$) linear assignment problems. Nevertheless, after having early and well-known lower bounds like the GL bounds, the problem of poor lower bounds was attenuated, e. g., by Anstreicher and Brixius [10] to achieve computational solutions of larger instances. They applied bounds based on convex quadratic programming by solving a *semidefinite programming* problem based on spectral decompositions of transformed A and B matrices (cp. description (6), details can be found in Sotirov [171]). This led to the solution of QAP instances with size $n = 30$ on a computational grid in about 7 years of single CPU time (on an HP9000 − C3000) [6]. For example, several more bounds and their quality are reported by Loiola et al. [128].

An important early class of bounds for QAP are '*eigenvalue bounds*'. Eigenvalue bounds [61, 158, 85, 86] exploit the fact that the solution of QAP instances with symmetric real flow and distance matrices can be put into the constraints from Theorem 1 based on the (therefore all solely *real*) eigenvalues of the flow and the distance matrix.

Theorem 1. *Let* A *and* B *in (6) be symmetric with (not-decreasingly sorted) eigenvalues* $\lambda_1 \leqslant \lambda_2 \leqslant \cdots \leqslant \lambda_n$ *and* $\mu_1 \leqslant \mu_2 \leqslant \cdots \leqslant \mu_n$*, respectively. For any permutation* π *of the set, the following constraint holds true:*

$$\sum_{i=1}^{n} \lambda_i \mu_{n-i+1} \leqslant \sum_{i=1}^{n} \sum_{i=j}^{n} a_{ij} b_{\pi(i)\pi(j)} \leqslant \sum_{i=1}^{n} \lambda_i \mu_i \tag{21}$$

$\sum_{i=1}^{n} \lambda_i \mu_{n-i+1}$ and $\sum_{i=1}^{n} \lambda_i \mu_i$ are in fact the exact lower and upper bounds of the estimation when relaxing the set of feasible matrices from *permutation matrices* to *orthogonal matrices* (cp. Rendl and Wolkowicz [158]). As the permutation matrices X_n are the intersection of the *orthogonal matrices* and the *doubly stochastic matrices*, the relaxation to orthogonal matrices due to the obtained tight bounds for these leads to (more or less tight) bounds for the

permutation matrices. Thereby, eigenvalue bounds are generally based on the relaxation to orthogonal matrices.

Thus, $\sum_{i=1}^{n} \lambda_i \mu_{n-i+1}$ provides a lower bound under the given restrictions at the price of computing the eigenvalues of A and B in $\mathcal{O}(n^3)$ time (theoretically in $\mathcal{O}(n^{2.38})$ with the method of Coppersmith and Winograd [47, 41]). Such simple eigenvalue bounds are, like Gilmore-Lawler bounds, known to be still rather weak in general (cp. Anstreicher [9]). Strategies to sharpen them can be based on reducing the matrices *spreads* (the range of the eigenvalues), e. g., by decomposing the flow and the distance matrix and transporting a preferably large proportion of the problem to the linear term (B) of the QAP formulation as described in Finke et al. [61]. Another possibility are *gradient projection* methods like in Hadley et al. [86]. Even though sharpened eigenvalue bounds can be much better than GL bounds, their calculation is, in general, very time-consuming and therefore not applicable in a branch-and-bound procedure. This fact is strengthened by the observation that the bounds quality deteriorates in lower levels of the branch-and-bound tree (see Clausen et al. [39]).

Referring to the eigenvalue bounds, a general approach to get lower bounds for QAP is to relax the set of feasible solutions from the set of *permutation matrices* $\mathcal{X}_n$, which is the intersection of the sets of *orthogonal*, *non-negative* and *row/column-sum-equals-one matrices*, to only a subset of these restrictions (cp. Burkard et al. [29]).

Besides the important class of LP-based lower bounds, which was introduced by linearized MILP formulations in Section 3.1.3, the already mentioned SDP bounds based on *semidefinite programming*, see Zhao et al. [192] or Sotirov [171], became more and more important in recent researches on QAP solutions. Let $A \bullet B$ denote the sum of elements of the *Hadamard* product of A and B.

$$A \bullet B = \sum_i \sum_j A_{ij} \cdot B_{ij} \tag{22}$$

In an SDP, the variable to solve for is a *matrix* $\mathcal{X}$ and a general system is set up by m sets of equations, see formulation (23).

$$\begin{aligned} \min \quad & C \bullet \mathcal{X} \\ \text{s.t.} \quad & A_i \bullet \mathcal{X} = b_i \ \forall i \in \{1, 2, \ldots, m\} \\ & \mathcal{X} \succeq 0 \end{aligned} \tag{23}$$

SDP bounds in a specific way relax the set of *non-negative* matrices ($\mathcal{X} \geqslant 0$) to the set of (*positive* or *negative*, w. l. o. g. positive in the following) *semidefinite* matrices ($\mathcal{X} \succeq 0$) and thereby change the search space for $\mathcal{X}$ from a

non-negative orthant to the cone of semidefinite matrices. Another interpretation is to switch the search space from matrices with non-negative *entries* to matrices with non-negative *eigenvalues*. In addition, linear inequations are transformed to linear operators of the *matrix* $\mathcal{X}$ and the trace operator $\mathrm{tr}()$ becomes the inner product (scalar product) of two elements. A first access to the relaxation of SDPs '*from orthant to cone*' is to consider the relationship between an LP and an SDP by expressing an LP in the form of an SDP. Consider the LP in formulation (24).

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j \cdot x_j \\ \text{s.t.} \quad & \sum_{i=1}^{n} a_{ij} \cdot x_j \quad = b_i \ \forall i \in \{1, 2, \ldots, m\} \\ & x_j \qquad\quad \geqslant 0 \ \forall j \in \{1, 2, \ldots, n\} \end{aligned} \tag{24}$$

Setting up a matrix $\mathcal{X}$ with the x_j's on the diagonal and *zero* in all other entries, defining matrices A_i in the same way with only entries a_{ij} on the diagonal and a C matrix with only the c_j's on the diagonal, this system is the linear program expressed in the form of an SDP as in formulation (23). Many $\mathcal{NP}$-hard combinatorial optimization problems (like QAP) can be bounded by convex relaxations that, again, can be expressed by SDPs. The solutions of relaxations for certain problems not only provide lower bounds but can even be translated into a feasible solution of the original program with a provably good objective value. More explanations, the detailed transformation and a good overview about SDPs can, e. g., be found in the lectures 'Introduction to Semidefinite Programming (SDP)' by Epelman [56] or Freund [64]. A great benefit of SDP relaxations of QPs is that such systems often provide rather tight bounds compared to many of the other bounds. The SDP can be solved with interior-point or cutting-plane methods or other specialized algorithms like the bundle method in Rendl and Sotirov's work [157]. All this comes at the price of relatively large computational costs to solve the SDP what makes them not being well suited for practical use (cp. Huang [99]). Still, Anjos and Liers investigated in their work 'VLSI Layout: Global Approaches for Facility Layout and VLSI Floorplanning' that *"the semidefinite optimization approaches can provide global optimal solutions for instances with up to 40 facilities, and tight global bounds for instances with up to 100 facilities"* [7]. This result is particularly related to this work as the later described method handles, as already mentioned at the beginning of Section 3.1, a special case of floorplanning.

3.1.5 *The QAP polytope*

Following a linearization of the coefficients, Jünger and Kaibel [103, 104, 105] set up a Graph $\mathcal{G}_n(V_n, E_n)$ and showed that finding an n-clique with minimal weight in this graph is equivalent to solving QAP as the n-cliques correspond to the $n \times n$ permutation matrices. They also investigated relations to other well known polytopes. A clique is a set of nodes in an undirected graph with connections available in between all of these nodes.

The construction works as follows: Represent the coefficients c_{ghij} (cp. equation (7)) as weights of edges between two nodes (g, h) (representing x_{gh}) and (i, j) (representing x_{ij}) with $g \neq i$ and $h \neq j$. These restrictions are important to construct a legal assignment, as $g = i$ would mean that a facility g is placed on two locations at the same time and $h = j$ that two facilities are both placed on location h. In addition, consider the linear cost terms b_{ij} as weights of the nodes x_{ij}. Now, finding a *maximal clique* (an n-clique) with *minimal total vertex- plus edge-weight* is equivalent to solving the respective QAP instance.

In the following, only a basic example of the interpretation of the idea should be given, all the methodological details and the application of this model to elaborate new insights to the structure of QAP are explicitly described in the mentioned publications.

Let x^C_{ij} be the index-vector of *nodes* in the n-clique C with

$$x^C_{ij} = \begin{cases} 1 & \text{if } (i,j) \in C \\ 0 & \text{else .} \end{cases} \tag{25}$$

and y^C_{ghij} the index vector of *edges* in the n-clique

$$y^C_{ghij} = \begin{cases} 1 & \text{if } ((g,h),(i,j)) \in C \\ 0 & \text{else .} \end{cases} \tag{26}$$

An illustrative example for an n-clique in $\mathcal{G}_n$ with $n = 4$ (cp. Jünger and Kaibel [103], Section 2) is shown in Figure 14. The highlighted clique represents the assignment of unit 1 to location 2, unit 2 to location 1, unit 3 to location 4 and unit 4 to location 3.

Any n-clique can only contain one node of each row and one node of each column as the row and column nodes are not interconnected to each other. The relationship to permutation matrices is thereby obvious.

The QAP polytope of size n can, by this construction, be described by

$$\mathcal{QAP}_n := \text{conv}\left(\{\, (x^C, y^C) \mid C \text{ is an } n\text{-clique in } \mathcal{G}_n \,\}\right) . \tag{27}$$

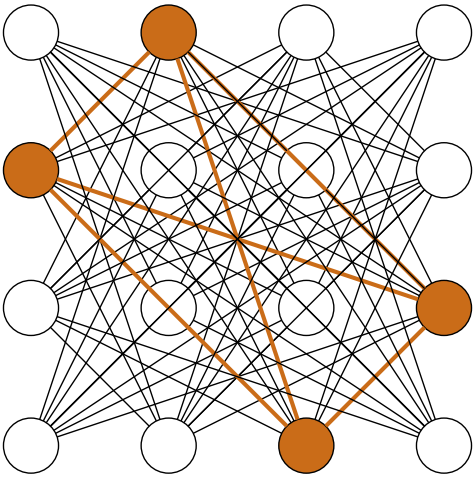


Figure 14: A Clique in $\mathcal{G}_n$

By such constructions from basically other field than linear or quadratic programming, the understanding of polytopes, here the QAP polytope, and especially the understanding of its general structures and its facets, *can and has* greatly be enhanced.

Remark 25. *It can be shown that the QAP polytope of size n is isomorphic to the Birkhoff polytope of second order $\mathcal{B}_n^{[2]}$. $\mathcal{B}_n^{[2]}$ is the convex hull of permutation matrices of second order defined by $\left(X_\pi^{[2]}\right)_{ghij} = (X_\pi)_{gh}(X_\pi)_{ij}$ (the index $[ghij]$ is often denoted by $[gh, ij]$ to separate the indices of the permutation matrices) corresponding to the linearization $y_{ghij} = x_{gh} \cdot x_{ij}$. A polytope can be minimally described by its facets, the $n-1$ dimensional faces of the n dimensional polytope. While the Birkhoff polytope has only n^2 many facets, there are exponentially many known facets for the Birkhoff polytope of second order $\mathcal{B}_n^{[2]}$ (cp. Aurora and Mehta [13]). The facets are important to obtain a possibly minimal description of the polytope and to generate, in that way, good cuts in a branch-and-cut procedure applied to such problems. The facets are particularly precious as they **cut** off relatively large portions of the relaxation and bound the polyhedron. For example, Erdoğan and Tansel proposed an approach based on branch-and-cut to solve a linearized QAP [58].*

Even though all these works and deliberations underline the high complexity of QAP in general, there are small classes of instances that can be solved efficiently. For example, Christofides and Gerrard [38] showed that, if both matrices A and B are weighted adjacency matrices of *trees*, the problem can be solved in polynomial time by dynamic programming approaches. As

soon as one matrix is not a 'tree-matrix', the problem remains $\mathcal{NP}$-complete as the *Traveling Salesman Problem* (TSP) can be reduced to that case (see Pardalos et al. [150]). Several other conditions for efficiently solvable QAPs where reported by Christofides and Gerrard and also in the works of Erdoğan and Tansel [57] and Burkard et al. [29, Section 10]. In '*A note on a polynomial time solvable case of the quadratic assignment problem*', Erdoğan and Tansel showed that if the distance matrix represents a Manhattan grid and the flow follows a path structure (like in the '*chr18b*' example from the QAPlib [28]), the QAP can be solved in polynomial time. Unfortunately, the instances that will come up in this work do not meet these conditions in general.

Many other well known $\mathcal{NP}$-hard problems (apart from TSP) have been formulated as QAP instances, e. g., *bin-packing*, *max-clique* or *graph-isomorphism* (cp. Loiola et al. [128]). In comparison to the very small sizes of optimally solved QAPs (in general with $n < 40$), TSPs with sizes up to 89500 elements (a micro-chip layout from the Bell Laboratories solved by *Cook et al.* in 2006 [12][1]) have already been solved to optimality. Even though TSP is also an $\mathcal{NP}$-hard problem, this emphasizes that it is obviously not as hard as QAP.

3.1.6 *QAP in chip layout*

QAP appears directly in the chip layouting problem introduced by Steinberg in 1961 [173]. The task was to place 34 units with 2625 connections on a backboard with 36 slots. To meet the QAP conditions, two dummy units were inserted (an extension that is similarly performed later in this work to match the 'idealized FPGA architecture' in Section 3.2). Steinberg considered the L_1-, L_2- and the squared L_2-norm for the distance calculations. Due to the already mentioned properties of many wiring architectures in Section 3.1.1, most research was invested for the L_1-norm, cp. Brixius and Anstreicher [24]. In 2004, they analyzed the quality of different bounds in this work, including the specialized *triangle decomposition bound* [107] for L_1-norms on grids and proposed a complete solution method based on branch-and-bound for the L_1 case. After 186 hours of CPU time on a single 800 MHz Pentium III, their method confirmed the optimality of a first 1990 found solution with costs 9526 based on a tabu search method [169]. This is a result that also shows that (meta-)heuristics like Tabu Search can be of great value in this field, more details on such techniques are given in Section 3.2. The best bound that was applied to the problem in Brixius and Anstreicher's approach was a dual LP based bound, assessing the costs to 7860 (gap of 17%).

1 http://www.tsp.gatech.edu (accessed 15 Jul 2016)

3.1.7 *Towards QAP heuristics*

Due to the difficulty of the problem and the importance of its solution in several fields, many heuristical approaches arose. While classical pure heuristics are designed to find a *good* solution for a specific problem or even instance, *metaheuristics* are able to process a wide range of problems, e. g., the class of QAP with arbitrary flow and distance matrices. Metaheuristics therefore operate as a general algorithm to find a good solution step-by-step. Many of the most popular metaheuristics (e. g., *local search*) base on neighborhood search principles, starting from an initial solution and improving the solution by *good choices* of *neighboring* solutions until no further improvement is possible in that way. Section 3.2 presents different examples for metaheuristics and investigates their behavior in terms of *quality* and *time* needed to reach the solution dependent on the *size* of the problem and the *quality* the initial solution. The experiments are performed on the base of an input with known optimal solution that is a simplification of the real instances later used in this work. Other metaheuristical approaches for general QAP can, e. g., be found in Hussin and Stützle's work [100], in Stützle and Dorigo [174] or, realized by a combination of different metaheuristical methods, in Kováč's work [115]. In addition, the study of Abd El-Nasser and Mahmoud [163] provides an interesting experimental comparison of different metaheuristics performed on recent hardware for QAPlib problems. A well known example for a heuristic in the field of chip layouting is the '*Kernighan-Lin-Algorithm*' [109], solving the graph-partitioning problem to minimize the wirelengths on a chip by swapping elements between two partitions to reduce the edge *cut* between them (the number of edges among both partitions).

3.1.8 *Why this work is not based on exact solutions*

All described techniques towards exact solutions of general QAP instances have in common that the solution time is impractical for an approach like the one presented in this work. Li et al. [124] provide a great survey on the computation (time) of lower bounds of QAPlib instances and Loiola et al. [128] additionally give an overview of the CPU times spent to solve the popular Nugent instances [145]. Even though the hardware used is partially from earlier decades and therefore extremely outdated from today's point of view, the results show the trend of runtime behavior for several different sizes of instances from the de facto standard in benchmarks for QAP, the QAPlib. With single CPU solution times of days, month or even years for instances with mostly less than 30 units, it is obvious that without a breakthrough in the methodology or in the compute architecture (e. g., *maybe with*

quantum computers) there will be no possibility to solve instances with several hundred or thousand elements exactly in the near future.

To complete the picture, Loiola et al. [128] also provide some insights into the quantity of research efforts that were put into approaches towards exact solutions or (meta-)heuristics. They counted the publications about QAP theory and applications, investigated which solution approaches and formulations have been used, what kind of bounds were applied and much more. Reading these statistics, one has to take into account that traditional ideas that arose years or decades ago naturally tend to have a higher count in their inquiry. Following their statistics, it can be summarized that formulations based on permutations, IPs and MILPs dominate the literature and are, for example, by far more present than SDP formulations. The most intensively studied class of lower bounds is (again, quite naturally) the GL bound. Due to the complexity and, at the same time, the importance of the problem and considering the costs of compute time to solve it exactly, it is finally not surprising that there is more work on heuristics than on exact methods. With more than 300 referenced publications (with growth over the years) alone in this survey, the presence of the problem and the interest into this, extraordinarily difficult, problem is nicely confirmed.

However, this work is *not* about exact solutions of the problem. The reasons for this are manifold: First and most importantly, the exorbitant solution time of today's methods is inapplicable in the proposed framework. The solution for inputs of real-world netlists would, even for the smallest input of our benchmark set, *explode* (at least 54 elements for the smallest '*stereovision3*' example, cp. Table 12 on page 222). In addition, a repeated optimization approach is desired in this work learning from previous layouts to meet other goals than only the wirelength minimization. For example, Benjafaar [17] showed in his work that layouts which minimize the *work-in-progress* (WIP) may drastically differ from the ones obtained by a general QAP formulation minimizing the wirelength. Likewise, similar characteristics for FPGAs are investigated in this work, e. g., that minimizing the *critical path delay* may result in different assignments than minimizing for *overall wirelength* while the former is, at the same time, often more important. Another example for an adjusted optimization goal is presented by Miranda et al.'s [45] heuristical approach based on QAP considering specifically the *thermal effects*.

To conclude, the purpose of this developed framework is to provide flexibility concerning the optimization goal and adaptivity within the procedure, all the details follow later in this work.

3.1.9 *Why this work is not using QAP lower bounds*

Even though lower bounds for QAP can help to get an impression of an assignment's quality, it is not necessarily easily possible to derive meaningful lower bounds with well known QAP bounds. First, these bounds (of course) only estimate a defined cost function based on the underlying QAP model and, as already stated in the previous section, wirelength is not for all situations the 'cost function of choice'. In fact, the FPGA placement problem has further restrictions as it demands for *different types* of units with appropriate locations. In addition, an FPGA (or the available part of an FPGA) will generally be significantly larger than the desired set of logical units, a fact that would in turn call for many dummy elements 'blurring' the bounds' qualities. It has also already been referenced that generating an ϵ-approximation of QAP is again $\mathcal{NP}$-hard. Finally, the calculation costs of QAP lower bounds are, in general, too high for an incorporation in a productive framework, at least in relation to their additional value. It will be shown that approximations of the achieved quality can be accomplished rather quickly and, most of all, stronger related to the actual optimization goal.

3.2 ITERATIVE APPROACHES TOWARDS SOLVING QAP INSTANCES

In this section, different iterative metaheuristical approaches are compared to show peculiarities, potentials, costs and options of such techniques for the desired layouting idea. The implemented algorithms are not meant to be 'the cutting edge' of metaheuristics in this field but rather to give an impression about the different basic characteristics. Finally, the idea of using *force-directed graph layouting methods* as the base of the technique is motivated by comparing the outcome of such a layout to the metaheuristics'.

All different neighborhood search algorithms examined in this section have been implemented based on *METSlib* [132], an object oriented metaheuristics optimization framework written in C++. The METSlib framework was designed to realize and adapt problem-independent metaheuristic solvers for optimization problems. The implementation is part of COIN-OR [92], the '*Common Optimization INterface for Operations Research*'.

Remark 26. *Due to the implementation of all methods on the same base and benchmarking on the same machine, the results are well comparable in terms of quality and computing time consumption. The machine is described in Section 1.4. The main goal of this section is to show the dependence of the solution* ***time*** *and* ***quality*** *on the* ***input's size*** *and the* ***input's quality****. To start off with, in FPGA- or chip-layout in general, especially (the also considered) 'simulated annealing' approaches (see Section 3.2.7) are widely used.*

First of all, the idealized models used for the comparison are described.

Definition 4 (Idealized chip architecture)**.** *For simplicity and to make QAP directly applicable, the architecture to be mapped on is described by a regular* $N \times N$ *grid with only one type of* ***locations****, just as in Section 3.1.1.*

Definition 5 (Idealized layout graph)**.** *The idealized layout graph (see Figure 15), itself representing the* ***units*** *(facilities) to be assigned and their* ***interconnections****, actually consists of* $(N^2 - 4)$ *nodes. To fulfill the QAP requirements,* ***four dummy nodes*** *are added (the corners). Besides these nodes on the outer frame of the graph (orange), each node is connected to each of its four neighbors. The nodes on the outer frame play a special role, as they represent the access to the chip from outer regions. They are typically referred to as* I/O *(Input/Output) nodes.*

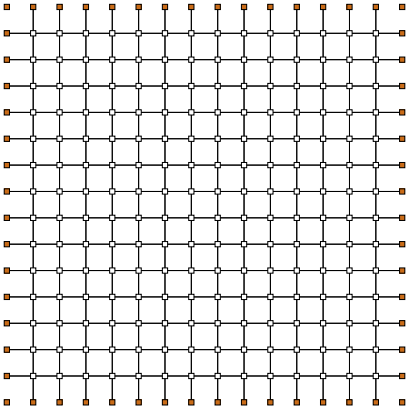


Figure 15: Idealized layout graph with $N = 16$ (16×16 grid)

Due to this construction, an 'immediate' global optimal assignment can be obtained as shown in Figure 15. Each row and each column (except the first and the last one) contains exactly $(N - 1)$ connections and the resulting costs in terms of wirelength can be calculated as described in formula (28).

$$\tilde{c} = 2 \cdot (N - 2) \cdot (N - 1) \tag{28}$$

Because each connection on the regular grid is exactly of length *one*, it is obvious that such an assignment is optimal concerning the overall wirelength.

Remark 27. *Moreover, the optimal placement is, apart from rotations of* $k \cdot 90°$ *with* $k \in \mathbb{Z}$ *and apart from the positions of the dummy corner vertices,* ***unique****. Placing any non-I/O unit on the outer frame would directly lead to higher costs as it would not be possible to place the four connected neighbors each with a distance of one on the*

grid. The four 'white inner corner nodes' can only be placed optimally in such 'inner corners', as they are connected to ***two*** *I/O nodes. Therefore, the outer frame and the 'inner non-I/O corners' are fixed apart from the mentioned* $90°$ *rotations and the dummy nodes. The* $(N-2)$ *actual I/O nodes on each face of the frame consequently form a group that has to be placed together on one face (due to the inner node with two connected I/Os). Now, all the* $(N-2)$ *horizontal paths of* $(N-1)$ *segments from the left face of the frame to the right one have to be on the same vertical coordinate and interchanging entire horizontal paths would increase the vertical distances between connected nodes.*

It can particularly be concluded that an optimal assignment of this model has to place the orange I/O nodes onto the outer frame of the grid, an observation that is, for example, used in Section 3.2.8.

The following explanations of the different algorithms are based on the work of Mirko Maischberger [132], who designed and implemented METSlib. METSlib has specialized structures to deal with QAP instances and is therefore perfectly suited to develop and analyze own implementations of QAP metaheuristics. For example, a QAP instance can be loaded from a simple *ASCII* '.dat' file containing the number of facilities and locations n, the flow matrix $\mathcal{V}$ and the distance matrix $\mathcal{D}$. For the benchmarks, a framework was developed setting up this file for different sizes $N \in \{8, 10, 12, 14, 16\}$ ($N \times N$ grid) with a flow matrix representing the idealized layout graph (cp. formulation (13)) and a distance matrix with Manhattan distances of locations on the idealized chip architecture (cp. formulation (12)). In addition, the resulting assignments can be exported to a standard *graph modeling language* '.gml' file, some resulting graphs for the different approaches are shown in Figure 26.

All algorithms are applied to five different initial assignments to show the dependence of achieved solutions and solution times from the quality of the initial situation. To make this dependence measurable and comparable, the optimal solution was taken as a base and perturbed by a number of random swaps. The more swaps are performed on the original optimal solution, the further away from this optimal solution is the starting point of the optimization.

All experiments for all sizes and every initial solution were performed 10 times. The results present either average measures of these ten runs or the best and the worst achieved solution to additionally mark the *span of solutions*.

The nomenclature in the algorithms is largely adopted from the referenced work of Maischberger. More details about metaheuristics can, for instance, be found in Gonzalezes book 'Handbook of Approximation Algorithms and Metaheuristics' [75] or in Gendreau et al.'s book 'Handbook of Metaheuristics' [70].

3.2.1 *Problem definition*

Consider a general discrete optimization problem

$$\min_{s \in \bar{S}} c(s) \qquad (29)$$
$$\bar{S} \text{ is discrete}$$

with a set of feasible discrete solutions $\bar{S}$ and a cost function $c(s)$.

The cost function $c(s)$ is a function of a permutation/assignment returning the overall wirelength resulting from this permutation/assignment. It can be calculated with the set up flow and the distance matrices as described in the beginning of this chapter.

As already stated in Section 3.1, *QAP* can be formalized as a permutation problem between the set of *facilities* F and the set of available *locations* L.

First, the size of the problem's search space will be investigated to derive some reasonable constants for the implementations later in this section, e. g., the size of a *reduced neighborhood*.

Definition 6. *Let* $X = \{x_1, x_2, \ldots, x_n\}$ *be a set of* n *elements. A* k*-permutation of* n *is a re-arrangement of a* k*-element subset in the* n*-set* X*.*

Lemma 1. *The number of* k*-permutations (without repetition) of an* n*-set is* $P = \frac{n!}{(n-k)!}$

Proof (informal). The subset of the entire set with n elements to be permuted consists of k elements. The first element can be swapped with any of the n elements (even with itself). The second element can now only be swapped with the $n-1$ remaining elements, etc. The last element of the k-subset can consequently be swapped with $n-k+1$ elements. This leads to $n \cdot (n-1) \cdot \ldots \cdot (n-k+1) = \frac{n!}{(n-k)!}$ possible k-permutations. □

Corollary 1. *The number of permutations (***n*-*permutations***) of an* n*-set is* $P = n!$

Due to the fact that a linear order can be defined on every finite set (e. g., by assigning successive numbers to each element of the set), a *permutation* can be represented by a bijective correspondence between a set of n elements and itself $\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ (cp. formula (15)). Therefore, a reordering of the elements on the chip architecture can simply be expressed by a permutation of the elements.

Under these assumptions, the system to be solved can be constrained to formulation (30).

$$\min_{s \in \bar{S}} c(s) \qquad (30)$$
$$\bar{S} \text{ is the set of permutations } \pi$$

The *search space* of formulation (30) therefore contains $n!$ feasible solutions for the allocation of n facilities to n locations.

3.2.2 *Neighborhood exploration techniques*

Neighborhood exploration techniques are frequently and effectively used in heuristics (and metaheuristics) for discrete optimization problems, e. g., in form of the popular Lin-Kerninghan heuristic [126] for TSP. A neighborhood exploration starts with a feasible solution $s_0 \in \bar{S}$ as the *current solution* s_{cur} and successively tries to improve the current costs by examining a *neighborhood* $\mathcal{N}(s_{cur})$ of this point. A neighborhood is, abstractly said, a set of points that can be reached from the current solution, e. g., by *minor* modifications of s_{cur}. For the later examples, a neighbor is constructed by swapping the positions of two random points in the current solution.

The number of different k-sets in an n-set is known to be $\binom{n}{k} = \frac{n!}{(n-k)!k!}$, called the *binomial coefficient*. For swapping two elements (*pairwise swap*) in an n-set, there are thus $\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n\cdot(n-1)}{2}$ possibilities. This corresponds directly to a construction of all possible swaps by choosing the first element out of all n elements and the second element out of the remaining $(n-1)$ elements. As each swap is counted twice in this construction, this number is divided by two to neglect the order of the two elements. The set of all possible pairwise swaps in s_{cur} is called the *full neighborhood* $\overline{\mathcal{N}}(s_{cur})$ of s_{cur}.

Corollary 2. *The number of pairwise swaps in an n-set is $\binom{n}{2} = \frac{n\cdot(n-1)}{2}$. This is thus the size of the **full neighborhood** $\overline{\mathcal{N}}$.*

Instead of examining the full neighborhood $\overline{\mathcal{N}}(s_{cur})$ to find an improving neighbor s'_{cur} of s_{cur}, only a subset $\mathcal{N}(s_{cur})$ of the full neighborhood may be considered to reduce the search time. Rather than taking all $\binom{n}{2}$ pairwise swaps (cp. Corollary 2) of point positions in the full neighborhood of the current solution s_{cur} into account, a random selection of only $n\sqrt{n} < \binom{n}{2} (\forall n \in \mathbb{N}_{\geqslant 6})$ swaps could be examined for a permutation problem like the prospected one. This is an $\mathcal{O}(n^{1.5})$ set size instead of the full $\mathcal{O}(n^2)$ set of swaps. This reduction provides a good balance between examining enough elements to find good solutions and not too many to keep the time to find a new neighbor in a reasonable time range, especially for large n. Nevertheless, many other choices are for sure possible and arguable. It is important to note that this neighborhood is, in general, only containing the next step in a series of many improvements. Examining all neighboring solutions does not necessarily lead to better solutions as local minima can also be reached earlier with larger neighborhoods. The mentioned reduction of the

neighborhood is used in several of the following algorithms. Thus, a neighborhood with $n\sqrt{n}$ elements is called *reduced neighborhood* in this section and denoted with $\mathcal{N}$.

Definition 7. *The **reduced neighborhood** $\mathcal{N}$ is a random selection of $n\sqrt{n}$ pairwise swaps extracted from the full neighborhood.*

The number of elements in a *full* and in a *reduced* neighborhood of pairwise swaps for the following benchmarked square $N \times N$ grids with $N \in \{8, 10, 12, 14, 16\}$ (resp. with $n \in \{64, 100, 144, 196, 256\}$ elements) are shown in Figure 16.

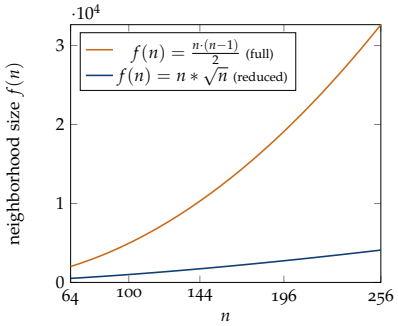


Figure 16: Number of elements in a *full* and a *reduced* neighborhood

A new solution s'_{cur} is chosen based on the *acceptance rule* of the specific algorithm and is, when applicable, assigned to be the best found solution s^*. For example, an acceptance rule could either be to choose *the best* or to choose *the first improving* solution in the neighborhood. If there is no new solution meeting the acceptance rule, the algorithm can, for example, stop and return s^*.

Remark 28. *One-dimensional functions are suitable for illustrative purposes in this section. For example, a neighborhood of a value x of a function $c(x)$ could be the values $(x - \varepsilon)$ and $(x + \varepsilon)$. The 'better' neighbor is therefore the one with better function value, e. g., $(x + \varepsilon)$ if $c(x + \varepsilon) < c(x - \varepsilon)$ in case of minimizing the costs.*

3.2.3 *Global and local optima*

A function can contain *global* and *local* optima. Figure 19a on page 71 shows how a local search routine gets stuck in a local optimum and is not able to reach a nearby better value on the right of it. In general, many simple heuristics tend to reach only local (and often *poor*) optima. Thus, one important

task is to define strategies that can escape local optima and reach improved solutions (*here:* permutations) to find better local optima or even a global optimum without loosing the majority of improvements that were already achieved by the heuristical method.

3.2.4 *Local search*

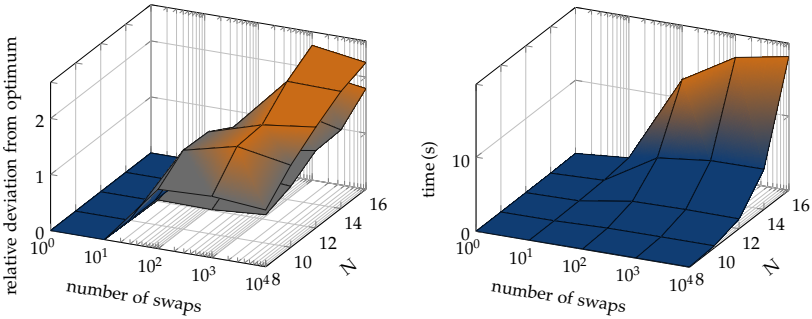
As a first metaheuristic to solve QAP, a simple *local search* (LS) method was implemented (see Algorithm 1). The local search method gets an initial solution as the input and therefore makes it the first solution s_{cur} and accordingly the, so far, best found solution s^*. It then generates a full neighborhood of s_{cur} with elements s'_{cur} and looks for the best element in the neighborhood which is successively the new s_{cur} and, if it improves s^*, also the new s^*. This procedure is repeated until a neighborhood occurs in which no improving neighbor is found. The algorithm then stops and returns s^* as the optimal solution. A possible modification would be to choose the first improving s'_{cur} instead of the best one in the full neighborhood. This would generally reduce the search time in the neighborhood. An analogue local search approach for maximization problems is, for example, the '*hill climbing*' algorithm.

Algorithm 1 Local search

```
procedure LocalSearch(s_0)
    s* ← s_0
    repeat
        s_cur ← s*
        for all s'_cur ∈ N(s_cur) do          ▷ or until a better sol. was found
            if c(s'_cur) < c(s_cur) then
                s_cur ← s'_cur
            end if
        end for
        if c(s_cur) < c(s*) then              ▷ improving neighbor was found
            s* ← s_cur
        end if
    until no improving neighbor was found
    return s*                                 ▷ local optimum
end procedure
```

The local search algorithm monotonically iterates to a local optimum, as depicted in Figure 19a. Figure 17b illustrates the relative deviation of the cost value of the found local optimal solution to the cost value of the known

global optimal solution, formally $\frac{c(s^*)-c_{opt}}{c_{opt}}$. A relative deviation of *one* therefore means that the costs of s^* were two times as high as the optimal cost value. A relative deviation of *zero* confirms optimality of the obtained solution s^*.



(a) Quality (min and max deviation)

(b) Time (average)

Figure 17: Local search (LS) [10 runs]

Figure 17a shows that the more swaps were performed to generate the initial solution s_0 by perturbing the optimal assignment s_{opt} (see Figure 15), the poorer is the finally found local optimum. The results of the ten repeated runs contain solutions that occupy a high relative deviation of up to more than factor *two* from the optimum with a moderate span of the quality which is visible by the *gap* between the smallest and the largest achieved deviation from the optimal cost value $c(s_{opt})$. For the larger examples of up to 16×16 elements with accordingly larger search space, the quality of of the final placement also decreases slightly.

On the other hand, all initial instances with only 1 or 10 random swaps distance to the optimal value were solved to optimality. In fact, any constellation of each size that is only one swap away from the optimum would be optimized after examining the first *full* neighborhood naturally. The initial perturbing swap is found, recognized as the best swap in the neighborhood and reversed. Consequently, the second neighborhood offers no further optimization and thus the algorithm stops. Already for the inputs with 'swap-distance' 10 to the optimum, this behavior could not have been guaranteed as a swapped element could also be iteratively moved nearer to a good destination and it is not assured that a swapped element reduces the overall costs when moved back to its originally optimal location because the other swapped elements may influence the costs negatively. Nevertheless, local

search with full neighborhood achieves the best results of all implemented methods for these 'near-optimal' initial situations.

The time that the procedure spends before it decides to terminate expectedly increases with both the size of the instance and the degree of perturbation in the initial solution. For both very small instances and very good initial solutions, the runtime of local search is exceptionally small.

3.2.5 *Tabu search*

Tabu search [73] (TS) can be seen as a progression of local search to escape local optima. Instead of terminating as soon as a neighborhood contains no improving neighbor, tabu search always takes the best solution in the neighborhood and proceeds. This is done for a number of neighborhoods (specifiable by TS_STOP in this implementation, 250 in these experiments) which lead to no improvement of the costs $c(s^*)$ of the best obtained solution s^*. Whenever an improving solution is found in the process, this 'counter' is reset to zero. As this metaheuristic investigates much more neighborhoods than the local search method, it is reasonable to shrink the neighborhood size by examining only a reduced neighborhood $\mathcal{N}$ made of $n\sqrt{n}$ randomly chosen moves from the full neighborhood $\overline{\mathcal{N}}$ (cp. Section 3.2.2).

To avoid cycling in the method by inversely performing a previous move again, there is a mechanism called the *tabu memory*. This memory stores a number of previous moves and inhibits to make them again for a certain period of time. This period is called the *tenure* of the tabu search. In the presented implementation, the tenure is set to a random value $\tilde{r} \in [5, n\sqrt{n}]$.

In certain circumstances, a neighbor from the tabu list may still be accepted as a new solution. Such rules in a tabu search are called '*aspiration criteria*'. A common aspiration criterion is to allow a move to new solution even though this move may be in the tabu list if this leads to a solution that is better than the current best one s^*.

Figure 19b shows how the tabu search proceeds rightwards after the first local optimum was found. Without the tabu list, the algorithm would, after going one step further to the right, turn back left to the local optimum as its costs are less than the one of the right neighbor. This would directly end in an infinite loop of non-improving swaps, whereby an escape from the local optimum would not be possible without the tabu list. If the algorithm reaches a new optimal solution s^* within the specified range of acceptable non-improving swaps (marked by the recommencement of the *blue* curve after the *orange* escape time in Figure 19b), it resets the counter and quasi continues with a new tabu search.

If a sequence of too many non-improving neighborhood searches appears (like in the second *orange* region in Figure 19b), the algorithm terminates. The randomly chosen tenure and the defined range TS_STOP can therefore greatly influence the outcome of the method. Furthermore, the aspiration criterion and the definition of the tabu list are 'adjusting screws' that influence both the quality and the runtime of a tabu search.

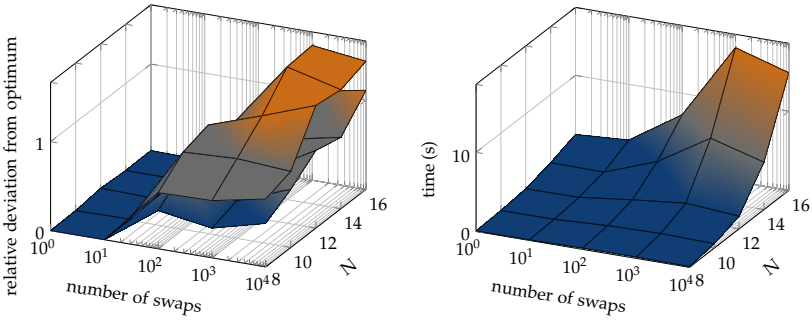
Summarizing, the algorithm (see Algorithm 2) can be described as

- start with an initial solution s_0 as the current and the best solution s_{cur} and s^*,
- search the solution s'_{cur} in the reduced neighborhood of s_{cur} which has the lowest costs and is not in the tabu list *unless* it is the best solution ever found (*aspiration*) and accept it in any case as the new current solution,
- if this solution improves s^*, formally $c(s'_{cur}) < c(s^*)$, update s^* and,
- if a certain number of consecutive neighborhood searches without improvements is exceeded (TS_STOP $= 250$ in the experiments), return s^*.

Algorithm 2 Tabu search

procedure TABUSEARCH(s_0)
  generate random tenure $\tilde{r}$
  $s_{cur} \leftarrow s_0$
  $s^* \leftarrow s_0$
  repeat
    $\tilde{c} \leftarrow \infty$
    for all $s'_{cur} \in \mathcal{N}(s_{cur})$ **do** ▷ or until a better sol. was found
      if $(c(s'_{cur}) < \tilde{c}) \wedge (\neg \text{tabu}(s'_{cur}) \vee \text{aspirate}(s'_{cur}))$ **then**
        $s_{cur} \leftarrow s'_{cur}$
        $\tilde{c} \leftarrow c(s_{cur})$
      end if
    end for
    if $\tilde{c} < c(s^*)$ **then** ▷ improving neighbor was found
      $s^* \leftarrow s_{cur}$
    end if
  until sequence of TS_STOP non-improving swaps appears
  return s^* ▷ local optimum
end procedure

Considering Figure 18a and Figure 18b, tabu search generates substantially better results than local search and this, due to the significantly reduced neighborhood for larger instances (cp. Figure 16), in rather comparable times. While the runtimes of small instances are still extremely small, larger near-optimal initial instances take much longer to achieve relatively good results than in a local search as the tabu list (due to its construction and its purpose) allows for local deteriorations of the current solution. Due to the reduced neighborhoods, tabu search can (in contrast to the local search) not even guarantee to fully optimize the 1-swap initial solutions. However, it produces, *in general*, better results especially for heavily perturbed initial solutions in times comparable to the ones of local search.



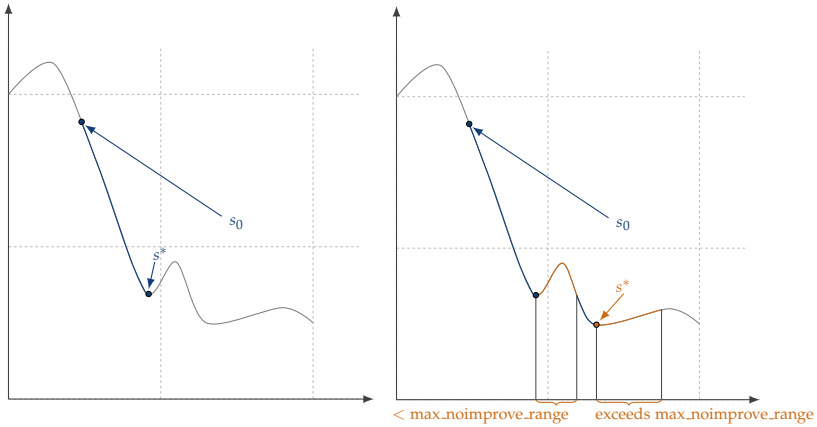
(a) Quality (min and max deviation)

(b) Time (average)

Figure 18: Tabu search (TS) [10 runs]

3.2.6 *Iterated Tabu search*

A further improvement of tabu search (in terms of quality) is an *iterated tabu search* (ITS). The idea is to call the tabu search method multiple times while giving the solution a moderate 'tap' in between these repetitions to let it escape the current local optimum s^*. In fact, after a tabu search stops (configured with the same parameters than in the previous paragraph), s^* is *perturbed* by applying a number of random swaps to it. It is obvious that a very small number of swaps will, in general, not be sufficient to escape a local optimum, as the next tabu search could easily iterate back to the previous solution. On the other hand, too many swaps may shatter the solution remarkably, so that all the already achieved improvements may be lost. A perfect perturbation would break up congested regions while preserving good structures. In the presented implementation, a perturbation is



(a) **L**ocal **S**earch (b) **T**abu **S**earch

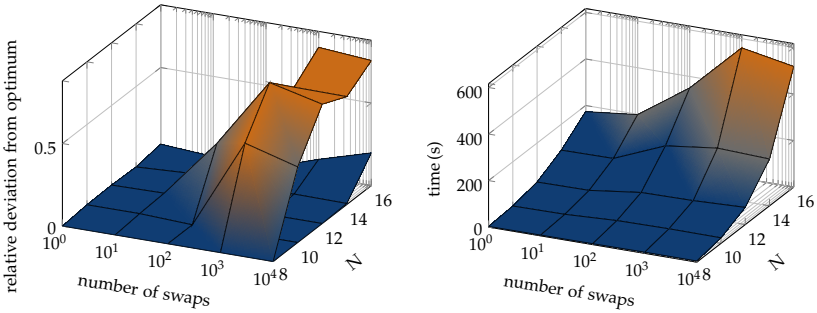
Figure 19: Principles of local search and tabu search for a 2D example

realized by a random number of $\bar{r} \in \left[\frac{n}{5}, \frac{n}{2}\right]$ swaps applied to the best known solution. If a sequence of ITS_STOP non-improving tabu searches appears (ITS_STOP $= 30$ in the experiments), the algorithm terminates and returns s^*. The runtime of the overall method therefore strongly depends on the two stopping criteria of the inner tabu search and the outer repeating iteration method.

Algorithm 3 Iterated tabu search

```
procedure IteratedTabuSearch(s_0)
    s* ← TabuSearch(s_0)
    repeat
        generate new random tenure r̃
        generate new random perturbation size r̄
        s'_cur ← perturb(s*, r̄)
        s_cur ← TabuSearch(s'_cur)
        if c(s_cur) < c(s*) then                ▷ improving neighbor was found
            s* ← s_cur
        end if
    until sequence of ITS_STOP non-improving repetitions appears
    return s*                                   ▷ local optimum
end procedure
```

The results in Figure 20a and Figure 20b show that even though this approach (of course) has a much longer runtime than a simple tabu search approach (ITS averagely runs about 30 times as long as TS in the experiments, which directly corresponds to the value of ITS_STOP), it produces very good results. Considering the lower *surface of best results* in Figure 20a, it can be seen that the obtained results are near to the global optimum. Only for the largest grid and the poorer input solutions, the *deviation from the optimum* of the best performing runs becomes considerable. However, the span of quality in the results is relatively large, based on the multiple random influences in the method and, of course, on the inherent 'local search'-characteristics of these methods in general.



(a) Quality (min and max deviation)

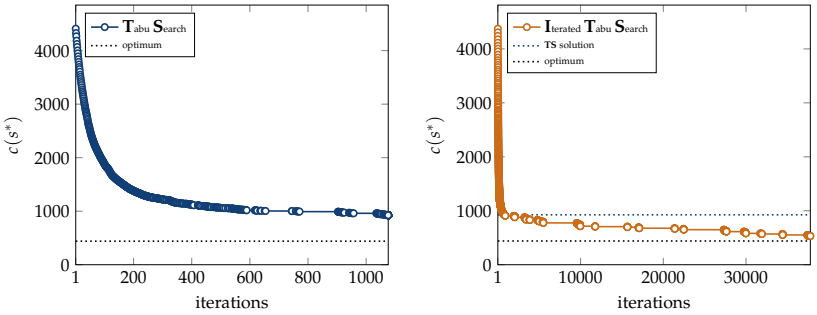
(b) Time (average)

Figure 20: Iterated tabu search (ITS) [10 runs]

Figure 21 depicts the convergence behavior of a tabu search and an iterated tabu search run for the largest instance (16×16) with maximal number of swaps (10^4) on the input instance. Each node marks a new found improved solution s^* and each iteration on the horizontal axis represents a neighborhood exploration. Figure 21a shows that, at the beginning of the method, a new improving solution is found in almost every iteration. The further the methods proceeds, the more iterations with no improvements occur resulting in larger gaps between two successive nodes. After 1076 neighborhood explorations, a 250 iterations long sequence without improvements appears and the algorithm stops returning a best solution s^* with costs $c(s^*) = 925$ and, therefore, with a deviation of $\frac{925-420}{420} \approx 1.2$ to the global optimum with costs of 420 (cp. equation (28) and Figure 15).

Figure 21b accordingly shows the convergence of an iterated tabu search run with $c(s^*) = 531$ after 37773 iterations. The blue dotted line marks the solution obtained by tabu search in Figure 21a. The tabu search optimum

is, in this run, reached after 773 of the 37773 iterations and larger gaps between improved solutions become more frequent after that. At the end of the procedure, a sequence with 30 non-improving tabu searches appears. The results show that the final additional enhancement of iterated tabu search over a simple tabu search comes at the price of many more necessary neighborhood explorations. Typically, the necessary effort for further improvements increases vigorously the nearer the method is to the global optimum.



(a) **T**abu **S**earch (b) **I**terated **T**abu **S**earch

Figure 21: Convergence comparison of the two tabu search approaches

3.2.7 *Simulated annealing*

Simulated annealing (SA), first proposed by Kirkpatrick et al. [111] in 1983, is a heuristical optimization technique that bases on *simulating* the behavior of an *annealing* physical system in the field of *statistical mechanics*. In fact, the method was initially applied to solve a microprocessor layout (floorplanning) problem by Kirkpatrick et al. and came into operation for many different problems in combinatorial optimization.

To get a tangible impression of the basic idea, one can imagine a metal plate that gets heated and cooled down again. By heating the plate, atomic irregularities in the crystalline structure of the metal are provided with energy to escape the unfavorable situation, consequently strengthening the structure of the material. With decreasing temperature, more and more arrangements become fixed due to the absence of energy, finally reaching a more stable *cold* state.

Following fundamental ideas from statistical mechanics, the *transition probability*, estimating the likelihood that a state s_{cur} moves to a neighboring

state s'_{cur}, can be described through the *Boltzmann distribution* (also *Gibbs distribution*, cp. McQuarrie [139]) neglecting the Boltzmann constant by

$$\tilde{p}_t(\Delta c) = e^{\frac{\Delta c}{t}} \tag{31}$$

with

$$\Delta c = c(s_{cur}) - c(s'_{cur}) \, . \tag{32}$$

Definition 8. *The **thermodynamic temperature** is a principle parameter in thermodynamics. It is also called the absolute measure of temperature and is defined by the third law of thermodynamics, declaring the theoretically lowest temperature as null. In this 'absolute zero' circumstance, particles' constituents of matter have minimal motion and cannot become colder.*

Δc measures the difference of the two states in terms of costs or, more precisely, the *absolute improvement* of a neighboring solution to the current solution. If such a neighboring solution s'_{cur} is an improvement (solution with *lower costs*) compared to the best known solution so far (s_{cur}), it follows that $\Delta c = c(s_{cur}) - c(s'_{cur}) > 0$. Thus, as the thermodynamic temperature t is positive by definition (Definition 8), the exponent in equation (31) is positive and $\tilde{p}_t(\Delta c) > 1$, which means that such an improving swap will always be accepted. If $\Delta c < 0$, it follows that $\tilde{p}_t(\Delta c) \in (0, 1)$. Now, a threshold $r \in [0, 1]$ has to be defined to decide for which probabilities $\tilde{p}_t(\Delta c)$ such a non-improving swap will be accepted. As $\tilde{p}_t$ shall map Δc on a probability, equation (31) can be limited to 1, resulting in p_t in equation (33).

$$p_t(\Delta c) = \min\left(e^{\frac{\Delta c}{t}}, 1\right) \tag{33}$$

It is important to note that the transition probability is strictly positive ($p_t(\Delta c) > 0$), called a '*non-zero transition probability*', so that any non-improving swap, regardless of its deteriorative intensity and the temperature, can potentially be accepted (the larger the degradation of costs and the colder the system, the less probable).

Figure 22 shows resulting transition probability functions for some chosen thermodynamic temperatures t and an exemplary threshold of $r = 0.63$.

Remark 29. *The Boltzmann distribution (including Boltzmann's constant k) itself describes the probability that a system is in a state with energy level E at a thermodynamic temperature t through the distribution $e^{-\frac{E}{k \cdot t}}$. The negative sign in the exponent disappears in equation (33) by reformulating the difference from originally $-(c(s'_{cur}) - c(s_{cur}))$ to $(c(s_{cur}) - c(s'_{cur}))$.*

Simulated annealing starts with a relatively high thermodynamic temperature t_0, determines a random *threshold* $r \in [0, 1]$ and calculates the transition probability $p_{t_0}(\Delta c)$ for a solution in the reduced neighborhood of the initial solution. If $p_{t_0}(\Delta c) > r$, the move to s'_{cur} is performed, otherwise not. A special characteristic of a simulated annealing approach therefore is that, besides all improving moves, also non-improving moves are randomly accepted (if the probability of the move exceeds the random parameter r), giving the method a good chance to escape local optima. The warmer the system is, the more non-improving moves are accepted, see the *acceptance ranges* denoted by the arrows in Figure 22. A new threshold r is randomly computed for each neighborhood.

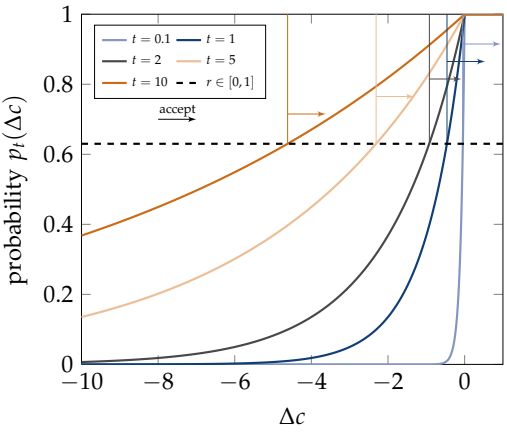


Figure 22: Transition probability $p_t(\Delta c) = min(e^{\frac{\Delta c}{t}}, 1)$

Algorithm 4 sketches the implemented simulated annealing algorithm. Starting with an initial solution s_0 and an initial temperature t_0, a reduced neighborhood (see Definition 7) of the current solution s_{cur} is constructed. As the *first* (and not the *best* like in the other approaches) improving neighboring solution will be accepted, it is generally reasonable to use relatively small neighborhoods to keep the time for the neighborhoods' creation down. After accepting a move to a neighbor s'_{cur}, the temperature of the system is decreased by a cooling function. This cooling function can, e. g., be a linear cooling $t_{i+1} = t_i - \hat{t}$ with $\hat{t} \in (0, t_0)$ or an exponential cooling $t_{i+1} = \hat{f} \cdot t_i$ with $\hat{f} \in (0, 1)$ and resulting $t_i = t_0 \cdot \hat{f}^i$. The exponential cooling approach can inherently never generate negative temperatures and approximates a cooling of a warmer object in a cold environment following *'Newton's law of cooling'* $(t(x) = t_{env} + (t_0 - t_{env}) \cdot e^{-kx}$ with environmental temperature

t_{env}, a parameter depending on the object's properties k and a point in time x). In general, any non-increasing progression of temperatures can represent the cooling function. Successively, a new neighborhood of the new s_{cur} is constructed and the procedure is repeated. The algorithm stops if the desired final temperature t_ω is reached.

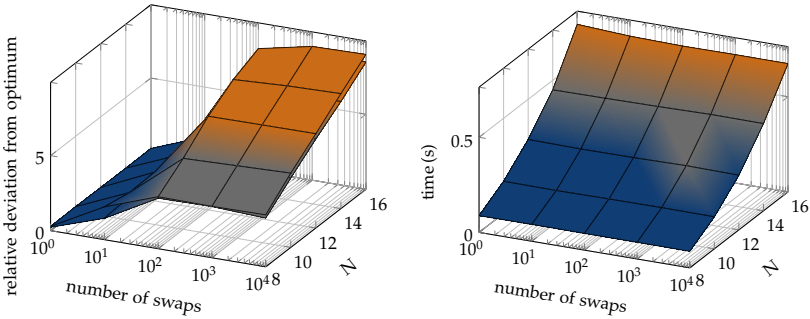
In the presented implementation (see Algorithm 4), an exponential cooling with $\hat{f} = 0.999$ was applied in the function $\mathtt{reduce}(t)$.

Algorithm 4 Simulated Annealing

```
procedure SimulatedAnnealing(s_0)
    s_cur ← s_0
    s* ← s_0
    t ← t_0
    while t > 0 do
        for all s'_cur ∈ N(s_cur) do
            generate random threshold r ∈ [0, 1]
            if min{e^((c(s_cur) − c(s'_cur))/t), 1} > r then        ▷ acceptance rule
                s_cur ← s'_cur
                break
            end if
        end for
        if c(s_cur) < c(s*) then                ▷ improving neighbor was found
            s* ← s_cur
        end if
        t ← reduce(t)
    end while
    return s*                                                    ▷ local optimum
end procedure
```

Combined with the cooling function, both the *starting* and the desired *final* temperature of the simulation essentially influence the characteristic of such a simulated annealing. Figure 23a shows the quality and Figure 23b the runtime for the presented simulated annealing algorithm with a high initial temperature $t_0 = 10000$ and a still relatively high desired final temperature $t_\omega = 100$. It is obvious (and reasonable) that the *runtime* for a specific size is *mostly independent from the quality of the input*, as the algorithm explores exactly $\log_{\hat{f}}\left(\frac{t_\omega}{t_0}\right) \overset{\hat{f}=0.999}{=} 4602$ neighborhoods of size $n\sqrt{n}$ for the mentioned temperatures. The quality of obtained results is naturally rather poor (relative deviations from the optimum of up to 10) when performing simulated annealing only in such 'warm' temperature regions. Due to the 'disturbing'

behavior of simulated annealing in warm phases of the algorithm, the resulting solution is, in many cases, not (or not much) better than s_0 as the simulation did often not improve the solution until it stopped. For more details on the convergence behavior of the simulated annealing approach, see Section 3.2.8. It should be clear that simulated annealing should not be applied in such a way. However, this parameter set is shown to illustrate this behavior.



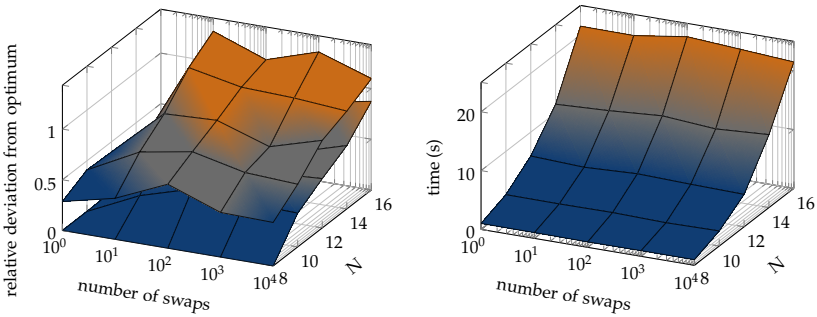
(a) Quality (min and max deviation)

(b) Time (average)

Figure 23: Simulated Annealing (SA), $t_0 = 10000$ and $t_\omega = 100$ [10 runs]

Running the same implementation with $t_\omega = 0.0001$ leads to the results shown in Figure 24. Again, the runtime is independent of the quality of the input solution s_0. The quality of the solutions for the small inputs is comparable to the ones obtained by the tabu search approach in Section 3.2.5. However, one main difference when comparing these results is that not only the runtime, but also the *quality* of the simulated annealing method is *rather independent from the quality of the initial solution* (correspondingly the number of initially applied swaps to the optimal solution) except for the near-optimal example input with only one initial swap (the reason for that is given at the end of Section 3.2.8). The conformity of the results is based on the high starting temperature allowing for a high amount of non-improving swaps and is one of the core features of simulated annealing compared to the other presented iterative methods when dealing with arbitrary inputs.

Remark 30. *The results already show that the quality of a simulated annealing method strongly depends on its defining parameters. The cooling schedule itself, another very important influence, was not even varied at all in the presented experiments. There are exceptionally tuned simulated annealing schedules for specific problems, optimized towards producing good resulting assignments while spending as little time as possible. One such simulated annealing routine (which was fine-*



(a) Quality (min and max deviation)

(b) Time (average)

Figure 24: Simulated Annealing (SA), $t_0 = 10000$ and $t_\omega = 0.0001$ [10 runs]

tuned over many years) is taken as the comparison method in the later parts of this work as it is known to be very effective for good FPGA placement.

Remark 31. *Even though several works have investigated under which circumstances simulated annealing reaches global optima with a high probability or even guaranteed (e. g., the work of Granville et al. [77] or Rajasekaran [153]), the optimality of results obtainable by simulated annealing approaches is often of minor practical importance. Instead, it is often more relevant in practice that simulated annealing tends to find good solutions in relatively short times and mostly independently from the initial quality.*

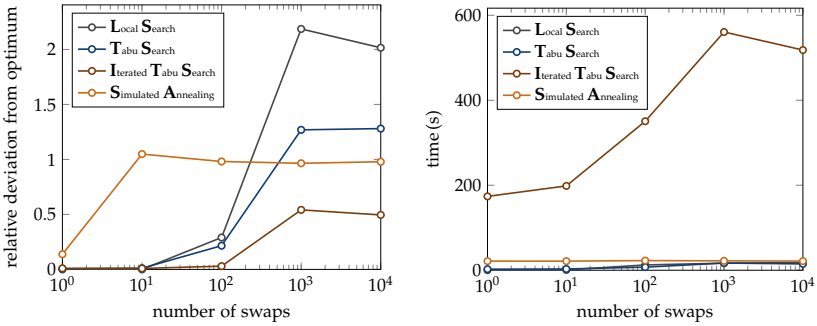
3.2.8 *Comparison*

To give a final overview of the presented methods, this section compares only the inputs on the largest grid (16×16).

The runtimes and the assignments' qualities are presented in Figure 25. The resulting 2D view (by neglecting one dimension - *the sizes*) provides a clearer view on the proportions of the measurements.

Remark 32. *As the **smallest** real-world examples of layout graphs that are investigated later in this work contain a comparable number of elements, it is justifiable to concentrate only on the* 16×16 *example here.*

Figure 26 shows the best resulting assignments for $N = 16$ attained in the repeated experiments. While the local search result is still rather perturbed, tabu search an simulated annealing already generate rather good assignments. Nevertheless, simulated annealing is (in this case) slightly better while needing almost the same amount of time. Iterative tabu search



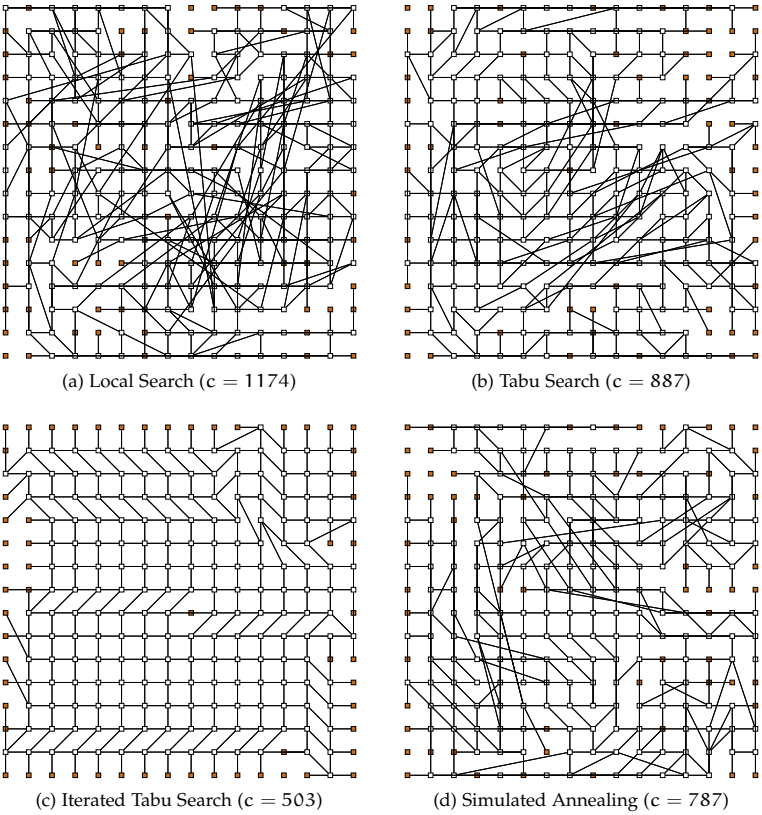
(a) Quality (average)

(b) Time (average)

Figure 25: Comparison for 16×16 instance [10 runs]

leads to very good assignments at the price of exceptionally high runtimes compared to the other methods (see Figure 25b). Figure 26c shows that a great portion of the I/O nodes are already placed on or near the frame of the grid (cp. Remark 27) and that the overall structure of the assignment is near to the desired grid structure. Nevertheless, it also descriptively presents the difficulty to escape the final local ITS optimum. For example, consider the two unconnected nodes on the left face of the frame. They would have to be swapped with any of the corner nodes to reach their optimal final location. However, such a swap with a corner node of this assignment would increase the wirelength still moderately when taking the nearby *north-west* corner to swap and more considerable when taking any of the other corners. There are several rectangular parts containing a perfect grid structure. However, pairwise swaps would first of all increase the overall costs to move such a complete block within the entire layout.

Figure 25a shows that the quality of the search methods can be rated by LS $<$ TS $<$ ITS, an outcome that could have been expected as these methods successively improve each other in the manners described. The runtime could be rated by LS $<$ TS $\ll$ ITS. Basing on local search's behavior, these methods all perform particularly well for input instances that are near-optimal (with small number of swaps in the input instance), but their quality worsens for intensively perturbed inputs. Simulated annealing instead reaches a rather constant quality (except for the almost optimal input with only one swap to the optimum) and outperforms local search and tabu search for the inputs with higher perturbation. Even though iterative local search still reaches better results than simulated annealing for these, the runtime of ITS is higher by a factor of 24.



(a) Local Search ($c = 1174$)

(b) Tabu Search ($c = 887$)

(c) Iterated Tabu Search ($c = 503$)

(d) Simulated Annealing ($c = 787$)

Figure 26: Best assignments for 16×16 grid with 10^4 initial swaps [10 runs]

Figure 27 shows the convergence behavior of s_{cur} in the simulated annealing method and thereby reveals the reason for the qualitative positive outlier in case of the near-optimal initial solution with only one swap. The simulated annealing approach does not lead to the good result, it is simply the very good initial solution that is not improved at all during the annealing process. For better visibility, the dashed lines mark the initial costs $c(s_0)$ for all constellations until the solution is actually improved by the algorithm. Only for the '1-initial-swap', and therefore in case of an extremely good first assignment, the annealing costs never fall below this initial value. The data series in Figure 27 expose that all runs follow a very similar annealing process, regardless of the initial solutions s_0. Due to the hot temperature in the

system at the beginning of the process, the solution is initially *vigorously* perturbed before the system cools down (quite drastically at the beginning due to the exponential cooling function) to converge to a stable solution s^*. Like in Figure 23, the annealing convergence also shows that the two best initial constellations can not achieve any improvement when setting $t_\omega = 100$. This confirms the assumptions from Section 3.2.7 that the resulting qualities for $t_\omega = 100$ often directly correspond to the quality of the input s_0. It is also noticeable that the process (almost) reaches its *stable state* long before the termination of the procedure due to an unnecessarily too lowly defined t_ω in this case (*in terms of time saving*). This is another evidence that tuning the procedure's parameters can, for example, greatly reduce the required time to solution without loosing notable quality.

The tuning of parameters to obtain a high qualitative assignment in short time with simulated annealing is therefore (once more) unveiled to be extremely valuable. Even though the input size in these experiments is fixed, it is important to note that the acceptance probability directly bases on the difference in costs caused by a swap. Therefore, it also depends on the input size as larger input grids allow for swaps to more distanced locations (*yet another parameter*).

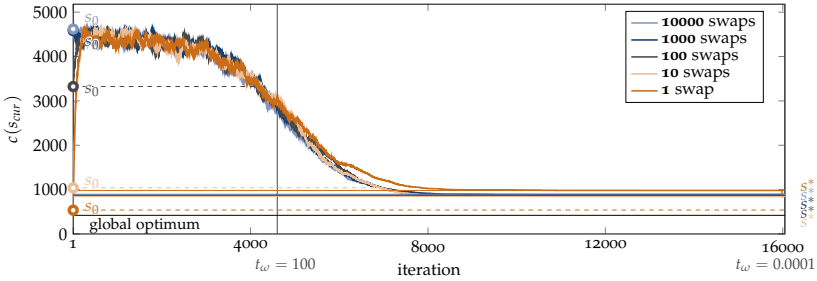


Figure 27: Convergence of SA for $N = 16$ and various s_0 qualities

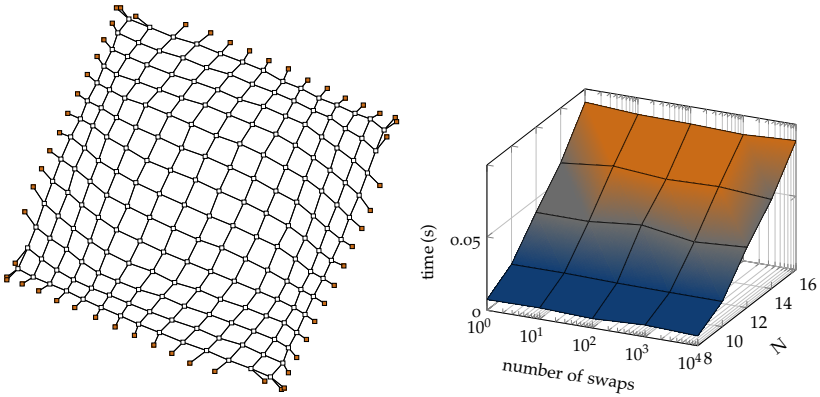
Note that Figure 27 reports $c(s_{cur})$ and therefore *all* (also non-improving) current solutions in contrast to Figures 21a and 21b which contained only the improving interim solutions $c(s^*)$.

Interim Result 1. *The independence of the initial assignment s_0 along with the good results and the relatively short runtime approves **simulated annealing** as a good choice for solving the assignment problem in time critical circumstances with unknown input quality. If the initial assignment s_0 is known to be rather good already, a fast local search based approach can nevertheless still be favorable to incrementally iterate to a near local optimum while making use of the good initial quality.*

3.3 A LAYOUT THROUGH FORCE-DIRECTED GRAPH DRAWING

With all the previous experiments as a base, the following section gives a first brief motivation why force-directed methods (see Chapter 4) are used as a fundamental component in this work.

Figure 28a shows the result of a force-directed graph layout of the *idealized layout graph* (cp. Figure 15) with $N = 16$, constructed as described in the following chapter through the *Fast Multipole Multilevel Method* (FMMM or FM^3) by Hachul and Jünger [82, 83, 84].



(a) Resulting layout

(b) Time (average)

Figure 28: Force-directed graph layout

Without going into the details of this method *in this section*, the outcome in Figure 28a shows a very balanced grid with surrounding (*orange*) I/O nodes, principally similar to the optimal assignment in Figure 15. However, this layout is no mapping to an integer grid. It is directly possible to create such layouts even with integer coordinates, but this usually comes at the price of large coordinates by multiplying the final real coordinates with a constant so that they occupy different integer points. Nevertheless, such methods often produce large gaps in between coordinates for more complexly connected graphs. An a priori set restriction of the layouting region in terms of size and coordinate type withdraws degrees of freedom in the method that are crucial for the result and the performance, especially to escape local optima.

The general resulting graph layout is profoundly *independent from the quality of the initial solution* as it creates, in contrast to all presented metaheuristics, a *good* initial graph layout exploiting a *multilevel method* basing on the *connection structure of the graph*. The layout is then iteratively improved based on a

physical model minimizing the distances between connected nodes (and thus the wirelengths) while keeping spaces between nodes by repulsive forces until an *equilibrium state* is reached.

Simulated annealing in contrast achieves the independence by extreme initial perturbation of the graph and therefore in a rather 'immethodical' manner.

While the *quality* of the result for the *idealized layout graph* looks very promising already, it is particularly imposing to see the corresponding *time* (Figure 28b) that was needed to produce such layouts. A layout for the $N = 16$ took averagely *not even a tenth of a second*! Compared with the runtimes of the presented iterative methods, this is extraordinarily fast. The independence of the input solution comes naturally, as the method produces its own starting point based on the graph structure. To keep the overall runtime moderate, especially for larger graphs, several time reducing approximations were developed in the field of force-directed layouts. These and other benefits are directly carried over into the presented method.

Even though such a force-directed graph layout looks promising for the desired assignment, such a layout is not directly fitted to the available slots as it does not match a restricted integer grid (due to the already mentioned *arbitrary* and *real* coordinates) and also due to its potential inner rotation angle.

Remark 33. *It should be noted that the very good results of the force-directed method concerning quality and time are partially based on the idealized regular structure of the graph. The following chapters will deal with the behavior of the method for more general input graphs. They will also show the demands on and the influences of configurations of the method to achieve good results in general.*

The basic idea of this work is to take such created layouts and embed the nodes to the restricted integer grid as similar as possible to how they appeared in the generic force-directed layout.

4

Force-directed graph layouts

"May the force be with you."
— Yoda —

Contents

4.1 FORCE-DIRECTED GRAPH LAYOUTS

Force-directed graph layouts (like the one shown in the previous chapter in Figure 28a) are the foundation of the approach presented in this work. Thus, the basic concepts and their emerging will be pictured in the beginning of this chapter. Based on these, advanced techniques for such force-directed graph layouts are presented to establish different benefits that the chosen method for the presented framework has compared to traditional ones and also compared to other current implementations.

In the following, a graph $G = (V, E)$ is an *undirected* graph with nodes (vertices) V and connections (edges) E. It therefore holds true that $(u, v) \in E \Leftrightarrow (v, u) \in E$ for $u, v \in V$.

Definition 9. *A graph $G = (V, E)$ is called **undirected** if its edges have no orientation. Thus, the edges (u, v) and (v, u) are identical. This work generally operates on such undirected graphs (unless stated otherwise).*

Definition 10. *A graph $G = (V, E)$ is called **simple** if at most one edge exists between any pair of nodes in the graph.*

Definition 11. *A graph $G = (V, E)$ with nodes V and edges E is called **complete** if E contains each possible node-to-node connection in the graph. An undirected complete graph contains exactly $|E| = \binom{|V|}{2} = \frac{|V| \cdot (|V|-1)}{2}$ edges (cp. Corollary 2).*

Definition 12. *A node $v \in V$ is a **neighbor** of node $u \in V$ ($v \in N(u)$) in the undirected graph $G = (V, E)$ if, and only if, $(u, v) \in E$. The nodes v and u are then called to be **adjacent**.*

Definition 13. *A node v in a graph $G = (V, E)$ has **degree** $\delta(v)$ if it has exactly $\delta(v)$ neighbors.*

Definition 14. *A graph $G = (V, E)$ with nodes V and edges E is called **connected** if there exist no two nodes $u, v \in V$ such that G contains no path of edges with u and v as its endpoints.*

Definition 15 (Schrijver [164])**.** *A (not complete) graph has **connectivity** k if there does not exist a set of $(k-1)$ vertices whose removal disconnects the graph. The graph is then called **k-connected** or, more precisely, **k-vertex-connected**.*

Definition 16. *An **embedding** of a graph $G = (V, E)$ is an assignment of the nodes to the plane (with two dimensional coordinates) and of the edges to plane curves.*

Definition 17. *An **embedding** of G is called **planar** if no two edges intersect each other.*

Definition 18. *A **graph** G is called **planar** if a planar embedding of G exists.*

Definition 19. *The **canvas** of an embedding of a graph G is the smallest rectangle that contains all nodes of G.*

4.1.1 *Basic idea of Tutte*

Tutte [180, 181] introduced the idea of drawing graphs based on barycenter mapping in the 1960s. He presented a method that can be applied to produce a convex planar embedding (crossing-free with straight edges) of a simple 3-vertex-connected planar graph with each node being located in the barycenter of its neighbors in the graph and each face in the graph forming a convex polygon.

Remark 34. *Aside from the graph-theoretical term of '3-vertex-connected planar graphs', such graphs are (**geometrically**) also called to be **polyhedral**.*

To calculate a *Tutte embedding* (or *barycenter embedding*), a system of linear equations can be created and subsequently solved.

To achieve a situation in which each node is in the barycenter of its neighbors, the x and y coordinates of all nodes have to satisfy equations (34).

$$\begin{aligned} x(v_i) &= \sum_{v_j \mid (v_i, v_j) \in E} x(v_j) \\ y(v_i) &= \sum_{v_j \mid (v_i, v_j) \in E} y(v_j) \end{aligned} \tag{34}$$

It is obvious that such a system with no further restrictions has a trivial solution with $x(v_i) = y(v_i) = 0 \; \forall v_i \in V$. However, the result that all nodes are placed onto the the same point $(0, 0)$ is not Tutte's desired solution. The goal of a Tutte embedding is a *tidied* drawing of the graph with no edge-crossing if it is a 3-vertex-connected planar graph. In order to achieve this, antagonistic forces have to be present in the system. For a Tutte layout, the coordinates of at least *three* nodes (*corners*) have to be *fixed* while the other nodes of the graph can be placed each in barycenter of its neighbors which will consequently be in the convex envelope of all fixed nodes.

Let V^0 be the set of fixed nodes and V^1 be the set of free nodes with $V = V^0 \cup V^1$ and $V^0 \cap V^1 = \emptyset$. Let $N^0(v_i)$ denote the set of fixed and $N^1(v_i)$, consequently, the set of free neighbors of v_i and let the superscript * highlight that a coordinate is fixed (and therefore not *variable*). For each free node, the barycentric coordinate with respect to all neighbors can be calculated as stated in equations (35) and (36).

$$\begin{aligned} x(v_i) &= \frac{1}{\delta(v_i)} \cdot \left(\sum_{(v_i,v_j)\in E} x(v_j) \right) \\ &= \frac{1}{\delta(v_i)} \cdot \left(\sum_{v_j \in N^1(v_i)} x(v_j) + \sum_{v_k \in N^0(v_i)} x^*(v_k) \right) \quad \forall v_i \in V^1 \end{aligned} \tag{35}$$

$$\begin{aligned} y(v_i) &= \frac{1}{\delta(v_i)} \cdot \left(\sum_{(v_i,v_j)\in E} y(v_j) \right) \\ &= \frac{1}{\delta(v_i)} \cdot \left(\sum_{v_j \in N^1(v_i)} y(v_j) + \sum_{v_k \in N^0(v_i)} y^*(v_k) \right) \quad \forall v_i \in V^1 \end{aligned} \tag{36}$$

This forms two independent systems of linear equations $A \cdot x = b^x$ and $A \cdot y = b^y$ for the two coordinate vectors x and y, each with $|V^1|$ variables (column j represents the free nodes' (v_j's) coordinates) and $|V^1|$ equations (row i represents the connections of v_i to its neighbors).

Following equations (35) and (36), the matrix $A = (a_{ij}) \in \mathbb{Z}^{|V^1|\times|V^1|}$ and the vectors $b^x = (b^x_i) \in \mathbb{Z}^{|V^1|}$ and $b^y \in \mathbb{R}^{|V^1|}$ can be constructed as shown in equation (37).

$$\begin{aligned} a_{ii} &= \delta(v_i) && \forall v_i \in V^1 \\ a_{ij} &= -1 && \forall (v_i, v_j) \text{ with } v_i \in V^1 \text{ and } v_j \in N^1(v_i) \\ b^x_i &= \sum_{v_k \in N^0(v_i)} x^*(v_k) && \forall v_i \in V^1 \end{aligned} \tag{37}$$

An example of the calculation and the outcome of a Tutte embedding is shown in Appendix A.2.

Remark 35. *This barycentric idea of Tutte simulates that connected nodes* ***attract*** *each other and derives an energy/force-minimal state of such a system (see Theorem 2).*

*If a node v_i would **not** be placed in the barycenter of its neighbors, there would be a resulting **force (vector)** $\vec{F}_{attr}(v_i) \neq \vec{0}$ acting on node v_i which can be quantified by equation (38).*

$$\vec{F}_{attr}(v_i) = \sum_{v_j|(v_i,v_j)\in E} \left(\begin{pmatrix} x(v_i) \\ y(v_i) \end{pmatrix} - \begin{pmatrix} x(v_j) \\ y(v_j) \end{pmatrix} \right) \tag{38}$$

Tutte has proven in his work [181] that the equation system is *non-degenerate* for 3-vertex-connected planar graphs. Thus, the system has a unique solution under these assumptions. With a given set of fixed nodes, such a graph therefore has a unique Tutte embedding which can be computed by linear equation solvers naively in $\mathcal{O}(n^3)$ time in general.

Given a graph, one non-trivial task for a Tutte embedding is to choose a set of fixed nodes and assign respective coordinates to them that generate a *good* layout. Figure 30 shows different Tutte embeddings of the 'Sierpiński Sieve Graph'. This graph is constructed by starting with an isosceles triangle and recursively subdividing it in four equally sized isosceles triangles. This fragmentation is performed in every recursion step for each but the central appearing *bottom up* triangle. The graph in recursion number n is called the *Sierpiński Sieve Graph of order* n ($\mathcal{S}_n$).

The Sierpiński Sieve Graphs are only *2-vertex-connected* so that the resulting layout may not be planar for all choices of fixed nodes. However, the method places the nodes in the barycenter of their neighbors and the connectivity preserves unique solvability of the equation systems. Altogether, the constructed graphs are good examples to show characteristics of the Tutte embedding. A graph $\mathcal{S}_{n>1}$ consists of of $\frac{3}{2} \cdot (1 + 3^{n-1})$ vertices and 3^n edges.

The example graph obtained from this construction is shown in Figure 29 ($\mathcal{S}_5 \Rightarrow |V| = 243, |E| = 123$).

Figures 30a-30c show different embeddings occurring from various choices of *three* nodes of the graph that are fixed and equally distributed on a circle. Figure 30d depicts the Tutte embedding when all nodes that define the *largest face (in terms of surrounding nodes)* are fixed and also positioned equally distributed on a circle.

Two general observations for Tutte embeddings become very clear by these examples:

- A Tutte embedding often wastes a lot of space (a few large and many small polygons are formed). *The nodes are not evenly distributed on the canvas.*
- The *choice of fixed nodes* influences the derived embedding vigorously.

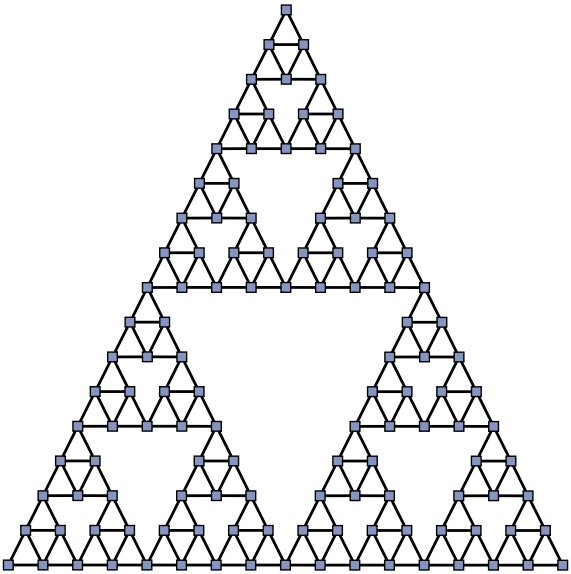


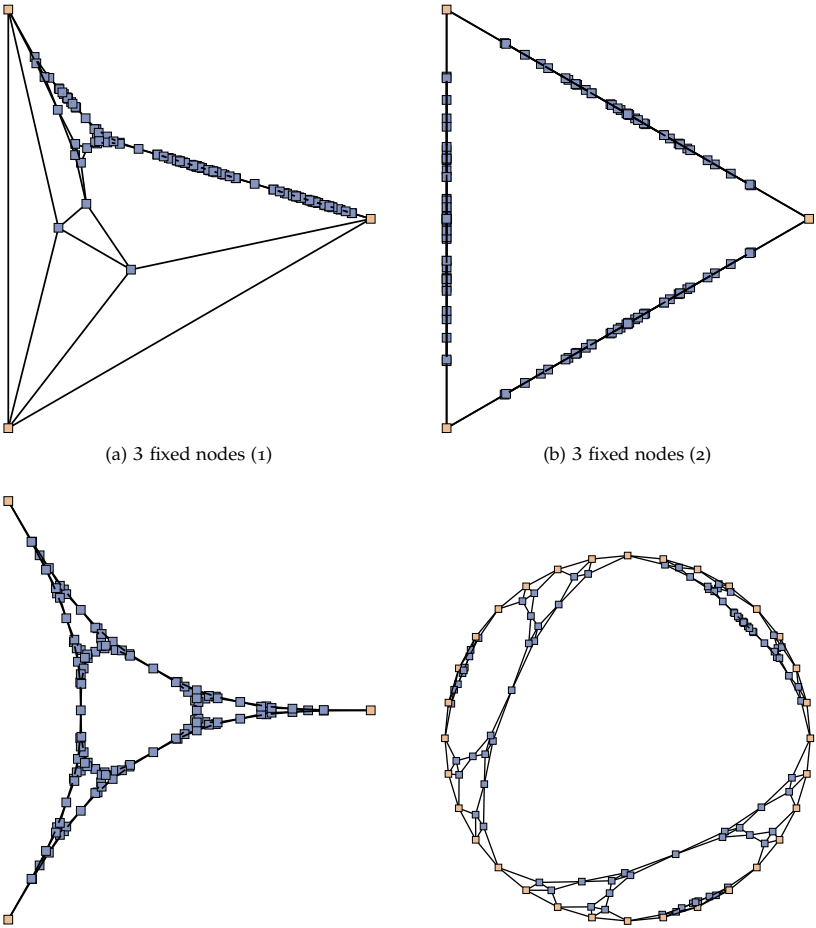
Figure 29: Sierpiński Sieve Graph S_5

Remark 36. *In their work 'Drawing Stressed Planar Graphs in Three Dimensions' [51], Eades and Garvan have proven that the worst case resolution of a (two-dimensional) Tutte Drawing is* $\Omega(k^n)$ *with* $k > 1$. *Thus, exponential space is needed for a Tutte embedding in the worst case.*

To give a second example, Tutte embeddings of the 'Crack' graph (planar, 3-vertex-connected, with $|V| = 10240, |E| = 30380$) taken from the 'Open Graph Drawing Framework' OGDF [36] library have been calculated. Figures 31a-31c show the results for different numbers of fixed nodes and Figure 31d, again, presents a layout resulting from fixing all nodes of the largest (*outer*) face on a circle.

As before, fixing only few nodes results in embeddings that waste a lot of space on the canvas. Fixing the nodes of the entire outer face (in a successive order following the face's surrounding nodes *clockwise* or *anti-clockwise*) leads to a good and biased embedding due to the regular connectedness of the graph.

However, the strategy of placing all nodes in the barycenter (or the *centroid*) of their neighbors is a meaningful approach for a *good* embedding as it minimizes the *sum of squared distances* between a finite set of points (see Theorem 2).



(a) 3 fixed nodes (1)

(b) 3 fixed nodes (2)

(c) 3 fixed nodes (3)

(d) Nodes of largest face fixed

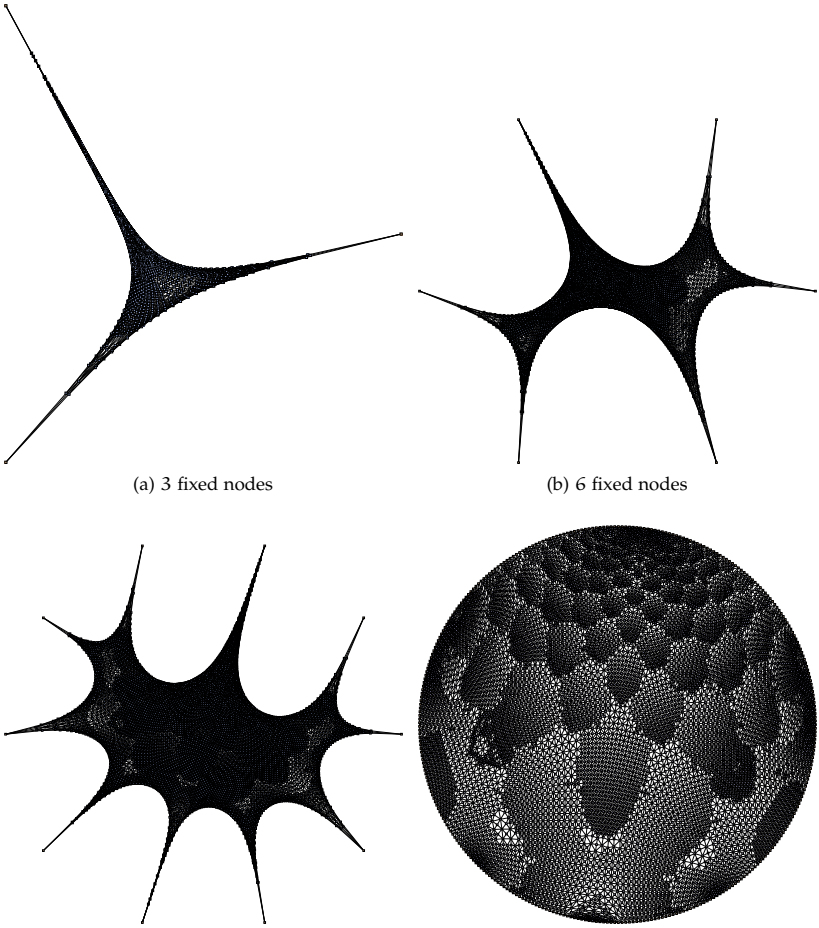
Figure 30: Tutte node fixing examples for 'Sierpiński' graph S_5

Theorem 2. *The **centroid** of a finite set of n elements p_i in $\mathbb{R}^d$*

$$C(p_1, \ldots, p_n) = \frac{p_1 + \cdots + p_n}{n} = \frac{1}{n} \cdot \sum_{i=1}^{n} p_i$$

*minimizes the **sum of squared distances** between itself and all points of the set.*

Proof. Mathematical folklore. □



(a) 3 fixed nodes

(b) 6 fixed nodes

(c) 10 fixed nodes

(d) Nodes of largest face fixed

Figure 31: Tutte node fixing examples for 'Crack' graph

Even though Tutte's method is a comprehensible and direct approach which is conceptually simple while following good ideas for common embeddings (*like edge length minimization*), the previous examples also show some general drawbacks of Tutte's method: a) the embeddings often *waste a lot of space* on the canvas, b) the *choice of fixed nodes* and their positioning are very influential on the resulting embedding and, finally, c) the method is only applicable for a small subset of (*3-vertex-connected planar*) graphs. The

considered input graphs of the framework presented in this work are usually *not* of this nature.

Furthermore, solving the equation system by direct methods like the Gaussian elimination requires $\mathcal{O}(n^3)$ (in this case $\mathcal{O}(|V^1|^3)$) operations and is, thus, still relatively *time-expensive*.

However, one goal that Tutte's method achieves by generating a guaranteed planar embedding (for *3-vertex-connected planar graphs*) is the minimization of edge-crossings (in particular *to zero*) and it additionally places all nodes in the convex hull of the fixed nodes (naturally by the barycenter placement).

Remark 37. *The minimization of the overall* ***edge-length*** *and the number of* ***edge-crossings*** *are important for the method presented in this work as the edges represent the* ***connections*** *on the reconfigurable chip that have to be routed on the architecture. Too many edge-crossing will cause the necessity of longer detours for the edges ('wires') on the chip and longer edges, in general, fill the limited routing resources more quickly.*

Apart from the already mentioned *aesthetic criteria* for 'good' embeddings (*short edges, as few edge-crossings as possible*), a common criterion (e. g., in the field of graph drawing) is that nodes should keep a certain distance to each other.

Remark 38. *Another early graph drawing technique related to the fundamental goals of this work is a method for direct embeddings* ***of planar graphs on a Manhattan grid*** *which was presented by Tamassia [177] in* 1987*. In particular, Tamassia proposed a method for region preserving grid embeddings with a minimum number of bends on edges by using network-flow techniques.*

4.1.2 *Generalization of the model - Spring Embedder*

The basic *'barycentric'* idea of Tutte's embedding can also be applied *to arbitrary* graphs with at least 3 fixed nodes (not only 3-vertex-connected planar graphs) by *iteratively* calculating the forces $F(v_i)$ acting on each free node ($\forall v_i \in V^1$) and moving the node (possibly only 'a bit') in this direction. After all free nodes of the graph have been moved once in that way, the procedure can be repeated again and again until either a fixed number of iterations has been performed or until the system reaches a stable state (formally until the lengths of all force vectors are *almost zero*).

As it has already been mentioned in Chapter 4.1.1, a system without fixed nodes that minimizes the overall edge-length (or the edge-*forces*) would result in an assignment of the same position to each point in the graph. Tutte's mechanism to overcome this was to fix a set of nodes (V^0) which consequently act as *counterforces* on the outer face of the resulting embedding. It

has also been shown in the previous section that the selection of fixed nodes and their positioning has a great influence on the resulting embedding.

Instead of fixing nodes to obtain a counterforce, the edges can be considered to be *mechanical helical springs* with an ideal length l connecting the nodes with each other. The ideal edge length is the 'zero-energy length' of the spring and thus the length of the spring in its unbent equilibrium state.

Remark 39. *Tutte's approach can be considered as such a spring model with ideal edge lengths $l = 0$ for all springs.*

The force of such a *linear* spring can be estimated by Hook's law (39), which states that the force needed to stretch or compress a spring from its *natural relaxed* state is *linear* to the difference ΔL of the spring's current length (in other words the distance d between spring's endpoints) and its zero-energy length l. If $d > l$, the spring is in a stretched state and the difference $\Delta L = (d - l)$ is greater than zero, whereas ΔL becomes smaller than zero if the spring is in a compressed state. This force strength of the spring can further be formalized by introducing a constant c_a depending on the material of the spring (cp. equation (39)).

$$f_a = c_a \cdot \Delta L = c_a \cdot (d - l) \tag{39}$$

Hook's law quantifies the strength of the force acting through the spring by a linear approximation of the 'real' force. Nevertheless, it is obvious that this assumption can not be true 'far away' from the zero-energy state. On the one hand, any material would break when being stretched too drastically and, on the other, it is not be possible to compress a physical spring to length *zero*.

Instead of using this linear approximation, Eades [50] proposed that a logarithmic relation, like in equation (40), behaves better in practice for far distanced pairs of points because the linear approximation is 'too strong' in such situations.

Remark 40. *In the following, p_u denotes the position (**localization**) of node u.*

Let p_u and p_v be two points in the Euclidean space ($\mathbb{R}^2$) and let $l(u, v)$ be the zero-energy length of the string between the two points. Then, the strength of an attractive force in Eades' model between two connected points can be approximated by equation (40).

$$f_a = c_a \cdot \log\left(\frac{d}{l}\right) = c_a \cdot \log\left(\frac{\|p_u - p_v\|_2}{l(u, v)}\right) \tag{40}$$

Remark 41. *Note that the zero-energy length in equation (40) is **individual for each spring** while Eades modeled it as a system-wide constant. This will in particular become important in Section 6.3.*

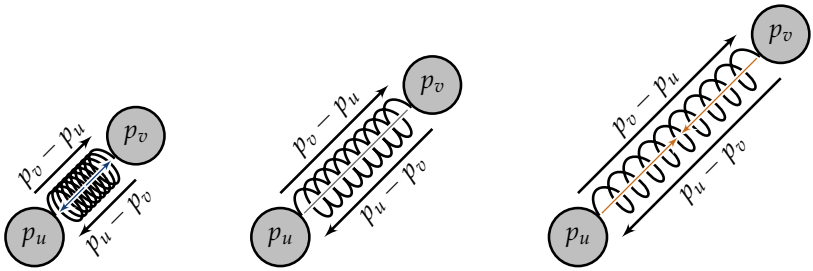
If the distance d of two adjacent points is equal to the zero-energy length l of the respective string, the fraction in the logarithm is *one* and, therefore, the force strength f_a becomes zero just like in the linear model in formula (39). As a result, the spring is in its 'relaxed' state and no force acts to either compress or stretch it (see Figure 32b).

Besides a model (approximation) for the strength of the force that acts on a node p_v, the direction of the force is needed to calculate the consequent force vector. Assume that the force that acts on point p_v through the connection of p_u and p_v is acting in the direction of $(p_u - p_v)$ and that $c_a = 1$. Then, the force vector $\vec{F}_{attr}^{(u,v)}$ can be calculated as shown in equation (41).

$$\vec{F}_{attr}^{(u,v)}(v) = f_a \cdot (p_u - p_v) = \log\left(\frac{\|p_u - p_v\|_2}{l(u,v)}\right) \cdot (p_u - p_v) \tag{41}$$

If $d > l$ (cp. Figure 32c), the spring is in a *stretched state* and therefore *tends to further contract itself* to reach the zero-energy state. Thus, the force strength f_a that acts on p_v in the direction of $(p_u - p_v)$ is *positive* in that case.

However, if $d < l$ (cp. Figure 32a), the spring is in a *compressed state* and therefore *tends to push its ends further apart* to reach the zero-energy state. Thus, the force strength f_a that acts on p_v in the direction of $(p_u - p_v)$ is *negative* in that case.



(a) $d < l$ (b) $d = l$ (c) $d > l$

Figure 32: Force strengths with distance d and zero-energy length l

As a result, the logarithm of the fraction of d and l in equation (40) qualitatively models the force strengths just like the linear assumption of Hook in equation (39), but it quantifies it differently. To be more precise, the logarithm is more 'moderate' in quantifying the forces' strengths for far distanced node pairs.

In addition to the *attractive* forces introduced by the strings between connected nodes, Eades extended the model by introducing *repulsive* forces to

the nodes so that (*only*) *non-adjacent* nodes in the system repel each other. The idea is to simulate the forces as repulsive forces F_{rep} of static *electrically charged particles*. The resulting repulsive forces F_{rep} can therefore be assumed to be proportional to $\frac{1}{d^2}$ due to Coulomb's law (*also called Coulomb's inverse-square law*).

Assume that two points p_u and p_v are *not connected* in the graph and therefore not encounter a reciprocal attractive force through a connecting spring. Then, the *strength* of the repulsive force between these two nodes can be quantified by equation (42).

$$f_r = \frac{1}{\|p_v - p_u\|_2{}^2} \tag{42}$$

The resulting *vector* of the repulsive force $\vec{F}^u_{rep}$ starting out from node u that acts on node v (and therefore in direction $(p_v - p_u)$) is calculated by formula (43) in Eades' model.

$$\vec{F}^u_{rep}(v) = f_r \cdot (p_v - p_u) = \frac{1}{\|p_v - p_u\|_2{}^2} \cdot (p_v - p_u) \tag{43}$$

Remark 42. *Both types of force vectors (**attractive** and **repulsive**) are calculated by the product of the **forces strengths** ($\mathbf{f_a}$ and $\mathbf{f_r}$) and the direction (vector) of the **force** ($(\mathbf{p_u} - \mathbf{p_v})$ and $(\mathbf{p_v} - \mathbf{p_u})$, respectively).*

With the two principal force sources (41) and (43), the overall system is able to converge to a stable state of balanced forces that is *not* a single point for all nodes and does not require any fixation of nodes. In contrast to Tutte's approach, in which the counterpart for the attractive forces was introduced by fixed nodes on the outer face, this *generalized* and *extended* model contains the repulsive forces between non-adjacent nodes (and also the zero-energy lengths for connected nodes) as 'opponents' to the attractive forces of connected nodes.

Thus, even if the graph is *complete* (eventuating in a system without repulsive forces), it does not collapse to a single point as long as the zero-energy lengths l of the springs are not zero.

In Figure 34b on page 102, the functions of the two forces' strengths are shown. The overall force strength acting on node v levels off to zero if the repulsive and the attractive forces *neutralize each other* and if, therefore, the pairwise distances d between all connected nodes correspond to the desired zero-energy lengths. Such a force model is often called a '*system of springs and magnets*' due to its *real-world counterpart*.

The process of iterating towards an equilibrium state with Eades' force model is summarized in Algorithm 5. After positioning all nodes randomly to obtain an initial configuration, the iterative procedure of minimizing the

forces in the system is started. In each iteration, the repulsive and the attractive forces that act on each of the nodes are calculated according to the previously defined force model. A resulting force $F(v)$ is derived for each node v by summing up the relevant forces and potentially scaling the sum of forces with parameters called the *stiffness factor* λ_{attr} for the attractive forces and the *repulsion factor* λ_{rep} for the repulsive forces. After these calculations have been completed for all nodes, each node v is moved 'a little' (more formally by the proportion δ) in the direction of the resulting acting force $F(v)$ whereupon the entire process is repeated with the new forces acting on the nodes due to their updated positions. If a fixed number of such iterations has been performed, the process terminates returning the final positions of all nodes.

Algorithm 5 Spring Embedder (Eades)

procedure SPRINGEMBEDDER(G,nb_iterations,δ)
  for all $v \in V$ **do** ▷ initially assign random positions
    $p_v \leftarrow \mathtt{random}(x, y)$
  end for
  $i \leftarrow 0$
  while $i <$ nb_iterations **do**
    for all $v \in V$ **do** ▷ calculate forces
      $F^u_{rep}(v) \leftarrow \frac{1}{||p_v - p_u||_2{}^2} \cdot (p_v - p_u)$
      $F^{(u,v)}_{attr}(v) \leftarrow \log\left(\frac{||p_v - p_u||_2}{l}\right) \cdot (p_u - p_v)$
      $F(v) \leftarrow \lambda_{rep} \cdot \sum_{u|(u,v)\notin E} F^u_{rep}(v) + \lambda_{attr} \cdot \sum_{(u,v)\in E} F^{(u,v)}_{attr}(v)$
    end for
    for all $v \in V$ **do** ▷ move nodes
      $p_v \leftarrow p_v + \delta \cdot F(v)$
    end for
    $i \leftarrow i + 1$
  end while
  return positions ▷ set of coordinates for each $v \in V$
end procedure

In each iteration, the attractive forces arising from the springs (itself representing the $|E|$ edges of the graph) and the repulsive forces between each non-adjacent pair of nodes (each pair of nodes *but* the connected ones in the graph: $|V|^2 - |E|$) have to be calculated. Thus, the overall time complexity of *one* such iteration is $\mathcal{O}(|E|)$ for the attractive forces and $\mathcal{O}(|V|^2 - |E|)$ for the

repulsive forces, therefore $\mathcal{O}((|V|^2 - |E|) + |E|) = \mathcal{O}(|V|^2)$ in total. With a fixed number of iterations, this is likewise the complexity of the entire procedure.

Remark 43. *The termination criterion could easily be formulated more adaptively and dynamically, e.g., by stopping either as soon as the sum of all acting forces $\sum_{v \in V} F(v)$ in the system falls below a certain predefined (small) **threshold** (as this indicates an equilibrium state) or if a maximum number of iterations has been performed.*

The overall complexity of Eades' approach is independent from the number edges in the graph, whereas the calculation of the repulsive forces is the dominating compute part of the method for general graphs (especially for rather *sparse* graphs with $|E| \ll |V|^2$ edges).

Definition 20. *A graph $\mathcal{G} = (V, E)$ will be called **sparse** in the following if, and only if, the number of edges in the graph is **much smaller** than the maximal possible number of edges $\frac{|V|(|V|-1)}{2}$ (cp. Corollary 2), or - to put it simply - if $|E| \ll |V|^2$. Generally, graphs are divided roughly in dense and sparse graphs by a property called the **density** of the graph which can, for example, be formalized by $\rho_{\mathcal{G}} = \frac{2|E|}{|V|(|V|-1)}$ for **undirected** and **simple** graphs. A complete graph consequently has a density of 1, the **sparsest** possible **connected** graph, consisting only of a path with n nodes, has a very low density of $\rho_{\mathcal{G}} = \frac{2(n-1)}{n(n-1)} = \frac{2}{n}$ with this quantification. By setting a certain threshold, it is possible to divide graphs into either the one or the other class, but it is rather a vague separation in general.*

The fact that the *complexity* of one single iteration is $\mathcal{O}(|V|^2)$ makes Eades' approach inapplicable for larger graphs. The initial random positioning of the nodes can additionally prevent the algorithm from reaching a good *local* or even a *global optimum*. Both aspects have consequently been improved by more advanced techniques, whereas an early improvement for the expensive calculation of repulsive forces has been introduced by Fruchterman and Reingold in 1991.

4.1.3 *Grid approximation of repulsive forces*

The general principle behind a *spring embedder* model is that, on the one hand, connected vertices should be placed near to each other and, on the other hand, vertices should generally not be placed too close to each other (cp. Fruchterman & Reingold [67]).

While this can principally be achieved with the model introduced by Eades, Fruchterman & Reingold's goal was to extend Eades' model in some certain directions and to reduce the overall computation time.

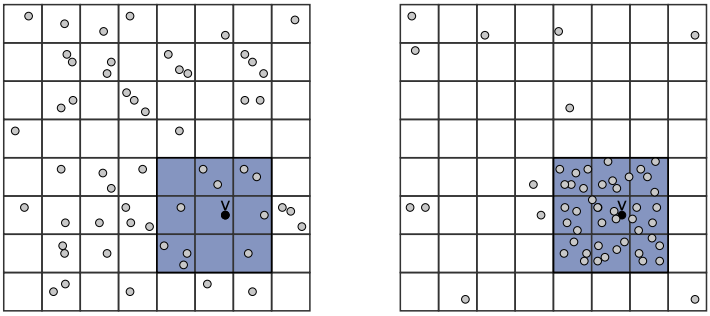
Calculating *all* pairwise *repulsive forces* between nodes in a graph $\mathcal{G} = (V, E)$ takes $\mathcal{O}(|V|^2)$ time as there are $\binom{|V|}{2} = \frac{|V|(|V|-1)}{2}$ such pairs in the graph (cp. Corollary 2). Considering that this set of calculations would have to be performed in every iteration of the spring embedder, it is obvious that this is relatively expensive in terms of runtime. Thus, it could only be applicable for rather small graphs if a 'precise' layout (with many iterations and tendentially small δ) of the graph is desired in a relatively short time. Combined with the fact that '*ordinary*' graphs (and also the ones that are present in the application of this work) *are* rather *sparse* ($|E| \ll |V|^2$), the bottleneck concerning the calculation time is unambiguously the repulsive force computation as the attractive force evaluation only needs $\mathcal{O}(|E|)$ calculations. On the other hand, it is obvious that repulsive forces between nodes that are far away from each other are not too influential.

An early attempt to reduce the bottleneck was published by Fruchterman & Reingold in 1991 based on considerations from *particle physics* that arise in the related field of *n-body* simulations (e. g., planetary simulations). Their idea was to calculate only *those* repulsive forces acting on a node $v \in V$ that arise from nodes in *its neighborhood*. For this purpose, the graph is placed on a *canvas* and the neighborhood is now constructed by dividing this *canvas* for the drawing with an equidistant $\left(\frac{\sqrt{|V|}}{k} \times \frac{\sqrt{|V|}}{k}\right)$ grid (with $k = 2$ in the original formulation). Now, only nodes in v's own and the eight neighboring cells are taken into account for the calculation of repulsive forces acting on node v.

Remark 44. *The main idea behind their approach is the observation that the gravitational forces between two protons in a free system attract each other while their electrical (charge) force is repelling them from each other. If the nodes are far away from each other, none of the two forces acts between the two nodes significantly. At a certain distance, the attractive force starts to become active and contracts both particles. At some point in a relatively near distance, the repulsive force from the nodes is starting to take significant effect on the overall acting force between both and reduces the overall attractive force until an equilibrium state between both forces is reached. This model explains how* ***atomic nuclei*** *cohere and why they do not collapse.*

Remark 44 gives a reasoning to neglect or at least less consider very far distanced nodes' repulsive forces what could, in turn, reduce the calculation time for these forces. Nevertheless, the grid approach of Fruchterman & Reingold has a great general disadvantage as it strongly depends on the actual distribution of the nodes across the grid. While the distribution pictured in Figure 33a is rather *uniform* and would thus reduce the number of considered repelling nodes on v drastically, Figure 33b shows how a *non-uniform*

distribution of the same number of nodes can still create situations in which the calculation of almost all repulsive forces for v would be necessary.



(a) uniform (b) non-uniform

Figure 33: Node distributions and their influence on the neighborhood size

Remark 45. *This drawback of a 'static' fragmentation of the canvas can be overcome by methods based on tree decompositions like* ***quadtrees*** *(see Section 4.2.1).*

As already stated in Section 4.1.2, the linear assumption in (39) for the *attractive forces* is not realistic for mechanical springs between nodes that are far distanced from each other. Even more, Fruchterman & Reingold stated that following Hook's linear law can also prevent the spring-system with repulsive forces from escaping local optima in graph drawing simulations. Thus, Fruchterman & Reingold [67] (in contrast to Eades) proposed to use a factor that is *quadratic* in the distance d for the attractive forces while the repulsive forces are modeled to decrease only *inversely proportionally*.

An additional cooling function in their approach descendingly limits the allowed quantity of displacement for a node per iteration to improve the drawing from a coarse grained to a fine grained manner. This idea is comparable to the mechanisms of simulated annealing approaches in general (cp. Section 3.2.7) and, more specifically, to the one published by Davidson and Harel [44] for graph drawing in 1989.

While Eades *weakened* Hook's linear assumption to a logarithmic coherence to be more moderate for far distanced nodes, Fruchterman & Reingold instead *strengthened* it to a quadratic dependence as the linear rule was often (especially for complex graphs) not strong enough to escape local optima. They additionally reported that their configuration of the overall system with the chosen pair of force functions (shown in Figure 34c) led to results comparable to Eades' but reduced the calculation effort by not using the relatively compute expensive *logarithm* function.

Remark 46. *As the calculation of transcendental functions on modern compute architectures has been continuously improved in common libraries and the hardware itself (see, for example, Harrison et al. [90]), it is nowadays indeed absolutely reasonable to use the logarithm if the algorithm behaves better in that way. It can therefore, e. g., be considered how much a more expensive calculation may reduce the amount of iterations.*

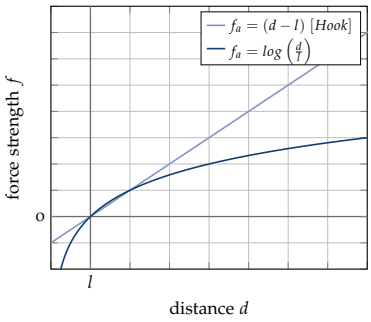
Fruchterman & Reingold additionally integrated mechanisms to keep all nodes on a predefined canvas, see Chapter '*The frame*' of their work. In fact, even the idea to overcome the drawbacks of their 'static' method in case of non-uniform distributions by the use of a *tree structure* for an approximation of far distanced sets of nodes as single poles (a well known technique from the field of n-body simulations to reduce complexity from $\mathcal{O}(|V|^2)$ to $\mathcal{O}(|V|\log(|V|))$ [11]) or as multipoles, like in Greengard and Rokhlin's work [80], was named in Fruchterman & Reingold's publication from 1991. It was actually not applied for the repulsive force calculation as the complexity of the *grid method* is as little as $\mathcal{O}(|V|)$ for uniform grids. However, for *worst-case* scenarios, like the one depicted in Figure 33b, the calculation time for the repulsive forces can increase up to $\mathcal{O}(|V|^2)$. As a result, the overall worst-case complexity of the grid method from Fruchterman & Reingold is $\mathcal{O}(|V|^2 + |E|)$ $\left(= \mathcal{O}(|V|^2)\right)$.

Remark 47. *Depending on the input graphs, their structure and their general density, the just presented basic approach can (**under certain circumstances**) be a practical choice. However, for general graphs it is advisable to use 'more advanced' models for the repulsive force approximation like the one of Hachul and Jünger presented in Section 4.2, which bases on an efficient clustering of repulsive influences from far distanced sets of nodes in a tree structure combined with an accurate approximation of the forces by multipole moments.*

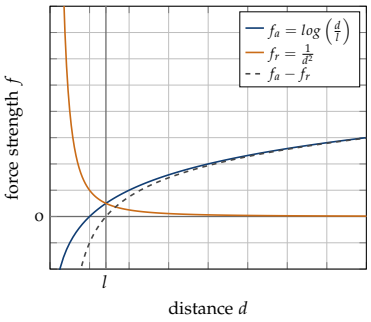
Figure 34 depicts the different force models mentioned in this section. Other force models (e. g., *linear, quadratic, logarithmic or inversely proportional models* but also *combinations* of these) and many different static or dynamic configurations of the parameters (like *the displacement factor* δ, *the repulsion factor* λ_{rep}, *the stiffness factor* λ_{attr} etc.) are possible and have been developed and tested in many researches.

It is an 'an art of its own' to find suitable configurations not only for the desired layout and its basic properties but also for a fast convergence. To, finally, say it with the words of Fruchterman & Reingold:

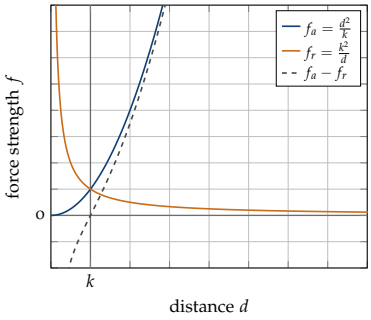
> *"We need not faithfully imitate a celestial, chemical, or atomic system - we desire only that the results be pleasing."* ([67])



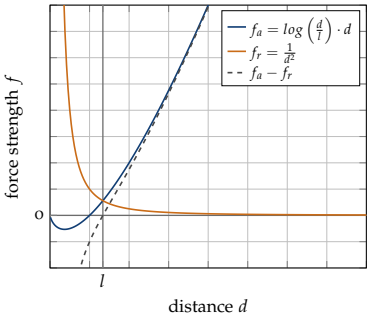
(a) *Hook*'s law & attractive forces *Eades*



(b) Attractive & repulsive forces *Eades*



(c) Attractive & repulsive forces *Reingold&Fruchterman*



(d) Attractive & repulsive forces *Hachul* (FM^3)

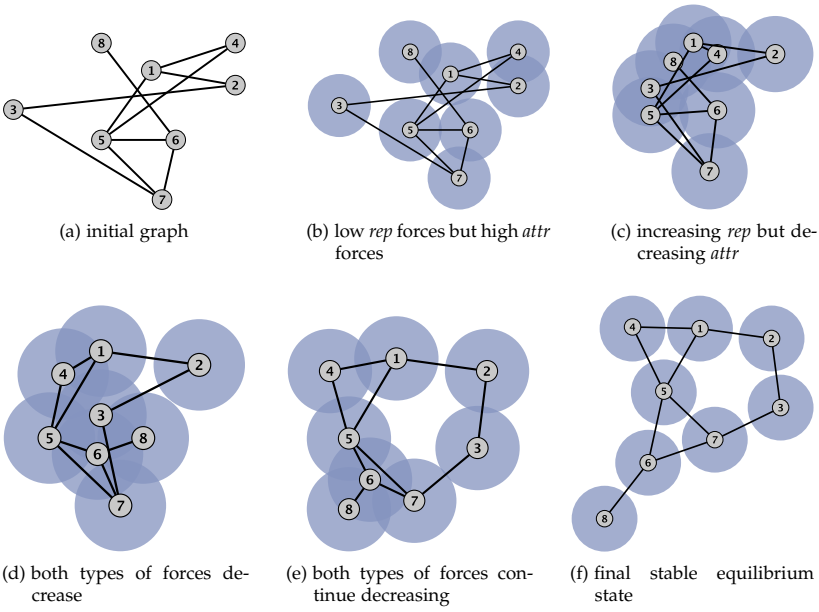
Figure 34: Attractive and repulsive forces strengths

Remark 48. *Beyond the already mentioned aesthetic criteria, force-directed models also naturally tend to create symmetric structures due to their 'undirected and locally identical' model.*

4.1.4 *A force-directed layout by spring embedder*

Figure 35 depicts the iterative process in a spring embedder procedure. The blue circles around nodes are drawn to indicate the repulsive force of the node acting in its close spatial proximity. Starting with a random initial layout, the *spring forces (edges)* attract connected nodes that are far away from each other. During the process, nodes have to pass regions with higher re-

pulsive forces to overcome local optima. This is possible in case of strong attractive forces for larger distanced connected nodes. It is important to note that, in this process, a node can undoubtedly be stuck in such a situation. It is therefore desirable to start with a *good* initial layout to avoid such situations as much as possible. One strategy to do so is a *multilevel* layout which is used in the presented implementation and explained in Section 4.2.3. When all the distances between connected nodes are more or less balanced, the repulsive forces become more influential by arranging *all* node pairs more consistently distanced to each other.



(a) initial graph

(b) low *rep* forces but high *attr* forces

(c) increasing *rep* but decreasing *attr*

(d) both types of forces decrease

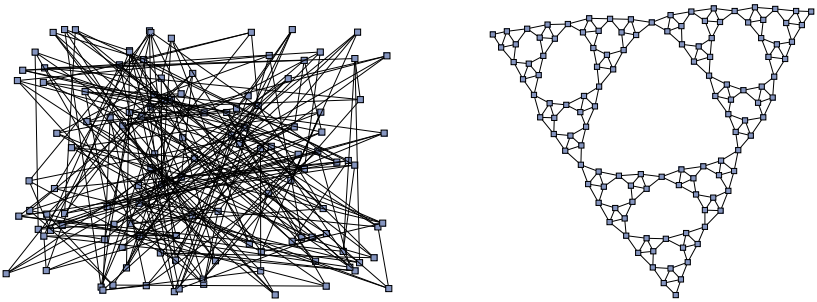
(e) both types of forces continue decreasing

(f) final stable equilibrium state

Figure 35: Iterations of a force-directed graph layout

Figure 36b shows a force-directed layout for the Sierpiński Sieve Graph of order 5 (cp. Figure 29) obtained after 'deranging' the initial layout randomly as shown in Figure 36a. The energy minimized constellation that was finally achieved by the force-directed simulation is similar to the constructed idea of the Sierpiński graph. Equilateral triangles are (*approximately*) formed and the final drawing is perfectly planar. The layout additionally shows the strong general trend to form symmetrical structures (see Remark 48).

Another important observation for the later parts of this work is that the drawing has no real orientation. Any rotation of this embedding has the



(a) Random layout

(b) Simple force-directed layout

Figure 36: 'Sierpiński Sieve Graph' of order 5

same amount of acting forces between all nodes, a fact that is formally based on the application of the L_2-norm in the force calculation methods. This characteristic will be discussed in more details in Section 5.6.2.

4.1.5 *The ideal edge length* l

To make use of all techniques presented in this work, it is important (though, in general, not *compulsory*) to involve zero-energy edge lengths for the desired graph layout to steer the distances between connected vertices in the force-directed layout. Even though these l-values have already been introduced in the previous section, the following paragraph should give a little deeper insight to their influence in the force model.

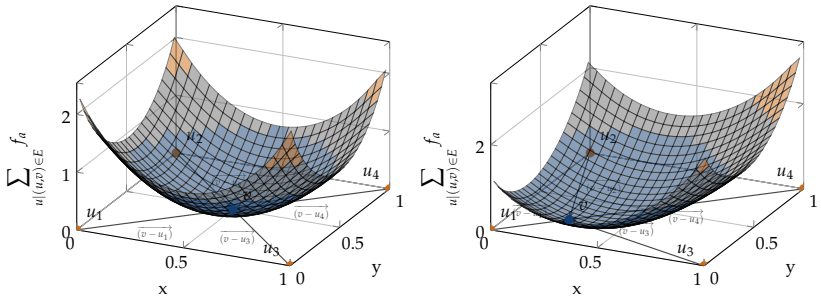
Consider the force model of Hachul and Jünger (Figure 34d) with an attractive force which is modeled following equation (44).

$$f_a = \log\left(\frac{d}{l}\right) \cdot d = \log\left(\frac{\|p_v - p_u\|_2}{l^{zero}(e)}\right) \cdot \|p_v - p_u\|_2 \tag{44}$$

If an edge is *longer* than its zero-energy length ($d > l$), the acting attractive force is *positive* while it becomes negative when the edge's length is *smaller* than it's zero-energy length ($d < l$). For $d = l$, the argument of the logarithm is exactly *one*, though the logarithm is *zero* by which means there is no acting attractive force. A state in which all distances of connected nodes correspond precisely to their zero-energy lengths can be called an *attractive force equilibrium*. The zero-energy lengths can therefore be seen as *dampers* or *amplifiers* of the attractive forces' strengths. Consider a node v which is connected to four neighboring nodes (u_1, u_2, u_3, u_4) that are, at the moment of the force-calculation, located around v at the four corners of the unit

square. The strengths of the attractive forces acting on v depends on the position of node v. The sum of the strengths of the acting forces $\sum_{u|(u,v)\in E} f_a$, depending on the position of v in the unit square, *with* $l = \frac{\sqrt{2}}{2}$ *for all edges* is depicted in Figure 37a. Under these circumstances, the sum of the forces' strengths is minimal in the barycenter of the four surrounding nodes (the *attractive force equilibrium*). Consequently, node v is pulled/pushed towards the barycenter of u_1, u_2, u_3 and u_4. If l would be larger than $\frac{\sqrt{2}}{2}$, the surrounding nodes would additionally be pushed further away in the process as the force strengths in the barycenter would fall below the value of *zero* (the position would still be optimal).

However, consider the same situation *but* with *differing* zero-energy edge lengths, e. g., with $l^{zero}(u_1, v) = \sqrt{0.25^2 + 0.25^2}$, $l^{zero}(u_2, v) = l^{zero}(u_3, v) = \sqrt{0.25^2 + 0.75^2}$ and $l^{zero}(u_4, v) = \sqrt{0.75^2 + 0.75^2}$. Figure 37b depicts the resulting forces sum $\sum_{u|(u,v)\in E} f_a$ and the consequent positioning for v in its *attractive force equilibrium* $(0.25, 0.25)$.



(a) Equal zero-energy edge lengths

(b) Different zero-energy edge lengths

Figure 37: Sum of acting forces of surrounding nodes

Remark 49. *Apart from this general incorporation of the zero-energy lengths in the force model,* FM^3 *additionally contains post-processing procedures to readjust the desired lengths of edges very accurately by a few extra iterations of the embedder with extremely reduced repulsion factor* λ_{rep} *and increased stiffness factor* λ_{attr}*. Under these assumptions, a small number of improving iterations that almost neglect the repulsive forces is performed after the* ***main*** *simulation. As a result, the final lengths of the edges of an* FM^3 *layout correspond rather precisely to the* ***a priori*** *desired user-defined edge lengths. In Chapter 7.5 of his dissertation, Hachul experimentally emphasizes the advantage that the post-processing provides (see also Section 4.2.4). Due to the zero-energy lengths, connected nodes would still retain a minimal distance to each other even if there were no repulsive forces.*

4.2 THE FAST MULTILEVEL MULTIPOLE METHOD FMMM

The force-directed layout method that was applied, extended and adapted in this work is the ***F**ast **M**ultilevel **M**ultipole **M**ethod* (**FMMM** or **FM³**). It was developed and implemented by Stefan Hachul in his PhD-work at the computer science chair of Prof. Dr. Michael Jünger and is practically available in the already mentioned graph drawing framework OGDF [36]. Detailed explanations of the method are available in several publications like [82, 83, 84] and, of course, in Stefan Hachul's PhD thesis '*A Potential-Field-Based Multilevel Algorithm for Drawing Large Graphs*' from 2005 [81]. The method and its implementation contain a great deal of specific features and many extensions to make it *fast* and *accurate* while providing a vast amount of steering functions for users to control the resulting layouts' properties. Several (but by far not all of such) functionalities of the algorithm were used in this work and the presence of many implemented mechanisms was, besides its speed and accuracy, the main argument to use FM³ as the basis for this work. However, the force-directed layout algorithm incorporated in this framework is easily exchangeable by any other layout algorithm (not even necessarily a force-directed layout method) through a very generic interface in the implementation. Details are discussed in Section 5.5.2.

The force model used in FM³ is summarized in (45). It can be seen as a mixture of *Eades* and *Fruchterman & Reingold* (cp. Figure 34). The repulsive forces are assumed to act inversely proportional to the square of the distance while the attractive force strength f_a is modeled through $\log\left(\frac{d}{l}\right)\cdot d$. As already stated at the end of Section 4.1.3, the tuning of the force model's parameters is important to obtain a good balance between short runtime and high quality of the simulation.

$$
\begin{aligned}
F^{u}_{rep}(v) &= \begin{cases} \frac{1}{\|p_v - p_u\|_2{}^2}\cdot(p_v - p_u) & p_v \neq p_u \\ 0 & \text{otherwise} \end{cases} \\
F^{(u,v)}_{attr}(v) &= \begin{cases} \log\left(\frac{\|p_v - p_u\|_2}{l^{zero}(e)}\right)\cdot\|p_v - p_u\|_2\cdot(p_u - p_v) & p_v \neq p_u \\ 0 & \text{otherwise} \end{cases} \\
F_{rep}(v) &= \sum_{u\in V\setminus v} F^{u}_{rep}(v) \qquad F_{attr}(v) = \sum_{u|(u,v)\in E} F^{(u,v)}_{attr}(v) \\
F_{res}(v) &= \lambda_{rep}\cdot F_{rep}(v) + \lambda_{attr}\cdot F_{attr}(v) \overset{\substack{target\\ force\\ equilibrium}}{\rightarrow} 0 \qquad (45)
\end{aligned}
$$

Remark 50. *For the later experiments, the* FM^3 *option 'qualityVersusSpeed* $\rightarrow$ *qvsBeautifulAndFast' was chosen. This is the 'medium' choice of the three predefined and implemented* ***option sets*** *that are either optimized towards* ***speed*** *or more in the direction of* ***quality****. Among other things, the option controls the number of the already mentioned fine-tuning iterations for the zero-energy lengths in the post-processing of the method (Remark 49).*

Forces are only calculated when two nodes are not in the same position (which is particularly important when dealing with integer coordinates), otherwise the force is neglected. However, the are implemented mechanisms to avoid such situations.

After calculating the *attractive* and *repulsive* forces acting on each node v, the resulting force $F_{res}(v)$ acting on node v is accumulated and the node is consequently moved in this direction.

4.2.1 *Quadtree for approximation of repulsive forces*

To overcome the drawback of Fruchterman & Reingold's rather *static* grid based algorithm to speed up the calculation of the repulsive forces (Section 4.1.3), which is for *average* relatively sparse graphs ***the*** *bottleneck* of the embedding routine, *tree data structures* like the 'reduced bucket quadtree' are likely used for further improvement.

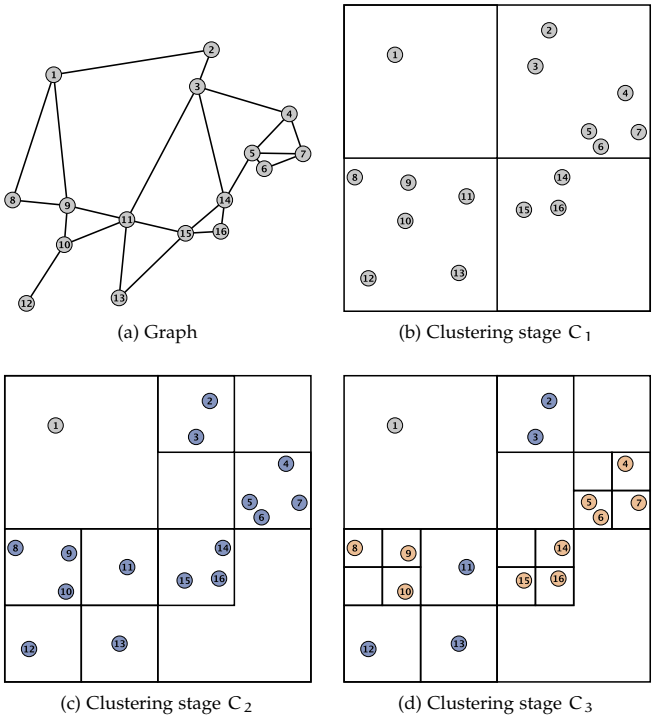
In the following paragraph, the basic idea how to create and how to use this data structure for faster repulsive force calculations is presented. Its application in FM^3 bases on early works of Appel [11] and Greengard [80] in the field of *n-body* simulations. Appel presented a 'gridless' variant to calculate potentials and forces of many-body simulations. Appel's goal was to approximate influencing forces of sets of nodes that are far away from a prospected node v by a representing *monopole* (also called '*center of mass*') instead of simply neglecting them like Fruchterman & Reingold did.

To create a quadtree data structure of a graph $\mathcal{G}$, the canvas of the graph can recursively be split in four (equally sized) sub-cells whereas each cell represents the set of nodes located in the cell. This procedure is repeated recursively for each cell until it contains only a constant (small) number of nodes K, at the extreme just a single one. If $K > 1$, a *leaf* of the tree can consequently contain more than one node. Such a leaf is called a *bucket* and the corresponding quadtree is more precisely called a '*bucket* quadtree' with *bucket capacity* K. Each created cell in the coarsening procedure becomes a node in the quadtree and, as there are at most four non-empty children for a cell by this construction, each node in the quadtree has at most four children. After the i-th of such recursions, the current 'coarsening' of the graph is called the coarsening of the graph to stage C_i and it is represented by level

i of the quadtree. After a number of n recursions, each cell contains at most K nodes and the construction terminates. Thus, the *leaves* of the tree contain the (buckets of) nodes of the graph.

Whenever a cell contains the same nodes than its child or, in other words, whenever a cell has only one child, this stage can be 'skipped' for this cell and the *technically* two cells become one in the tree which is consequently called '*reduced* (bucket) quadtree'. This shrinking of so-called *degenerated* paths in the tree structure has been proposed by Aluru et al. [5, 4]. If a series of such nodes forms a path $(v_1, \ldots, v_p)$, this entire path is shrunk and replaced by a simple edge (v_1, v_p). One example is the bottom right cell on stage C_1 and C_2 in Figure 38 and Figure 39.

Figure 38 shows how this process clusters a graph in three clustering stages with $K = 2$.



(a) Graph

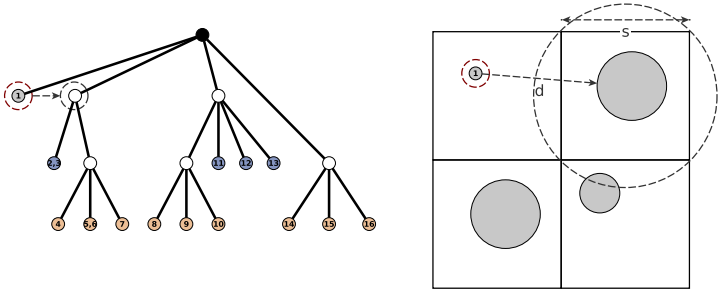
(b) Clustering stage C_1

(c) Clustering stage C_2

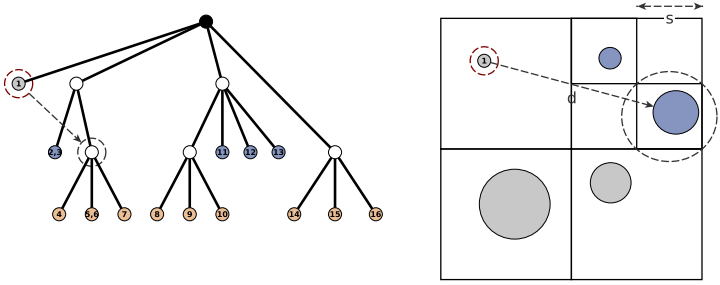
(d) Clustering stage C_3

Figure 38: Construction of reduced bucket quadtree with $K = 2$

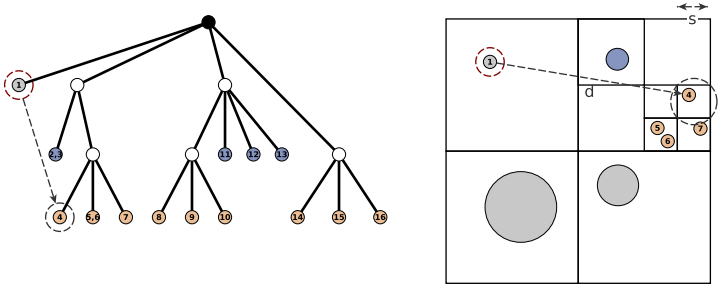
A simple approximation theme using this quadtree to speed up the repulsive force calculation is to represent the repulsive force of each quadtree node by an accumulated *cluster force* located in the barycenter of all nodes in the cell (cp. Figure 39). Consider a node v in the graph and a cloud of nodes that is 'far away' from v.



(a) Coarsening to stage C_1



(b) Coarsening to stage C_2



(c) Coarsening to stage C_3

Figure 39: Approximation through coarsening in the reduced quadtree

Instead of calculating the repulsive forces that act from each of the cloud's nodes onto v, the overall force can be approximated by the *cluster force*. This can reduce the amount of calculations for v drastically if there are many nodes in the cloud. Nevertheless, such an approximation introduces inaccuracies (*errors*) with respect to the underlying force model. Furthermore, the influence of such inaccuracies is larger the larger the acting force of the cloud is. Since the repulsive force in FM^3 is inversely proportional to the square of the distance of two nodes, the influence of the repulsive force and thus the influence of the introduced error decreases quickly with the distance. The question at which distance from the node to approximate forces can clearly not be answered conclusively. The earlier the force is calculated approximately, the more calculations can be saved but the larger is the influence of the error.

The following algorithm from Grama et al. [76] is a relatively simple traversal of the quadtree which approximates the force (acting on v) introduced by a cloud of distanced nodes (cell) only if the ratio of the size of the cell s (horizontal or vertical expansion) and the distance d between the clouds barycenter and v is smaller than a specific *threshold* t (thus, if $\frac{s}{d} < t$). The threshold t can consequently be used to control the accuracy of the approximation and the runtime of the method. Figure 39 depicts the quadtree of the construction from Figure 38 and the three coarsening stages for the approximation.

To constructively calculate all repulsive forces acting on a node v, the algorithm starts at the root of the tree with an initial force $F_{rep}(v) = 0$. Now, all (at most *four* and at least *two*) children $u_1, u_2, \ldots$ are visited with the following rule for the recursion step: if v *is* a leaf in the subtree rooted by the quadtree child node (or the *cell*) u_i, visit all children of u_i. If v *is not* in the subtree rooted by the child node u_i, the following three cases are possible:

- if the child u_i is a leaf, add the repulsive force acting from cell u_i on v to $F_{rep}(v)$,
- if u_i is not a leaf but 'far enough' away ($\frac{s}{d} < t$) from v, add the *approximative* cluster force acting from the cloud in cell u_i on v to $F_{rep}(v)$,
- else visit all children and proceed analogously.

The larger t is, the earlier the algorithm stops while traversing the tree and the fewer calculations have to be performed *but* the larger is also the introduced error. It is, again, a balancing act to choose a 'good' parameter t. The repulsive influence of node 4 on node 1 in Figure 39 is considered more and more accurately the deeper in the quadtree the repelling cluster force (containing node 4) is. While the approximation on coarsening stage 1 (Figure 39a) may be too imprecise (if $\frac{s}{d} \geqslant t$), stage 2 (Figure 39b) could be sufficient to approximate the influences of nodes $4-7$.

Even though this method needs $\mathcal{O}(|V|\log|V|)$ time only for uniform distributions, there are very sophisticated methods that can guarantee this complexity for the worst cases. One of such is applied in the presented and used FM^3 algorithm, all details can be found in the dissertation of Stefan Hachul [81]. The $\mathcal{O}(|V|\log|V|)$ approach using a *reduced bucket quadtree* also bases on the work 'Truly distribution-independent algorithms for the N-body problem' from Aluru et al. [5].

Even though the approximation can speed up the repulsive force calculation by reducing the number of calculated influencing forces per node, the construction of the quadtree introduces a potentially relevant additional overhead. To profit from the use of a quadtree in the explained manner, it is therefore necessary that the number of nodes in the graph is sufficiently large. In Section 4.2.2, the influence of a quadtree approximation on the performance and the outcome for two differently sized example graphs is elucidated by way of example.

Remark 51. *For three-dimensional graphs, an analogue construction leads to an* ***octree*** *representation.*

4.2.2 *Multipole approach for accurate and fast approximation of repulsive forces*

In comparison to *grid* or *mesh* based approaches like P^3M (particle-particle/particle-mesh) [96] for n-body simulations whose efficiency strongly relies on uniform node distributions (just like the grid-based graph drawing approach of Fruchterman & Reingold), Greengard refined Appel's '*gridless*' tree idea from a *monopole* to a *multipole* approximation for influences of distanced *clouds* of nodes in 1997 [80] (see Figure 40). Multipoles can be used to approximate the interaction of charges in a potential field instead of calculating all pairwise interactions.

Remark 52. *The multipole approach is rather complex (****not only in the sense of complex numbers****) and its application is not essential for the presented approach. A monopole approximation would also work and would speed up the calculation of repulsive forces, whereas a multipole approach is more accurate. Furthermore, the repulsive force calculation was not modified for the presented approach in this work. It is used simply due to the fact that it is very well implemented in the favored* FM^3 *algorithm. Thus, the basic idea of multipoles is only explained briefly at this point while more details can be found in the referenced publications.* ***The following basic explanations are mainly based on the works of Greengard [80] and Hachul [82].***

The goal of a multipole approximation is to describe the influence of a large set of charges onto other charges in a potential field by decomposing

this influence into the sum of 'basic' influences, whereas the incorporation of more such basic components increases the accuracy of the approximation. These *basic* components of the interaction in a potential field are *monopoles*, *dipoles*, *quadrupoles*, *octupoles*, etc. The effort to determine the approximation can be kept small by taking only few such components into account. The '*right*' choice of a number of basic components is, again, a question of *weighing up* between *costs* (in terms of time) and *accuracy* of the approximation.

Remark 53. *The idea of a multipole expansion is comparable to the one of a* ***Taylor series****. A Taylor series approximates a function (in the neighborhood of a point* a*) by an infinite sum of polynomial terms (basic components) that are associated with the function's derivatives (part of the coefficients) at this point* a*. Developing the series only to a specific degree results in a* ***Taylor polynomial*** *which can be taken as an approximation of the function at point* a*. The higher the degree, the more accurate is the approximation, or, in other words, the smaller is the introduced error. Periodic functions can, similarly, be modeled by the sum of sines and cosines (or complex exponentials) with appropriate coefficients in a* ***Fourier series****.*

A multipole expansion can be used to approximate the influence of the potential field introduced by a cloud of charges that acts on another *distanced* cloud of charges. To perform the multipole approximation, charges (or nodes) can be identified by points in the complex plane $x_R + i \cdot x_I \in \mathbb{C}$ directly equivalent to their Cartesian coordinates (x_R, x_I). The attribute that two clouds (two sets of nodes $\{x_1, \ldots, x_m\}$ and $\{y_1, \ldots, y_n\}$) are *distanced* is generally formalized by the term '*well-separated*'. According to Greengard, two clouds are called *well-separated* if there exists a radius r so that two circles with centers p_1 and p_2 and radii r exist that contain all nodes x_i and y_i, respectively, and themselves have a distance of at least r to each other.

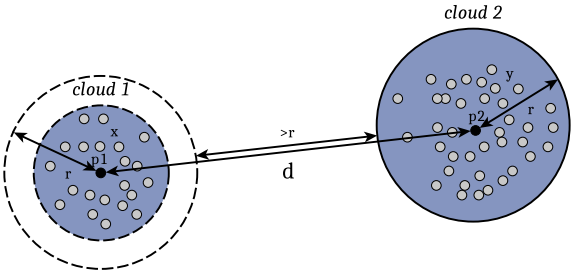


Figure 40: Two 'well-separated' clouds of nodes as $d > 3r$

Formally, the two sets $\{x_i\}$ and $\{y_i\}$ are *well-separated* if there exist points $p_1, p_2 \in \mathbb{C}$ and a radius $r > 0$ such that

$$\begin{aligned} |x_i - p_1| &< r \qquad \forall i \in \{1, \dots, m\} \\ |y_i - p_2| &< r \qquad \forall i \in \{1, \dots, n\} \\ |p_1 - p_2| &> 3r \end{aligned} \tag{46}$$

A tangible approximation of acting forces between the two sets can be derived by the theory of potential and force fields. It bases on the assumption that the electrostatic potential in the free two-dimensional space satisfies Laplace's equation (47) and thus is *harmonic*.

$$\nabla^2 \Phi = \Delta \Phi = \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0 \tag{47}$$

A multipole expansion approximates the distanced potential field and thereby the potential energy $\mathcal{E}(z)$ induced by m particles in the distanced cloud acting on any $z \in \mathbb{C}$ outside the cloud as described in Theorem 3.

Theorem 3 (Multipole Expansion (cp. [80] and [82]))**.** *Suppose that m charges of **strengths** $\{q_i, i = 1, \dots, m\}$ are located at **points** $\{z_i, i = 1, \dots, m\}$ with $|z_i - z_0| < r$. Then for any $z \in \mathbb{C}$ with $|z - z_0| > r$, the potential energy $\mathcal{E}(z)$ induced by the m charges is given by:*

$$\mathcal{E}(z) = Q \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k}$$

with

$$Q = \sum_{i=1}^{m} q_i \quad \text{and} \quad a_k = \sum_{i=1}^{m} \frac{-q_i \cdot (z_i - z_0)^k}{k}$$

Using the *Cauchy-Riemann* equations, the corresponding force $\mathcal{F}$ to that energy $\mathcal{E}$ (with $\mathcal{E}'$ being the derivative of $\mathcal{E}$) can be derived as described in equation (48).

$$\mathcal{F}(z) = (\mathrm{Re}(\mathcal{E}'(z)), -\mathrm{Im}(\mathcal{E}'(z))) \tag{48}$$

Now, the infinite series in the multipole expansion may only be developed up to an index p (cp. Taylor polynomial in Remark 53), whereas the resulting approximation through the truncated Laurent series is called *p-term multipole expansion*. Thus, p becomes a further steering parameter to increase accuracy by including more multipole moments or reduce computation costs by taking only a few into account.

Details on the comprehensive *theory* and on the specific *practice in* FM^3 can, e. g., be found in the dissertations of Greengard [79] and Hachul [81], respectively. The *well-separated* sets can basically be obtained from the quadtree construction presented in the previous section.

By assigning a constant value to p (Hachul supposes $p = 4$ for a sufficiently exact approximation), computing the coefficients of the multipole expansion to get the repulsive force acting from one cloud of m particles onto another cloud of n particles takes $\Theta(m)$ time. The calculation of the derivative of the p-term multipole expansion (which is a simple Laurent series) only needs constant time $\Theta(p)$. Therefore, the acting force can likewise be calculated in $\Theta(m)$ time.

All approximately acting repulsive forces can now be derived by traversing the quadtree up and down and calculating multipole expansions for the nodes of the tree. Hachul presented an algorithm that guarantees a complexity of $\mathcal{O}(|V|\log|V|)$ for the quadtree construction and the calculations, which is independent from the particle positions (related to Aluru's approach [5], all details in Hachul's dissertation [81]).

Apart from this, the worst-case running time of the method only depends on the density of the graph for the attractive force calculations and sums up to $\underbrace{\mathcal{O}(|V|\log|V|)}_{F_{rep}} + \underbrace{\mathcal{O}(|E|)}_{F_{attr}} = \mathcal{O}(|V|\log|V| + |E|)$.

Remark 54. *While the already mentioned* ***medium*** *accuracy option 'qualityVersusSpeed → qvsBeautifulAndFast' (cp. Remark 50) develops the multipole expansion up to* $p = 4$*, the lesser and higher accuracy options set* $p = 2$ *and* $p = 6$*, respectively.*

An experimental comparison

To show the the influence of the quadtree approximation with multipole moments and also the one of the force model in FM^3 (on the runtime and on the resulting graph) and the general runtime behavior of the approaches, a set of benchmarks is presented in the following. Two approaches were applied to generate a force-directed layout starting from an initial *random* placement of the nodes in the plane within a rectangular region.

On the one hand, a layout based on the force model of Fruchterman & Reingold (see Figure 34c) was performed *without (!)* making use of a grid approximation (denoted by **F&R (exact)**). Thus, the repulsive forces are derived by summing up all pairwise repulsive forces between nodes in $\mathcal{O}(|V|^2)$.

On the other hand, the FM^3 method (denoted by **FM³**) with its different force model (see Figure 34d) was applied with $p = 4$ for the multipole expansion approximating repulsive forces through a quadtree in $\mathcal{O}(|V|\log|V|)$.

For both approaches, the attractive forces are computed exactly (*naively*) in $\mathcal{O}(|E|)$ time. The implementations were once again realized in *OGDF*.

To show how the runtime of different parts of the simulation increases with increasing graph sizes, the 'Crack' graph with 10240 nodes and 30380 edges (cp. Figure 31) was taken as a base and was shrunk by deleting outer regions of the graph resulting in $\frac{i}{8}$-th ($i = 1, \ldots, 8$) fractions of the entire graph. The edges of the regions were accordingly deleted so that the edges per node ratio $\frac{|E|}{|V|}$ is pretty much constant for all these fractions of the entire 'Crack' graph. The precise statistics about the resulting benchmark graphs are shown in Table 2 including the achieved overall speedup due to the approximation of repulsive forces. All times report the average time of 10 repeated runs on the machine posed in Section 1.4.

Figure 41b shows the runtime behavior for the different graph sizes and additionally contains the subdivision into time needed for the *repulsive force calculation*, *the attractive force calculation* and also the amount of time spend for *miscellaneous parts*. In addition to the measured times, interpolated curves of the theoretical runtime behaviors are sketched by dotted lines.

The results of the Crack graph show that the approximation scheme speeds up the simulation enormously and that the achieved speedup increases the larger the input is due to the significantly reduced time complexity. For the entire Crack graph, an overall speedup of 21.73 was achieved by computing *approximative repulsive forces* instead of all pairwise exact ones. The amount of time needed to compute the *attractive forces* is (as expected) very small and becomes, due to the lower time complexity, less relevant the larger the graphs are.

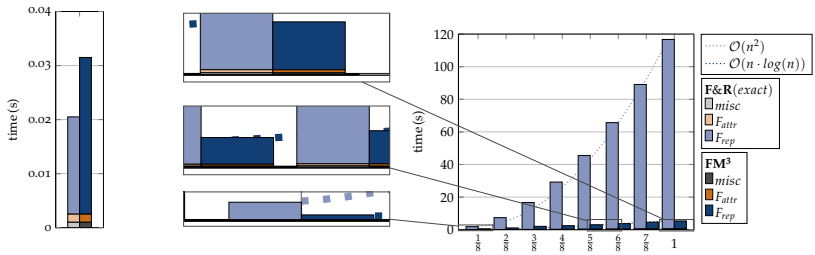
However, the fact that the approximation scheme can play off its advantages for all Crack graph fractions bases on the relatively large sizes of all these input graphs (in terms of number of *nodes*). The picture does change when considering small graphs like the Sierpiński graph S_5. The layout time and its subdivision is shown in Figure 41a. In this case, applying the approximation scheme within FM^3 leads to an increased runtime compared to the naive calculation of all pairwise repulsive forces. This can be explained by

Fraction	$\frac{1}{8}$	$\frac{2}{8}$	$\frac{3}{8}$	$\frac{4}{8}$	$\frac{5}{8}$	$\frac{6}{8}$	$\frac{7}{8}$	1
Nodes $\|V\|$	1280	2560	3840	5120	6400	7680	8960	10240
Edges $\|E\|$	3691	7498	11278	15081	18870	22697	26510	30380
$\|E\|/\|V\|$	2.88	2.93	2.94	2.95	2.95	2.96	2.96	2.97
Speedup	3.30	6.80	8.12	11.90	15.01	17.63	19.24	21.73

Table 2: Graph properties and FM^3 speedups of different 'Crack' fractions

the fact that the quadtree creation and its traversal need additional time and this overhead only pays off if the graph is sufficiently large (cp. Section 4.2.1).

Certainly, there is no constant number that can define this point exactly for every graph and every system. In the default implementation of FM^3, repulsive forces are calculated exactly for any graph with less than $|V| = 175$ nodes. This also means that in the process of coarsening the graph in a multilevel approach like the one introduced in the following chapter, exact calculations of all repulsive forces are performed on coarsened representations of the graph.



(a) S_5 (b) Different fractions of 'Crack' graph

Figure 41: Runtime with approximative repulsive force calculation in FM^3

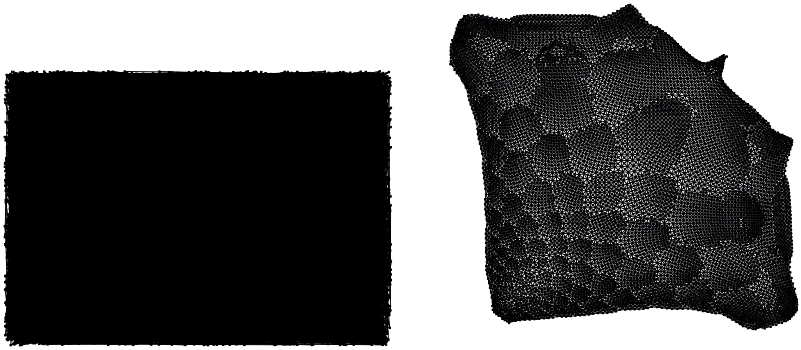
Figure 42 shows the initial random layout of the entire Crack graph and the final FM^3 layout. The random layout offers no visible structure of the graph while the FM^3 layout is, apart from very few exceptions in the outer regions of the graph, *planar* and, referring to all the mentioned aesthetic criteria, *well structured*.

In addition, Appendix A.3 contains some close-up comparisons of the layouts obtained by both benchmarked methods. The results show that the speedup due to the force approximation in FM^3 does not come at the price of an inferior layout to the exact method as the approximation with multipoles is quite accurate. The results also show that the two differing force models that were applied result in comparable layouts.

Remark 55. *Although it is not a primary criterion for this work, it should be noted that the implementation of* FM^3 *only has linear memory requirements.*

4.2.3 *Hierarchical multilevel approach to overcome weak initial placements*

One crucial challenge for force-directed graph layouts, as for other iterative techniques, is the dependence of the methods behavior and its outcome on the initial placement of the nodes. While direct approaches like the one of



(a) Random layout (b) FM^3 layout

Figure 42: 'Crack' graph

Tutte calculate a placement without the need of randomized initial coordinates, they are not comprehensively applicable and also not as customizable as, for example, the spring embedder algorithm is. However, an utterly randomized placement of the initial nodes as it is sketched in Algorithm 5 can lead to extremely long times needed until a stable equilibrium state is reached *or* (or *and*) to a resulting local minimum of low quality. All the good results and runtimes presented of both FM^3 and F&R have been achieved by embedding the discussed algorithm in a *multilevel* framework in OGDF.

The main goal of a multilevel approach in the field of graph drawing is to create a good initial placement of nodes (or groups of nodes) for the layouting phase. Specifically, a commonly used approach is to create (potentially recursively) *coarsened* representations $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ of the graph $\mathcal{G}$ and consequently layout the graphs in the inverse order starting with $\mathcal{G}_n$ while transferring improvements that were made on a coarser representation onto the finer ones. Even when dealing with an efficient layouting approach like FM^3, the layouts of such smaller (coarsened) graph take significantly less time than the layouting of the finer or even the original representation $\mathcal{G}$ (resp. $\mathcal{G}_0$).

A graphically intuitive coarsening strategy has been presented by Walshaw [187]. The idea is to choose the *edges* of a (*good* or even *perfect*) *matching* (see Definition 21) of $\mathcal{G}$ and shrink all the edges to single nodes in a coarsened graph.

Definition 21 (Matching)**.** *A subset of edges* $\tilde{E} \subset E$ *of a graph* $\mathcal{G} = (V, E)$ *is called a* ***matching*** *(in* $\mathcal{G}$*), if for all pairs of edges* $(u_1, v_1) \in \tilde{E}$ *and* $(u_2, v_2) \in \tilde{E}$ *it follows that* $(u_1, v_1) \cap (u_2, v_2) = \emptyset$*.*

Definition 22. *A matching* $\tilde{E}$ *is called* ***maximal*** *if there is no additional edge* $e \in E$ *so that* $\tilde{E}' = \tilde{E} \cup \{e\}$ *still is a matching.*

Definition 23. *A matching* $\tilde{E}$ *is called* ***perfect*** *if every node* $v \in V$ *is part of one edge in the matching.*

By Walshaw's method, shrinking edges of a *perfect* matching in $\mathcal{G}_i$ would half the number of nodes in the coarsened graph $\mathcal{G}_{i+1}$. However, a perfect matching is not present in every graph. Even more, deriving a solution of the maximum cardinality matching problem needs at least $\mathcal{O}(|V|^{2.5})$ time (cp. Papadimitriou and Steiglitz [149]) which is thus not applicable to speed up the $\mathcal{O}(|V|\log|V|)$ process in general. Walshaw therefore proposed the heuristic of Hendrickson & Leland [93] to find a 'good' matching. The heuristic is a periodically applied variant of the well-known Kerninghan-Lin partitioning scheme [109] and its time complexity is linear in the number of edges and nodes.

The shrinking process can be applied iteratively until a coarsened graph $\mathcal{G}_n$ contains only a constant (predefined) number of nodes. Now, $\mathcal{G}_n$ is layouted with the force-directed layout method. Note that the coarsened graphs contain a much smaller number of nodes that the original graph. Thus, this layouting does need only a relatively little amount of time. As soon as $\mathcal{G}_n$ reaches a force equilibrium state, the shrunken nodes of the graph $\mathcal{G}_n$ that represent the matching edges of $\mathcal{G}_{n-1}$ are 'unfolded' again, each to two connected nodes. Now, the same procedure is applied to $\mathcal{G}_{n-1}$, $\mathcal{G}_{n-2}$, etc. until the original graph $\mathcal{G}_0$ has been layouted. The fact that the nodes after the expansion of the matching edges on each level are already near their equilibrium position facilitates that the layout of the refined graphs in the multilevel calculations can be performed in fewer iterations.

The multilevel strategy that is used in this work is the one of Hachul implemented in FM^3. Instead of using matchings for the coarsening, Hachul proposed his '*sun-planet-moon*' model in analogy to galaxies and solar systems. The idea works as follows. The entire graph $\mathcal{G}$ is considered as a *galaxy* partitioned into a set of *solar systems* that each contain a central object called the *sun* of the system. The sun's direct neighbors in the graph are called *planets* and these planets may have neighbors called *moons*. To formalize the property of being neighbors or, more general, the distance of nodes to each other, a metric called the *graph theoretical distance* (Definition 24) is needed. An example for graph-theoretical distances in the graph of Section 4.2.1 is given in Appendix A.4.

Definition 24. *Given a (connected) graph* $\mathcal{G}(V, E)$*, the* ***graph-theoretical*** *distance* $d_{\mathcal{G}}(u, v)$ *between two nodes* $u, v \in V$ *is the number of edges on a shortest path between* u *and* v *in* $\mathcal{G}$*.*

To mark the nodes representing the *suns* in $\mathcal{G}$, a 'working-copy' V' of all nodes V is created. A random node v from V' is picked and its 'twin-node' in V is marked to be a *sun*-node. Now, all nodes u with graph-theoretical distance $d_{\mathcal{G}}(u, v) < 3$ to v are deleted from V'. This procedure is performed until no node is left in V'. After that, the further steps are performed solely with the original set of nodes V.

Now that all sun nodes are marked in this way, the direct neighbors $u \in V$ of each sun node v are marked as *planets* of v's solar system. As all nodes with $d_{\mathcal{G}}(u, v) < 3$ were excluded from the set of potential suns after marking v as a sun, no direct neighbor of v has become a sun afterwards. The planet assignment can therefore be performed without any conflicts. Finally, the remaining nodes are *moons* in the galaxy and have a graph-theoretical distance smaller than *three* to at least one of the suns in the galaxy (*by construction*). Each of such moons is now assigned to a nearest planet and its solar system. The method is formalized in Algorithm 6.

After partitioning a graph $\mathcal{G}_i$ in this way, the coarsened graph $\mathcal{G}_{i+1}$ is created by collapsing all solar systems to a single point which represents this system on the coarser level $(i+1)$. All paths connecting different solar systems in $\mathcal{G}_i$ (*inter solar-system paths*) are represented by respective edges connecting nodes in $\mathcal{G}_{i+1}$. Thus, even several edges or paths between two solar-systems may be collapsed to one. To inherit the zero-energy lengths (which can be very important for this approach) from one level of the graph $\mathcal{G}_i$ to the coarser representation $\mathcal{G}_{i+1}$, each edge (s_k, s_l) in $\mathcal{G}_{i+1}$ between two collapsed solar systems with suns s_k and s_l gets a zero-energy length which is the average zero-energy length of *all paths* between s_k and s_l in $\mathcal{G}_i$. In this context, the zero-energy length of a path is the sum of all edges' zero-energy lengths on the path. By this construction, the general aspirations concerning edge lengths are inherited from one coarsening level to the next.

Remark 56. *As said before,* FM^3 *is also capable of creating drawings with user-defined node sizes. To consider such sizes in the coarsening step, each node on a coarser level* $(i+1)$ *gets a desired node size that is the sum of the node sizes of its ancestors on level* (i).

As already mentioned, the overall process in the *coarsening phase* creates the coarser representations of the graph $\mathcal{G}$ until a graph $\mathcal{G}_n$ with a predefined constant number of nodes is created. This maximally coarsened representation $\mathcal{G}_n$ is subsequently layouted with the force-directed method. After that, all *planets* and *moons* apparent in $\mathcal{G}_{n-1}$ are placed near their respective *suns* and therefore already near their final position. Then, $\mathcal{G}_{n-1}$ is layouted by the force-directed method. This process is repeated until $\mathcal{G}_0 = \mathcal{G}$ is reached. The layouting on the coarse representations can be performed very fast due to a very small number of nodes in the representations. However, the finer the

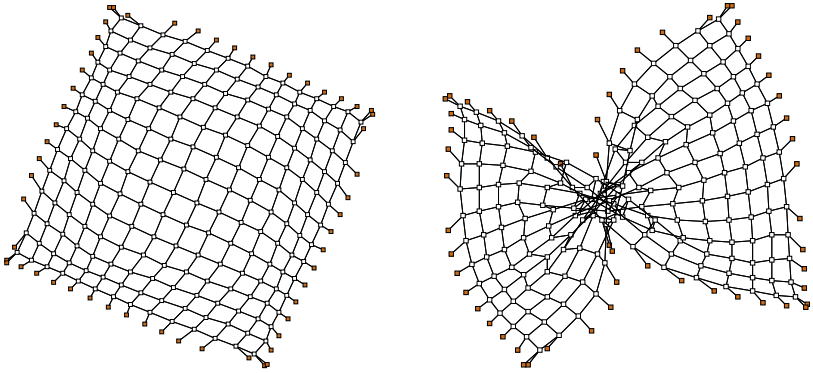
graphs become, the more nodes have to be layouted. Nevertheless, the simulations on the finer levels converge relatively quickly as the inserted nodes are already near their desired equilibrium state position.

Algorithm 6 Galaxy Partitioning (Hachul)

```
procedure GalaxyPartitioning(G(V, E))
    create a copy V' of V
    associate each v ∈ V' with its pendant v ∈ V                    ▷ by ID
    give each v ∈ V a property called 'type'
    for all v ∈ V do                                                ▷ initialize type
        v.type ← FREE
    end for
    while V' ≠ ∅ do                                                 ▷ mark suns
        pick random v ∈ V'
        v.type ← SUN                          ▷ add v to the set of sun nodes in V
        delete v and all nodes u ∈ V' with d_G(u, v) < 3 from V'
    end while
    for all v ∈ V with (v.type == SUN) do                           ▷ mark planets
        for all u ∈ N(v) do                                         ▷ all connected nodes in V
            u.type ← PLANET
            associate u with v's solar system
        end for
    end for
    for all u ∈ V with (u.type == FREE) do                          ▷ mark moons
        find a nearest node u with (u.type == PLANET)
        u.type ← MOON
        associate u with u's (sun's) solar system
    end for
    return G's solar system partitioning
end procedure
```

It is crucial to perform the layout very accurately on the coarser representations of the graph. An inaccuracy that was made on any coarse level of $\mathcal{G}$ *propagates* on the finer levels and therefore affects a larger number of nodes. Such *errors* would create initial layouts on the next level that are local minima and difficult or, accordingly, time consuming to escape. An example is given in Figure 43. While Figure 43a shows how a good and accurate placement has been achieved by this technique, Figure 43b instead shows what can happen if the layout on the coarsest representation has not been performed to the end. Two of the four 'corner nodes' of the coarsest graph were flipped

and, as a result, a large amount of nodes on the finer levels would have to be moved to 'unknot' this situation.



(a) Good multilevel layout (b) Bad multilevel layout

Figure 43: Force-directed layout and local minima

If the number of applied iterations on the finer levels is not sufficiently large to 'correct' this, the layout can end in a weak local optimum. As the calculation of repulsive and attractive forces on the finer levels takes much longer, the process of 'untangling' such a situation on a *fine* level would take very much time. Instead, applying many iterations on the coarser levels is very cheap (in terms of time) due to the small number of nodes and edges. A *rapidly growing* function for the number of iterations on coarsening level i can be applied to obtain a *good* final layout *quality* by avoiding local minima in a *small* amount of *time* due to many cheap iterations on the coarser graphs and (due to good initial placements) decreasingly many on the finer and 'more expensive' graphs.

In contrast to the meta-heuristics presented in Chapter 3, this multilevel strategy takes the *actual structure* of the problem into account to create good initial placements and thereby to avoid weak local optima. It is a much more '*precise*' and *problem-related* technique than, for example, the strong perturbation in the beginning of a simulated annealing approach. This is one of the reasons why a multilevel strategy is used to *improve the quality and the runtime* of the desired chip placement.

Remark 57. *The node degree of the selected **sun**-nodes directly influences the 'amount of coarsening' in the process. The more **planets** and **moons** a solar system contains, the stronger is the reduction when collapsing the system to a single node on the next level. Thus, a strategy for a rapid coarsening could be to sort the sun candidates in*

V' *decreasingly according to their* ***degree*** *(number of adjacent nodes). If the list is sorted in ascending order, a tendentially moderate coarsening can be achieved.*

The overall complexity of the multilevel approach of Hachul (*without the layouting procedures*) is linear in the number of nodes and edges ($\mathcal{O}(|V_i| + |E_i|)$) on each level and, by restricting the construction to a fixed number of maximum levels, $\mathcal{O}(|V| + |E|)$ overall. The same holds true for the memory requirements. The method guarantees that every solar system contains at least 2 nodes and that consequently each coarsening step at least halves the number of nodes $\left(|V_{i+1}| \leqslant \frac{|V_i|}{2}\right)$.

Remark 58. *In the multilevel process, the repulsive forces on very coarse-grained representations may even be calculated exactly for better performance by avoiding the quadtree creation overhead (see Section 4.2.2).*

Remark 59. *To give an additional illustration, Appendix A.5 shows a detailed example of a multilevel layout for a 'chip graph'.*

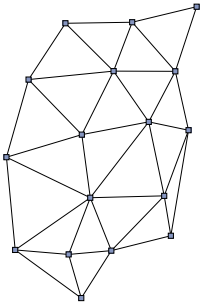
Figure 44 depicts the multilevel layout of the Crack graph with FM^3. Three coarsened levels of the graph ($\mathcal{G}_1$-$\mathcal{G}_3$) are created with the *sun-planet-moon* model. The number of nodes and edges for each level of the graph, as well as the time spent for layouting it, are listed in Table 3. The number of performed iterations of the spring embedder on each level is derived by the (FM^3) rule in Algorithm 7. This rule follows a linear dependence between the level i and the number of iterations on the level with a maximal number of $(\mathrm{ItFac}_{max} \cdot \overline{It})$ iterations on the coarsest graph and a minimal one of $\overline{It}$ iterations on the finest (*original*) graph. The implementation of FM^3 has been applied with default parameters $\overline{It} = 60$ and $\mathrm{ItFac}_{max} = 10$.

The results in Table 3 show that the time needed to layout the *finer* levels of the graph is still relatively short as the number of performed (*and necessary*) iterations is consequently small. The overall time of 5.35 seconds for FM^3 *with multilevel support* contains the time spent for the layouting including all overheads caused by data structure constructions etc. With *deactivated* multilevel approach, FM^3 performed 600 iterations directly on the original graph

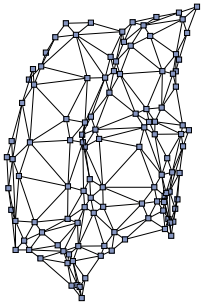
	in level				total	
Level	0	1	2	3	with ML	no ML
$\|V\|$	10240	1025	115	17	10240	10240
$\|E\|$	30380	2954	304	37	30380	30380
iterations	60	240	420	600	1320	600
time (s)	2.39	0.9	0.03	0.00	5.35	28.95

Table 3: Properties of multilevel representations & time spend on the levels

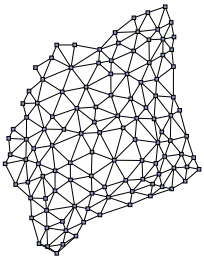
(a) Initial random graph $\mathcal{G}_3$



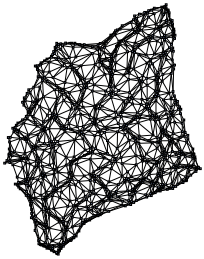
(b) Layouted graph $\mathcal{G}_3$



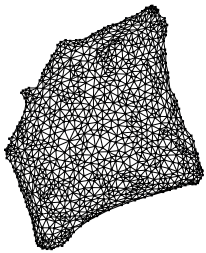
(c) Refined graph $\mathcal{G}_2$



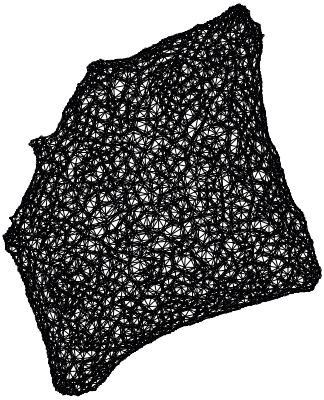
(d) Layouted graph $\mathcal{G}_2$



(e) Refined graph $\mathcal{G}_1$



(f) Layouted graph $\mathcal{G}_1$



(g) Refined graph $\mathcal{G}_0$

(h) Layouted graph $\mathcal{G}_0$

Figure 44: Multilevel steps of a force-directed 'Crack' graph layout

(according to Algorithm 7). *For one thing*, this takes much longer (28.95 seconds) as each iteration operates on all 10240 nodes and 30380 edges of the graph. *For another thing*, it leads to inferior results as the algorithm has not yet reached a force equilibrium. Four different results of such a simulation *without* the multilevel approach are shown in Figure 45.

Algorithm 7 Calculate number of iterations on a level (FM^3)

procedure IterationsOnActLevel(i, $\mathrm{ItFac}_{max}(=10)$, $\overline{\mathrm{It}}(=60)$)
  $i \;\widehat{=}\; actLevel$ ▷ description of variables
  $\mathrm{It}_i \;\widehat{=}\; IterationsOnActLevel$
  $\overline{\mathrm{It}} \;\widehat{=}\; fixedIterations$
  $i_{max} \;\widehat{=}\; maxLevel$
  $\mathrm{ItFac}_{max} \;\widehat{=}\; maxIterFactor$
  if $maxLevel > 0$ **then** ▷ calculate It_i
    $\mathrm{It}_i = \overline{\mathrm{It}} + \left\lfloor \frac{i}{i_{max}} \cdot (\mathrm{ItFac}_{max} - 1) \cdot \overline{\mathrm{It}} \right\rfloor$
  else
    $\mathrm{It}_i = \overline{\mathrm{It}} + (\mathrm{ItFac}_{max} - 1) \cdot \overline{\mathrm{It}}$
  end if
  return It_i
end procedure

Remark 60. *In fact,* FM^3*'s stopping criterion can not only check for the maximum number of (**fixed**) 'commissioned' iterations on each level but can, e. g., also query whether the forces-sum falls below a certain threshold and thus reached a stable state.*

Finally, applying the same force model *without* the multilevel approach until it converges completely would result in significantly longer runtimes. Performing a few such benchmarks on the Crack graph graph showed that the time needed to layout the graph completely *without the multilevel method* (and without stopping until the force model reached an equilibrium state) can even take up to 3 orders of magnitude longer (depending on the initial random situation). The runtime *with the multilevel framework* is pretty constant. However, it is important to note that the coarsening of $\mathcal{G}$ works particularly well and accurately for the Crack graph due to its regular structure (and a therefore very balanced distribution of node degrees, cp. Remark 57).

Remark 61. FM^3 *not only incorporates many mechanisms to produce a layout whose edge lengths match the desired zero-energy lengths but also techniques to similarly assign different desired **node sizes**. For now, this feature is not used but it could be interesting to increase the node size of such nodes that have many connections to other nodes (a high **node degree**). In this way, the **stress** in the region around such heavily 'loaded' nodes could pro-actively be reduced.*

Remark 62. *In the explanations of the previous chapters, $\mathcal{G}$ was assumed to be connected. However, the presented methods and especially* FM^3 *are not restricted to connected graphs. Hachul implemented sophisticated techniques to handle* ***unconnected*** *graphs by layouting all* ***connected components*** *of a graph separately and finally arranging them compactly on a joint canvas. Once again, details can be found in [81]. This fact is pertinent for this work as the handled graphs may be or become unconnected due to some preprocessing in the method (see Section 5.5.4 - Multiple components in the design).*

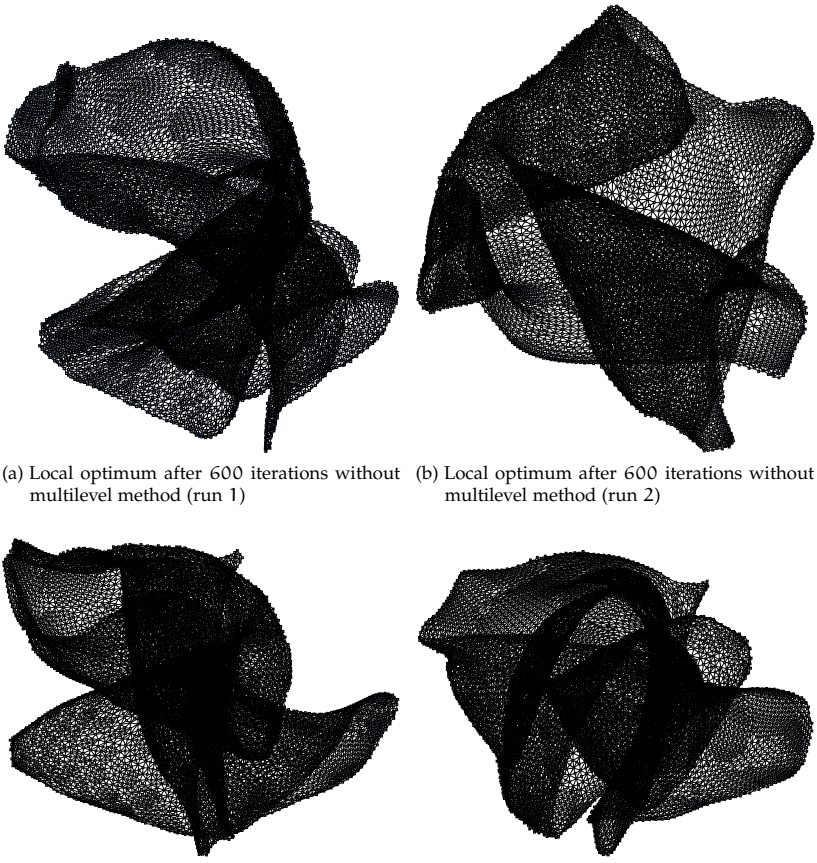
Remark 63. *Regarding the (still) expensive force calculation step, it is important to note that the procedure is potentially highly parallel as all present forces of one iteration can be calculated independently from each other. Thus, a parallelization is easily possible and even architectures like GPUs have a great further speedup potential as shown by Frishman and Tal [66] in general or, specifically for the* FM^3 *algorithm, by Godiyal et al. [66, 74].*

Interim Result 2. *The previous sections presented the main sources of the good performance and the high accuracy of* FM^3*. It is based on a radical* ***reduction of repulsive force calculations*** *for distanced nodes combined with an* ***accurate approximation*** *based on multipoles and a fast multilevel approach which leads to* ***good initial placements*** *of the nodes. The multilevel approach also (mostly) eradicates the dependence of the final solution's quality on the initial arrangement by layouting the coarser graph representations very precisely for 'low costs' (in terms of necessary time).*

4.2.4 *Alternative force-directed layout methods*

There is a large number of available force-directed graph layouting approaches differing not only in the force model but also in their principal technique to obtain a layout. Apart from approaches that are based on the iterative *spring embedder* idea with and without a multilevel mechanism and, for example, using different approximation schemes for the repulsive forces (e. g., *The Grid Variant Algorithm (GVA)* [67], *Graph Drawing with Intelligent Placement (GRIP)* [69, 68], *A Fast Multi-scale Method (FMS)* [88]), there are also several 'direct' approaches (like the one of Tutte based on solving a system of linear equations) using *eigenvectors* and *eigenvalues* of a matrix constructed from the adjacency structure (for example, the *Laplacian matrix*) of the graph (e. g., the *Algebraic Multigrid Method (ACE)* [114], *High-Dimensional Embedding (HDE)* [89]).

Hachul and Jünger published an experimental study in 2007 [84] comparing all these approaches with each other and to FM^3 in terms of time consumption and quality of the outcome. For the quality comparison, the *deviation of edge lengths* in the final layout was measured assuming and configuring



(a) Local optimum after 600 iterations without multilevel method (run 1)

(b) Local optimum after 600 iterations without multilevel method (run 2)

(c) Local optimum after 600 iterations without multilevel method (run 3)

(d) Local optimum after 600 iterations without multilevel method (run 4)

Figure 45: Force-directed layouts without multilevel method

that uniform edge lengths are desired. Anyway, many available implementations only support uniform lengths, some due to their implementation and some due to the methods' general characteristics.

Even though the experiments showed that direct approaches can be even faster than FM^3, concerning *edge length accuracy* (which can be of major importance for the presented framework depending on its specific configuration and usage) FM^3 has clearly shown to be the best algorithm in the test field while the direct approaches tendentially fail here.

Furthermore, FM^3 layouts tend to produce a relatively *small number of edge crossings* in the drawing (also favorable for the developed framework).

Finally, FM^3 was absolutely stable in generating drawings for all benchmark graphs while several other implementations failed for some of them because of memory restrictions or simply due to failures in the executables. Due to the absence of the source code for some of the algorithms, the source of errors could often not even be detected.

To conclude, FM^3 is a very good choice for the intended ideas of the framework and a good basis for further developments. An integration of some of the mentioned (very fast) alternatives is nonetheless possible and planned. Even the closed-source versions can be integrated due to the very generic and unspecific interface of the presented framework (see Section 5.5.2).

FURTHER IMPROVEMENTS LIKE WSPD Applying the *FastMultipoleMultilevelEmbedder* approach developed and implemented by Gronemann in OGDF will be an interesting attempt for the future. Compared to Hachul's FM^3 implementation, it includes a different quadtree space partitioning and a *well separated pair decomposition (WSPD)*. Combined with an approximation of repulsive forces by 'simple' monopoles, the approach can perform an order of magnitude faster than FM^3 (or even more). However, to apply it directly in the presented framework, the recognition of zero-energy lengths and the mentioned post-processing for these should be added. In general, the layouting process can be speeded up or refined by further *parameter tuning* or other *force* and *force approximation* models. The implementation of FM^3 is used for the framework as it provides many useful mechanisms and a very good balance between speed and accuracy. However, the presented framework provides a flexible structure to exchange the graph layouting procedure (cp. Section 5.5.2).

4.3 FROM VLSI PLACEMENT TO GRAPH DRAWING AND BACK

The works of Eades, Fruchterman & Reingold (and consequently all the ones that have been presented and build upon these) base on the early work '*A force-directed component placement procedure for printed circuit boards*' of Quinn and Breuer from 1979 [152]. Their general idea of using force-directed placement methods for *application-specific integrated circuits'* (ASICs') designs will again be addressed in Section 5.1. The approach presented in this work is intended to, in a sense, bring progress that was made in the field of graph drawing back to a discipline from the VLSI domain, along with further *extensions* and *adaptions*.

4.3.1 *Force-directed graph layouts for FPGAs placement*

Just like for general VLSI placements of ASICs, there are several approaches using force-directed methods for FPGA placement procedures (see, for example, Section 5.1). This work aims at providing a framework *for my own and for further research* in this field bridging the gap between the rather technical scope of *Logic synthesis* and the 'visually interpretable and applicable' field of *graph drawing*.

A novel approach, which performs, among other things, iterative adjustments of the layout and which offers different objectives to optimize for, has been developed. It is presented in the following chapters. In addition, it is compared to, and partially combined with, a long-term tuned established simulated annealing approach.

Besides the fact that the force-directed layouts meet several already mentioned aesthetic criteria, the drawings seem *tidied* and do heuristically minimize the overall edge lengths by attractive forces while avoiding 'hugely stressed' regions by repulsive forces. The results maintain a good balance between these two goals.

Figure 43a on page 121 shows how well the force-directed layout meets the desired requirements by being principally near-optimal (cp. Section 3.2) neglecting the facts that a) the graph is *not embedded on an integer grid* and that it is b) *not oriented (rotated)* as its shape does not necessarily follow the axes of the surrounding drawing frame (canvas). Another interesting fact of such a force-directed layout is that the I/O pins (marked orange) are 'automatically' placed on the outer frame of the grid graph (cp. instead the local optimum of the basically very good, albeit slow, *ITS* approach in Figure 26c of Section 3.2.8). This is based on the fact that all I/O pins in this synthetical example only have one connection to inner elements of the chip and thereby only have one edge keeping the node near its neighbor while all repulsive forces carry such nodes outwardly. Even if an I/O node is connected to multiple inner nodes of the chip, this I/O node would still be placed on a position balancing the distances to all connected nodes. Section 5.5.4 explains how this fact is used to efficiently place the I/O nodes obtained from a force-directed layout on the outer I/O frame of the (*island-style*) FPGA.

The following list summarizes the motivations why force-directed graph drawings were used for the practical FPGA placement routine.

The **force-directed drawings** of FM^3

- tend to create **symmetrical**, '**tidied**', **crossing-reduced** and '**balanced**' drawings with **controllable** edge lengths and
- **without wasting** space that naturally **match** common (especially *island-style*) FPGA architectures,
- while heuristically **minimizing the overall edge lengths** through attractive forces and **avoiding 'heavily stressed' regions** by repulsive forces,
- **fast** and **accurately** in a **stable** and **extendable** framework.

Part IV

How can this be transferred to FPGAs?

This chapter presents the core of the **FieldPlacer** *method to embed 'the graph onto the grid'. The force-directed graph layout is realized with no restrictions concerning the coordinates, neither their data type nor their range. The* `FieldPlacer` *takes such a layouted graph as its input and assigns all the different elements structure-preservingly on the FPGA grid architecture. Therefore, the 'inner'* ***logic units*** *are distributed by sorting them according to their vertical and horizontal coordinates in the graph layout respecting a previously defined distribution strategy. The surrounding* ***input and output pads*** *are distributed with regard to their angle to the barycenter of the graph. To complete the first basic embedding of the graph to the grid, special and rare units like* ***memory*** *or* ***multiplicator*** *elements are finally placed nearest to the barycenter of their connected and already placed elements on the chip. In addition to the* `basic FieldPlacer` *method, a subsequent local refinement of the placement in form of a local search can be performed. This last step can be interpreted as a 'cold annealing' and is implemented based on VPR's simulated annealing method with temperature* 0*. This post-processing iterates until it (approximately) reaches the nearest local optimum. It is shown, how this local refinement can additionally improve the quality of the placement in terms of several FPGA related quality norms.*

The entire process can roughly be summarized as follows:

1. *setup a graph with the design's netlist (representation of connections between blocks)*
2. *generate a force-directed layout*
3. *embed this layout to the architecture's discrete grid*
4. *perform some additional improvements of the assignment*
5. *refine this assignment with a concluding local refinement (through a local search) directly on the grid*

5

Architecture-aware field embedder for FPGAs

"Don't Panic"
— The Hitchhiker's Guide to the Galaxy —

Contents

5.1 ESTABLISHED CHIP PLACEMENT TECHNIQUES

5.1.1 *FPGA placement*

As a part of the compilation flow for FPGAs, the *placement* step has already been introduced in Section 2.4. The task is to assign logic blocks to suitable positions on the chip architecture while optimizing a cost function, for example, the overall wirelength on the chip. The following paragraphs introduce the main techniques used in FPGA placement approaches today to finally position the approach presented in this work in Section 5.2. Later in this Chapter and in Chapter 6.4, the achieved results are compared to the widely referenced *simulated annealing* approach implemented in *VPR*.

Simulated annealing-based placement

Simulated annealing (SA) is generally regarded as the most popular technique to perform placements for FPGAs (see, for example, Ludwin and Betz [129]). The method itself has already been introduced and discussed in Section 3.2.7. One great advantage compared to other algorithms is that a *basic* SA approach is relatively simple to implement while creating results of relatively high quality if its main defining components (like the *temperature schedule* and the *cost function*) are configured appropriately. VPR's placement [19, 21] is based on a temperature schedule with *exponential cooling* (cp. Section 3.2.7). In contrast to the 'classical' approach, the temperature in VPR's placement is not statically reduced but follows an adaptive scheduling scheme $t_{i+1} = \alpha \cdot t_i$ with α depending on the number of previously accepted moves (in the iteration with temperature t_i). The actual update function for α is depicted in Table 4. Following the work of Swartz and Sechen [175] from 1990, the default setting applies an initial temperature of $t_0 = 10 \cdot (N_{\text{blocks}})^{1.33}$ (therefore depending on the total number of available blocks N_{blocks} of all types), whereas this value can be varied to influence the time needed for the placement and the consequently achieved quality (cp. Remark 16). However, the

Fraction of accepted moves (R^{accept})			α
	R^{accept}	$\leqslant 0.15$	0.8
$0.15 <$	R^{accept}	$\leqslant 0.8$	0.95
$0.8 <$	R^{accept}	$\leqslant 0.96$	0.9
$0.96 <$	R^{accept}		0.5

Table 4: VPR's temperature update schedule

default value of t_0 and also the temperature update schedule have been tuned intensively to achieve very good results. Two years ahead of the publication of Swartz and Sechen, Lam and Delosme [120] have presented a new annealing schedule and benchmarked it for its application in *standard cell placements* (ASICs). With reference to both mentioned works, VPR's approach tends to keep the R^{accept} value near 0.44, which is achieved by varying the *frame*-size (D^{limit}) out of which pairs of elements are taken to check whether they should be swapped or not (see Section 2.4 [VPR Placer]). The D^{limit} value is updated through equation (49).

$$D_{i+1}^{limit} = D_i^{limit} \cdot \left(1 - 0.44 + R_i^{accept}\right) \tag{49}$$

The annealing is finally terminated when the temperature falls below a certain threshold ($t_\omega = 0.005 \cdot \frac{cost}{N^{nets}}$). This threshold depends on the *cost function*, more precisely, on the relative cost per *net* in the design.

Remark 64. *Lam and Delosme [120] have additionally shown how their simulated annealing approach could successfully be applied to solve TSPs. The generality of simulated annealing is one of its core advantages compared to specialized approaches.*

As already discussed in Section 3.2.7, SA also has the great advantage of being rather independent from the initial solution. This fact is important as VPR starts with a random initial assignment of all blocks.

Remark 65. *In the later experiments, those and other* ***default values*** *are used (see Appendix A.6). The predefined parameters provide a good basis for acceptable runtime while reaching good resulting layouts. In addition, the results are thereby comparable to other published works.*

For example, the *Quartus II placement algorithm* (P2Q), which is incorporated in Altera's design tools, uses simulated annealing as it is actually based on an early version (v4.30) of *VPR*. Ludwin and Betz [129] showed how this procedure can be parallelized.

Analytical (numerical) placement

While Xilinx's *ISE* software suite (like most of the tools available in the 1990s) also used to apply simulated annealing, they switched to an *analytic placement* approach in 2012 with the release of *Vivado* [60] in order to overcome the challenges that emerged from smaller production generations and a consequently extremely increased number of logic blocks per chip. Analytic approaches model the problem of placement in systems of *mathematical equations* and subsequently solve these to obtain a solution in form of coordinates for the blocks. In contrast to the simulated annealing approaches, which *inherently operate on the integer grid* with valid locations for *each block* at *each time* in

the process, a solution vector of the equation system may contain arbitrary *real* coordinates. Generally, restricting the components of the solution vector to *distinct integer* coordinates while occupying solely a *predefined area* would make the problem in fact a *QAP* and, thus, not efficiently (*exactly*) solvable with today's hardware and solution methods (see Chapter 3). Due to this fact, analytic placers generally operate in two phases: Firstly, an *arbitrary arrangement* is calculated and then, secondly, the achieved arrangement is *projected* onto the integer grid by, for example, *equally* distributing them with respect to the calculated relative assignment. This second step is often called *slot assignment* or, sometimes, *legalization*. The setup of the mathematical system of equations and also the *slot assignment* step can both be used to influence the desired optimization goal (cp. Section 5.5.3).

The basis of most analytic placers is the attempt to *minimize the quadratic wirelength* between connected blocks (with respect to the *Euclidean distance*). Assuming that a connection-matrix $\mathcal{V}$ of the design is given (like in Section 3.1.1), the quadratic distances d_{ij}^2 between connected blocks b_i and b_j can be added up as described by equation (50).

$$\sum_{(b_i,b_j)} d_{ij}^2 = \sum_{(b_i,b_j)} v_{ij} \cdot \Big((x(b_i) - x(b_j))^2 + (y(b_i) - y(b_j))^2 \Big) \tag{50}$$

In addition, the connections could be *weighted* to target further optimization goals, e. g., by their *criticality* for timing-driven placement.

Section 5.1.2 provides a further comparison of different *analytic* approaches to the one presented in this work. As the equation systems, in general, model and minimize the wirelength between the logic blocks and are thereby realizing a *force-directed* approach with *wires* representing the *springs*, the method presented in this work can be classified as an *analytic* placement procedure, even though it does not solve an equation system by a linear equation solver. The mathematical system in this work is the *force model* of the spring embedder (see Section 4.2).

For example, Xilinx's Vivado Design Suite models equations to minimize a given cost function. *Multiple possible optimization targets* can be incorporated in its models such as the *timing*, *wirelength* and *congestion metrics* for both *timing-driven* and *routability-driven* placements. Due to its generally rising importance in the world of computing and also due to energy-critical application environments, even the *power*-consumption can be taken into account.

Partitioning-based (Min-Cut) placement

Partitioning-based (or *Min-Cut*) placement approaches divide the chip recursively into *subregions* (e. g., by horizontal and vertical *bisection*) and assign *subsets of nodes* (or *subnets* of the overall design) to such regions. The classical

optimization target is to minimize the *cut* between the partitions, thus, the number of edges *connecting* the partitions *with each other*.

Partitioning-based approaches often use *iterative improvement* techniques (see Section 3.2) to improve the current layout while starting with a random assignment just like the classical simulated annealing placers. The *Kernighan-Lin-Algorithm* (see Section 3.1.7) can be applied to iteratively check whether pairwise swaps of blocks between partitions improve the layout or not until no more improving swap is available (see, for example, Udar and Sharma [182]).

The main idea is that densely connected parts (*net-parts*) of the circuit are thereby 'packed closely together' into a common region of the chip in order to minimize the overall wirelength. Performing this approach recursively then creates improved assignments of nodes to partitions, *from a coarse-grained* level *to a fine-grained* one.

Selvakkumaran et al. [165] presented a partitioning-based placement method for *heterogeneous* FPGAs and Maidee et al. [131] published an approach of this class specifically targeting a *routing-aware* partitioning.

Remark 66. *In addition, an extensive comparison of placement (and routing) techniques for FPGAs can be found in the dissertation of Tessier [179] from* 1999*.*

Remark 67. *In* 2000*, Haldar et al. [87] compared different approaches for parallel FPGA placement which were also benchmarked within the VPR framework. They were able to achieve some speedups paired with high quality results with a partitioning-based method, while parallelizing a simulated annealing method was not successful due to many synchronization barriers. Their negative influences on the method's runtime dominated the parallel speedup, leading to no improvement in the runtime by parallel processing. They also presented two versions (**synchronous** and **asynchronous**) of a parallel placement method based on Markov chains with a **near-linear** speedup.*

Comparison to general chip placement

There are obvious similarities between the placement for FPGAs and the placement for classical (*hardwired*) computer chips like *ASICs*. Both disciplines call for *short connections* and in both disciplines, the available environment (*chip area*) has to be considered. However, an important difference is that FPGA placement is, in a sense, *discrete* and that the area of the logic blocks is negligible due to the *predefined* available and identical *slots* on the FPGA architecture. A placement for a hardwired chip instead has to take further properties like the *size* of the logic elements into account for the placement in order to *avoid overlapping*. Furthermore, the positions of the chip-elements are generally *not* (or *not utterly*) restricted to predefined slots.

However, *whitespace management* between blocks on the general chip that maintains routability between the blocks is a technique from general chip placement (e. g., for ASICs) that is related to routability-driven placement for FPGAs. Two well-known frameworks for such *general chip placements* are NTUplace3 [35], an *analytical* placer, or Kraftwerk2 [172], also a *force-directed* approach. There are various other approaches in the field of chip placement that, for example, apply *evolution-based* methods like the one of Kureichik et al. [118].

Chang et al. [33] compared the three main classes of placers *for general chips (e. g., ASICs*). In terms of FPGAs, they came to the conclusion that **simulated annealing** generates placements of *good quality* for *small* designs while it is easily possible to *consider multiple objectives* simultaneously but with a *relatively high runtime* for larger circuits. Instead, **Min-Cut** placement was classified to be more efficient and scalable for larger circuits while it is more difficult to handle *multiple objectives* simultaneously. Finally, **analytical placement** was appraised to be *efficient and scalable* for *large circuits* with results of *high quality* while being able to relatively easily handle *multiple objectives*.

5.1.2 *Related placement methods*

The following section discusses works that are closely related to the presented approach. It is pointed out where ideas from other published approaches are *similar* to the presented method and also how they *differ* from one another.

General chip placement

Even though this work is about *FPGA* placement, basic ideas, like the application of a force-directed model in order to minimize wirelengths, come from the general field of chip placement, e. g., for ASICs (cp. Quinn and Breuer from 1979 [152] and Section 4.3). Thus, such basic related works should at first be named. *GORDIAN* [112] is a widely used framework for VLSI placement from 1991 which combines an *analytical approach* based on quadratic programming with a *partitioning technique* and a final *slicing* optimization based on the work of Dunlop and Kernighan from 1985 [49]. **The application of a *force model* and basic features of the *slicing* technique are similar to some ideas of the presented work.** The main idea of their approach is to model the *wires* between connected *modules* (blocks) *as springs* in a *force model* so that such nodes attract each other. Minimizing the overall spring forces in the system consequently results in minimizing the overall wirelength in the system (cp. equation (50)). The authors of *GORDIAN* (Kleinhans et al.) set up *linearly constrained quadratic programming problems* (LQPs) for multiple

levels l in the form of equation (51) and for both the x and the y coordinates separately.

$$\min_{x \in \mathbb{R}^m} \left\{ \phi(x) = \frac{1}{2} x^T C x + d^T x \;\middle|\; A^{(l)} x = u^{(l)} \right\} \tag{51}$$

This system contains a *quadratic objective function* $\phi(x)$ modeling the *wirelengths* (or *spring-forces*) in the system and a *linear* system of constraints to consider not only the initial *free (movable) or fixed* nodes but also inserted *fixed* 'dummy' nodes.

On a level l, the overall (*root*) region has already been partitioned by bisection l times. The algorithm starts on level $l = 0$ by optimizing the system with respect to the *entire chip* region and *all* modules while assuming that all the surrounding *(I/O) pads* on the chip have *fixed coordinates*. In that case, the matrix C is *positive definite* if only every module is (*directly* or *indirectly*) connected to such fixed pads. This again is a reasonable assumption for usable designs, because a part of the logic without any connection to the outer regions of the chip would not be useful at all. If there was a node with no connections to pads, it would have no *fixed* reference points in whose center it should be placed (see Section 4.1.1). In the extreme case that *all* nodes should be treated as *movable* nodes, such a system with only attractive forces would collapse to a single point with an overall wirelength of zero. This is similar to the situation in *Tutte's approach* for 3-vertex-connected planar graphs from Section 4.1.1.

However, **fixing** the arrangement of the **outer pads** can influence the placement considerably (cp. Figure 30) and may, especially in the prototyping phase of a design, **often not be necessary**. A free (*unconstrained*) assignment of the pads, like it is applied in the presented approach in this work, can (**if desired**) indeed lead to remarkably better overall results (see Betz et al. [21, Section 5.4.3]).

As many analytical placement approaches ground on the same basic quadratic programming construction that GORDIAN also applies, this construction should now briefly be explained together with some GORDIAN-specific additions.

The *force system of the springs* or, analogously, the *system of wirelengths* from equation (50) can be modeled in the general form of the function $\phi(x)$ in equation (51). With a *positive definite* system matrix C (and therefore a convex objective function $\phi(x)$) and together with the *linear equality constraints*, the minimization problem can be solved by minimizing the *combined (transformed)* target function $\psi(x_i)$ in equation (52) which incorporates the linear system and $\phi(x)$.

$$\min_{x_i \in \mathbb{R}^{m-q}} \left\{ \psi(x_i) = \frac{1}{2} x_i^T Z^T C Z x_i + c^T x_i \right\} \tag{52}$$

In other words, the linear system is transformed and substituted into $\phi(x)$ to form $\psi(x_i)$, which now additionally contains information about inserted partition-central fixed nodes that attract all nodes of the respective partition for a more even distribution.

Since C is positive definite and since the columns of the transformation matrix Z form a basis of the new *search space* for the optimization (an $(m-q)$-dimensional subspace of $\mathbb{R}^m$), the new system matrix $[Z^T C Z]$ of $\psi(x_i)$ is likewise positive definite. Thus, the (*unique*) optimal solution for equation (52) can be obtained by equating the gradient of $\psi(x_i)$ to zero and solving this system ($\nabla\psi(x_i) = 0$) in equation (53) efficiently (*like it is implemented in GORDIAN*) with a *conjugate-gradient*-solver and *sparse-matrix* datastructures.

$$Z^T C Z x_i^* = -c \tag{53}$$

One main problem of such systems incorporating *only attractive forces* and springs with *zero-energy lengths of zero* (see Section 4.1.2) is that obtained optimal solutions use to **waste a lot of space**, a fact that has already been shown for the related approach of *Tutte* in Section 4.1.1 (except for rather regular inputs such as the 3-vertex-planar *Crack* graph in Section 4.1.1). Optimizing the *root* system in GORDIAN with *fixed* surrounding pads generally leads to intensive overlays in the densely packed center region of the chip while there occur larger whitespace areas in the outer regions (see the publication of Kleinhans et al. [112], the same effect is discussed in the presented method in Section 5.6.1). The resulting overlays are certainly not allowed for the 2-dimensional placement problem of general chips but they play an *important role even for FPGA placements* using this quadratic optimization technique with the mentioned force model. On the other hand, incorporating the *repulsive forces* between all node pairs would at least *diminish the sparsity of the system* (equivalently to inserting inversely acting springs between all nodes). Consequently, the solver time would generally be increased remarkably.

Remark 68. *In contrast to the approach of GORDIAN, the presented model in this work* ***does not use such numerical techniques*** *and is able to optimize designs where* ***all*** *nodes are* ***free*** *(if desired, not necessarily). To overcome the problem of extreme overlapping (or* ***space-wasting****), an extended force model incorporating* ***repulsive forces*** *is used (the one of* FM^3*) and new chip distribution methods targeting FPGAs have been developed.*

To get a better (*more regular*) distribution on the entire chip, the optimization process in GORDIAN is iteratively repeated through a recursive partitioning of the *root* region, realized by successive bisection in *horizontal* and *vertical* direction. All created subregions on level l get a respective subset of the modules from level $(l-1)$ and each partition's subsystem additionally obtains a so called *center of gravity* which attracts all nodes of this partition.

While other approaches optimize each partition separately fixing the nodes of all other partitions, the GORDIAN approach behaves more globally with these centers of gravity. However, for such other approaches, the order of optimizing the partitions also influences the result. Illustrations of the entire process and a discussion about different partitioning schemes can be found in the main publication about GORDIAN [112, Chapter IV] as the partitioning itself also influences the final results considerably. By *including repulsive forces*, this work aims at a direct global optimization without any such iterative partitioning-based postprocessing steps.

Another technique that is used in GORDIANs approach is the *Standard Cell Final Placement* [112, Section 5.1]. Modules on a general chip (e. g., an ASIC) may have different heights and, more often, very different widths. To achieve horizontal rows of modules that are approximately of the same length, the elements in the set of modules M obtained from the layout after partitioning are first of all sorted by their *vertical* coordinate. Now, this list is divided by $(r-1)$ horizontal cuts to generate r *pairwise disjoint subsets* (*rows*) of modules M_r with $\bigcup_{i\in\{1,\dots,r\}} M_i = M$ and with approximately equal accumulated widths. Thus, the rows may contain different numbers of modules but, by the preceding sorting of the y-coordinates, it is ensured that the vertical coordinates of modules in consecutive rows are always increasing $\big(y(m \in M_i) < y(n \in M_j) \text{ iff } i < j\big)$. The modules within each row can be ordered horizontally by their x-coordinates obtained from the placement after quadratic optimization and partitioning. The **CLB assignment** introduced in Section 5.5.3 basically applies a **similar** approach, though with different objectives as it is a method for FPGAs and not for general chips. The authors of GORDIAN already stated in the mentioned publication that this procedure tries to change the available global placement after partitioning *as little as possible*. A further publication by Sigl et al. [168] from 1991 showed that using a *linear* objective function for the optimization in GORDIAN (or at least some linear assumptions to better match the *Manhattan* distances on the routing architecture) can improve the placement compared to the classical quadratic objective function. The linear function was in fact applied in order to measure the distances between modules in the constructed rows. The **general** use of **different norms** is one **main feature** of the presented method in this work and is discussed in Section 5.6.2. While an integration of the *not everywhere differentiable Manhattan distance* introduces several difficulties for the *direct numerical quadratic programming approaches*, the usage of different norms in the presented *iterative* approach is directly possible.

Another *force-directed* approach for general chips which **includes repulsive forces** (due to the already mentioned drawbacks of a simple model without them) was published by Vorwerk et al. [186]. Referring to the just mentioned

publication of Sigl et al., they also use a *linear* objective function. However, they **fix the I/O pads** in their model for general chips and apply a **numerical solver** to find an optimum for the system. Like in FM^3 (and therefore as in this work), the authors use a **quadtree** for approximations of the repulsive forces and **multipoles** for high accuracy. In addition, the optimization of the circuit is iteratively repeated while the forces in their model are **dynamically adapted** to reduce overlapping in dense regions. Even though this method is directly targeted towards tackling challenges in *general (ASIC) designs* (which are not present in FPGA placement), the idea of **dynamically adjusting properties** of the model in repeated optimization runs is **similar** to the *slack graph morphing* of this work (see Section 6.3).

Finally, the work of Chan et al. [30] applies a *force-directed multilevel technique* to create general chip placements using a relatively complex objective function (see the patent of Naylor et al. [144]). This function considers the semi-perimeter wirelength approximation (see Section 2.4) in a *differentiable* way (to be suitable for their numerical solver) instead of a linear or quadratic function what consequently led to better results compared to many other placers from this field.

FPGA chip placement

Motivated by the results of general chip placement methods, *analytical* approaches also arose as FPGA placement routines. In Section 5.1, it has already been mentioned that Xilinx switched to an analytic approach because simulated annealing tends to need too much time for the *ever rising* amount of resources an today's FPGAs. While overlapping plays an important role in general two-dimensional placement for general chips (also called *floorplanning*), it is important for the discrete FPGA architecture in the sense that one available slot can only be occupied by one block of the design. Overlapping in the FPGA placement therefore represents situations where multiple blocks are potentially assigned to exactly the same position on the chip. While *simulated annealing* avoids such situations *by construction* of the algorithm itself, a solution of an analytical placer (as introduced in the previous section) does not restrict different elements to different integer locations. Thus, analytical placements obtained from solvers like the already mentioned one in GORDIAN have to find appropriate slots on the FPGA. Generally speaking, such approaches (**just like the one presented in this work**) look for a unique location for each element which is possibly near the calculated position. This assignment is often followed by a **local iterative improvement**, like the *local refinement* in this work (see Section 5.6.3).

In the year 2013 and thus **in parallel to the development in this work**, Lin et al. [127] published an analytical placer for FPGAs and compared

their approach to several other analytical ones that were available. In contrast to all but one of their 'competitors' (namely *FastPlace* [185]), they used a **multilevel approach** (*like this work does*) and applied the norm of Naylor et al. [144], just like Chan et al. [30] did for general chips. They also incorporated information about the **slack** into their model (*similar to this work*) to create an approach that is both **timing-driven** and **wirelength-driven** (*like the presented one can be*). Additionally, they performed a **low temperature simulated annealing** at the end of their procedure for detailed refinement (which allows non-improving swaps, see Section 3.2.7). Instead, the presented approach in this work optionally performs a local refinement after the main force-directed method, which only is a *quasi-simulated-annealing with temperature zero* (in fact a *local search*, see Section 3.2.4). Moreover, Lin et al. **consider only homogeneous FPGAs** made of *CLB*s and **fixed I/Os** to meet the requirements of their **numerical solver** instead of the (optionally) **totally free spring embedder-based** approach of this work. The overall technique proposed in their publication therefore has **some similar rudiments but differs in several main and fundamental points** (many of such, which are new ideas of this work, have not even yet been named, see Chapters 5 and 6). Nevertheless, Lin et al. thereby also showed that (at least for homogeneous FPGAs) such a *force-directed approach can be very beneficial* compared to other state of the art techniques, and this is a good foundation for this work. Mak and Li [133] considered **homogeneous** FPGAs with only *CLBs* and *I/Os* and, in addition to their general force-directed placer approach, they address a special (constrained) I/O placement problem in more detail. For the case that different I/Os need different voltages and assuming that groups of I/O pads form so called *banks* which are all served by the same voltage (like on Altera's Stratix devices), they set up and solve an *integer linear program* to assign each I/O pin to its optimal legal position in terms of an adjustable cost function. In their force-directed approach, they iterate over differently detailed representation levels of the design and they include **repulsive forces**. A *simulated annealing method* is applied on the coarsest representation (due to its small number of elements) while force-directed methods are used on the finer representations. Finally, the obtained coordinates are simply converted into integer coordinates. If more than one logic block is assigned to the same location, they place '*the outstanding ones into nearby available CLBs*' without going into details *how* this is specifically implemented. In fact, especially for FPGA designs that use almost the entire available chip, such a technique can have *great disadvantages compared to the CLB distribution method presented in this work* (see Section 5.5.3) as a *reassigned* CLB can potentially be placed very far away from its initially desired position. As a result, very dense regions could be spread widely over the FPGA. In their cost functions, they use the **Manhattan norm** (like the approach of this work can)

and in contrast to all the other mentioned approaches, this one uses a kind of a **spring embedder** method (instead of numerical solvers) until an equilibrium state is reached. It was shown in the publication that the approach can create *better placements than VPR's placer* (concerning both *net* and *critical path delay*) while being *slightly faster*. Even though the authors discuss the problem of I/O placement, they treat both problems (*CLB* and *I/O* placement) **separately** instead of the **global** approach presented in this work. In the force-directed placement routine, the I/O pads are not specifically placed. Instead, if a CLB needs to be connected to I/O resources, the distance to the nearest chip-boundary is simply taken as the respective contributed distance. In summary, their approach shows that such an analytical placement (for homogeneous FPGAs in their work) can achieve promising results. Malpuri and Hauck [142] came to the same conclusion by implementing different placer types and comparing them for **homogeneous** FPGAs. The force-directed approach they benchmarked achieved the best results of all in terms of *critical path length* and performed slightly better than VPR's simulated annealing placer. They especially investigated how the placement of the methods improves over time. Based on this, a *runtime-quality trade-off* could be defined and steered by the user to achieve either a very fast placement or a more time-consuming one of higher quality by stopping the iterative routines at the desired time. **Different trade-offs** between time and quality can also be chosen **in the presented work by different strategies for single placements** (see Section 5.5.3) **or with respective termination criteria for repeated runs** (see Section 6.6.1).

An early and *more related* approach (again for **homogeneous FPGAs**) was published by Raman et al. [155] in 1996. They set up a force-model like *Eades* with *attractive* forces between connected nodes and *repulsive* forces between non-connected nodes (see Section 4.1.2) representing the complete design as a whole (just like the **global** nature of the presented approach in this work). However, the I/O pads are again regarded **as fixed** and a system of equations is solved by a **numerical solver** whereas the nodes are then iteratively moved into the calculated direction until an equilibrium state is reached. The **Manhattan distance** is used to estimate the distance between blocks, like it is possible in this work. They additionally integrate a **timing-driven** approach called *adaptive netweighting* which is similar to the *slack graph morphing* of this work presented in Section 6.3. Their overall method works as follows. A set of upper bounds for the net lengths is obtained from the *zero-slack algorithm* to meet the timing requirements (see Nair et al. [143]). They are incorporated into the force-model as **weights** to modify the strengths of the forces. The force-directed approach calculates *optimal positions* for the *logic blocks* and the nodes are iteratively moved into this direction until an equilibrium state is reached. The nodes from this *continuous plane* are then assigned

to *near slots* on the FPGA in the order of the impact that this assignment has on the overall net lengths. Due to this, the necessary moving of blocks in the *slot assignment* should be reduced. Now, the incorporated *timing analysis* checks which connections are (*in simple words*) too slow and reduces the weights accordingly. This is repeated until no more significant improvement can be achieved. In contrast to this work of Raman et al., the approach presented here not only shortens critical paths, but also relaxes those that have a high slack. This gives the simulation in this work 'more space for nodes on critical paths' even though previous non-critical paths may become critical in this process. However, a good resulting '*timing-equilibrium*' was achieved by a detailed iterative adjustment of the underlying *model of weights* (which is comparable to the *slack graph morphing in Section 6.3*). Finally, a **limited and local pairwise exchange** is carried out to legalize paths that still violate any timing-constraints (see also Remark 9). This is comparable to the **local refinement** in this work (see Section 5.6.3) but towards the specific goal of legalization instead of general refinement. The *adaptive netweighting* payed off as it remarkably improved the overall timing and consequently increased the feasible speed in the circuit. They experimentally verified that optimizing for better timing can increase the overall wirelength while the circuits speed is still higher due to the additional optimization. Furthermore, they showed that routability was not remarkably affected by the timing-driven *adaptive netweighting*. These results motivate the **heterogeneous** approach from this work with all its incorporated methods which are discussed later in this Chapter (including **timing optimization**).

While writing this thesis, another related approach was published by Upadhyay [184]. He presented a placement method directly based on the **combined analytical and partitioning-based *GORDIAN*** approach using the mentioned quadratic formulation of the problem (see Section 5.1.2). It treats **homogeneous** FPGAs and was finally implemented in *MATLAB* to **compare its results to VPR's placer (version 7.0)**. The elements in the continuous result of the equation system were **assigned to slots** by searching a free slot starting from the calculated position in a **spiral** manner. The author already mentioned that this heuristic **can destroy** the results of the analytical placer for denser circuits **significantly** (like in the approach of *Mak and Li*). Finally, a simulated annealing with cold temperature for the final **local refinement** was applied (comparable to this work, while this work applies a *pure local search*). Overall, the results without the local refinement were achieved in 38% shorter time on average than the simulated annealing based placer in *VPR*, but the overall wirelength quality was rather poor. Together with the **local refinement**, the achieved wirelength was comparable to the one accomplished by VPR at the cost of 11% higher runtime. Finally, it has to be noted that the method needs an (optimized) I/O assignment as an input which

was received by **additionally running VPR's placer in advance**. This adds remarkable runtime (in addition to the reported one) and is not necessary in the presented approach in this work due to its **global** optimization with an **extended force model**. In any case, locations for the I/O pads have to be fixed in Upadhyay's model due to the application of the numerical method with only **attractive forces**.

Apart from such similarities between this work and others' analytical placers for **homogeneous** FPGAs, Hu [97] proposed an approach especially targeting a designated class of **heterogeneous** FPGA architectures. Like in the presented approach of this work (see Section 5.5.1), the set of different types of blocks in the overall design is partitioned into the available basic types of blocks (*computational blocks*, *memory blocks*, *LUTs* and *pads*) while the **pads are fixed** (*again*) to create a system with *attractive* forces only (like *Tutte*, Section 4.1.1). Hu assumes a special type of heterogeneous architectures which consist of an array of equal *basic processing units* (BPUs) which *each* contain different types of elements (*compute units*, *memory blocks* and *lookup tables*). Such an FPGA therefore has a *fine-grained heterogeneity*. Each *compute block* (CBs) (basically a part of the design performing a task) is represented by a geometric shape that models what kind of neighboring resources are needed for its included heterogeneous functionality (see the publication of Hu for some demonstrative examples). For example, LUTs are considered much smaller than compute units as the former ones are available in much larger numbers on the chip. However, these shapes of the CBs (and therefore the local arrangement of elements) are **predefined** and **constant** throughout the process (what is *not at all the case in this work*). Now, a force-directed optimum of the nodes is iteratively calculated for each density layer solving a non-constrained quadratic optimization problem by a VLSI placer (see Hu et al. [98], similar to GORDIAN in Section 5.1.2). The method stops as soon as the density on the two dimensional *density-fields* (for each block type) is balanced. For this, several iterations are necessary as the positions achieved for the different types obviously affect each other. In general, high-density (overlapping) regions are resolved by adding respective attractive forces (*fixed-points*) to the system to pull the nodes away from such high density regions (also comparable to GORDIAN). Such effects are **directly avoided** in the presented **global** approach in this work by **repulsive forces** in the system and an **appropriate slot assignment** instead of such a rather *local* approach. Hu's method quasi creates a *floorplanning for each type* of resources (like for ASICs). Even though the idea of the method is not too related to the presented approach in this work, the **partitioning of the overall design into the different block types** and individual assignment routines for these are also necessary in this work whereas the optimization in this method is always performed **globally as a whole**. The approach of Hu is also **timing-driven** as it modifies

force-strengths in the system throughout the iterative process relatively to the connections' criticalities by timing analyses (similar to the timing-driven *slack graph morphing* presented in Section 6.3). Hu performs the timing analyses within the necessary iterative process of the force-directed method to adjust the forces. Even though an iterative process is not *necessary* in the presented **global** approach of this work, repeated independent runs can deliver and include such information in the mentioned *slack graph morphing*.

In summary, the works of Raman et al. [155] and the one of Upadhyay [184], both for homogeneous FPGAs with fixed I/O pads, can be regarded as the two mostly related works in this field (in the author's opinion).

5.2 HETEROGENEOUS FORCE-DIRECTED PLACEMENT

One main idea behind this work is to bring progress that was made in the field of graph drawing within the last decades, in particular in the field of *force-directed graph drawing*, back to the contemporary field of FPGA placement (see Chapter 4.3) together with a new assignment methodology and additional studies. The presented method in this work is pursuing several goals.

First of all, a **force-directed** placement routine for **heterogeneous FPGAs** should be developed to meet the requirements of today's FPGA architectures, principally basing on an iterative **spring embedder** simulation (see Section 4.1.4) with a **force system** including **attractive** and **repulsive** forces (in this case FM^3, see Section 4.2). It is desired to build a method incorporating **multilevel** coarsening and **multipole** approximations of repulsive forces for a **fast** simulation with **high accuracy**.

A characteristic of almost every available analytical placement method is that the surrounding I/O pads have to be fixed to create a uniquely solvable equation system finding the optimal coordinates of all inner nodes. The technique presented in this work should be usable **without any fixing of nodes** to optimize the system more **globally**, as the initial fixing of nodes can either negatively affect the quality of the resulting placement or, if it is calculated in advance, be very time consuming. Anyway, I/O positions and inner logics' position influence each other considerably so that all nodes should be declared **free** if the situation allows it (e. g., in a prototyping phase).

Both facts, the influence of the actual assignment of fixed nodes to predefined positions and the unbalanced distribution in the absence of repulsive forces (see Section 4.1.1 for examples from the related approach of *Tutte*) emphasized the application of a **global and entirely free** spring embedder approach. The direct usage of **graph drawings** that could be provided by **any**

graph drawing software within the workflow (due to **universal interfaces**) should generate an **intuitive** entry point and **flexible** access as a basis for future developments in this field supported by other researchers, e. g., from the field of general graph drawing. In particular, the **abstraction** of the chip design into a basic **graph** will be used to create **placement strategies with different targets**, e. g., **wirelength-driven**, **timing-driven** or **routability-driven placements** or even their **weighted combinations**. In addition, various operation modes with different **trade-offs** between **time** and **quality** should be available (*depending on the development state of a design*) and the principal **distribution of elements** on the chip should also be adjustable (*for different demands*).

Finally, the system should have a **modular design** to make it possible to use parts of it in combination with other methods. As many previous analytical placers need fixed I/O pads, a rapid global optimization from this workflow could, for example, also be used to obtain a good initial I/O distribution for such analytical methods.

It has been discussed in Section 5.1.1 that the design flows of today's (and *especially upcoming*) architectures with an extremely increased number of logic elements have a great need for faster techniques than the traditional simulated annealing, which is why Xilinx already switched to analytical placement in 2012. The method presented in this work could be incorporated in any FPGA CAD software in the future. However, to compare against *the* simulated annealing method in the academic FPGA world, it will first be incorporated into *VPR* in this work.

5.3 SETUP OF THE BASIC DATASTRUCTURES

The *place-and-route* tool *VPR* is embedded into the comprehensive *Verilog-To-Routing* (VTR) CAD flow [130] and basically needs two input files to run the FPGA compile chain (see Section 2.4).

First, it needs *the design* that should be implemented into the hardware in form of a netlist. A high-level *Verilog* description of the design is transformed into a text file using the *Berkeley Logic Interchange Format* (BLIF) in the *elaboration* step performed by ODIN [101]. This file is subsequently taken by the *synthesis* step (*ABC* [176]) for hardware-independent optimizations and the like.

Second, it needs a description of the targeted *hardware architecture*. VPR expects a principal description of the architecture in an *Extensible Markup Language* (XML) file. **For all *VTR (VPR) benchmarks*, the heterogeneous flagship architecture of VTR, the '*Comprehensive Architecture*' file, is used.** This *heterogeneous* architecture consists of *CLBs* (with *fracturable LUTs* that can be

used either as one 6-LUT or two 5-LUTs, see Section 2.2.1), fracturable *multipliers*, configurable *memories* and *I/O pads*. Every eighth column of the *Comprehensive Architecture* is a column of multiplier blocks and, with an offset, every eighth column consists of memory blocks. Both *special block types* span several rows as they are larger than the ordinary LUTs. As the multipliers are fracturable, a 36×36 multiplier can also be used as two independent 18×18 multipliers or these again as two 9×9 multipliers. The memory blocks are fracturable by their *word size*. The I/O blocks are surrounding the architecture while each *I/O block* holds 8 I/O pins which can either be used as output or as inputs (cp. Section 2.2.1).

Remark 69. *A modern architecture can also contain so called* ***carry-chains*** *to transport results from one LUT to a neighboring one directly. These can, e. g., be used to implement* ***adders*** *in hardware efficiently using columns of LUTs. However, it does not affect the* ***abstracted architecture*** *and is, thus,* ***not explicitly considered*** *in the presented method. The architectures used for the benchmarks later in this work do not model carry-chains.*

All architecture assumptions in *VTR* are made based on real architectures from Xilinx and Altera, see the publication of *VTR 7.0* [130] for more details. However, the presented method is not restricted to such an architecture in any sense. Further *special* block types and also different *CLB* or *I/O* types can be easily added to the model in the future (see Section 5.5.5 and Section 5.8).

VPR first reads the design description and *packs* (or *groups*) it into basic blocks available on the architecture (*I/O*s, *CLB*s, *MEM*s, *MUL*s - see Section 2.4). After this, it is known how many resources of each type are necessary for the design. An architecture with the aforementioned properties that contains all such basic blocks in an adequate number can *automatically* be created instead of manually passing a size for the FPGA. This automatic mode is used in all benchmark runs of this work (apart from the outlook in Section 7.3). In *real* implementations (in contrast to such *simulations*), a suitable subarea of the overall FPGA is often chosen.

The size of the automatically derived *squared* $N \times N$ *architecture* is determined by simple *bisection* starting with $N_0 = \sqrt{\#\texttt{blocks}}$. If there are enough resources of all types, the size N_0 is halved to $N_1 = \frac{N_0}{2}$ (*otherwise it is doubled*) and the routine subsequently checks whether enough resources *of all heterogeneous types* are available on an architecture of this size. If not, $N_2 = \frac{N_0+N_1}{2}$ is checked and so forth. As soon as the suitable size is found so that enough resources of each type are available, the position of all blocks on the architecture is exported to an appropriate data structure named `FPGAArch` (see Section 5.3.1).

VTR comes with a set of *heterogeneous* benchmark circuits which will be used throughout this work to show the *working principles*, the *characteristics*,

the *quality* and the *performance* of the developed methods. The necessary number of each block type (after packing with *ABC* in *VTR 7.0*) for all benchmarks in this heterogeneous set is shown in Table 12 on page 222. These benchmarks (included in *VTR 7.0*) are mostly identical to those of earlier VTR releases so that results can be compared to former implementations and publications. Table 12 also contains information about the *representing graph* that is set up by the presented method for the force-directed placement as its input (see Section 5.5.1). Section 6.5 will additionally provide results for another set of benchmarks for logic-synthesis, the well-known classical (*though rather outdated*) and formerly commonly used *MCNC benchmarks* [188]. The presented methods of this work were originally developed and integrated within *VPR* version *6.0*. With the release of version *7.0*, several structures changed in VPR and, more globally, it was even programmed in *C++* instead of *C* before. With this version change, several methods were refined whereas the performance of *VPR 7.0* is generally significantly better than the one of *VPR 6.0*. In particular, the packing was greatly *accelerated* while producing solutions of *higher quality*. However, the placer routine has also been accelerated by about 24% (e. g., by the *incremental bounding box update*, see Luu et al. [130] for details and comparisons). The presented method was adjusted accordingly and integrated into the *VPR 7.0* code. The final version of the presented placer can consequently be used *in both version with the same code base*.

For comparability to earlier works, *both VPR versions* are supported by the implementation while only the *faster and most recent* version *VPR 7.0* (with its *faster simulated annealing approach*) is finally benchmarked and compared.

5.3.1 *Model the architecture*

Before the placement routine starts, an appropriate *architecture size* was either defined by the user manually or found by the bisection-based routine introduced in the previous section. Anyway, the automatic bisection-based method guarantees that the *packed design* can be implemented in the defined hardware architecture. If the user defines the size manually, this is checked and the program *does not proceed* if the size is not sufficient.

An architecture that was created by *VPR* with automatic sizing is shown in Figure 46 with an initial random assignment of the 44 *CLB*s (in white), 258 *I/O*s (in orange), 5 *MUL*s (36 × 36 in gray) and no *MEM*s (in blue) from the `diffeq1` VTR design. The *unoccupied* slots of all types on the architecture are shown in a *lighter shade* of the respective (type's) color. There are 96 *CLB* slots, 384 *I/O* slots (at 48 8-way locations), 6 *MUL* slots and 4 *MEM* slots on the depicted architecture.

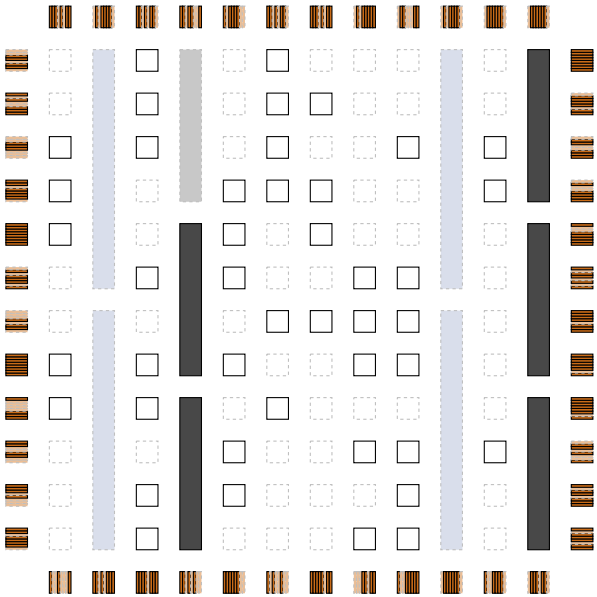


Figure 46: The heterogeneous architecture in VPR (code: `diffeq1`)

The overall architecture (and therefore also the placement) can be represented on a two dimensional grid (`FPGAArch[x][y]`) containing *logic blocks* and *routing resources* (RR). Each element on this grid contains a *type information* (*CLB*, *I/O*, *MEM*, *MUL* or *RR*). The I/O blocks additionally need to contain the number of available I/O pads in this I/O block (e. g., *eight* in Figure 46 for each I/O block). The routing wires, for example, supplementary store their channel widths (see Section 2.3). In general, each block on the architecture is represented by a *two-dimensional* reference point in its center. All (e. g., *eight*) **I/O** pads in an I/O block are represented by the center (x, y) coordinates of their block together with an additional z-coordinate (e. g., ranging from 0 to 7) to distinguish the different pads in a block.

In addition to the general availability of the *CLB*s' coordinates in the `FPGAArch` array, the number of CLBs *in each row* of the FPGA is stored in a separate array for fast creations of appropriate distributions later (see Section 5.5.3). For the depicted architecture in Figure 46, this *CLB-on-architecture* distribution contains 8 CLBs in each of the 12 CLB rows.

All heterogeneous blocks (**MEM**s and **MUL**s) are solely represented by a central reference point.

5.3.2 *VPR norms*

BOUNDING BOX COST (BB) The predominant norm in VPR is the *bounding box cost* norm, which takes the *semi-perimeter bounding box sizes* of all nets and the available average *channel width* in a region of the FPGA into account (see equation (5)). The implementation in VPR also includes an estimation about wire-crossings within the boxes and rates the consequent wire elongation in the routing based on the previously named bounding boxes' parameters (the *RISA* model, see Cheng [125]). Based on this estimation of wire-crossings (which essentially uses statistics about *Steiner Trees*), the *bounding box cost* is scaled by a factor $q(i)$ depending on the number of *terminals* in a net.

The simulated annealing approach in *VPR* uses this norm in its cost function. Thus, the optimization in *VPR* considers both the overall *wirelength* (by the *semi-perimeter bounding box sizes*) and the routability (by the *channel widths* and the *estimation of crossings*) simultaneously.

CRITICAL PATH DELAY (CPD) *VPR* includes a method to estimate the *critical path delay after* the actual routing and also *before* it. If this norm is used *after the placement* and, therefore, *before the routing*, the wire-delay has (of course) to be roughly estimated as the concrete routing tracks are not known. This norm estimates the *maximum delay* of a clock-cycle in the design and consequently the maximum possible speed to run it validly ($\frac{1}{\mathrm{CPD}} \cdot 10^3$ MHz). See Section 2.2.4 for a detailed explanation how the calculation of this norm is performed.

MAXIMAL CHANNEL OCCUPANCY (MCO) The router in VPR tries to route all connections of a net with the available routing tracks on the architecture by applying an iterated maze router similar to the *PathFinder negotiated congestion-delay algorithm* [138]. As already described in Section 2.4, the nets are ripped up and rerouted in each *routing iteration* with adjusted parameters (*timing-driven* in the applied default configuration) if the previous routing was not successful under the restrictions of the routing architecture (consequently prioritizing critical connections in the next routing iteration). This process is repeated until a successful routing is created which satisfies all guidelines (including some additional cost functions) or until a predefined number of routing iterations has been performed (which is 50 in the default setup, see *RouterOpts.max_router_iterations* in Appendix A.6). After the routing process, the numbers of routing tracks that are used on each wire segment (their *occupancies*) are reported. The *Maximal Channel Occupancy* (MCO) is consequently the largest number of channels used on one wire segment by the final routing. If the routing satisfies the requirements of the architecture, the occupancy is smaller than the channel width for each wire segment. Oth-

erwise, it exceeds the guidelines and would not be realizable on the given architecture.

Remark 70. *As this work is not particularly about the routing, this 'simplified' view of it should be sufficient. More details about the routing process can be found in Betz et al. [21].*

However, the *maximal channel occupancy* can be taken as a norm to rate the *congestion* in a specific routing and the routability of a placement. Together with the finally realized *critical path delay* after routing, the overall quality of a design's layout can be rated.

Remark 71. *During and after the simulated annealing iterations in VPR, several additional norms like the overall delay sum of all point-to-point connections are reported. However, for the investigations in this chapter, only the named norms will be used.*

5.4 ADDITIONAL INTRODUCED NORMS

Along with the presented `FieldPlacer` method, additional norms were introduced to rate the quality of a placement towards different objectives. All these norms operate on a model considering the *global routing* on the architecture (see Section 2.4). The precise setup of the graph model is described in Section 5.5.1. Generally, there is an *initial graph representation* $\mathcal{G}_D$ with arbitrary coordinates to perform the basic force-directed layout $\mathcal{G}_D^{\text{layout}}$ and an embedded representation $\mathcal{G}_D^{\text{arch}}$ on the architecture after slot assignment with *constrained integer coordinates*.

5.4.1 *Point-to-point **W**ire**L**ength*

The bounding box cost norm in VPR includes the approximation of the overall wirelength by the semi-perimeter bounding box size of nets. As the actual routing of connections is not known in the placement phase, this approximation is reasonable and, as the upcoming chapter will show, very well suited and rather accurate. However, the force-directed layout approach that will be used in this work tends to minimize the *point-to-point* wirelength sum in the targeted force-equilibrium of the introduced graph model of the design by attractive forces (see Section 5.5.1) while keeping distances between nodes by repulsive forces. Thus, a further norm was implemented that iterates over the edges of the graph representation of the design ($\mathcal{G}_D^{\text{layout}}$) and sums up all distances between connected nodes. Due to the characteristics of

the routing architecture, the distance is measured as the *Manhattan distance*. Consequently, the wirelength in the graph can be derived by equation (54).

$$\mathbf{WireLength}_{\mathcal{G}_D^{\text{layout}}} = \sum_{(u,v)\in E_D} |x(v) - x(u)| + |y(v) - y(u)| \tag{54}$$

After the embedding of the graph on the integer grid of the chip, each node $v \in \mathcal{G}_D^{\text{layout}}$ receives the coordinate of its assigned *slot* on the *architecture* in $\mathcal{G}_D^{\text{arch}}$. Thus, the point-to-point wirelength on the chip can be calculated analogously on the corresponding graph $\mathcal{G}_D^{\text{arch}}$ with $\mathcal{G}_D^{\text{arch}} = (V_D^{\text{arch}}, E_D)$. Notice that the connection information (E_D) in the graphs does not change by the *layout* or the *embedding*.

VPR uses the semi-perimeter bounding box approximation not only because the actual routing is not known, but also because the bounding box updates can be performed much faster than recalculating all connections' lengths of a node after a position change. This is extremely important for VPR's placer, as the simulated annealing method needs to recalculate the distances frequently in every iteration. The point-to-point wirelength is not applied within the optimization process of the graph, but after it in order to get a final evaluation of the achieved quality so that the time consumption is 'negligible'.

5.4.2 *An approximation of congestion*

As mentioned previously in Chapter 2, not only the estimation of the wirelength in the resulting layout plays a role for a placement. Other criteria may even be much more relevant, though not easily assessable. For example, it would often be more desirable to minimize the *critical path length* instead of the overall wirelength. However, in a simulated annealing process, the estimation of the critical path length in every iteration would be way too time consuming (see Section 2.2.4). Thus, such complex estimations are generally performed once after the placement.

Besides wirelength, VPR's placer also includes a statistical evaluation of wire crossings (see Section 5.3.2) to take *routability* into account in the annealing process. In general, rating routability is a difficult task but as the routability affects the later routing *time* and *quality*, an approximation of it is desirable to compare different placements. The following section describes a model to rate the routability of a placement based on several idealized assumptions while taking the actual routing architecture on the chip into account.

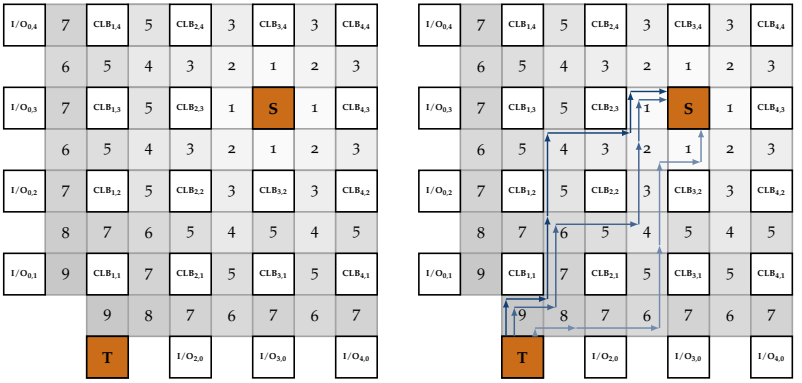
The FieldPlacer congestion-driven maze router

As *VPR* iteratively applies a *maze router* to route the nets, this behavior is *imitated* by the 'FieldPlacer congestion norm'. In fact, the routing of *each point-to-point connection* between logic blocks is simulated by searching a shortest route via *wave propagation* and *backwards tracking*. As it has been described earlier in this work, there are, in general, multiple shortest routes (concerning the *Manhattan* distance). The idea is to make *one* routing attempt under the assumption of an *infinite* number of routing tracks in each routing channel. In addition, the routing cell with the smallest current *congestion* is (*greedily*) chosen among the possible cells on shortest routes. After that, the *overuse* of all routing wires is measured by summing up the congestion on all routing tracks. This process will be illustrated in the following.

Figure 47a shows a part of an FPGA architecture with *I/O*s and *CLB*s. In this example, a connection from the marked CLB (***S**ource*) to the marked I/O (***T**arget*) has to be found. Therefore, a wave is expanded from the source point, marking the Manhattan distance of every routing cell back to the source. As the overall architecture not only consists of routing cells but also contains the logic units, the expanded wave has *holes*. However, due to the regular grid of logic and routing resources in the model, a shortest track back to the source is available for every connection. The wave is expanded until the target point is reached with a final Manhattan distance n. This part of the process is called *wave expansion*.

After this, the route is determined by starting at the *target* point and following the wave back to the source by choosing routing resources with declining Manhattan distances from n to 1. This part of the process is called *backwards tracking*. Figure 47b shows three different routes with minimal Manhattan distance. Due to the 'holes' in the wave that occur from the logic cells, the process always proceeds from one *switch box* to the next *traversing one wire segment*.

All depicted routes in Figure 47b obviously have the same (*minimal*) wirelength by this construction. Now, the *occupancy* of the cells caused by already routed connections is taken into account. If two wire segments (routing cells), *both on optimal tracks*, are available as the next cell, the 'FieldPlacer congestion-driven maze router' *greedily* takes *the* next wire segment with the smallest current *occupancy* to continue the routing. This decision is *locally optimal* but not necessarily *globally*. Figure 48 visualizes the process for two example situations. Starting with the current wire usage (in form of the *occupancy array*), the wave is expanded from the source and tracked back from the target. Reaching the first switch box, the track could continue *upwards* or *rightwards*. In both examples, the *upwards* wire segment has a smaller current *usage* (or *occupancy*) and is therefore chosen. After choosing the next routing



(a) Expanded wave on architecture

(b) Backwards tracking of possible routes

Figure 47: Wave expansion on the architecture

cell, the occupancy array is updated. Finally, the wire usage on the cells that are chosen for the actual route are *increased by one* each. This procedure is consecutively performed for every connection between blocks in the design.

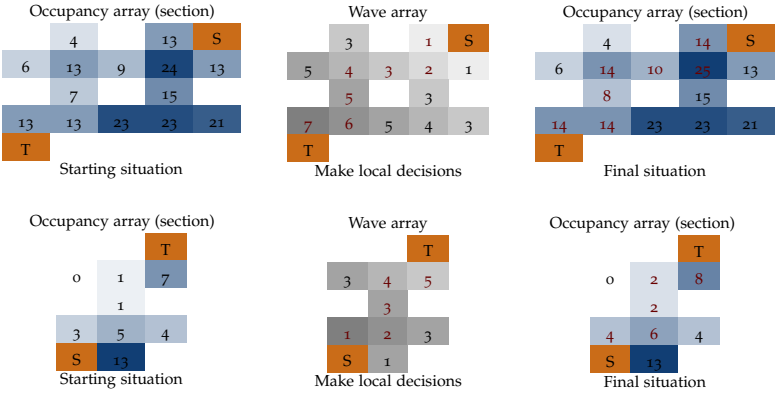


Figure 48: Congestion-driven maze router (two examples)

After every connection has been routed in a *globally optimal* way concerning *wirelength* and *locally optimal* concerning the *congestion* of routing tracks, the

overall *overuse* of routing resources in this simplified model is obtained by summing up all cells' overuse ratings (see equation (55)).

$$\mathbf{O}\text{ver}\mathbf{U}\text{se} = \sum_{\text{wire segments } \mathbf{w}} \max(0, \text{occupancy}(\mathbf{w}) - \text{capacity}(\mathbf{w})) \tag{55}$$

Thus, the *overuse* norm considers only those wire segments that would not be routable in the described manner on the defined architecture. It rates in which quantity *routing cells would be overused*. In the actual routing phase, such *congestions* would have to be resolved by routing uncritical paths on a detour. This enlarges the final overall *wirelength* and also increases the *routing time*. Consequently, a placement with small overuse is generally desired. The entire process is summarized in Algorithm 8.

Remark 72. *The capacity of **switch boxes** is not easily estimable because not every change of direction at 'intersecting' wire segments is possible (see Section 2.3, especially Figure 10). A **detailed routing** would be necessary for accurate investigations (see Section 2.4). Thus, the switch boxes' capacities are not considered in this model.*

Algorithm 8 Congestion-driven maze router

```
procedure CongMaze(Arch FPGAArch, NodeList* BlockPlacement, EdgeList* BlockConnections)
    combine point-to-point connections from BlockConnections
    with their coordinates from BlockPlacement

    for all such point-to-point connections (S, T) do
        place S and T from BlockPlacement on the FPGAArch

        start at S                                              ▷ wave expansion
        repeat
            expand wave by one unit on the routing segments' cells
        until T is reached after n expansions

        start at T                                              ▷ backwards tracking
        for all n, n − 1, . . . , 1 do
            find next wire segment with smallest current occupancy
            update occupancy array
        end for
    end for

    OverUse = Σ_{wire segments w} max(0, occupancy(w) − capacity(w))
    return OverUse                                              ▷ return the norm
end procedure
```

Like in the '*real*' routing, the order of the connections plays a role for this process, but as the norm is only used to get an impression of the *stress* on the routing architecture, this fact is not considered in the norm calculation. Figure 49 shows the wiring and the occupancy from such a *simulated routing* run with the '`FieldPlacer` *congestion-driven maze router*'. The resulting *overuse* is illustrated and discussed in Figure 71 in Section 5.5.6.

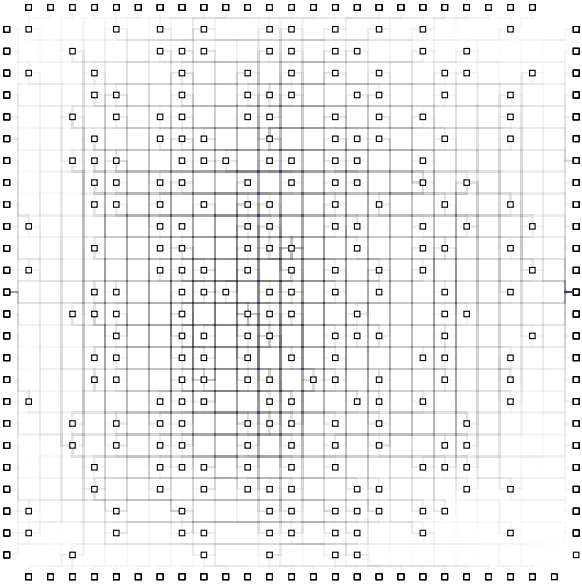


Figure 49: Congestion Router result, cp. Figure 71d (code: `or1200`)

Instead of simply accumulating the overuse of wire segments, a *superlinear assumption* could be made to, for example, penalize heavily overused resources more than only slightly overused ones, as the rerouting may take overproportionally longer in such cases (see Figure 50).

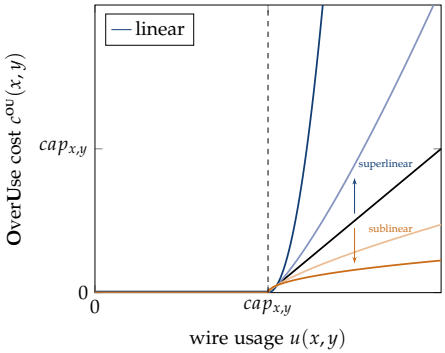


Figure 50: **O**ver**U**se in a cell

Finding a suitable function for the *overuse cost* would in any case be difficult and would have to be done on basis of experiments, as there are many *influencing and unknown* factors. In this work, a *linear* behavior of the overuse cost is assumed. Thus, the function is piecewise linear (0 up to the point of overuse and directly proportional with slope 1 afterwards). As a result, the more a segment is overused, the higher is the penalty, while non-overused segments are neglected.

Remark 73. *In this context, VPR similarly rates the channel width in its cost function linearly as it led to the best results in practice (see Section 2.4).*

5.5 THE FIELDPLACER METHOD

Remark 74. *In this work,* ***VPR*** *is* ***always*** *used in its* ***default*** *configuration (see Appendix A.6). Only the* ***seed*** *for the initial random assignment of all blocks onto the architecture is varied (via* `command line` *parameter) for repeated runs of one method. However, the set of* 10 *different seeds for* 10 *repeated runs is reused for the different methods to generate the same set of* 10 *inputs for each. Unless otherwise mentioned, each measurement is repeated* 10 *times with the already mentioned* 10 *different seeds for each version and each input code. Finally, the minimum and the maximum of the* 10 *runs (concerning the different norms) are ignored while the other values are averaged. The used hardware is described in Section 1.4.*

The `FieldPlacer` method creates a heuristically energy-minimized graph layout as a basic 'arrangement-draft' of the design and embeds this *unrestricted* graph with *arbitrary* (e. g., floating-point) coordinates on a given heterogeneous FPGA architecture (and therefore on a *constrained integer grid*). The algorithm to create the initial graph-layout is arbitrarily exchangeable although a *force-directed graph layout* is, for this approach, advisable to match the presented embedding-process. The *basic* `FieldPlacer` method mainly bases on *nested sort-techniques* and *barycenter- and angle-calculations* paired with *user-definable distributions*. It is composed of several consecutive *steps* and can be extended for upcoming FPGA architectures with other block types and also with further methods. The development of the `FieldPlacer` itself has been an iterative process adding more and more refinements and functionalities step by step. The method sets up a graph that represents the design that has to be be placed, creates a *free (unconstrained)* force-directed layout and places it by assigning each element to a suitable (*fitting and adequate*) integer position on the restricted grid of the FPGA chip following different (*selectable*) strategies. This *basic* `FieldPlacer` method has then been extended by further optimization steps like, for example, a *local refinement* (see Section 5.6.3), the application of *different distance norms* in the layout phase (see Section 5.6.2),

a *second energy phase* (see Section 5.6.1) or even the repeated application of (parts of) the method in a statistical framework (see Chapter 6).

Remark 75. *In this work, the algorithmic ideas of the* `FieldPlacer` *and their effect are described and compared. However, details about the implementation are not part of this work. The implementation contains several additional experimental features that are not even described. Any interested reader is very welcome to contact me!*

5.5.1 *1st Step: Setup of the graph representation*

The representation of the *design*, stored in the `FPGAGraphRep` structure, is the fundamental basis of the procedure and the input for the force-directed graph layout. For its setup, each *packed (CLB, I/O, MEM, MUL) block* of the FPGA design becomes a *node* in the `FPGAGraphRep` and all *point-to-point connections* of the nets of the design are traversed and incorporated as *edges* in the `FPGAGraphRep`. In this process, only connections that do *not belong to global nets* are considered because global nets do not have to be routed on the normal routing architecture and do generally not influence the placement, routing and timing of a layout (see Section 2.2.2).

The *initial position* of the nodes is only needed if there are *fixed* nodes in the design that should not be moved. For now, this is not the case. Thus, the graph representation contains no information about the position of nodes but solely their *connectivity* (or *adjacency*) and their heterogeneous block *type*. It is consequently a pure abstract graph with *no definite geometry* (no *embedding*).

Depending on the subsequently applied graph layouting approach, it can be desirable to remove *parallel edges* in the graph that result from multiple connections between pairs of logic blocks. This is optionally possible in the `FieldPlacer` method. For the presented ideas of this work, this option is actually *always activated* as it becomes particularly important in the *slack graph morphing* procedure in Section 6.3 to steer the connections' lengths in the layout.

In summary, the `FPGAGraphRep` structure represents the netlist of the *input design* as a graph $\mathcal{G}_D = (V_D, E_D)$ with *all* heterogeneous blocks (V_D) and their interconnectivity (E_D). This graph may have *multiple components* processing different *independent* tasks. However, the inputs that were taken from the heterogeneous benchmark set in *VTR 7.0* (see Table 12) mostly contain *one single* or at least *one predominant component* (in terms of number of blocks/nodes).

Algorithm 9 Create the FPGA representation graph

```
procedure CreateFPGARep(NodeList* BlockPlacement , EdgeList* BlockConnections)
    for all nodes in BlockPlacement do                          ▷ create the nodes
        create a node in the FPGAGraphRep (G_D)
        store the block type with the node in the FPGAGraphRep
    end for

    extract point-to-point connections from BlockConnections that are NOT on GLOBAL nets

    for all such point-to-point connections (S, T) do           ▷ create the edges
        if Option.no_parallel_edges then
            if the nodes S and T are not yet connected in the FPGAGraphRep then
                insert the edge (S, T) into the FPGAGraphRep
            end if
        else
            insert the edge (S, T) into the FPGAGraphRep
        end if
    end for
    return FPGAGraphRep (G_D) and .gml representation           ▷ return and export FPGAGraphRep
end procedure
```

5.5.2 *2nd Step: A force-directed graph layout*

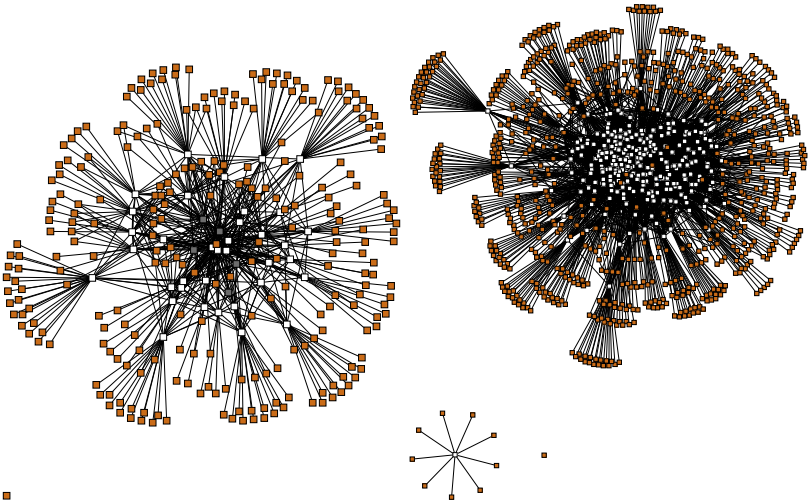
◁ *. . . Energy Layout*

The FPGAGraphRep from Section 5.5.1 is internally stored in a structure and additionally exported to a common *Graph Modelling Language* (GML) [95] file (see Algorithm 9). In that way, the graph layout can be performed by *any* graph layouting software that is able to read and write such files. Additional interfaces can be implemented. In the FieldPlacer method, this GML file is passed to a slightly modified and enhanced version of the FM^3 algorithm implemented in OGDF (see Section 4.2), FieldFM^3 and FieldOGDF in the following. The extensions are methodologically described in in the following Sections and some technical insights are given in Section 5.9.1.

Remark 76. *The **GML**-interface may require some small additional (and **negligible**) time to write the graph to harddisk, but it makes the layouting approach easily exchangeable, even by commercial products with closed sources.*

In VPR's simulated annealing approach, all blocks are randomly assigned to suitable slots on the architecture to create a legal initial solution. The graph layouting in FieldFM^3 gets the general graph description (*without embedding*) and starts with a random initial assignment on the coarsest representation of the multilevel framework.

Remark 77. *All nodes are randomly assigned in the beginning as long as there are no user-defined **fixed** nodes. As there were no inputs with such fixed blocks in the benchmark set, this is always assumed in this work. However, an extension with initially fixed nodes is directly possible due to the extensions implemented in* FieldFM^3.



(a) code: `diffeq1` (b) code: `or1200`

Figure 51: Force-directed graph layout

Figures 51 shows the resulting layout obtained from FieldFM^3 for two example codes. Even tough these layouts were produced *without any restrictions* concerning the resulting coordinates of the nodes, the results show some distinct peculiarities of achieved force-directed layouts in general. First of all, each node is approximately placed in the barycenter of its neighbors as the energy-minimized solution tends to minimize the sum of distances between connected nodes (see Section 4.1.1).

In addition, *I/O nodes* (depicted in orange) tend to the border of the layout. This is based on the fact that most I/O pads are only connected to *one single inner* logic block (CLB, MEM or MUL). Thus, such I/O nodes are (the) *leaves* of the graph and there is no *force* that pulls the node 'inwards' the graph layout except for these single connections. This perfectly matches the structure of FPGAs (e. g., the considered *island-style* FPGAs) or chip architectures *in general* as the I/O connections are naturally surrounding the other elements.

Remark 78. *There may also be I/O nodes that are connected to multiple inner nodes. These are then, again, placed near the barycenter of their neighbors.*

Every block of another type (than *I/O*) generally has *in-* and *outputs* to *process data* and is therefore carried to the inner regions of the graph layout. As a result, while the I/O nodes are pulled outwards the layout due to the *repulsive forces*, the neighboring inner nodes reside near them. The inner nodes

are ordinarily much stronger connected to *several* other nodes (e. g., a CLB, depicted in white, contains multiple LUTs and these have multiple in- and outputs). In fact, the *MEM* (in blue) and MUL (in gray) elements often have exceptionally many in- and outputs and are therefore strongly connected (and placed near the barycenter of their neighbors).

As extensively discussed in Chapter 4, the layout of the $\mathcal{G}_D$ follows some physically motivated properties. Like in many published approaches from the field of *analytical placement techniques*, connected elements *attract each other* by the attractive forces in the force model. However, the FieldPlacer method additionally considers *repulsive forces*. As a consequence, the I/O nodes can be *freely* distributed to find good positions for them (like for all other nodes) *without a collapse* of the system. Another very positive effect is that the nodes are quite evenly distributed and not too much space is wasted in the layout. Due to the implication of the *repulsive* forces for each node in V_D, nodes tend to repel each other and this reduces the problem of *overlapping* (see Section 5.1). On the other hand, each edge in E_D generates a *contracting* force between connected blocks, conceivable as a spring. By these attractive forces, connected nodes still tend to be placed closely together.

Summarizing, this phase generates a low energy arrangement of the system (the *design*), it concentrates 'clusters' that are strongly connected together (so that the many wires between nodes in the cluster are kept short) and it preserves larger distances between groups of nodes with smaller numbers of connections. The method thereby keeps the overall edge length sum small and generates a consistently distributed node arrangement of the design whereas the repulsive forces ensure that the *stress* (resp. the *overlapping*) in all regions of the layout remains moderate.

The main idea is now to take this force-directed layout as a 'preliminary sketch' of the later embedding. For the depicted graph layouts, the FieldFM^3 implementation (which uses the FM^3 algorithm [81]) was used as it is extremely fast while being accurate at the same time (see Section 4.2). Nevertheless, the method is absolutely exchangeable. No matter which particular force-directed layout approach and implementation is used, the mentioned *positive core effects* are in the nature of these methods and should therefore generally be present for each individual implementation.

Extensions

The later placement can easily be controlled, adjusted and extended by modifying the properties of the graph model $\mathcal{G}_D$ as it is, e. g., done in the *slack graph morphing* (see Section 6.3) by adjusting the *zero-energy lengths* of the edges to iteratively reduce the overall slack in the design and consequently the *critical path delay ('length')*.

Minimizing the critical path delay is reasonable when the primary optimization goal is performance. If developers are, for instance, aiming at a better thermal distribution, the graph model and the layouting algorithm could be modified in that direction before applying the force-directed layout, e. g., towards further reduction of stress by a more 'aggressive' function for the repulsive forces. Even completely different layout-techniques could be used in such a case.

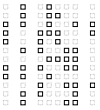
Another possibility to adjust the generated placement is to choose different norms for the distance within the layouting phase. For the depicted graphs in this section, the *Euclidean* norm has been used to calculate the strength of attractive and repulsive forces. Section 5.6.2 discusses how other norms can be applied to influence the layout in a positive manner.

Remark 79. *Note that the **horizontal coordinates** in all following calculations (and also in all visualizations) **rise from west to east and** the **vertical** ones **rise from north to south** (**NOT** from south to north).*

Remark 80. *To represent the order of the nodes in the linked **node-lists**, the graphs in the next paragraphs depict succeeding list elements connected with an arrow. These **arrows** do **not** represent physical connections **in the design**.*

5.5.3 *3rd Step: CLB placement*

CLBs... ▷



Definition 25. *Revisiting the notation from the previous Sections, let $\mathcal{G}_{\mathbf{D}}$ be the graph of the basic description of a design's blocks' **connectivity** (**adjacency**), $\mathcal{G}_{\mathbf{D}}^{\textbf{\textit{layout}}}$ the output graph of the **force-directed layout** with arbitrary coordinates and $\mathcal{G}_{\mathbf{D}}^{\textbf{\textit{arch}}}$ the **embedded graph on the architecture after slot assignment** with constrained integer coordinates.*

The nodes of the force-directed layout in $\mathcal{G}_{\mathrm{D}}^{\mathrm{layout}}$ from step 2 in the previous Section can have arbitrary (*floating-point* and *arbitrarily sized*) coordinates. The next steps *embed* this graph layout onto the restricted integer grid of the FPGA architecture.

As *CLBs* are the basic logic blocks of ordinary FPGAs (the '*general purpose FPGA workers*', see Section 2.2) and therefore are, in general, the predominant (non-I/O) block type in a design, these are embedded first of all. For that, the *GML* output from the graph layouting method is taken as the input of the following steps.

To be able to create an embedding of the CLBs from the graph layout, the CLB *nodes* and their coordinates are extracted from $\mathcal{G}_{\mathrm{D}}^{\mathrm{layout}}$ (which contains *all* block types) while the connections can be neglected in this step (see Figure 52). The extracted set of CLB nodes is denoted by $V_{\mathrm{CLB}}^{\mathrm{layout}}$ and they should

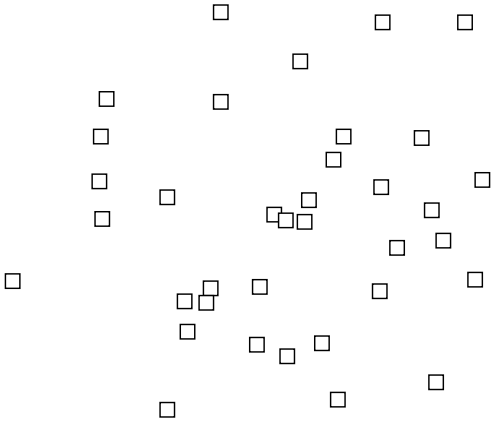


Figure 52: Inner CLBs after force-directed layout (code: `diffeq1`)

now be placed on the integer grid, generally *preserving their two-dimensional arrangement to each other in the obtained layout*.

Remark 81. *In the following, the nodes of a node pair v and $\tilde{v}$ are always representing the same node of the FPGA design while $v \in \mathcal{G}_D^{layout}$ has the arbitrary coordinates of the layout and $\tilde{v} \in \mathcal{G}_D^{arch}$ is its counterpart embedded on the architecture.*

Vertical sort

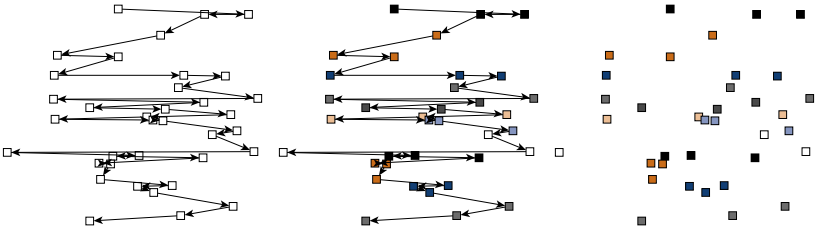
At first, a node $v \in V_{CLB}^{layout}$ that is below another node $u \in V_{CLB}^{layout}$ in $\mathcal{G}_D^{layout}$ *may not* be placed *over* v in the final embedding on the architecture in $\mathcal{G}_D^{arch}$. If $\tilde{v}, \tilde{u} \in V_{CLB}^{arch}$ are the embedded nodes with integer coordinates, then $y(v) < y(u) \Rightarrow y(\tilde{v}) \leqslant y(\tilde{u})$ should hold true. Therefore, the set of CLB nodes V_{CLB}^{layout} is arranged in a linked list (the `CLBNodeList`) which is first *sorted ascendingly* according to all nodes' *vertical coordinates*. The resulting list for the depicted example is shown in Figure 53a.

At this stage, successive nodes in the `CLBNodeList` have ascending vertical coordinates. In the next step, this list is *partitioned* into disjoint subsets of nodes R_i whereas all nodes of one subset R_i (depicted by the coloring in Figure 53b) will be placed in the same row r_i of CLBs on the FPGA architecture (with $\bigcup_i R_i = V_{CLB}^{layout}$). For the partitioning, different desired `CLBDistributions` can be chosen.

- CENTER DISTRIBUTION - The *CENTER* distribution places the CLBs densely in the center of the FPGA. For this purpose, the smallest central square field of size $N_{SQUARE} \times N_{SQUARE}$ on the architecture containing a sufficient number of CLB slots ($>$ #CLBs) is chosen and the `CLBNodeList` is partitioned into N_{SQUARE} respective sets of nodes.

Remark 82. *In case of an homogeneous architecture, this field is of size* $\left\lceil \sqrt{\#CLBs} \right\rceil \times \left\lceil \sqrt{\#CLBs} \right\rceil$. *However, due to special blocks on the architecture, this size may vary (see Figure 56a as an example). In this example, the centered square is of size* 19×19 *as there are only* 14 *CLB columns in the region to finally gather the* 257 *CLBs of the design.*

- NO, EQUAL & DISTANCE DISTRIBUTION - *All other* distributions place the CLBs consistently across the rows of the FPGA. Thus, the number of elements per row is calculated as $\left\lceil \frac{\#CLBs}{\#CLBRowsOnArch} \right\rceil$. The last used row may consequently get a smaller number of *CLBs* and some final rows on the architecture may remain empty due to rounding. Under this assumption, the partitioning of all CLBs into the group that will be placed in the first row of the FPGA (R_1), the second row (R_2), etc. is defined (see Figure 53c) and, therefore, the assignment of *vertical coordinates* in $\mathcal{G}_D^{arch}$ can be conducted.



(a) Vertically sorted `CLBNodeList` (b) Partitioned `CLBNodeList` (c) Resulting partitioned set of CLBs

Figure 53: Vertical sorting and slicing of the `CLBNodeList` (code: `diffeq1`)

Remark 83. *For the **CENTER** distribution, empty sets* R_k *appear for*

$$k = 1, \ldots, \left\lfloor \frac{\#LOGICRowsOnArch - N_{SQUARE}}{2} \right\rfloor$$

and for

$$k = \left\lfloor \frac{\#LOGICRowsOnArch - N_{SQUARE}}{2} \right\rfloor + N_{SQUARE} + 1, \ldots, \#LOGICRowsOnArch \quad .$$

The next step additionally generates the *horizontal coordinates* of the nodes in $\mathcal{G}_D^{arch}$ within each row following the different *distribution strategies*.

Horizontal sort

For the horizontal assignment of the nodes, each packed row R_i (*slice*) is independently considered and the subset of the linked list is sorted by its nodes' horizontal coordinates from $\mathcal{G}_D^{\text{layout}}$ (see Figure 54b). After this step, $x(v) < x(u) \Rightarrow x(\tilde{v}) < x(\tilde{u})$ will hold true *for every pair of nodes u, v from the same row*. In addition to their pure horizontal order, *penalties* (*free slots*) may be placed between nodes following the *distribution strategies*.

- CENTER DISTRIBUTION - To center each row, the first CLB is placed with an offset of *ColumnOffset* logic blocks to leave a *free margin* on the left and on the right (like it was left at the top and the bottom).

$$\text{ColumnOffset} = \left\lfloor \frac{\#\text{LOGICColumnsOnArch} - N_{\text{SQUARE}}}{2} \right\rfloor$$

The nodes of each row are placed from left to right onto the next free CLB slot and columns with heterogeneous 'special' block types (see Section 2.3) are skipped if they appear.

- NO DISTRIBUTION - The *NO* distribution uses all rows of the FPGA (see the previous paragraph). The CLBs are simply placed from the left to the right onto the next free CLB slot with no free CLB slots between them.

Remark 84. *This **NO** distribution is the baseline for the* EQUAL *and* DISTANCE *distribution which instead add free CLB slots within the rows.*

- EQUAL DISTRIBUTION - The *EQUAL* distribution aims at spreading the CLBs of each row evenly within this row. For this purpose, a *penalty counter* p is increased by $\frac{\#\text{CLBsToBePlacedInThisRow}}{\#\text{CLBColumnsOnArch}}$ after placing a CLB. Whenever this counter p becomes equal to or greater than 1, $\lfloor p \rfloor$ CLB slots are skipped and p is updated to $p - \lfloor p \rfloor$. This guarantees that all CLBs can be placed within the row and that the available penalties are approximately evenly distributed across the row.

- DISTANCE DISTRIBUTION - This is the *core distribution* of the `FieldPlacer`. Instead of distributing the penalties equally, free CLB slots between blocks of a row are assigned according to the respective open spaces in the force-directed layout $\mathcal{G}_D^{\text{layout}}$. Larger spaces between two nodes u, v in the layout are supposed to result in multiple free slots between the embedded nodes $\tilde{u}, \tilde{v}$ and vice versa. To realize this, the *minimal* and *maximal* horizontal coordinate is extracted from $\mathcal{G}_D^{\text{layout}}$ to derive the overall width of the set V_{CLB}^{layout} as shown in equation (56).

$$\text{width}_{V_{CLB}^{\text{layout}}} = \max_{\forall v \in V_{CLB}^{\text{layout}}} \{x(v)\} - \min_{\forall v \in V_{CLB}^{\text{layout}}} \{x(v)\}. \tag{56}$$

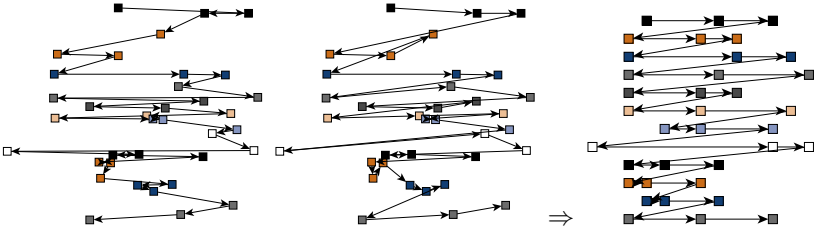
For each row R_i, the number of

$$\texttt{\#FreeCLBsInThisRow} = \texttt{\#CLBColumnsOnArch} - \texttt{\#CLBsToBePlacedInThisRow}$$

should now be distributed according to *horizontal distances* in $\mathcal{G}_D^{layout}$. Therefore, each *unit of distance* between two successive nodes in a row is basically penalized with $\frac{\texttt{\#FreeCLBsToBePlacedInThisRow}}{width_{V_{CLB}^{layout}}}$ free slots. The same is done for the initial free space in the row to the left border of the chip architecture. For two successive nodes in the `CLBNodeList` v_j and v_{j+1}, the penalty counter p is consequently increased by p_j, see equation (57).

$$p_j = (x(v_{j+1}) - x(v_j)) \cdot \frac{\texttt{\#FreeCLBsInThisRow}}{width_{V_{CLB}^{layout}}} \,. \tag{57}$$

Like for the *EQUAL* distribution, $\lfloor p \rfloor$ CLB slots are left free between v_j and v_{j+1} and p is updated to $p - \lfloor p \rfloor$. By this strategy, the distribution of nodes in $\mathcal{G}_D^{layout}$ is '*imitated*' on the architecture and therefore in $\mathcal{G}_D^{arch}$. Figure 54c shows the outcome of an embedding with the *DISTANCE* distribution. Figure 54 explicitly shows how the horizontal distances in Figure 54b are (*approximately*) imitated in the embedding in Figure 54c (by the *DISTANCE* distribution).



(a) Partitioned `CLBNodeList` (b) Horizontally sorted rows in `CLBNodeList` (c) Final embedded `CLBNodeList`

Figure 54: Horizontal sorting in the `CLBNodeList` (code: `diffeq1`)

Remark 85. *Currently, the* DISTANCE *distribution assigns the same number of nodes to each row R_i (except for the last row) and the layout-aware distribution on the architecture is solely realized by respective penalties between nodes within the rows. In the future, a more appropriate imitation of $\mathcal{G}_D^{layout}$ on the architecture could be achieved by a respective **vertical distribution** of nodes. For example, a histogram of the vertical distribution of nodes in $\mathcal{G}_D^{layout}$ could be calculated to partition the nodes more adaptively. Furthermore, other more sophisticated techniques are undoubtedly conceivable to consider **vertical penalties**.*

Remark 86. *The accumulation of penalties ensures that each basic penalty (fraction)* p_j *is legally inserted nearby the node* v_j*. Without the accumulation, smaller distances* ($p_j < 1$) *would simply be globally neglected. The next paragraph describes the motivations behind the different distribution strategies.*

Motivations of the distribution strategies

The *principal arrangement* of the nodes is *the same* for all presented distribution types as their relative position to each other is defined by their occurrence in $\mathcal{G}_D^{\text{layout}}$ and by the two-dimensional sorting. The inserted penalties follow different purposes.

The *CENTER* distribution places the CLBs densely in the center of the architecture. As the blocks are generally connected with multiple elements, nodes representing CLBs (just like for MEMs or MULs) have much higher node degrees than the I/O nodes on average. The CENTER distribution should therefore be well suited to keep the many *inner* connections small while enlarging only the fewer connections' lengths between I/O and inner logic blocks. Thus, the overall *wirelength* in the embedding should be relatively small. However, the *overuse* and the *maximal channel occupancy* can instead be expected to be relatively high as the connections have no good chance for low-stress detours. Thus, the routing time for a placement with the CENTER distribution is extended by the ripup and reroute phases (see Section 5.3.2). On the other hand, the expanded waves are relatively small so that creating the individual routes between two points should be possible in relatively short times.

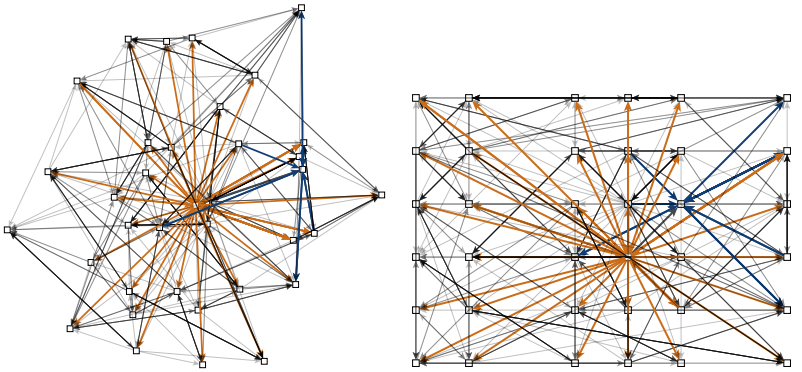
The *EQUAL* distribution should work in the contrary way. Due to the free spaces between the CLBs, the routing should be more 'relaxed' because of many opportunities for detours. However, the distances between nodes are relatively large so that the overall *wirelength* should consequently become large and the routing of the individual connections requires more time. The *EQUAL* distribution is intended to produce an even dispersion of the nodes among the architecture, like it was desired by the partitioning technique in GORDIAN (see Section 5.1.2).

Finally, the *DISTANCE* distribution is the main objective of the `FieldPlacer` method. It imitates the free (or *unconstrained*) situation in the force-directed layout with repulsive forces and should therefore find a good balance (*equilibrium*) between the two *extreme* options mentioned before. For example, regions with many nodes are expanded by the repulsive forces and *weakly* connected nodes can be carried further away from others due to small acting attractive forces.

Remark 87. ***One of the main ideas of this work is that the* equilibrium state *of such a force-directed layout with repulsive forces results in a profitable trade-off between wirelength and overuse.***

The *NO* distribution is primarily shown for comparison purposes. It has exactly the same order of nodes in each row than both the EQUAL and the DISTANCE distribution. Results for *this* distribution strategy can therefore be considered to investigate the impact of the introduced penalties on different measures of quality.

Figure 55 shows the embedding of the CLBs ($\mathcal{G}_{\text{CLB}}^{\text{layout}}$ in Figure 55a and $\mathcal{G}_{\text{CLB}}^{\text{arch}}$ in Figure 55b) onto the architecture with the CENTER distribution. The connections of two strongly connected nodes are highlighted in blue and orange. The two nodes themselves are connected by the gray edge. The figures show how the relative position of nodes to each other is preserved by the CLB embedding technique presented in this Section. Algorithm 10 shows a summarizing pseudo-code of the method. Examples for the embedding with the different distribution types are depicted in Figure 56.



(a) Force-directed layout result (b) Embedded on the grid

Figure 55: Embedding of the graph on the grid (code: `diffeq1`)

Remark 88. *The entire process was described with a constant number of CLBs per row. However, it is possible to have different numbers of CLBs in each row. Therefore, two arrays are used to store the information of how many CLBs are* ***available*** *and* ***used*** *per row, respectively.*

Algorithm 10 CLB placement

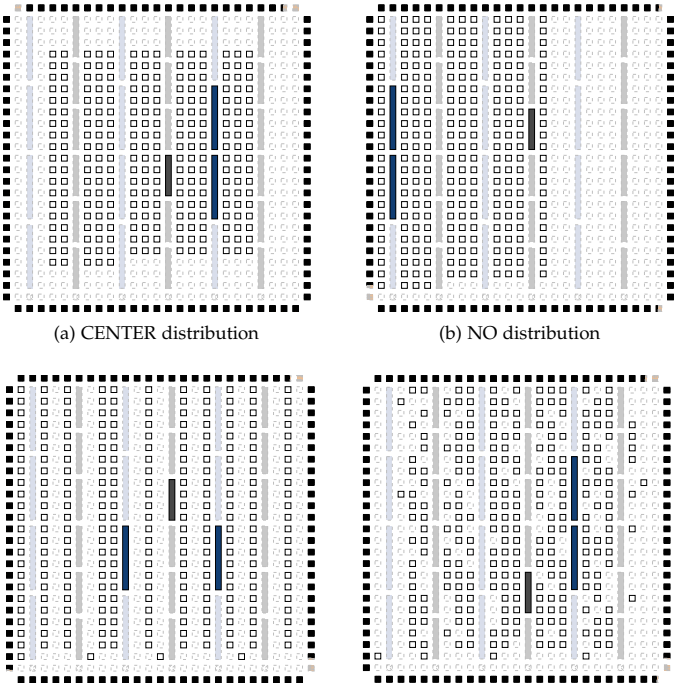
procedure createCLBplacement(Arch `FPGAArch` , Graph $\mathcal{G}_D^{layout}$, Enum `Option.dist_type`)
  linked list `CLBNodeList`← extract V_{CLB}^{layout} from $\mathcal{G}_D^{layout}$

  sort `CLBNodeList` ascendingly concerning **vertical coordinates**

  partition the list into rows R_i

  for all rows R_i **do**
    sort the nodes of R_i in `CLBNodeList` ascendingly concerning **horizontal coordinates**

    if `Option.dist_type==EQUAL` or `DISTANCE` **then**
      calculate penalties between nodes ▷ see the '*Horizontal sort*' subsection
      assign the sorted CLBs to repective CLB slots on `FPGAArch` skipping the free slots
    else
      assign the sorted CLBs to successive CLB slots on `FPGAArch`
    end if
  end for
  return the CLB slot assignment in the `CLBNodeList`
end procedure



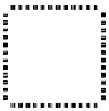
(a) CENTER distribution (b) NO distribution

(c) EQUAL distribution (d) DISTANCE distribution

Figure 56: Embedding with different distribution strategies (code: `or1200`)

5.5.4 *4th Step: I/O placement*

I/Os . . . ▷



Basic I/O partitioning

To be able to create the embedding of the I/O pins from the graph layout, the I/O *nodes* and their coordinates $V_{I/O}^{layout}$ are extracted from $\mathcal{G}_D^{layout}$ and stored in a linked list, the `IONodeList`. In addition, for each node in the list, the angle ω to the barycenter of the graph (BC) is stored as a further parameter. This angle is rotated to start (and end) with $-\pi$ (and π) in the *north-west*.

The I/O nodes are basically grouped into *four faces* by sorting them according to their ω-parameter. This results in a *clockwise enumeration* of the nodes in the `IONodeList` (beginning in the north-west). For the basic partitioning into the four I/O faces of the FPGA architecture, each node is assigned to either the ***N**orth* ($-\pi < \omega(v) \leqslant -\frac{\pi}{2}$), the ***E**ast* ($-\frac{\pi}{2} < \omega(v) \leqslant 0$), the ***S**outh* ($0 < \omega(v) \leqslant \frac{\pi}{2}$) or the ***W**est face* ($\frac{\pi}{2} < \omega(v) \leqslant \pi$), solely based on its position in $\mathcal{G}_D^{layout}$. This process is formally described in Algorithm 11.

Capacity legalization

Even though the FPGA contains enough I/O pins to satisfy the code's demands (ensured by the setup of the architecture, see Section 5.3), some faces may be overfull while others still have free capacities due to the simple partitioning concerning their angle to the *BC* described in the previous section. In that case, nodes should be redistributed into neighboring faces so that, in the end, no face is overfull.

Remark 89. *For the partitioning of the (sorted) list, only the indices that contain the last node of each face have been stored as the* ***split points*** *of faces in the list (see Algorithm 11).*

For the legalization of the capacities in the I/O faces, the following technique is applied.

First, traverse the sorted list's `split_points` *clockwise* (`North.end`, `East.end`, `South.end`). If the capacity of the prospected face is exceeded (`this.nb_nodes > this.capacity`), the exceeding amount of nodes is transferred from the end of `this` face to the beginning of the `next` face (*clockwise*) by adjusting the `split_point this.end` (last node's index of this face) for the sorted `IONodeList`. After this, the `West` face may still be overfull as it could, e. g., have 'received' exceeding nodes from the `South` face.

Therefore, the linked list's `split_points` are then traversed *anticlockwise* (`South.end`, `East.end`, `North.end`) and the number of nodes that exceed the `capacity` of the prospected face are passed to the *anticlockwise* neighbor by adjusting `prev.end`.

Algorithm 11 Extract basic I/O partitioning

procedure CREATEBASICIOPARTITIONING(Graph $\mathcal{G}_D^{layout}$)
linked list `IONodeList`← extract $V_{I/O}^{layout}$ from $\mathcal{G}_D^{layout}$

$x(BC) = \frac{1}{\left|V_{I/O}^{layout}\right|} \sum_{v \in V_{I/O}^{layout}} x(v)$ ▷ calculate the graph's **barycenter**

$y(BC) = \frac{1}{\left|V_{I/O}^{layout}\right|} \sum_{v \in V_{I/O}^{layout}} y(v)$

for all nodes v in $V_{I/O}^{layout}$ **do**
$\Delta x = x(BC) - x(v)$ ▷ get the **distance vector** from v to BC
$\Delta y = y(BC) - y(v)$

▷ **rotate** the node temporarily to start enumerating in the north-west
$\alpha = -\frac{3\pi}{4}$
$\Delta\hat{x} = \Delta x * cos(\alpha) + \Delta y * sin(\alpha)$
$\Delta\hat{y} = -\Delta x * sin(\alpha) + \Delta y * cos(\alpha)$

▷ **calculate** the α-rotated angle to the rotation point
$\omega(v) = atan2(\Delta\hat{y}, \Delta\hat{x})$

$$\text{with} \quad atan2(\Delta\hat{y}, \Delta\hat{x}) = \begin{cases} \arctan\left(\frac{\Delta\hat{y}}{\Delta\hat{x}}\right) & \Delta\hat{x} > 0 \\ \arctan\left(\frac{\Delta\hat{y}}{\Delta\hat{x}}\right) + \pi & \Delta\hat{x} < 0, \Delta\hat{y} \geqslant 0 \\ \arctan\left(\frac{\Delta\hat{y}}{\Delta\hat{x}}\right) - \pi & \Delta\hat{x} < 0, \Delta\hat{y} < 0 \\ +\frac{\pi}{2} & \Delta\hat{x} = 0, \Delta\hat{y} > 0 \\ -\frac{\pi}{2} & \Delta\hat{x} = 0, \Delta\hat{y} < 0 \\ \text{undefined} & \Delta\hat{x} = 0, \Delta\hat{y} = 0 \end{cases}$$

end for

sort `IONodeList` ascendingly concerning ω

▷ **partition** the list by finding the list's (the faces') `split_points` (indices)
`North.end` ← index of first element with $\omega > -\frac{\pi}{2}$
`East.end` ← index of first element with $\omega > 0$
`South.end` ← index of first element with $\omega > \frac{\pi}{2}$
return the `IONodeList` and the `split_points`
end procedure

An example for the *number of nodes in each face* in the iterative traversals is given in the following table. The last column shows how many nodes were transferred (*in brackets*) and between which faces. The initial situation is depicted in Figure 57.

N	E	S	W	
177	**206**	195	200	N → E (0)
177	**200**	**201**	200	E → S (6)
177	200	**200**	**201**	S → W (1)
177	200	**201**	**200**	W → S (1)
177	**201**	**200**	200	S → E (1)
178	**200**	200	200	E → N (2)

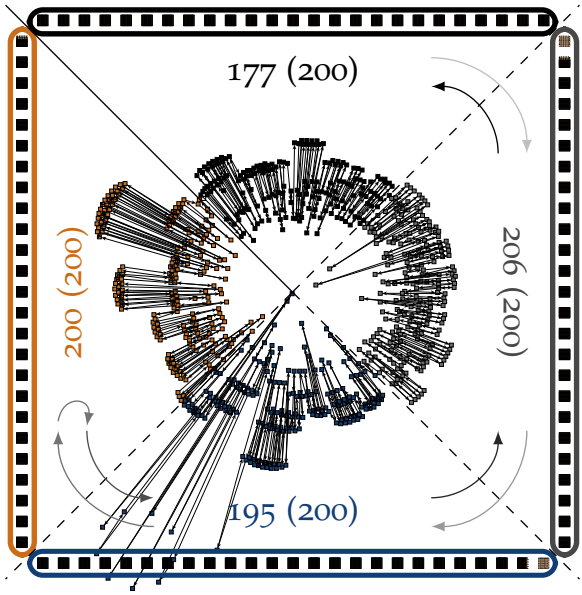


Figure 57: I/O legalization (code: `or1200`)

After both traversals, it is *guaranteed* (by construction of the method and the architecture) that *all faces' capacities are respected*. Finally, the set of I/O nodes in each face is assigned clockwise to the faces *slots* on the architecture (I/O nodes from the sorted `IONodeList` are successively assigned to the `North`, `East`, `South` and `West` face), whereas each integer I/O block on the architecture generally comprises multiple I/O pins (*eight* for the depicted architecture, see Section 5.3.1). The nodes have z-coordinates to distinguish between each blocks' *pins*. Thus, groups of eight successive nodes in the sorted `IONodeList` always have the same *horizontal* and *vertical coordinate*. In each face, the set of nodes is additionally centered so that free I/O pins in a face are equally distributed to the outer ends of the face.

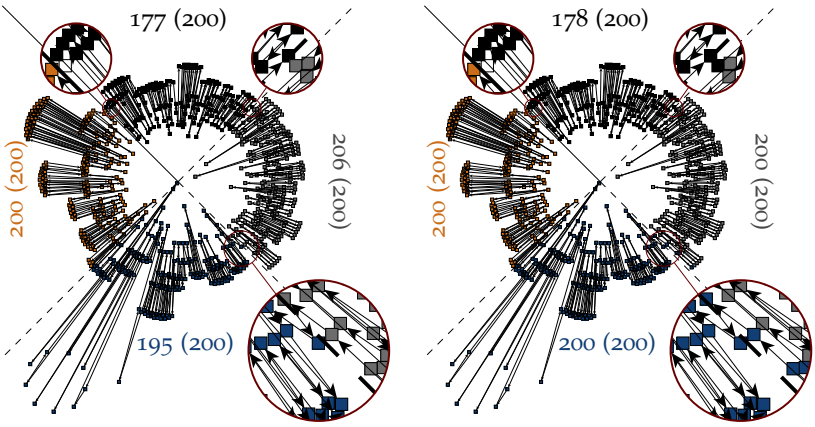
Remark 90. *This technique locally transfers as many nodes as necessary but as few as possible to neighboring faces (see Algorithm 12). The* ***time complexity*** *of the legalization method only depends on the number of faces and is thus* ***constant*** *for given architectures (independent from the design).*

Figure 58a shows the `IONodeList` and the partitioning after calling `create BasicIOpartitioning` while Figure 58b depicts the final *legal* partitioning after calling the `legalizeIOpartitioning` technique (with final *I/O slot assign-*

ment). The different faces are illustrated by different colors and the **white** node in the north-west is the start of the α-rotated `IONodeList`.

Algorithm 12 I/O legalization

```
procedure LEGALIZEIOPARTITIONING(Arch FPGAArch, NodeList* IONodeList, IndexList* split_points)
    extract capacity constraints of the I/O faces from FPGAArch

    for this=North, East, South do                                  ▷ traverse clockwise
        if this==North then prev.end = 0
        end if
        this.nb_nodes = this.end - prev.end
        if this.nb_nodes > this.capacity then
            this.end -= (this.nb_nodes - this.capacity)
        end if
    end for

    for West, South, East do                                        ▷ traverse anticlockwise
        if this.nb_nodes > this.capacity then
            prev.end += (this.nb_nodes - this.capacity)
        end if
    end for

    ▷ I/O slot assignment
    Place the I/O nodes of each face centered to the face by traversing
    the angle-sorted IONodeList splitting at the updated split_points

    return the legalized I/O slot assignment with updated split_points and the IONodeList
end procedure
```



(a) *before* legalization

(b) *after* legalization

Figure 58: I/O legalization (code: `or1200`)

Remark 91. *The close-ups in Figure 58 show how the **six** exceeding nodes from the* `East` *face in the list were finally transferred to the* `North` *face (**one** node) and the* `South` *face (**five** nodes).*

In the slot assignment obtained from the `legalizeIOpartitioning` method (which uses the force-directed graph layout $\mathcal{G}_{\mathrm{D}}^{\mathrm{layout}}$ as its base just like the `createCLBplacement` method), I/O pins that are connected to 'upper' regions of the CLB placement from the previous step (see Section 5.5.3) tend to be assigned to the *northern I/O face*, I/O pins that connect to the 'lower' part of the CLB placement are in the *southern I/O face*, etc. Thus, the methods lead to short connections from the *outer* I/O pins to the *inner* CLBs of the FPGA.

A resulting placement graph $\mathcal{G}_{\mathrm{CLB+I/O}}^{\mathrm{arch}}$ with *CLB* and *I/O* nodes is shown in Figure 59. Notice that multiple I/O nodes are on identical positions *over each other*.

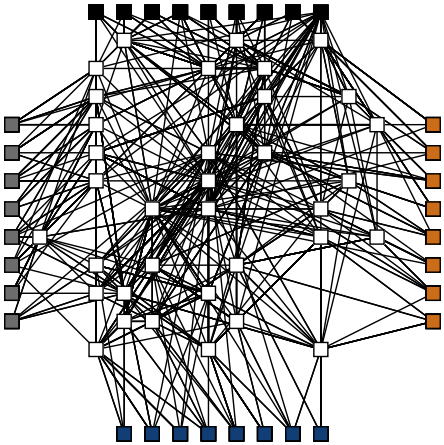


Figure 59: I/O and CLB placement (code: `diffeq1`)

I/O refinement with the barycenter heuristic

The relative positioning of CLB and I/O nodes to each other has essentially been optimized by the force-directed graph layout in $\mathcal{G}_{\mathrm{D}}^{\mathrm{layout}}$. Even though the introduced embedding of both the *CLBs* and the *I/Os* is principally preserving the relations of positions in the graph layout, the *detailed* positioning is inevitably disturbed by the embedding on the restricted integer grid. To take the actual embedding of the CLBs from the preceding step into account, an additional optimization step is performed after the *I/O legalization*.

The idea is to take the legalized I/O partitioning and rearrange the I/O pins in each face with a fast heuristic to minimize *wirelengths* and *wire-crossings*. Minimizing crossings can be very beneficial as the short connections between outer CLBs and I/O faces usually do not have too many detour possibilities in heavily used regions of the routing architecture.

The problem of minimizing the wire-crossings between I/O nodes and CLB nodes is basically a *'(one-sided) bilayer straightline crossing minimization problem'*. The *CLB nodes* of all *CLB*-to-*I/O*-connections define the *fixed layer* of the problem and the I/O nodes are on the *free layer*. The fact that the CLB nodes are not necessarily on one horizontal line can be neglected. Unfortunately, the problem is known to be $\mathcal{NP}$-hard (see Eades and Whitesides [52]). However, good heuristics have been developed. In this work, the *barycenter-heuristic* is applied.

To each node on the *free* layer, the barycenter heuristic assigns the *arithmetic mean* of the horizontal coordinates of connected *fixed* nodes (CLBs) and subsequently sorts the *free* nodes (I/Os) according to this average value.

Remark 92. *This procedure can analogously be performed for **vertical** parallel layers.*

In their work from 1996, Jünger and Mutzel [106] investigated the quality of different heuristics for this problem comparing the results to optimal solutions obtained from a branch-and-cut solver. They came to the conclusion that the simple *barycenter heuristic* leads to surprisingly good results and that smaller instances can even be solved to optimality with their branch-and-cut solver.

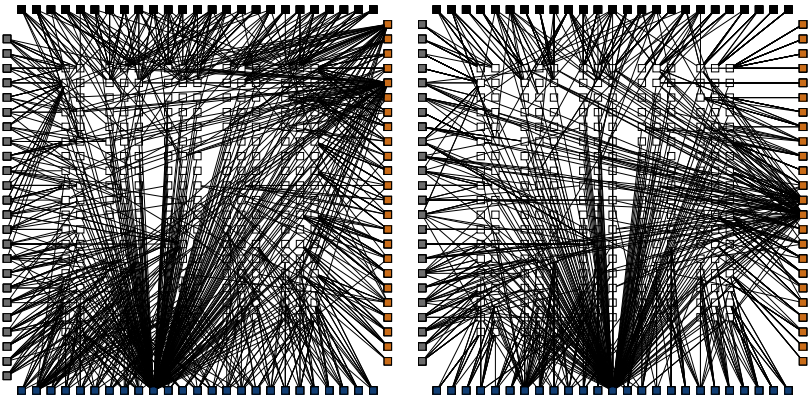
Remark 93. *In this work (and at this point in time), an exact solution is not considered at all as subsequent refinement-steps may disturb the placement and due to simplicity for the implementation and for short runtimes of the method.*

Figure 60 shows how the barycenter heuristic improves the I/O arrangement on the `North` face of the FPGA for one example code. Even though the assignment has already been quite good before, it is obvious that the application of the barycenter heuristic improves both wirelengths and wire-crossings. Connections from identical I/O blocks are 'bundled' by this procedure. Applying this heuristic to all four faces of the architecture (see Algorithm 13) leads to a *CLB-aware reordering* of the I/O nodes in each face. For the chosen example, the overall result is depicted in Figure 61. The improvement of wirelengths by the reordering can be measured by accumulating the pure *horizontal* (`North` and `South` face: $|x_{I/O}(v) - x_{CLB}(u)|$) or the pure vertical (`East` and `West` face: $|y_{I/O}(v) - y_{CLB}(u)|$) distance between connected nodes v (*I/O*) and u (*CLB*). For example, this displacement would be *zero* if a totally crossing-free arrangement is found where each I/O node connects to a CLB

(a) *before resort* (b) *after resort*

Figure 60: Connections from `North` I/Os to CLBs (code: `or1200`)

in the same column or row. The accumulated displacement in Figure 61a is 6963 while the application of the barycenter heuristic reduced the displacement of this (already quite decent) arrangement to only 3232. Considering the fact that there are 779 I/O nodes in the design (and therefore in $\mathcal{G}_D^{\text{layout}}$, cp. Table 12 on page 222), the average displacement per node was reduced from ~ 9 to ~ 4 through the application of the barycenter heuristic. Figure 62 shows the *improvement of the displacement* in the *CLB*-to-*I/O*-connections for *all codes in the heterogeneous benchmark set of VTR* (average of 10 repeated runs neglecting the *minimum* and *maximum*).



(a) *before resort* (b) *after resort*

Figure 61: Connections from I/Os to CLBs (code: `or1200`)

Summarizing, the barycenter heuristic can improve the placement of the I/O nodes after fixing the CLBs significantly, especially for codes with relatively high numbers of I/O blocks. Due to the centering in the I/O face, codes with only very few I/O connections can naturally not be improved too much. However, in such designs it is also not that important. After this *fourth step*, both the *CLB blocks* and the *I/O pins* are set to their *final* posi-

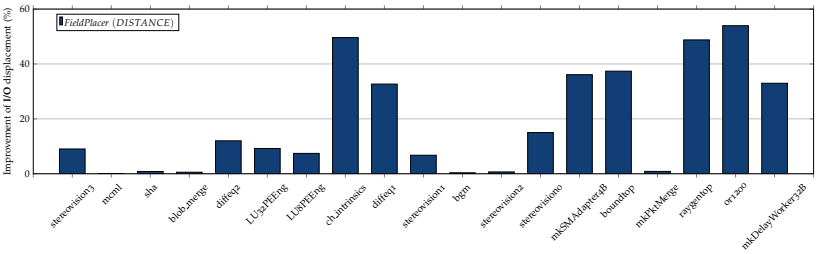


Figure 62: Improvement of I/O displacement by the barycenter heuristic (*codes sorted ascendingly by number of I/O nodes*)

tion in the *basic* `FieldPlacer` method (see Figure 63). Subsequently, final optimizing exchanges may (possibly) only take place in the *second energy phase* (Section 5.6.1) or the *local refinement* (see Section 5.6.3).

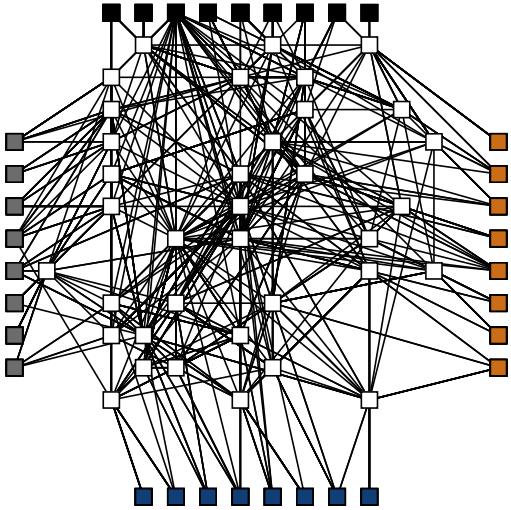


Figure 63: Final CLB and I/O placement (code: `diffeq1`)

Multiple components in the design

As already mentioned in Section 5.5.1, the *designs* and therefore the representing graph $\mathcal{G}_D$ may have more than one component (*unconnected subgraphs*). However, in the observed benchmark examples there is always a predominant graph component (in terms of number of nodes, cp. Appendix A.9).

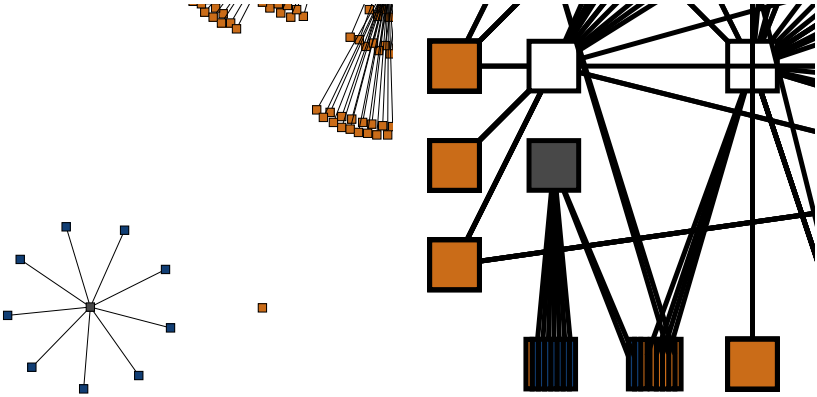
Algorithm 13 I/O refinement by barycenter heuristic

procedure IMPROVEIOARRANGEMENT(Graph $\mathcal{G}_D^{\text{layout}}$, NodeList `CLBNodeList`, NodeList `IONodeList`)
  extract NodeConnections E between I/Os and CLBs from $\mathcal{G}_D^{\text{layout}}$
  for `face=North,South` **do**
    for all nodes v in this `face` **do**
      extract coordinates of CLBs and I/Os from the `CLBNodeList` and the `IONodeList`
      $\overline{x}_{I/O}(v) = \frac{1}{\delta(v)} \sum_{(u,v)\in E} x_{CLB}(u)$ ▷ see Definition 13
    end for
    sort the `face`'s part of the `IONodeList` ascendingly concerning $\overline{x}(v)$
    assign updated coordinates to the nodes with respect to their sorted order
  end for
  for `face=East,West` **do**
    for all nodes v in this `face` **do**
      extract coordinates of CLBs and I/Os from the `CLBNodeList` and the `IONodeList`
      $\overline{y}_{I/O}(v) = \frac{1}{\delta(v)} \sum_{(u,v)\in E} y_{CLB}(u)$ ▷ see Definition 13
    end for
    sort the `face`'s part of the `IONodeList` ascendingly concerning $\overline{y}_{I/O}(v)$
    assign updated coordinates to the nodes with respect to their sorted order
  end for
  return the refined coordinates in the `IONodeList`
end procedure

The FM^3 algorithm has a powerful postprocessing step to combine the individual layouts of all components of a graph to a compact common drawing (see Hachul [81]). The entire placement of such designs in the `FieldPlacer` method is directly realized based on this common drawing. Figure 64a shows a second (small) component of the `or1200` design after the force-directed graph layout in $\mathcal{G}_D^{\text{layout}}$ including I/O and CLB nodes (cp. the entire graph in Figure 51b on page 162). After the embedding, the component has been compactly placed onto the FPGA architecture (see Figure 64b) with short connections. Due to the separate layouting of the components with subsequent consolidation, each component gets a *virtually separate part* of the chip architecture with short resulting connections on the routing architecture.

Remark 94. *The **basic** `FieldPlacer` method assumes a free (unconstrained) assignment of all nodes on the architecture. However, if there are (user-)fixed blocks in the design, they can also be taken into account as the enhanced `FieldOGDF` implementation can handle such **fixed nodes**. This is exploited in the **second energy phase** in Section 5.6.1 and further explained in Section 5.9.1. Thus, inputs with a priori fixed nodes can be considered in the `FieldPlacer` method in the future. All requisite preconditions for these procedures are already implemented, even maintaining the important multilevel functionality. In the case of a priori fixed I/O nodes, an initial free force-directed layout could be performed to extract a good scaling of the nodes' positions for the forces in the subsequent simulation.*

*The I/O assignment technique of the `FieldPlacer` can also be used for other numerical force-directed approaches basing on equation systems which **require a priori fixed I/O nodes** (like those presented in Section 5.1). In such cases, the `FieldPlacer`*



(a) Graph Layout with two components (b) Placement of the two components

Figure 64: Multiple graph components (code: `boundtop`)

could be used to create a good I/O assignment for other placers rapidly. The improvement achieved by the use of the barycenter heuristic alone shows that a free positioning of the I/O nodes can actually be quite beneficial (cp. also Betz et al. [21, Section 5.4.3]).

Nevertheless, fixed I/O blocks also appear in certain situations in 'productive design flows' and random pin assignment is a well known issue for FPGA designs (see Betz et al. [21, Section 3.2.1]). The application of the proposed ***rapid*** `FieldPlacer` *I/O assignment could be considered in the future in such cases (even without all the other embedding techniques of the* `FieldPlacer`*).*

In addition, different I/O block types - like mentioned in the publication of Mak and Li [133] - could be considered by performing a separate clockwise enumeration for each I/O type and subsequently splitting the `IONodeList`*s with respect to the architecture's equipping.*

Finally, a splitting of the clockwise enumerated `IONodeList` *into* ***less than four faces*** *can easily be performed, e. g., in the case that only a small fraction of a large FPGA is used for a design which is placed in one corner of the chip. In such cases, it can be desirable to connect only to the two nearby I/O faces. Due to the fact that the architecture is rather densely filled in the presented benchmarks (due to the bisection-based architecture creation), this situation does not occur in the investigations of this work. However, some designs with only few I/Os (compared to the number of CLBs) already use only some of the I/O faces with the introduced I/O partitioning technique.*

5.5.5 *5th Step: Special blocks (MEM+MUL) placement*

MULs & MEMs . . . ▷

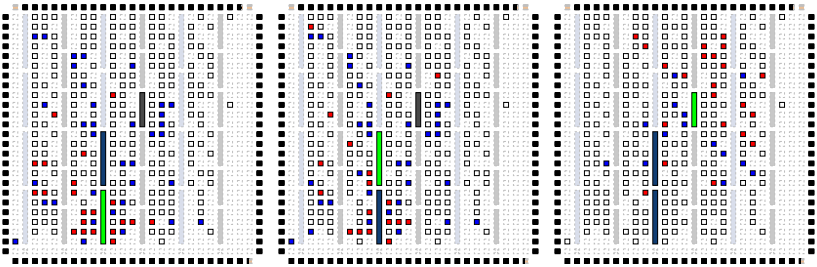
Finally, further heterogeneous block types are placed. Two main characteristics of such blocks in common designs are used. First, they appear only in *relatively small numbers* and, second, they generally have a *high node degree* (as they are strongly connected due to many in- and out-pins). This is due to their high inner complexity which is the essential motivation to use such *special* elements in a design. Each of these *special blocks* is now placed after the *general purpose elements* of the FPGA (*CLBs* and *I/Os*) have been assigned to suitable locations. For this purpose, all *memory blocks* (*MEMs*) are first extracted from the force-directed layout $\mathcal{G}_D^{\text{layout}}$ and stored in a linked list together with the information about connected *I/O* and *CLB* nodes for each of those. Then, the barycenter $(\overline{x}, \overline{y})$ of all such connected nodes' coordinates on the architecture (taken from $\mathcal{G}^{\text{arch}}_{\text{CLB+I/O}}$) is calculated for each memory block and the number of connections (the *node degree*) is stored. The memory blocks are subsequently *sorted descendingly* according to their node degrees. Finally, they are - *in their sorted order* - assigned to the positions that are nearest to the beforehand calculated barycenters of connected nodes (that have already been embedded on the architecture) until all blocks have been placed. The same procedure is performed independently for the *multiplicators* (*MULs*).

Remark 95. *Connections between different special types of blocks could also be considered for later placed special blocks.*

As already mentioned, these special elements are placed after all other '*ordinary*' ones. The idea is to assign them the *best suitable available place* with respect to the placed CLBs and I/Os and to priorize such elements that influence many connections on the chip to keep the *overall wirelength small*. Due to the fact that the number of such elements is nowadays still relatively small, a pretty naive implementation of the assignment is used (see Algorithm 14). In the future, more advanced techniques, for example, basing on a *quadtree* to find the nearest available slot, can be applied (see Section 5.7).

Remark 96. *To emphasize this fact once again, even though the number of heterogeneous blocks may be small, they often have many in- and output pins and, thus, their influence is important and can become crucial for the wirelength in the placement and for good routability. It is therefore reasonable to place these nodes with respect to all already placed blocks taking their final coordinates from $\mathcal{G}_D^{arch}$ into account.*

Figure 65 shows the three heterogeneous elements in the `or1200` design after embedding them with the presented method. The heterogeneous block itself is highlighted in green while *fanouts* and *fanins* to the block (cp. Section 2.2.4) are shown in *red* and *blue*, respectively. The final position in the barycenter of connected nodes is clearly visible.



(a) MEM block 1 (b) MEM block 2 (c) MUL block 1

Figure 65: Placement of special blocks (code: `or1200`)

After these 5 phases, *all* elements are placed on the FPGA and the *basic* `FieldPlacer` placement $\mathcal{G}_{\mathrm{D}}^{\mathrm{arch}}$ is produced by combining $\mathcal{G}_{\mathrm{CLB}}^{\mathrm{arch}}$, $\mathcal{G}_{\mathrm{I/O}}^{\mathrm{arch}}$ and $\mathcal{G}_{\mathrm{MEM+MUL}}^{\mathrm{arch}}$. An example for such a complete placement with *distance penalties* (for the CLBs) is shown in Figure 65 for the `or1200` code while Figure 66 depicts such a placement for the smaller `diffeq1` design (cp. also Figure 63). Figure 67 finally illustrates the overall workflow of the *basic* `FieldPlacer` method.

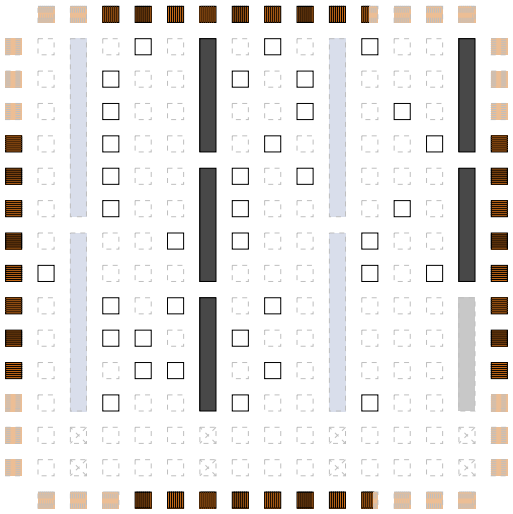


Figure 66: Placement of all elements with the *basic* `FieldPlacer` method

Algorithm 14 Special heterogeneous blocks' placement

procedure EMBEDSPECBLOCKS(Arch `FPGAArch`, Graph $\mathcal{G}_D^{layout}$, Graph $\mathcal{G}_{CLB+I/O}^{arch}$)

linked list `MEMNodeList` ← extract V_{MEM}^{layout} from $\mathcal{G}_D^{layout}$
linked list `MULNodeList` ← extract V_{MUL}^{layout} from $\mathcal{G}_D^{layout}$

linked list `MEMSlots` ← extract all *memory slots* from `FPGAArch`
linked list `MULSlots` ← extract all *multiplicator slots* from `FPGAArch`

sort the `MEMNodeList` **descendingly** concerning the nodes' degree in $\mathcal{G}_D^{layout}$
sort the `MULNodeList` **descendingly** concerning the nodes' degree in $\mathcal{G}_D^{layout}$

for all $v \in$ `MEMNodeList` **and for all** $v \in$ `MULNodeList` **do**
extract (x, y) coordinates of connected CLBs and I/Os from $\mathcal{G}_{CLB+I/O}^{arch}$

▷ calculate the barycenter of all connected and embedded elements $(\overline{x}(v), \overline{y}(v))$
$\overline{x}(v) = \frac{1}{\delta(v)} \sum_{(u,v) \in E_D} x(u)$ ▷ see Definition 13
$\overline{y}(v) = \frac{1}{\delta(v)} \sum_{(u,v) \in E_D} y(u)$ ▷ see Definition 13
▷ calculate the distances of the barycenter to each available slot
for all suitable slots $(x(s), y(s))$ on `FPGAArch` **do**
$\texttt{dist}(s) = ||(\overline{x}(v), \overline{y}(v)) - (x(s), y(s))||_p$ ▷ see Section 5.6.2
end for

▷ embed the special block
select slot s with minimum `dist`
assign coordinates of s to v in `MEMNodeList` or `MULNodeList`
remove slot s from the respective list (`MEMSlots` or `MULSlots`)
end for
return coordinates for all special blocks in `MEMNodeList` and `MULNodeList`
end procedure

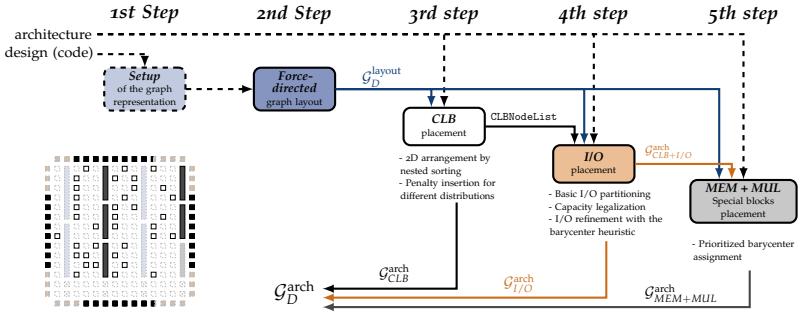


Figure 67: Overall workflow of the *basic* `FieldPlacer`

5.5.6 *Benchmark: Basic* `FieldPlacer`

In this section, results from the *basic* `FieldPlacer` are presented. Once again, all runs were repeated 10 times and the average over these runs is reported (neglecting the minimal and maximal value).

To get a rough overview, Figure 68 shows the average quality (over *all* 19 benchmark codes from Table 12 on page 222) that was achieved with the different distribution types concerning the *bounding box cost*, the *critical path delay*, the overall *wirelength* and the *overuse* (see Section 5.3.2 and Section 5.4). The quality is measured *relative* to results of the *simulated annealing* approach in *VPR* in its default configuration (see Appendix A.6). First of all, it is clearly visible that the quality of the *basic* `FieldPlacer` is (*on average*) inferior to the one of the simulated annealing results of VPR in *all* categories.

Remark 97. *As this is only the* ***basic*** `FieldPlacer` *approach, the absolute values of the quality are not too important right now. The measurements are primarily intended to show the relations between the different* `FieldPlacer` *options and the quality norms.*

However, there are distinct differences concerning the results of the different distribution strategies. While the *CENTER* distribution performs best concerning all measures that are principally related to the resulting *wirelengths* between connected nodes (namely the *bounding box cost*, the *critical path delay* and the overall *wirelength*), it achieves the *worst* result concerning the average *overuse*. This was expectable (see Section 5.5.3 - *Motivation of the distribution strategies*) as the densely packed *CENTER* distribution does not leave additional space for detours in the simulated 'stress-aware' routing of the *overuse* norm calculation (see Section 5.4). In general, the stress on the routing architecture is intensified by the dense assignment. The results for the *overuse* show that both the *EQUAL* and the *DISTANCE* distribution lead to significantly better results concerning this measure while the *EQUAL* distribution was expected to produce the placement with the lowest stress on the overall architecture beforehand due to the equally distributed free spaces between the CLBs.

It is interesting that the imitation of the force-directed graph layout through the *DISTANCE* distribution achieves the *second best average* results in *all* categories. It therefore combines the aforementioned positive characteristics of force-directed layouts with *attractive* and *repulsive* forces. The *attractive* forces tend to keep distances between connected nodes small (for small *bounding box cost*, *critical path delay* and overall *wirelength*) while the *repulsive* forces keep nodes away from each other and counteract highly dense regions in the layout (basically for reduced *overuse*).

Concerning all four categories, the results are in each case in similar ranges. This is based on the fact that the general arrangement of nodes to each other is the same for all distribution types (at least identical for all but the *CENTER* distribution and it is also rather similar for this one, see Section 5.5.3).

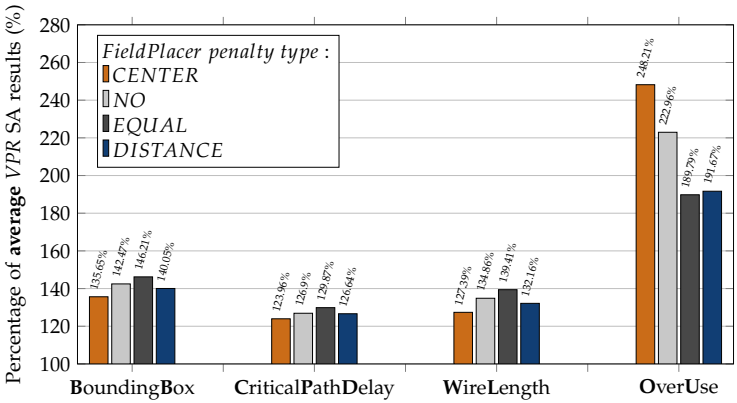


Figure 68: Pure `FieldPlacer` - Overview

Each reported value in Figure 68 contains the average quality of all 19 codes (each one run with 10 repetitions) and is therefore only outlining a rough **overview** of the situation. Figure 69 instead shows the average results for each code concerning *VPR*'s main optimization target, the *bounding box cost*. More precisely, the depicted values show how much the 10 different initial random assignments of each code were improved by the full *simulated annealing approach* in *VPR* and by the *basic* `FieldPlacer`. The measurements confirm that the *simulated annealing* results are still consistently better than the ones of the *basic* `FieldPlacer` but that the difference between both approaches vary from code to code. For the `ch_instrinsics` code, for example, both solutions are relatively close to each other concerning the bounding box cost while for the `mkPktMerge` code, the difference is much larger.

Figure 70 shows the correlation between all measurements for the different norms in scatter plots (again relative to the *VPR SA* results of each code, the colors match the distribution strategies from, e. g., Figure 68). A code achieving the same quality with the *basic* `FieldPlacer` than with the SA approach in VPR would consequently have a quality measure of *one* (100%). Without this *relative* measure, the comparison of different codes would, in general, not be meaningful. It is obvious that, for example, one code with 1000 times larger bounding box cost than another code will have a much larger absolute overall wirelength. However, the same would generally be the case for the compar-

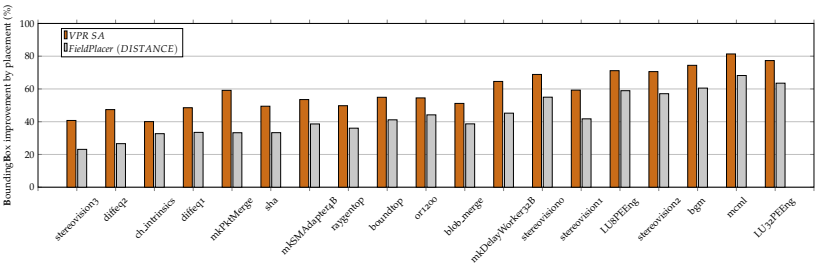
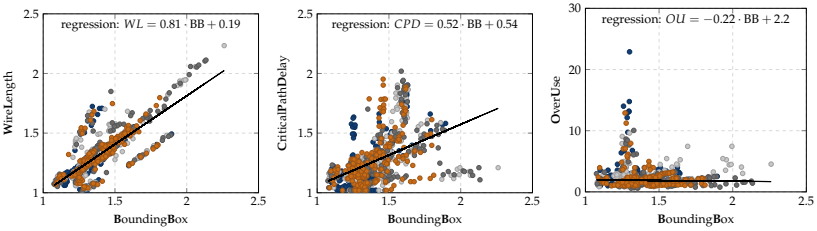


Figure 69: **B**ounding **B**ox cost improvement (*sorted ascendingly by* VPR SA *runtime*)

ison of *absolute* bounding box cost and *absolute* overuse as this measure is very *coarse-grained*. The relative measure for each code $\left(\frac{\texttt{FieldPlacer result}}{\textit{VPR SA result}}\right)$ instead answers the much more meaningful *fine-grained* question whether, e. g., a *larger bounding box cost* for *one code* due to the *distribution strategy* in the `FieldPlacer` does lead to an *appropriate increase* in the *overall wirelength*.



(a) **BB - WL** (b) **BB - CPD** (c) **BB - OU**

Figure 70: Pure `FieldPlacer` - Correlation

The results in Figure 70a and Figure 70b confirm the already assumed *positive correlation* between *bounding box cost*, the overall *wirelength* and the *critical path delay*. Figure 70c additionally shows that there is a *negative correlation* between these values and the resulting *overuse* in the system. However, due to higher outliers for this measure, the trend is not as 'clear' as in the other two plots. Finally, the *trendlines* (*linear regression*) of the scatter plots are shown in form of a black line and their mathematical function. The *bounding box cost* to *wirelength* comparison is in fact nearest to a *perfect* correlation (which would have to have a *slope* of *one*).

Besides the obtained quality of results, the *time that was needed* to produce the result (*runtime* in the following) plays an important role. Table 5 shows the average placer runtimes of *VPR*'s simulated annealing approach

and those of the *basic* FieldPlacer for the different codes along with the number of *nodes* and *edges* in $\mathcal{G}_D$, as the routines' runtimes directly depend on these values (see Section 3.1.2 and Section 5.7). Even though the quality of the *basic* FieldPlacer method is (*so far*) inferior to the *simulated annealing* approach of *VPR*, the *basic* FieldPlacer is up to 26 times faster than the *VPR SA* method. It can in fact play out its benefits concerning the runtime especially for larger inputs due to its relatively small runtime complexity (see Section 5.7).

Placer runtimes (s)	FieldPlacer	*VPR SA*	Nodes	Edges	Speedup
stereovision3	0.03	0.06	54	72	2.05
diffeq2	0.08	0.15	194	433	1.88
ch_intrinsics	0.12	0.20	267	560	1.64
diffeq1	0.14	0.26	299	661	1.84
mkPktMerge	0.23	0.34	497	614	1.48
sha	0.21	0.97	283	2793	4.64
mkSMAdapter4B	0.29	1.21	570	2373	4.18
raygentop	0.45	1.57	725	2648	3.49
boundtop	0.36	2.04	701	3327	5.68
or1200	0.62	2.68	1039	5048	4.33
blob_merge	0.60	4.29	679	9406	7.10
mkDelayWorker32B	1.20	5.62	1554	7178	4.68
stereovision0	0.85	8.49	1259	6223	9.93
stereovision1	0.98	9.23	1205	8579	9.39
LU8PEEng	2.45	32.65	2373	36616	13.31
stereovision2	3.39	46.70	2939	26002	13.79
bgm	3.89	53.90	3230	57686	13.86
mcml	8.27	220.68	6873	81390	26.67
LU32PEEng	11.57	287.83	7544	129453	24.87

Table 5: Comparison of average runtime in VTR 7.0 (*sorted ascendingly by* VPR SA *runtime*)

The following Sections will investigate the quality of the achieved *basic* FieldPlacer placement in more details. Section 5.6 then addresses further methodology to improve the *quality* of the *basic* FieldPlacer in the *extended* FieldPlacer method.

OverUse

The FieldPlacer congestion-driven maze router was introduced to simulate an 'ideal routing' without capacity restrictions on the routing architecture in order to extract a measure for the *routability* of a placement *before* the actual routing takes place (see Section 5.4.2).

While the reported relative *overuse* values in Figure 68 represent the average overuse sum on the architecture, Figure 71 shows the distribution of

overused routing resources for the `or1200` example code in *absolute values* (*best run out of ten*). It is clearly visible how the overuse of routing resources matches the distribution types' shapes as the overuse mainly takes place due to inter-logic-block connections. There is almost no overuse in the outer regions of the architecture because there are fewer connections to I/O than between the *inner* logic-blocks and probably also because of the good I/O placement through the application of the barycenter heuristic.

While the *DISTANCE* distribution reaches an *overuse* quality comparable to the *simulated annealing* approach in *VPR* (2982 compared to 2907 overused segments), the *EQUAL* distribution even performs remarkably better (*concerning this measure*) with an *overuse* of only 1154.

Table 6 contains the other quality measures for these single runs.

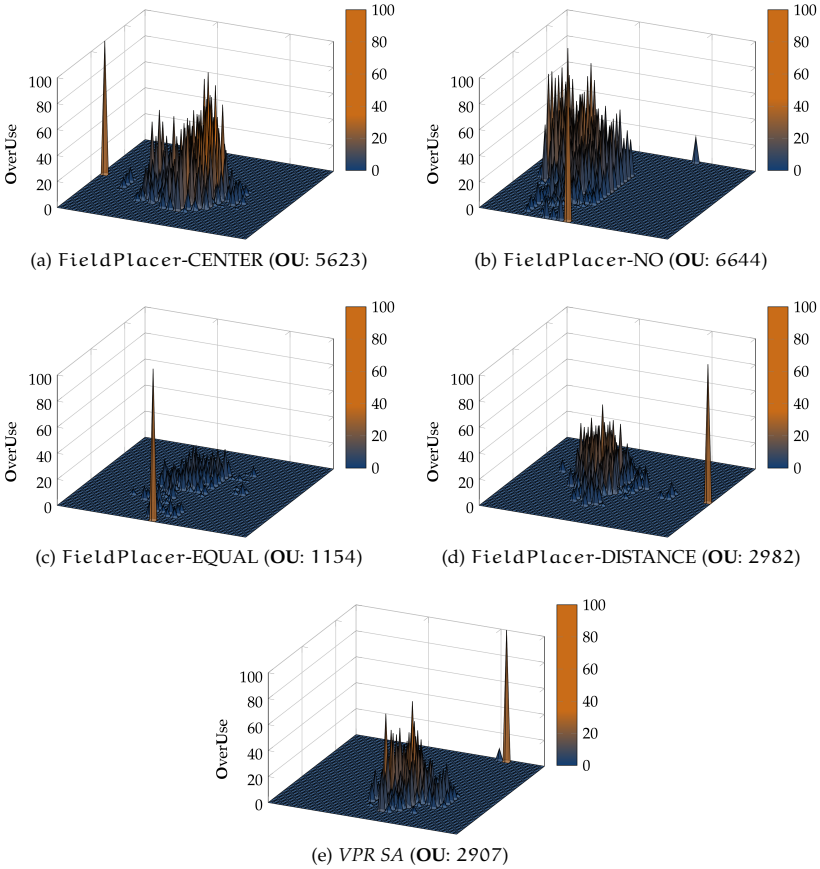
Placer	BoundingBox	CriticalPathDelay (ns)	WireLength	OverUse
CENTER	374.05	25.31	60516	5623
NO	388.87	26.13	62950	6644
EQUAL	447.67	26.26	76277	1154
DISTANCE	421.03	25.58	69652	2982
***VPR*-SA**	317.81	24.34	55843	2907

Table 6: Statistics of the runs from Figure 71

As the estimation of the *overuse* simulates the routing under idealized circumstances (see Section 5.4.2), the actual routing *will in fact look different* and may, due to many different influences (like the connections' *order* and the resolving of *congestions*) not necessarily behave as the *overuse* norm tries to predict. To investigate the discrepancy between the new simulating *overuse* metric and the actual routability, all benchmark codes were placed with the different distribution strategies and subsequently routed by *VPR*'s router.

Remark 98. *As the routing of all benchmark codes took approximately* 2 *days for each distribution strategy (almost* 10 *days in total), the* ***router*** *investigations in Figure 72 are, as an exception,* ***not repeated*** 10 *times but only measured once per code and distribution strategy.*

The *maximal channel occupancy* (see Section 5.3.2) reports the highest usage in a routing cell and thereby the maximal number of parallel routing wires that are necessary to realize this specific routing. The higher this value is, the higher is the *maximal* 'stress' on the routing architecture. Figure 72 shows that a correlation between the *overuse* estimation and the final *maximal channel occupancy* is, to a certain extent, measurable. For example, the distribution strategy with the smallest average *overuse* (*the EQUAL distribution*) is the one with the smallest *maximal channel occupancy*. However, both are still not '*exact*'



(a) FieldPlacer-CENTER (**OU**: 5623)

(b) FieldPlacer-NO (**OU**: 6644)

(c) FieldPlacer-EQUAL (**OU**: 1154)

(d) FieldPlacer-DISTANCE (**OU**: 2982)

(e) *VPR SA* (**OU**: 2907)

Figure 71: **O**ver**U**se among the chip for different distribution types (FieldPlacer) and *VPR SA* (code: or1200)

measures but only (more or less rough) estimates for the general routability of a design's placement.

As it has already been shown earlier in this Section, the *overuse* of the *EQUAL* and the *DISTANCE* distribution is generally much smaller than the one of the other two FieldPlacer distribution strategies. As a consequence, only for these two strategies, all codes that were *routable with VPR*'s placement by simulated annealing were *finally routable with the basic* FieldPlacer placement. For the *CENTER* distribution, *three* times as many codes were not

routable (under the given architecture restrictions concerning the channel widths, see Appendix A.6). This *actual routability* of the placement (cp. the amount of **N**ot**S**ucc**R**outed inputs in Figure 72) does in fact *correlate very well* with the introduced *overuse* estimation (cp. **O**ver**U**se in Figure 72) performed by the `FieldPlacer` *congestion-driven maze router* (Section 5.4.2).

Figure 72 additionally shows that the routing time does *not* necessarily follow the trend of the *overuse* or the *maximal channel occupancy*. This is based, among other things, on the fact that even though a placement of a design might be routable with *smaller maximal channel occupancy* than another placement, both can still be routable *at all* so that a legal routing could be generated in the same time, just with different requirements for the *channel widths*.

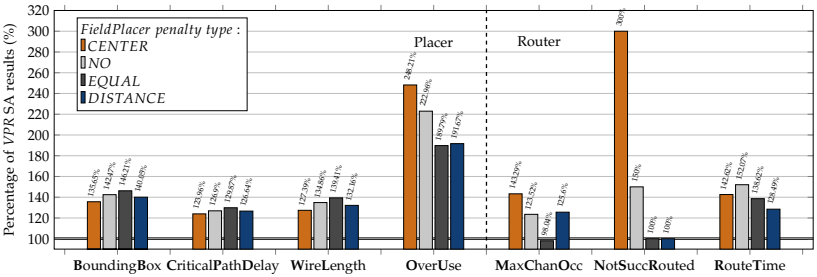


Figure 72: Pure `FieldPlacer` after routing - Overview

To measure the quality of the placement procedure, the *critical path delay* can only be *estimated* after the placement as the routing influences connections' *wiring* and *wirelengths* and thus the resulting (*wire*) delays (see Section 5.3.2). Figure 73 shows the correlation between the *estimated critical path delay* after the placement and the actual *final critical path delay* after routing (again relative to the *VPR SA* result and, as before in Figure 70, the colors match the distribution strategies from, e. g., Figure 72). The measurements show that the correlation between the *estimated* and the *final* critical path delay is *almost perfectly positive*. Thus, the estimated critical path delay after placement can definitely be considered to estimate the speed of the design on the architecture *already after the placement* and *without the actual routing* in the following.

Finally, it has to be noted that even though the estimation of the *overuse* is interesting and (*as the previous results showed*) valuable to measure the routability of a placement, this comes at the price of additional runtime by running the simulated routing of the *congestion-driven maze router*. If one assumes a completely filled architecture and neglects the fact that multiple I/O pins may share the same I/O block, the theoretical *worst case* runtime of

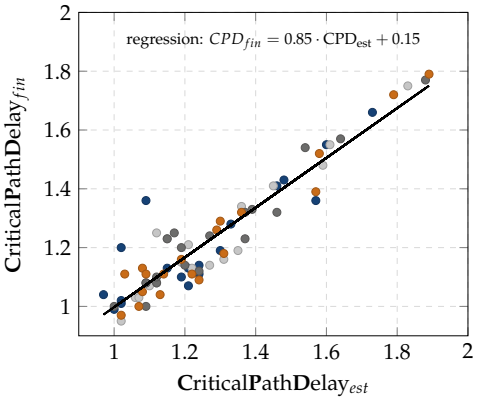


Figure 73: Correlation between estimated and actual critical path delay (*all* 19 codes with *four* distribution strategies each)

expanding the wave is $\mathcal{O}(N \times N) = \mathcal{O}(|V_D|)$ for each connection on a square architecture of size N times N (see Algorithm 8). Tracking the wave back of course takes less time ($\mathcal{O}(N)$) and has to be performed for each edge. Thus, the (combined) theoretical *worst case* runtime is $\mathcal{O}(|V_D| \cdot |E_D|)$. However, this is a theoretical consideration. With short connections, the runtime to route the single connection will be much smaller in general. Due to this fact, an exploration of the *real effective* runtime is indispensable. Figure 74 shows how long the execution of the *congestion-driven maze router* took for each code and each distribution strategy compared to the time spend in the entire placement procedure of the basic `FieldPlacer`. The different codes were sorted ascendingly concerning the runtime of the *simulated annealing* approach in *VPR*. The relative runtime of the *congestion-driven maze router* increases the 'larger' the inputs become. This is based on the higher theoretical runtime complexity of the norm calculation compared to the theoretical $\mathcal{O}(|V_D|\log|V_D| + |E_D|)$ runtime of the `FieldPlacer` implementation (see Section 5.7). In general, the time to calculate the *overuse* is smallest for the *CENTER* distribution, due to the relatively small *wirelengths* for this strategy. To sum up, the runtime of the *overuse* calculation is maximally $\sim$ 30% of the overall basic `FieldPlacer` runtime (*including this calculation*) and is, in general, much smaller for the small codes (or *designs*). Nevertheless, the estimation time of *overuse* can become noteworthy and a designer can consider whether it is important to get this information or, instead, to save this time.

Interim Result 3. *In this section, the* ***basic*** **FieldPlacer** *was introduced. The investigations showed that the different* ***distribution strategies*** *actually have the*

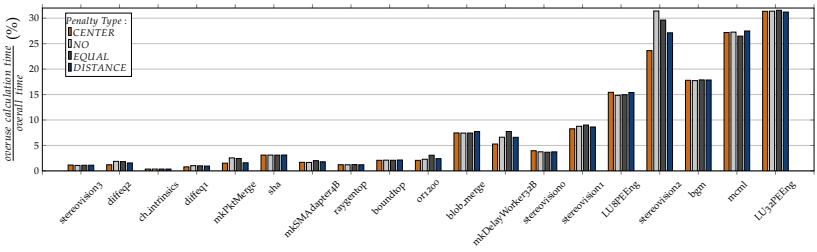


Figure 74: Percentage of the overuse calculation time of the overall `FieldPlacer` runtime (*codes sorted ascendingly by* VPR SA *runtime*)

previously assumed characteristics (see Section 5.5.3 - Motivation of the distribution strategies). It has been shown that the strategies differ concerning the different quality measures and that the new introduced ***overuse*** *norm can be helpful to predict the* ***routability*** *of a placement. It can therefore be used to choose either a 'well routable' placement out of several tries or to choose, for example, the right distribution strategy for the demands of the designer as better routability normally comes at the price of increased wirelength (which* ***correlates positively*** *with the critical path delay and the bounding box cost). Thus, wirelength minimization and routability can be* ***contradicting goals*** *in general. However, the* ***DISTANCE*** *distribution, which imitates the arrangement of the* ***force-directed graph layout****, is very promising at* ***combining good routability and short wirelengths*** *and is therefore* ***the strategy of choice*** *in the following.*

As the actual routing time is generally very long, it can be crucial to have a placement which ***supports the routing process****. In fact, the routing can take* ***many (3 to > 400) times*** *longer than the placement time with VPR's placer and router* ***in the default configuration*** *although this, undoubtedly, depends on the specific router (and placer) that is used. Apart from the routing time, the general* ***routability*** *of placements has been shown to be* ***influenceable by the distribution strategy****.*

Due to the good ***estimation of the critical path delay****, the* ***basic*** **FieldPlacer** *could, for example, be applied to quickly check how changes in the design may help to achieve a desired clock speed before placing and routing the design in detail.*

The next Sections will now show how the ***basic*** **FieldPlacer** *is used as the basis for* ***extended*** **FieldPlacer** *techniques to achieve further improved results.*

5.6 FieldPlacer EXTENSIONS

5.6.1 *5½th Step: Second energy phase*

The biggest portion of the runtime in the basic FieldPlacer is usually needed to calculate the repulsive forces in the force-directed graph layout in step 2 (see Section 5.5.2 and also Section 4.2.2 - *An experimental comparison*). Simulating (*or even solving*) a (*sparse*) system *without repulsive forces* (like in the GORDIAN approach based on equation systems, see Section 5.1.2), but instead with fixed I/O nodes, would work much faster (see Section 5.6.1 and Section 4.2.2). However, it was reasoned extensively that fixing the I/O nodes a priori can influence the later design a lot and is not the idea in this work. Nevertheless, assuming that the I/O nodes are as well distributed as with the basic FieldPlacer result, the question remains whether the attendance of *repulsive forces* is necessary or advantageous at all.

To investigate this, a *second energy phase* can be conducted after the basic FieldPlacer has been applied. In the *second energy phase*, all I/O nodes are *fixed* and *only attractive forces* between connected nodes are taken into account. The resulting force system is shown in equation (58).

$$
\begin{aligned}
F_{attr}^{(u,v)}(v) &= \begin{cases} \log\left(\frac{||p_v - p_u||_2}{l^{zero}(e)}\right) \cdot ||p_v - p_u||_2 \cdot (p_u - p_v) & p_v \neq p_u \\ 0 & otherwise \end{cases} \\
F_{attr}(v) &= \begin{cases} 0 & \text{if } v \text{ is fixed} \\ \sum\limits_{u|(u,v)\in E} F_{attr}^{(u,v)}(v) & otherwise \end{cases} \\
F_{res}(v) &= \lambda_{attr} \cdot F_{attr}(v) \overset{target}{\rightarrow} 0
\end{aligned}
\tag{58}
$$

Optimizing this system is twofold. On the one hand, the absence of repulsive forces should help to escape from local minima (that were facilitated by the repulsive forces in step 2) and, on the other hand, nodes are not kept away from each other by their *reciprocal repulsion*. Finally, such an 'amendment' respects the final I/O positions and could therefore be able to improve especially the CLB placement which was performed solely based on $\mathcal{G}_{CLB}^{layout}$ and *without* the explicit knowledge of the later embedded *I/O* positions in $\mathcal{G}_{D}^{arch}$.

As the initial layout of this phase is already quite good and should only be improved, the initial positions for all nodes are of course taken from the embedded layout of the FieldPlacer (instead of placing the nodes randomly like in the first phase). Due to the good initial solution, the *multilevel* ability of FieldFM^3 is deactivated (as it is generally included to create a good initial placement, see Section 4.2.3).

Figure 76 shows the layout after different numbers of improving iterations and nodes of the same row in the initial layout are depicted with identical colors. The surrounding *fixed I/O nodes* are again shown in orange. After a defined number of iterations has been performed in this way, the entire FieldPlacer procedure (Step 3-5) is called again with the new layout as its input. Figure 75 shows the *CLBs* in the second layout graph $\mathcal{G}_D^{\text{2ndlayout}}$ with a new final row assignment.

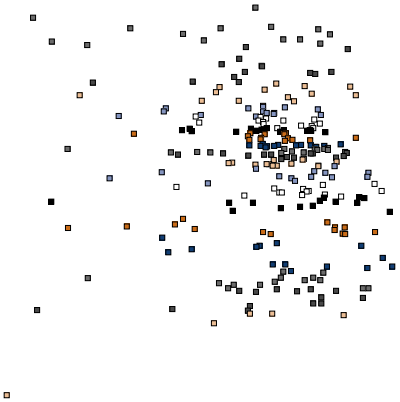
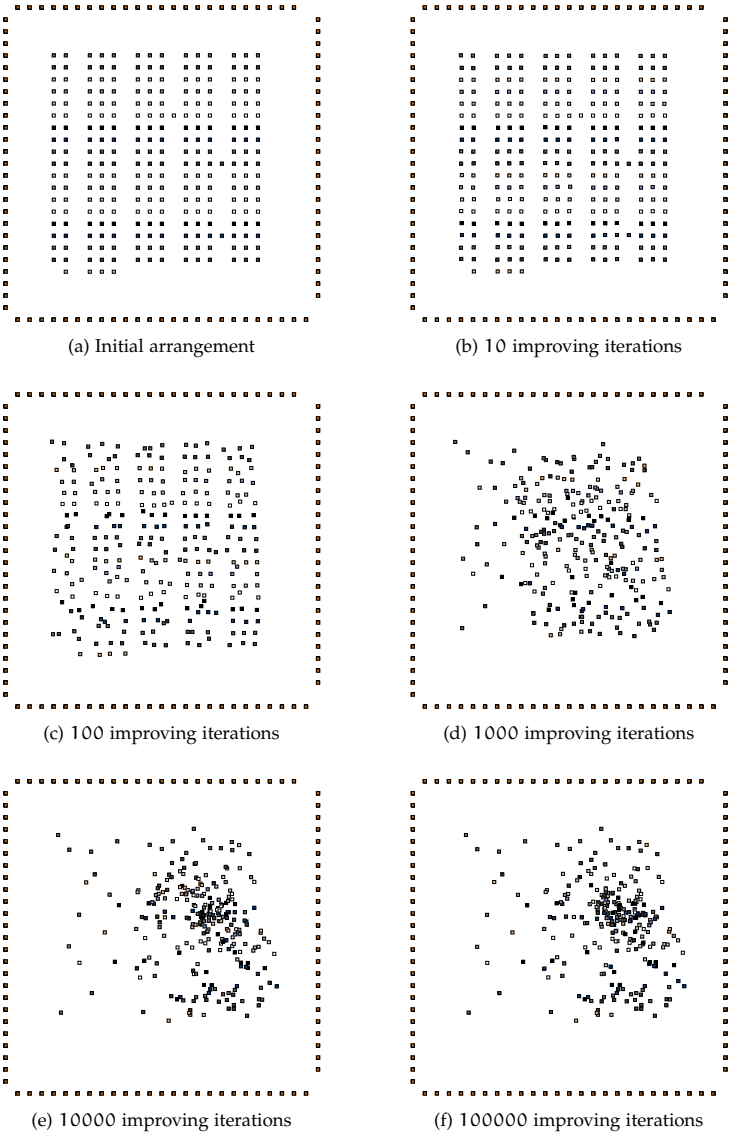


Figure 75: New slicing after second energy phase (code: or1200)

A closer look onto Figure 76f reveals different peculiarities of such layouts without repulsive forces that were already discussed several times before in this work (e. g., in the description Tutte's approach in Section 4.1.1). There are *heavily stressed regions* but also *sparely used* ones. The initial rows are partially preserved, but especially in the heavily stressed regions, the order of the elements is significantly perturbed. Overall, the different densities in the force-directed layout $\mathcal{G}_D^{\text{2ndlayout}}$ *without repulsive forces can not* be preserved well in the embedding $\mathcal{G}_D^{\text{2ndarch}}$ due to the restricted number of slots on the architecture.

Figure 77 shows a few randomly chosen connections in the or1200 design and how they are varied due to both *layouting* and *embedding* phases. While the first energy layout *with* repulsive forces (Figure 77a) can be embedded in a *structure-preserving* way (cp. Figure 77b), connections in $\mathcal{G}_D^{\text{2ndarch}}$ (Figure 77d) are more often *altered* compared to the energy layout *without* repulsive forces (Figure 77c). For example, the gray connection in the bottom part of the layout even changes its direction in the embedding and is short-



(a) Initial arrangement

(b) 10 improving iterations

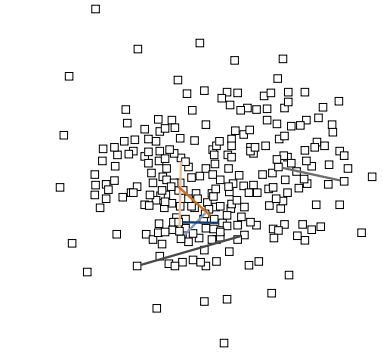
(c) 100 improving iterations

(d) 1000 improving iterations

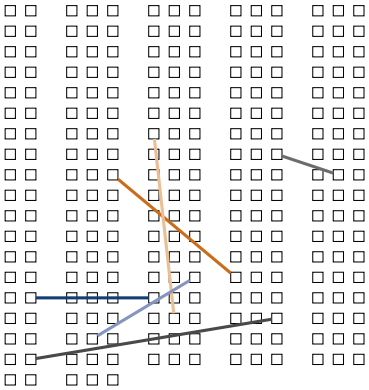
(e) 10000 improving iterations

(f) 100000 improving iterations

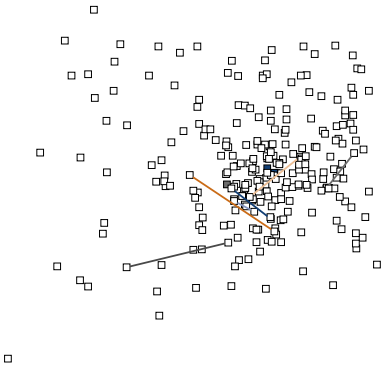
Figure 76: Second energy phase with fixed surrounding I/O nodes (code: `or1200`)



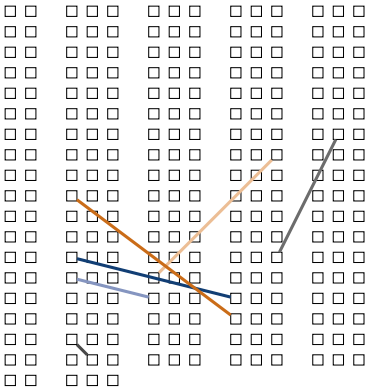
(a) Force-directed layout **Phase 1** $\mathcal{G}_D^{\text{layout}}$



(b) Embedded on the grid **Phase 1** $\mathcal{G}_D^{\text{arch}}$



(c) Force-directed layout **Phase 2** $\mathcal{G}_D^{\text{2ndlayout}}$



(d) Embedded on the grid **Phase 2** $\mathcal{G}_D^{\text{2ndarch}}$

Figure 77: Displacement in first and second energy phase (code: `or1200`)

ened a lot due to the large unused spaces in the layout. Appendix A.7 shows two further examples of embeddings to illustrate this effect.

Generally, the more iterations are performed with the fixed outer I/O nodes, the smaller becomes the wirelength in $\mathcal{G}_D^{\text{2ndlayout}}$. However, as the distances of the nodes to each other can not be preserved in the *embedding*, this does not necessarily mean that the embedded result on the architecture in $\mathcal{G}_D^{\text{2ndarch}}$ can also benefit from many iterations.

Figure 78 shows that for the `LU32PEEng` code example, the force-directed layout can improve the wirelength in $\mathcal{G}_D^{\text{2ndlayout}}$ by up to 69.43% after 100000 iterations while the wirelength in the embedded layout $\mathcal{G}_D^{\text{2ndarch}}$ becomes worse and worse. Even for codes where the embedding can be performed in a more structure-preserving way (like for the `mkPktMerge` example), the final improvement due to this *second energy phase* is rather small and such a refinement takes a significant amount of time (depending on the density of the graph), even though the repulsive force calculations do not have to be carried out. While 1000 iterations in the second energy phase of the `or1200` example (see Figure 76) took approximately the same time than the entire energy layout in *phase 1*, performing 100000 iterations took consequently 100 times as long. Together with the rather poor overall improvement, only very few, if any, iterations of such a second energy phase should be performed.

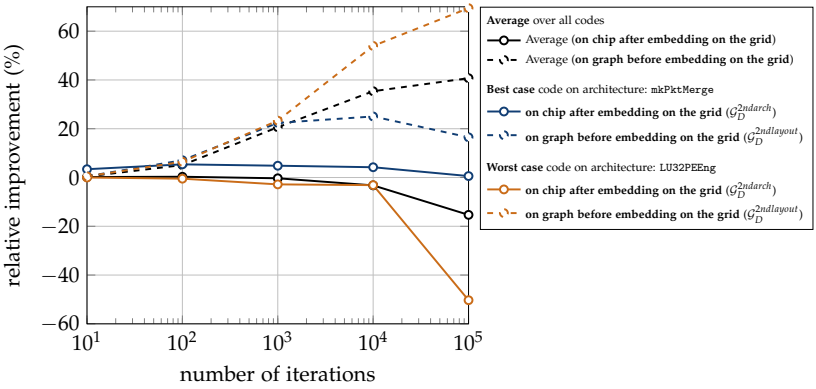


Figure 78: Second energy phase quality impact

This shows why the aforementioned approaches (see Section 5.1.2), which base on force systems without repulsive forces and solve the system for minimal wirelength, *need* an intensive hierarchical partitioning scheme after the basic layout. The usage of repulsive forces makes it possible to embed the created layout with only small displacements and, therefore, to benefit from the fine-grained properties of the layout *without the need for a 'universal' partitioning* like in the *GORDIAN* method. *Altogether, these results underline the good quality of the embedding through the* `FieldPlacer` *method.*

Finally, only very few additional iterations for fine-grained improvement of the situation could be performed in general. Figure 79 shows a further problem in case that there are relatively few I/O nodes in the design at all. Very much *space is wasted* in the outer regions of the graph because all nodes

are contracted to (*and by*) the inner regions. This is based on the very few *attractive* forces from the fixed surrounding I/O nodes which are *dominated* by the attractive forces in the strongly connected inner parts of the graph. A *structure-preserving* embedding onto a fairly *filled* chip architecture is not possible in such situations.

Remark 99. *To match the parameters of the force-system from the beginning, the coordinates of the initial layout with fixed I/O nodes are linearly scaled according to the final size of the previously obtained* $\mathcal{G}_{\mathrm{D}}^{layout}$ *graph.*

Remark 100. *Other choices of fixed node sets are directly possible in the* FieldPlacer*. For example, it can also be configured to* ***fix all but the CLB*** *nodes.*

Remark 101. *Due to the fact that the benefit of the second energy phase depends much on the specific example code, it is* ***not*** *performed in any of the following benchmarks. Nevertheless, the option is available as an experimental feature in the method. If the* ***number of I/O nodes is relatively small****, it should generally not be used. In such cases, the method therefore outputs an appropriate* ***warning****.*

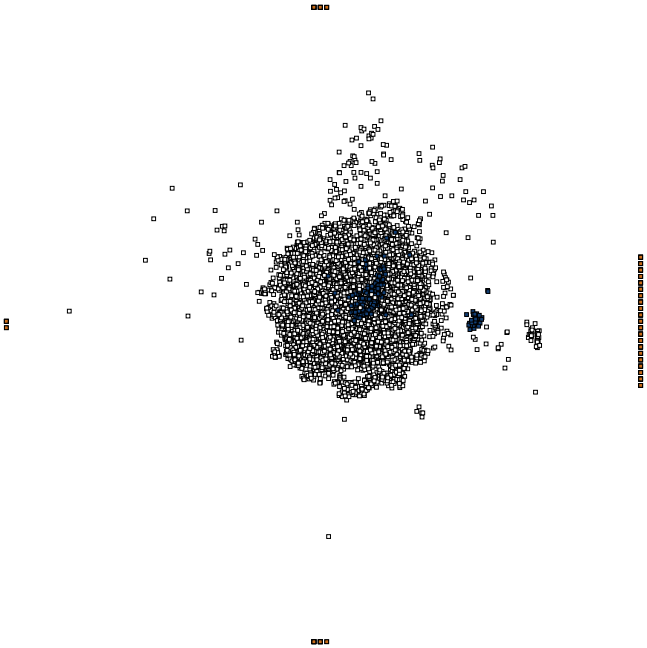


Figure 79: Second energy phase of the worst case (code: LU32PEEng)

5.6.2 *2nd Step with different distance norms*

Many available approaches in the field of force-directed placement techniques *measure and model* the distances between connected blocks usually with the *Euclidean* distance (see Section 5.1.2). However, the wiring on the architecture instead follows the *Manhattan* distance.

In order to pursue this fact, the FieldPlacer supports the usage of not only the *Euclidean distance* ($\|.\|_2$ *norm)* in the force model for the attractive forces, but also contains the option to use the *Manhattan distance* ($\|.\|_1$ *norm)* or the *Chebyshev distance* ($\|.\|_{max}$ *or* $\|.\|_\infty$ *norm)*.

Remark 102. *The reason for the consideration of the* ***Chebyshev*** *distance is given in the following.*

In general, such $\|.\|_p$ distances (metrics) are called p*-norms*. The p-norm of a vector $x \in \mathbb{R}^n$ is calculated as shown in equation (59).

$$\|x\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \tag{59}$$

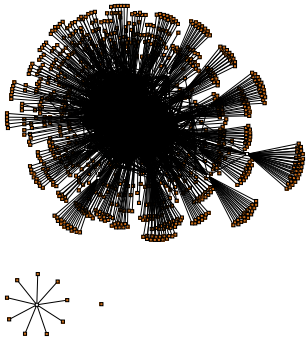
In the FieldPlacer method, the norm to calculate distances can be arbitrarily varied. If the FieldPlacer is configured to use, for example, the *Manhattan* distance to model the wirelength of connections, the original force model from FM^3 (see equation (45)) changes to system (60).

$$\begin{aligned}
F^u_{rep}(v) &= \begin{cases} \frac{1}{\|p_v - p_u\|_2{}^2} \cdot (p_v - p_u) & p_v \neq p_u \\ 0 & \text{otherwise} \end{cases} \\
F^{(u,v)}_{attr}(v) &= \begin{cases} \log\left(\frac{\|p_v - p_u\|_1}{l^{zero}(e)}\right) \cdot \|p_v - p_u\|_1 \cdot (p_u - p_v) & p_v \neq p_u \\ 0 & \text{otherwise} \end{cases} \\
F_{rep}(v) &= \sum_{u \in V \setminus v} F^u_{rep}(v) \qquad F_{attr}(v) = \sum_{u \mid (u,v) \in E} F^{(u,v)}_{attr}(v) \\
F_{res}(v) &= \lambda_{rep} \cdot F_{rep}(v) + \lambda_{attr} \cdot F_{attr}(v) \overset{\substack{target \\ force \\ equilibrium}}{\rightarrow} 0
\end{aligned} \tag{60}$$

Remark 103. *The norm calculations for the* ***repulsive*** *forces can also be performed with other norms in the* FieldPlacer*. However, as the* ***routing architecture*** *and the resulting* ***wirelengths*** *are the reasons to alter the norm, only the attractive forces are modified in the force system at this point.*

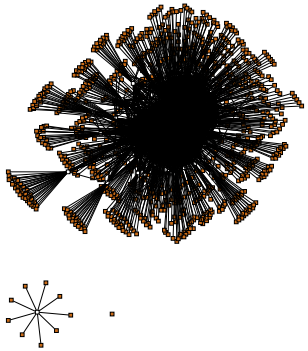
Figure 80 shows resulting force-directed graph layouts $\mathcal{G}_D^{layout}$ for the three already mentioned metrics.

(a) Euclidean distance: nodes



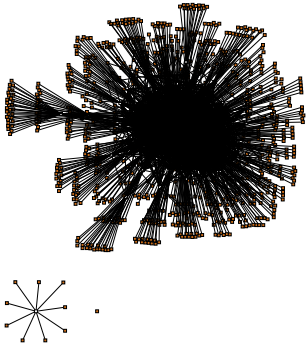
(b) Euclidean dist.: nodes and edges

(c) Manhattan distance: nodes



(d) Manhattan dist.: nodes and edges

(e) Chebyshev distance: nodes



(f) Chebyshev dist.: nodes and edges

Figure 80: Force-directed layouts under different metrics (code: `or1200`)

Naturally, nodes of a graph that are connected to a common *center* tend to be spread on circular perimeters around this center in a *force equilibrium* obtained from a force-directed layout method. This is based on the fact that the strength of attractive forces between nodes depends on their distance to each other and that, therefore, nodes with the same distance to the center experience the same force so that an equilibrium state is reached when repulsive and attractive forces compensate each other on the unit circle of the applied p-norm. Figure 81 shows two dimensional *unit circles* for different p-norms and it is obvious that the layouts from Figure 80 basically result in the shape of the respective p-norm's unit circle.

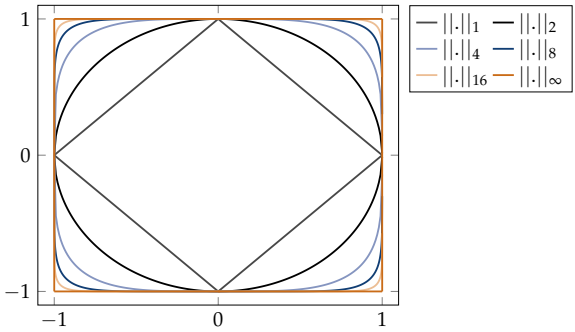
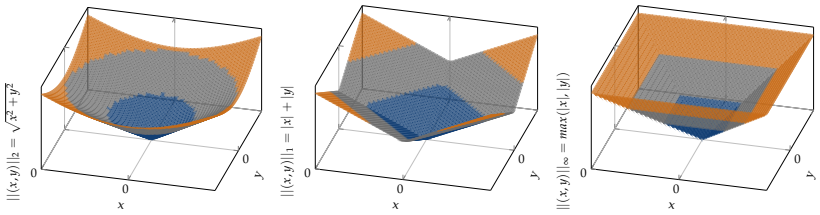


Figure 81: Unit circle for different p-norms

In *three dimensions*, the p-*norms* form the distance functions presented in Figure 82. For each point (x, y) in the two dimensional plane (as present in general 2-dimensional layouting approaches), the z-coordinate represents the *normed distance* of (x, y) to the origin $(0, 0)$. Consequently, the z-coordinate of each point $(x_2 - x_1, y_2 - y_1)$ corresponds to the distance between two points (x_1, y_1) and (x_2, y_2). The shape of the unit circles is revisited in Figure 82.



(a) Euclidean distance (b) Manhattan distance (c) Chebyshev distance

Figure 82: 3-dimensional p-norms

While the Euclidean graph layout results is *circular* graphs, the two other layouts have an '*orientation*' due to their different distance functions. The *Manhattan graph* has a *diamond* shape and the *Chebychev graph* has a *square* shape. Thus, as the resulting layouts in Figure 80 do not all have a quadratic shape (as the embedding on the chip has to have), it could be advantageous to *rotate* the layouts appropriately to match the shape of the chip. Without a rotation, the displacement of nodes in the embedding (from $\mathcal{G}_D^{\text{layout}}$ to $\mathcal{G}_D^{\text{arch}}$) can become large and could even eradicate the advantages of the other norm usage as the created diamond and the square chip shape do not match at all. Figure 81 highlights that the shape of the resulting force-directed layout matches better with the chip's shape the larger p is.

Thus, applying the *Chebychev* distance ($p \to \infty$) results in a perfectly matching (outer) graph shape. However, the *Manhattan* norm respects the routing architecture perfectly. To make the Manhattan result also match the chip's shape, it could be ordinarily rotated by 45° (*in the Euclidean sense*). Even though an '*ordinary Euclidean*' rotation does not change distances between any two points under the Euclidean norm, this is not the case for other metrics. Thus, as the distances on the chip have to be minimized concerning the Manhattan norm for the wiring, such a rotation actually influences the Manhattan distances between nodes in the graph.

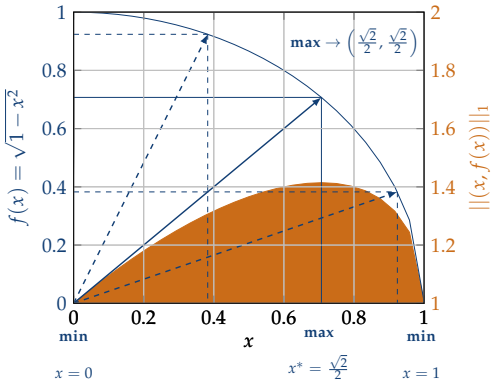


Figure 83: Influence of a $||.||_2$-rotation on the $||.||_1$-norm

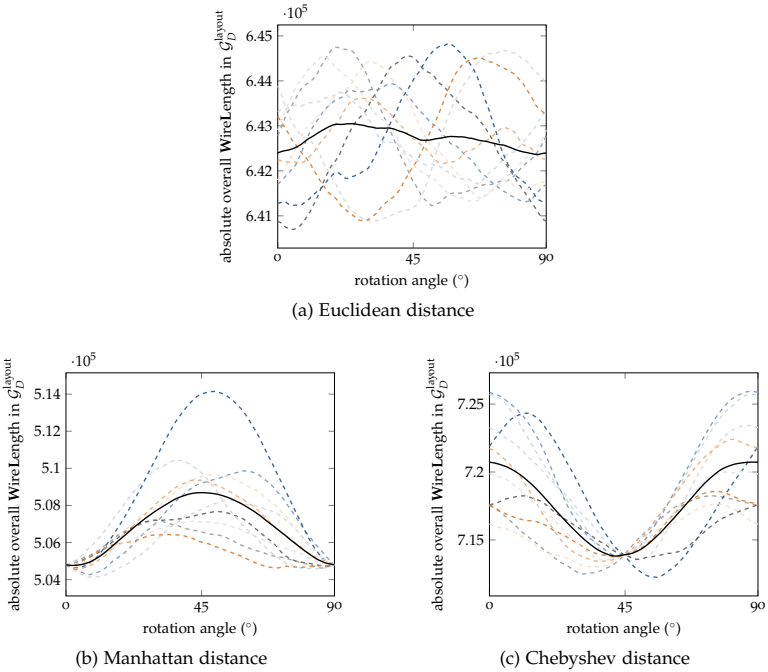
Figure 83 shows that a vertical connection of length 1 that is rotated has a minimal 'Manhattan'-length of 1 in the *vertical* and *horizontal* position while it is 'longest' ($\sqrt{2} \to 41.4\%$ longer) when it is rotated by 45°.

Remark 104. *For a complete graph, this degradation is the maximal possible value as the elongation of an edge caused by the rotation can (fully or partially) be compen-*

*sated by other edges with **different initial orientations**. The rotation benchmarks on 'real world graphs' (see Figure 84) - just as many other performed benchmarks - have shown that the effect is in fact much smaller due to rather 'un-orthogonal' graph structures.*

One question is whether it is profitable to optimize for the *Manhattan* distance directly in the force model and *either* 'loose wirelength quality' in the rotation step *or* have larger displacements in the embedding step.

Figure 84 presents the influence of such rotations on the wirelength (measured by the Manhattan distance) of the resulting layouts from Figure 80.



(a) Euclidean distance

(b) Manhattan distance

(c) Chebyshev distance

Figure 84: Rotation of $\mathcal{G}_D^{\text{layout}}$ (code: `or1200`)

Remark 105. *Larger rotation angles that are not between 0° and 90° lead to periodical results due to the 'horizontal and vertical (orthogonal) characteristic' of the Manhattan distance.*

First of all, Figure 84a shows that rotating the resulting graph $\mathcal{G}_D^{\text{layout}}$ created with the *Euclidean* distance in the force model can improve the graphs overall (*Manhattan*) wirelength, but the optimal angle is not predictable. This

is simply based on the fact that the graph produced with the Euclidean distance has no '*distinct orientation*', in other words, there is no trend how the final *circle* is rotated in the force-directed minimum because the rotation does not influence the layout's energetic potential. However, this also means that the *general* displacement (concerning the shape) is the same for all rotations.

Figures 84b and 84c show very conspicuous trends in the 10 benchmark runs (here for the `or1200` code) and consequently in the resulting average wirelength behavior in $\mathcal{G}_D^{layout}$ (black line) for the varied *rotation angles*. While the *Manhattan* graph has the smallest overall wirelength in its resulting energy-minimal layout (with *no* rotation as it was particularly optimized towards the distance norm of the wirelength), the *Chebyshev* graph has the smallest wirelength in $\mathcal{G}_D^{layout}$ if it is rotated by 45° and is, therefore, similarly oriented as the Manhattan graph.

Even though Figure 84 shows that the wirelength can be optimized when rotating the chip, this is still performed with the layout graph $\mathcal{G}_D^{layout}$ *before* the actual embedding onto the chip. Once again, as the wirelength is calculated by Manhattan distance between connected blocks, both the (*oriented*) Manhattan and Chebyshev results have the smallest overall wirelength in $\mathcal{G}_D^{layout}$ in the *diamond* shape rotation of the 1-norm unit circle. This means that the Manhattan result should *not be rotated* at all and the Chebyshev graph should be *rotated by approximately* 45° to minimize the distances in $\mathcal{G}_D^{layout}$. On the other hand, a *diamond* shape of the graph $\mathcal{G}_D^{layout}$ does not match the shape of the chip architecture. Thus, larger *displacements* are introduced when embedding the layout onto the restricted quadratic integer grid for $\mathcal{G}_D^{arch}$ (with all the different distribution strategies). This is based on the fact that nodes have to be moved further away from their relative position in a *diamond*-shaped $\mathcal{G}_D^{layout}$ to embed them on the *square*-shaped chip. In fact, further repeated benchmark results have shown that - *on average* - the *deterioration* of the wirelength by these *displacements* to create $\mathcal{G}_D^{arch}$ *redeems* the *advantage* of the *rotation* applied to $\mathcal{G}_D^{layout}$ in case of using the *Chebyshev* distance for $\mathcal{G}_D^{layout}$ while it is profitable to rotate the result of the Manhattan graph by 45°. *Both graphs consequently match the shape of the chip while not necessarily minimizing the wirelength in* $\mathcal{G}_D^{layout}$. Thus, matching the chip's shape to preserve the arrangement of nodes to each other as well as possible is more favorable.

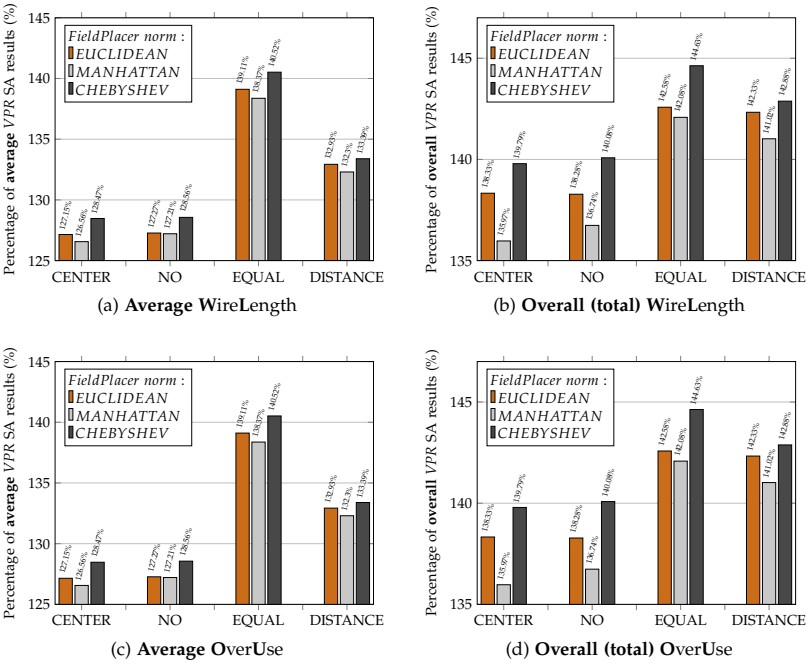
Due to the unpredictable optimal rotation angle for the Euclidean graph, it is desirable to try different angles between 0° and 90° and choose the best one. This rotation of the Euclidean graph is, due to its choice of the best angle and the relatively constant outer shape, undoubtedly profitable but accordingly more time consuming.

Considering the embedded results in $\mathcal{G}_D^{arch}$, the following strategies are defined for the final rotation of the energy-graphs (see Figure 80).

NORM STRATEGY: EUCLIDEAN Rotate the graph in 10 steps from 0° to 90° and choose the best rotation angle.

NORM STRATEGY: MANHATTAN Rotate the graph by 45° for small displacements in $\mathcal{G}_D^{arch}$.

NORM STRATEGY: CHEBYSHEV Do not rotate the graph at all for small displacements in $\mathcal{G}_D^{arch}$.



(a) **Average WireLength** (b) **Overall (total) WireLength**

(c) **Average OverUse** (d) **Overall (total) OverUse**

Figure 85: Influence of different norms on **WireLength** and **OverUse**

The results obtained by using these (in each case averagely optimal) rotation strategies are depicted in Figure 85. Even though the overall improvements after embedding are relatively small (partially due to the fact that the chips are principally very densely filled with logic blocks), the best strategy

is to perform a force-directed layout with the *Manhattan* distance in the force model (see equation (60)), rotate $\mathcal{G}_D^{\text{layout}}$ by 45° and embed it with the desired CLB distribution strategy. Figure 85 shows the impact on the *average wirelength* and also on the *overall wirelength* (after embedding) of all benchmark codes (again relative to the results of full simulated annealing runs in VPR). The fact that the advantage obtained from using the Manhattan norm is even higher regarding the *overall total wirelength sum* (Figures 85b) instead of measuring the *average advantage per code* shows that the impact of the 'right' norm is greater for larger designs with larger overall wirelengths.

Figures 85c and 85d additionally show that using the Manhattan norm with its 45° rotation strategy even results in a reduced *overuse*.

In summary, the impact of the different norms on the final embedded wirelength in $\mathcal{G}_D^{\text{arch}}$ is, nevertheless, averagely relatively small and is, therefore, rather a *fine-tuning* as the basic arrangement of nodes is not influenced too significantly by the norm itself. For example, the influence of the distribution strategy is generally *much larger* (see Figure 85).

However, the *CENTER* distribution, for example, results in *shortest overall wirelengths* when using the *Manhattan* distance in the force model with subsequent 45° rotation of the layout graph. This shows that the *Manhattan* norm not only improves pure distances between connected nodes (by smaller penalties in the *DISTANCE* distribution strategy, see Section 5.5.3), but it also slightly improves the general arrangement of nodes as the *CENTER* distribution packs all nodes densely (in $\mathcal{G}_D^{\text{arch}}$) into the same area of the chip for each outcome of $\mathcal{G}_D^{\text{layout}}$.

Remark 106. *The results, especially from the Chebychev graphs, show that the rotation towards a **diamond shape** is generally the best choice for $\mathcal{G}_D^{layout}$. However, as the chip architecture has a **square shape**, the diamond arrangement cannot be preserved and the displacements between $\mathcal{G}_D^{layout}$ and $\mathcal{G}_D^{arch}$ in the embedding phase can deteriorate the good original wirelength results.*

*The results also show that a **diamond**-shaped chip or, equivalently, a square chip with 'diagonal routing architecture' could actually be advantageous to minimize the wirelength on the chip.*

Interim Result 4. *Following the results of Section 5.6.1 and Section 5.6.2, the **second energy phase** will not be applied in the following benchmarks and the force system with the **Manhattan** distance (equation (60)) with subsequent 45° rotation of the layout graph will be used to obtain the best results.*

*Due to the results from Section 5.5.6, the **DISTANCE penalty** option will generally be used in the following for a good balance between **performance** (short wirelengths) and **routability** of the placed design.*

5.6.3 *6th Step: Local refinement*

Section 3.2 investigated the strengths and weaknesses of common iterative approaches to solve the *QAP* problem. It was shown that simulated annealing is a powerful method as it is relatively independent from the initial solution and leads to good results in reasonable times. Even though the fundamental *local search* approach (see Section 3.2.4) leads to *local optima* in short times, the quality of obtained solutions strongly depends on the initial configuration as the algorithm finally stops when the first (*nearest*) local optimum has been found (see Figure 17). However, Figure 17 also shows that local search can be a profitable technique to improve results that are already of good quality very fast as it does not deteriorate the solution at all.

In fact, after steps 1 to 5 of the `FieldPlacer` method, the obtained result *is* already of a relatively high quality. Thus, applying a final *local search* can be expected to be both *advantageous* and *fast* (cp. Interim Result 1).

The benchmarks of this Chapter also showed that the *simulated annealing approach* implemented in *VPR*, which was tuned over many years, leads to rather good results and that the *bounding box cost* function is an accurate norm for several optimization goals like *wirelength*, *critical path delay* or even the *overuse* as it actually takes all influencing effects into account. As a consequence, a local search (essentially based on the *VPR SA* approach and especially on its *cost function*) can finally be performed to improve the created placement. For that, the *VPR SA* method is called with an initial system temperature of 0. In that way, only improving swaps are accepted and the *bounding box cost* of the layout can consequently only become smaller. In addition, the idea of *shrinking* the regions (the *frames*) from which the pairs of blocks are taken is used to make *global* swaps in the beginning and become more and more *local* if fewer swaps are accepted in the process. Due to the fact that the temperature is 0 and that consequently *only improving swaps* are accepted, the stopping criterion of this '*cold annealing*' approach or, more precisely, of the *local search* method, can be simplified. In the *VPR SA* method, a number of inner iterations $\texttt{\#inner\_iter} = \texttt{annealing\_sched.inner\_num} \cdot |V_D|^{1.3333}$ (with the default configuration of $\texttt{annealing\_sched.inner\_num} = 1.0$, see Appendix A.6) is performed in each *iteration block*. Depending on the number of successful swaps, the *frame* to choose the swap-candidates from is shrunk or enlarged and the next *iteration block* is processed. For the following **extended** `FieldPlacer` benchmarks (Section 5.6.4), the method stops if the improvement of the *bounding box cost* in one such *iteration block* is smaller than 1%.

In summary, the 6th step (the *local refinement*) can be seen as a simulated annealing with starting temperature 0 and, therefore, improves the layout up to the nearest local optimum of the objective function (see Figure 86). While

a general *simulated annealing* approach overcomes local optima by accepting deteriorating swaps in the beginning of the method, this task is carried out by the preceding force-directed multilevel layout of the *basic* FieldPlacer which provides the *initial solution for this local search*.

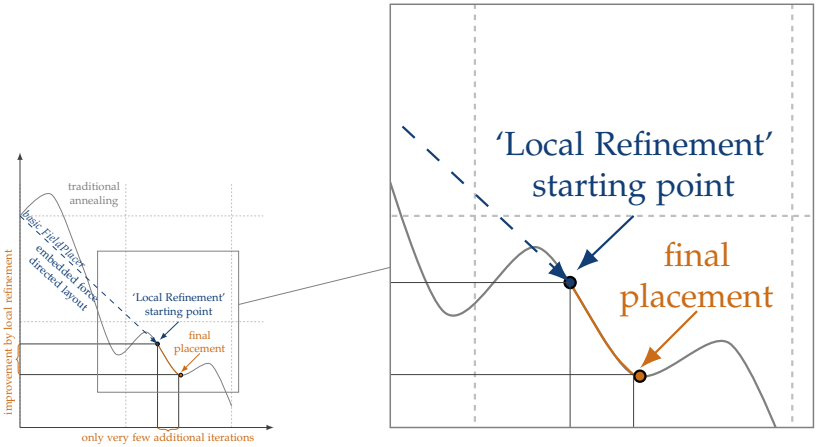
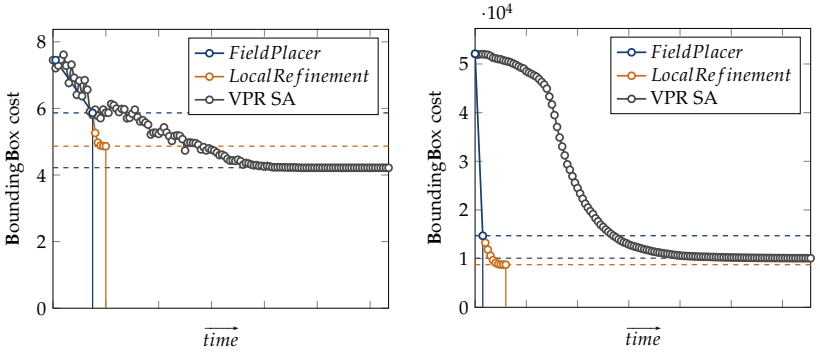


Figure 86: Local refinement scheme

Figure 87 shows how the local refinement performs in the **extended** FieldPlacer method compared to the *VPR SA* approach for three chosen example codes from the benchmark set. More specifically, the most *average* code (`stereovision2`), the *best* case (`mcml`) and the *worst* case (`stereovision3`) code are presented (in terms of resulting bounding box cost). Each *dot* marks the result of one *iteration block*. Due to the constant number of attempted swaps per *iteration block*, the evaluation of each block takes approximately the same time.

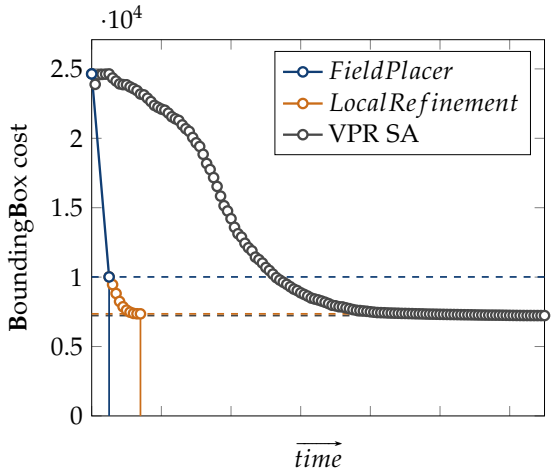
These results already show that, *after the local refinement*, the **extended** FieldPlacer can reach better (*bounding box*) results than *VPR SA* by finding the nearby local optimum after only very few iterations in short time (`mcml`). However, the small `stereovision3` code shows that, for other codes, the *VPR SA* approach can still perform better than the **extended** FieldPlacer. On average, the obtained results *concerning the bounding box cost* are comparable (like for the average `stereovision2` code with 4% higher bounding box cost than *VPR SA*, cp. Table 8). Section 5.6.4 will give a more detailed evaluation for the different quality norms.

However, it should already be mentioned that, even including the *local refinement*, the **extended** FieldPlacer approach is still remarkably faster than the *VPR SA* method for all three depicted examples. The number of per-



(a) **worst** case (code: `stereovision3`)

(b) **best** case (code: `mcml`)



(c) **average** case (code: `stereovision2`)

Figure 87: `FieldPlacer` and *VPR SA* iterations comparison (DISTANCE penalties)

formed *iteration blocks* was limited to 100 in the **extended** `FieldPlacer` implementation to restrict the maximal runtime. The actually performed number of *iteration blocks* until the improvement fell below the threshold of 1% has in fact been significantly smaller in all presented benchmarks.

Remark 107. *Some further insights to the* VPR SA *approach are given in Section 5.1.1. For a very detailed description of the* VPR SA *method, please refer to the book of Betz et al. [21].*

5.6.4 *Benchmark: Extended* FieldPlacer

In this Section, the performance of the **extended** FieldPlacer will be examined. Figure 88 presents the *bounding box cost per code* relative to the result achieved by the *simulated annealing approach* of VPR. The codes are sorted *ascendingly* by their *VPR SA* runtime and three bounding box results are presented for each code (averaged from 10 runs as before). First, the *INIT* bar shows the average bounding box cost of the *initial random placement*. The FieldPlacer bar marks the results of the **basic** FieldPlacer (in the chosen configuration with *DISTANCE* penalties and using the *Manhattan* metric in the force system) and the LocalRefinement bar finally represents the achieved *bounding box cost* result from the **extended** FieldPlacer.

In general, the relative improvement of both methods, *VPR SA* and FieldPlacer, compared to the initial assignment, tends to increase with growing *VPR SA* time (and with growing '*design size*'). While the *bounding box cost* after the **basic** FieldPlacer are generally still considerably higher than the ones of the *VPR SA* approach, the **extended** FieldPlacer is able to create comparable results along with the LocalRefinement. Even more, the **extended** FieldPlacer with LocalRefinement is even able to *beat* the quality of the *VPR SA* method for some codes (as it has already been presented in Figure 87b).

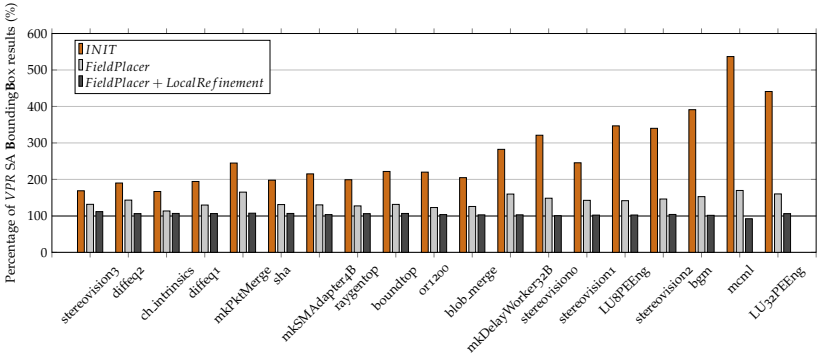


Figure 88: Local refinement **B**ounding**B**ox results (DISTANCE penalties, *sorted ascendingly by* VPR SA *runtime*)

For comparison, Figure 89 shows the time that is spend for the various codes in the different parts of the **extended** FieldPlacer method (relative to the *VPR SA* time). First of all, the relative runtime of the overall approach (compared to *VPR SA*) tends to decrease for larger code/design sizes. For the largest codes, the **extended** FieldPlacer with LocalRefinement is more

than 10 *times faster* than the *VPR SA* method while achieving a *comparable bounding box quality* in the placement. The fraction of the time that is spent in the **basic** FieldPlacer essentially becomes smaller the larger the codes are.

Within the FieldPlacer method, a very large proportion of the time is spent to create the force-directed graph layout by the spring embedder approach. The overall presented *embedding* itself only needs a very small time span for all codes.

The remarkable jump of the graph layouting time from `stereovision3` to `diffeq2` can be explained by the strategy switch in the multipole method (see Section 4.2.2 - *An experimental comparison*). While the graph representation of the `stereovision3` code has only 53 nodes, `diffeq2` contains 194 nodes (cp. Table 12) and, therefore, exceeds the threshold of 175 nodes to make use of multipole approximations for the repulsive force calculations. As it has been shown in Section 4.2.2, this threshold is not necessarily 'accurate'. Thus, the multipole approximation (including the setup of the *quadtree* etc.) may not pay off for codes with node numbers slightly above this value.

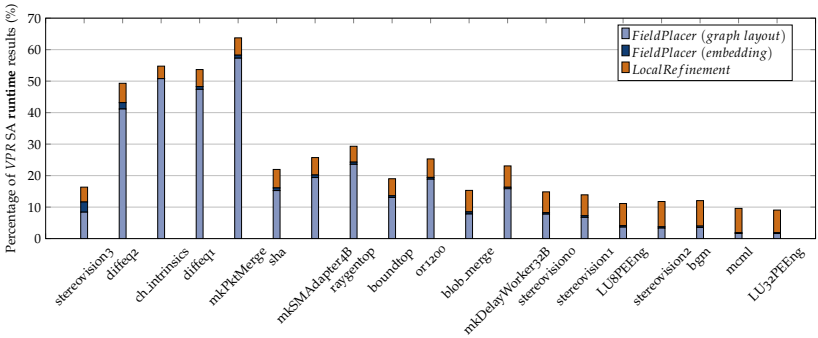


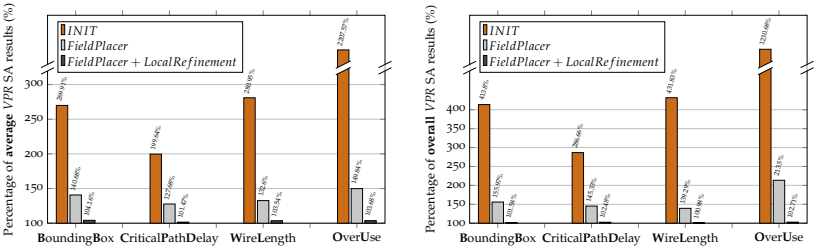
Figure 89: **Extended** FieldPlacer runtime (DISTANCE penalties, *sorted ascendingly by* VPR SA *runtime*)

To get an overview, Figure 90 shows the *average* and the *total* quality achieved for the three different stages concerning all considered quality measures and, again, relative to the *VPR SA* results. While the **extended** FieldPlacer with LocalRefinement is able to achieve results that are - *on average* - not more than 5% inferior to the ones of *VPR SA*, the situation concerning the *total* values is even better with less than 3% deviation. The *total* values again show that the FieldPlacer performs especially well (compared to *VPR SA*) for larger codes with, e. g., larger overall wirelengths.

Figure 91 additionally shows that some details in the trends of *advantages* and *disadvantages* for the different distribution strategies may no longer be as apparent as before because of the LocalRefinement. However, the general

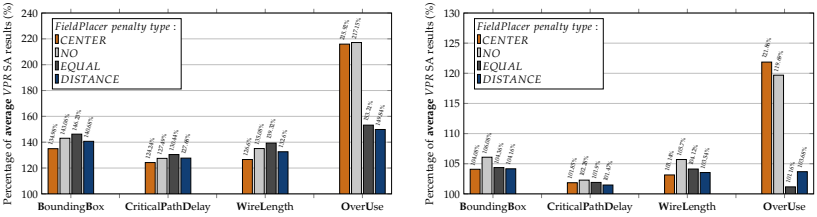
assumptions remain and the *DISTANCE* distribution still results in a good trade-off between all measures. Comparing Figure 91a and Figure 91b reveals the additional advantage that the LocalRefinement achieves.

In addition to these summaries, the Tables 8-11 contain the detailed average results for all three stages and all quality norms. The codes are again sorted ascendingly by their *VPR SA* runtime.



(a) **Average** (b) **Overall (total)**

Figure 90: FieldPlacer + LocalRefinement (DISTANCE penalties)



(a) **Basic** FieldPlacer (b) **Extended** FieldPlacer

Figure 91: FieldPlacer - Overview

Interim Result 5. *The results of the* ***extended*** FieldPlacer *along with the* LocalRefinement *show that, on average, this approach leads to resulting placements that are* ***comparable*** *to those of the* VPR SA *approach concerning all the mentioned* ***quality*** *metrics. The* ***basic*** FieldPlacer *layout is used as a good starting point for a* LocalRefinement *with reducing 'swap-frame'. Even though the resulting quality is comparable, the runtime of the* FieldPlacer *based placement is up to* 10 *times smaller (in* VTR 7.0*) and the runtime advantage of the* ***extended*** FieldPlacer *increases with larger inputs due to an* ***actually smaller runtime complexity*** *of the method.*

The remainder of this Chapter will investigate the ***theoretical runtime behavior*** *of the algorithm and point out some further extensions that are planned for the future. In addition, some very few insights to the actual implementation of the* FieldPlacer *and* FieldOGDF *are given. The following Chapter will finally present how these results can be further improved by* ***repeated execution*** *of the method.*

5.7 THEORETICAL RUNTIME BEHAVIOR OF THE FIELDPLACER

Like before, V_D and E_D represent the *nodes* and *edges* of the designs' graph representations ($\mathcal{G}_D$, see Table 12). Table 7 contains the summarized theoretical runtimes of all parts of the method.

Most of all steps are dominated by sorting of nodes that can be done in $\mathcal{O}(|V_D|\log|V_D|)$ time. The used `sort` function from the *C++ Standard Library guarantees* such a theoretical worst-case runtime of $\mathcal{O}(|V_D|\log|V_D|)$ (cp. the *Working Draft, Standard for Programming Language C++* [170, 25.4.1.1]).

Traversals of all edges take additional $\mathcal{O}(|E_D|)$ time so that the overall theoretical runtime of the FieldPlacer (without *local refinement*) is $\mathcal{O}(|V_D|\log|V_D| + |E_D|)$.

Due to the fact that the number of *MEM/MUL* nodes is very small (in general and) for the benchmark codes, the actual implementation uses a simple $\mathcal{O}(|V^2_{MEM/MUL}|))$ approach in Step 5 to avoid the setup of a *quadtree*. For upcoming architectures with other conditions, this can easily be exchanged to an appropriate method with $\mathcal{O}(|V_{MEM/MUL}|\log|V_{MEM/MUL}|)$ runtime. However, like in the FM^3 algorithm, this will only make sense for relatively large numbers of such special nodes (the maximal number is 200 for the benchmark codes in this work). As the slots on the architecture are relatively evenly distributed, a very simple quadtree construction could in fact be used. As a result, the FieldPlacer does not extend the theoretical runtime of the included FM^3 algorithm.

In practice, the runtime of FieldPlacer's embedding methods is significantly smaller than the time needed to perform the graph layout. A practically faster implementation of a spring embedder-based force-directed graph layout routine, like the already mentioned work of Gronemann in OGDF (see Section 4.2.4), could indeed help to improve the runtime of the overall FieldPlacer method even more. However, it has to be investigated if such an approach without multipoles is still as accurate as FM^3 and if an accurate edge length steering can be integrated.

The runtime of the LocalRefinement in the FieldPlacer implementation is $\mathcal{O}(|V_D|^{1.3333})$. This is due to the fact that (*per default*) each *iteration block* performs $|V_D|^{1.3333}$ swaps and that the number of these *iteration blocks* is limited to 100 in the FieldPlacer.

Step	Theoretical runtime	Reference
1st Step:	$\mathcal{O}(\lvert V_D \rvert + \lvert E_D \rvert)$	Algorithm 9
2nd Step:	$\mathcal{O}(\lvert V_D \rvert \log \lvert V_D \rvert + \lvert E_D \rvert)$	Section 4.2.2
3rd Step:	$\mathcal{O}(\lvert V_{CLB} \rvert \log \lvert V_{CLB} \rvert)$ $= \mathcal{O}(\lvert V_D \rvert \log \lvert V_D \rvert)$	Algorithm 10, sorting dominates
4th Step:	$\mathcal{O}(\lvert V_{I/O} \rvert \log \lvert V_{I/O} \rvert + \lvert E_{I/O} \rvert)$ $= \mathcal{O}(\lvert V_D \rvert \log \lvert V_D \rvert + \lvert E_D \rvert)$	Section 5.5.4, sorting dominates, edges are initially traversed in the sum calculation once
	$\mathcal{O}(\lvert V_{I/O} \rvert \log \lvert V_{I/O} \rvert)$ $= \mathcal{O}(\lvert V_D \rvert \log \lvert V_D \rvert)$	Algorithm 11, sorting dominates
	$\mathcal{O}(\lvert F \rvert) = \mathcal{O}(C)$	Algorithm 12, for constant number of I/O faces $\lvert F \rvert$
	$\mathcal{O}(\lvert V_{I/O} \rvert \log \lvert V_{I/O} \rvert + \lvert E_{I/O} \rvert)$ $= \mathcal{O}(\lvert V_D \rvert \log \lvert V_D \rvert + \lvert E_D \rvert)$	Algorithm 13, sorting plus the number of connections between I/O and CLB nodes
5th Step: (naive)	$\mathcal{O}(\lvert V_{MEM/MUL}^2 \rvert))$	generally with a very small $V_{MEM/MUL}$, see Section 5.5.5, $V_{MEM/MUL}$ is definitely smaller than the number of available MEM/MUL slots on the architecture
5th Step : (quadtree)	$\mathcal{O}(\lvert V_{MEM/MUL} \rvert \log \lvert V_{MEM/MUL} \rvert))$ $= \mathcal{O}(\lvert V_D \rvert \log \lvert V_D \rvert)$	Section 4.2.2
5½th Step:	$\mathcal{O}(\lvert V_D \rvert \log \lvert V_D \rvert + \lvert E_D \rvert)$	Section 4.2.2
6th Step:	$\mathcal{O}(\lvert V_D \rvert^{1.3333})$	Section 5.6.3 and runtimes in Figure 89, maximally 100 *inner iterations* with $\lvert V_D \rvert^{1.3333}$ swaps each

Table 7: Theoretical runtime of the `FieldPlacer` routines

Remark 108. *The memory requirements are not analyzed in this work as they are not limiting for the considered inputs on today's systems. However, this can become more important for future (larger) designs. In the benchmarked implementation, the available memory has always been more than sufficient.*

5.8 OTHER ARCHITECTURES

Apart from the introduced heterogeneous FPGA architecture with *I/O*, *CLB*, *MEM* and *MUL* blocks (see Section 2.3), *other types* or even utterly different architectures can easily be integrated into the framework. Depending on the typical number of elements of such a new block type and its influence on the placement quality, an appropriate execution point in the consecutive `FieldPlacer` steps has to be chosen. In the following, some ideas for such integrations are given.

NON-UNIFORM CLBS For extended architectures with non-uniform CLBs (e. g., with CLB types *CLB1* and *CLB2* having *different sizes/numbers of LUTs*), the `FieldPlacer` method could be applied with small adaptions. All steps, except for Step 3 from Section 5.5.3, could operate exactly the way that is explained in Section 5.5. Step 3 could take the energy-minimized graph layouts

from Step 2 (Section 5.5.2) and partition the CLB nodes into node sets of type 1 (*CLB1*) and those of type 2 (*CLB2*). After that, the CLB slot assignment can be performed independently for these types with the presented approach including the different distribution strategies. Therefore, a distribution solely for the *CLB1* slots on the target architecture can be considered (based on the availability of type 1 CLBs on the chip) and the type 1 CLBs can be embedded with a chosen distribution strategy. Subsequently, this could be done with the type 2 CLBs (*CLB2*) in the same way. For sure, more than 2 CLB types are possible in that way.

FURTHER SPECIAL BLOCK TYPES If there are *additional special block types* (like further *DSP*s) which are only sparsely available on the architecture, Step 5 could simply be repeated with the new block type to assign such nodes to appropriate slots on the chip.

3D-FPGAS The method can even be extended to place layouts for 3-dimensional FPGAs (cp. Kwon et al. [119]). For that purpose, a 3-dimensional force-directed layout (see Figure 92) can be performed similarly to the FM^3 approach with 3-dimensional coordinates, e. g., with a corresponding 3-dimensional multipole development (cp. Cottet and Koumoutsakos [42]) and a 3-dimensional version of *Hachul's 'sun-planet-moon'* multilevel model (see Section 4.2.3).

The 3-dimensional graph layout $\mathcal{G}_D^{3Dlayout}$ could be embedded by dividing the nodes in horizontal slices (basically ordered by their z-coordinate and partitioned according to an architecture-related distribution, similarly to how it is done in Step 3 of the `FieldPlacer` for the CLBs, see Figure 92). Then, a 2-dimensional embedding for each of these slices can be performed according to the corresponding layer in the 'FPGA cube' (just as in Section 5.5.3). To obtain a resulting placement, the 2-dimensional slices finally have to be stacked with respect to their z-coordinate.

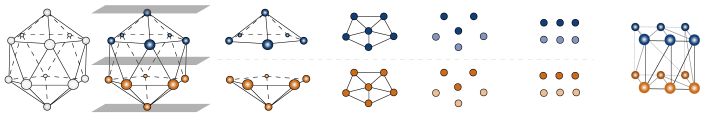


Figure 92: 3D Placement

However, it can be expected that the inter-slice communication in the layout and the arrangement of I/O resources (and many other more specific effects) in such hardware circumstances will demand for additional optimization steps.

5.9 ABOUT THE IMPLEMENTATION

5.9.1 *FMMM extensions (*FieldOGDF*)*

The FieldOGDF library is a modified subset of the original *OGDF* library [36] (version *v*2012.07) containing the energy-based layout techniques and all necessary miscellaneous functionalities. The library has therefore been reduced to a state without considerable dependencies to other non-system libraries. In other words, FieldOGDF is a stripped-down force-directed graph layouting library with some extra functionalities to communicate with the FieldPlacer method and including new methodological options. All of these 'special options' are *not mandatory* but only used for further refinements (as it has already been described previously). Every other 'ordinary' graph layouting method that outputs *GML* descriptions can be directly included. *OGDF* (and FieldOGDF) are written in *C++*.

DIFFERENT NORMS TO MATCH THE CHIP Apart from the 'traditional' usage of the *Euclidean* metric in the force system for the spring embedder, two other norms were additionally implemented as options in FieldOGDF's force systems, namely the *Manhattan* and the *Chebyshev* distance (see Figure 80 for resulting layouts). The different norms are applied in different parts of the force calculations and they can be used for *all* forces or (like in the presented method) *only* for the attractive forces by a simple switch in the implementation. The norm itself can even be chosen at runtime by setting an appropriate *member* of the `fmmm` class (e. g., by `fmmm.applied_norm(MANHATTAN_NORM)`).

The *rotation strategy* is automatically set with respect to the chosen *metric* but can be altered by a user in the same simple way (e. g., by setting `fmmm.innerrotationstrategy(ROT_45)` for a 45° rotation). Instead of rotating each *component* of a graph after the other (like it is done in the original FM^3 implementation before the *compaction* to optimize the area of each component in the drawing), the components are rotated simultaneously in FieldFM^3. This is important when using the *EUCLIDEAN* distance and choosing the best rotation angle out of, e. g., 10 angles between 0° and 90° as the evaluation of the Manhattan wirelength has to be conducted *for all graph components concurrently* for each angle to choose the best one. The check which norm to use is generally performed as far as possible outside of loops to reduce (time-consuming) branching within deeply nested loops. However, this potentially comes at the price of more code lines by 'duplicated' loop nests.

HANDLING AND USAGE OF FIXED NODES IN FMMM The `node` object in FieldOGDF got an additional parameter to store whether a node is *fixed* or *free*. This is necessary to perform the second energy phase and, in the future,

to handle a priori fixed nodes (see Section 2.2.1). To preserve the multilevel functionality in this idea, the *fixed* or *free* status is *inherited* from a finer level to a coarser one. Whenever a cluster node of a coarser representation in the quadtree (see Section 4.2.1) contains *at least one fixed child node*, this cluster node is accordingly also *fixed*. Consequently, a *(cluster) node* on any level is only moved if it is not *fixed*, otherwise the node always remains in its position. Whenever the *movement* of a *free* node that is connected to a *fixed* node has to be calculated, the *attractive* and *repulsive* forces are not split between both nodes but are acting entirely on the *free* node.

HANDLING SEVERAL NODES IN THE SAME PLACE IN FMMM Due to the fact that *several I/O nodes* may be in the *same position* on the architecture (see Section 5.5.4), the calculation of *repulsive* forces between such pairs has to be skipped. This is not an issue, as the two nodes are fixed anyway and will consequently not be moved at all. The original FM^3 implementation in the *OGDF* library generally handles such cases by moving the nodes slightly away from each other (within a *small epsilon radius*) to avoid the *singularity* in the repulsive force calculation from pairs of nodes with *zero distance*.

OTHER MODIFICATIONS Several other modifications were introduced to enable a flawless interaction with the `FieldPlacer` framework (see, for example, Section 6.3). However, the general behavior of the FM^3 method has, of course, been preserved and the modifications are not essential for the application within the `FieldPlacer`. For a *basic usage* in the `FieldPlacer` (*without the second energy phase*), *any* layouting method that takes a *.gml* description of the graph as an input and outputs a *.gml* description of the layout can be used (the nodes' numbers/labels should, in any case, be preserved to reassign the nodes appropriately).

Remark 109. *As a further extension, the* ***nodes' sizes*** *could be scaled with the* ***nodes' degrees*** *in* FM^3 *to create larger whitespace regions around nodes with higher node degrees to, finally, facilitate the routing.*

5.9.2 `FieldPlacer` *framework*

INTEGRATION OF THE SOFTWARE The `FieldPlacer` is an independent implementation and was tested in *VTR 7.0* and *VTR 6.0*. It can easily be integrated into these frameworks by adding the `FieldPlacer` sources to *VPR*'s *SRC* folder and running a patch script to incorporate the placement method into *VTR*'s (more precisely *VPR*'s) placer routine. In fact, the user can afterwards choose which placer to use in the *GUI* or by the command line option `--vprfieldplacer`. All options can initially be configured in a header

file and can also be *altered at runtime* in the FieldPlacer *GUI* (a GUI extension of the VPR GUI, see Chapter 6) to test different strategies and options interactively. As *VTR 7.0* is written in C++ while *VTR 6.0* was written in pure C, the FieldPlacer implementation provides *calls* for both and the script to patch VTR expects a parameter to decide which code base should be patched. In detail, the integration of the FieldPlacer into both *VTR* versions modifies several source code files to integrate the new *options*, *graphics*, *outputs*, etc. However, the FieldPlacer method itself remains as decoupled as possible and works rather *encapsulated*. To integrate the FieldPlacer into other FPGA 'compile flows', the framework only has to provide the *architecture* (see Section 5.3.1) and the *design's description* (see Section 5.5.1). After the entire process, an export routine has to pass back the block positions of the final layout to the compile flow.

GRAPH DEBUGGER/TRACER The graph debugger component of the FieldPlacer was a great help for the development of all presented features. It exports the graph representation from different steps of the FieldPlacer as *.gml* files to make the effects and results of all steps easily comprehensible. This feature is also very helpful to *find and fix* methodological bugs in the implementation. Several of the shown figures in this chapter are directly taken from this graph debugger/tracer. Finally, it should support the development of further extensions in the future. Furthermore, a command-line debugging mode can be activated which, e. g., outputs the *nodes' parameters* of the different NodeLists before and after being sorted (like *coordinates, angles to the barycenter, displacements, etc.*) and other helpful information especially for further development.

BoundingBox	INIT	(%-SA)	FieldPlacer	(%-SA)	LocalRef	(%-SA)	*VPR SA*
stereovision3	7.45	*(169%)*	5.81	*(132%)*	4.93	*(112%)*	**4.42**
diffeq2	74.66	*(190%)*	56.26	*(143%)*	41.72	*(106%)*	**39.28**
ch_intrinsics	40.52	*(167%)*	27.55	*(113%)*	25.92	*(107%)*	**24.29**
diffeq1	109.34	*(194%)*	72.96	*(130%)*	59.83	*(106%)*	**56.26**
mkPktMerge	221.55	*(245%)*	149.40	*(165%)*	97.15	*(107%)*	**90.49**
sha	263.94	*(198%)*	174.72	*(131%)*	142.28	*(107%)*	**133.38**
mkSMAdapter4B	320.03	*(215%)*	193.56	*(130%)*	153.94	*(103%)*	**148.76**
raygentop	352.90	*(199%)*	225.67	*(127%)*	188.09	*(106%)*	**177.26**
boundtop	428.28	*(222%)*	253.86	*(131%)*	205.96	*(107%)*	**193.11**
or1200	734.43	*(220%)*	410.25	*(123%)*	345.75	*(104%)*	**333.95**
blob_merge	1097.47	*(205%)*	674.06	*(126%)*	550.83	*(103%)*	**535.89**
mkDelayWorker32B	2390.31	*(282%)*	1352.46	*(160%)*	870.63	*(103%)*	**846.39**
stereovision0	2293.54	*(321%)*	1060.00	*(148%)*	717.69	*(100%)*	**714.26**
stereovision1	3316.74	*(246%)*	1925.95	*(143%)*	1379.28	*(102%)*	**1350.72**
LU8PEEng	10374.94	*(347%)*	4237.84	*(142%)*	3062.93	*(102%)*	**2991.71**
stereovision2	24640.44	*(340%)*	10594.99	*(146%)*	7530.05	*(104%)*	**7245.87**
bgm	16431.28	*(391%)*	6416.73	*(153%)*	4265.35	*(102%)*	**4201.02**
mcml	52093.60	*(537%)*	16466.07	*(170%)*	8940.50	*(92%)*	**9704.72**
LU32PEEng	64018.88	*(441%)*	23247.50	*(160%)*	15409.35	*(106%)*	**14516.25**
AVERAGE		***270%***		***141%***		***104%***	

Table 8: **Extended** FieldPlacer + LocalRefinement **B**ounding**B**ox cost (*sorted ascendingly by* VPR SA *runtime*)

CriticalPathDelay	INIT	(%-SA)	FieldPlacer	(%-SA)	LocalRef	(%-SA)	*VPR SA*
stereovision3	1.91	*(105%)*	1.86	*(102%)*	1.83	*(101%)*	**1.82**
diffeq2	18.50	*(120%)*	16.95	*(110%)*	15.57	*(101%)*	**15.47**
ch_intrinsics	3.87	*(120%)*	3.66	*(113%)*	3.21	*(100%)*	**3.23**
diffeq1	23.21	*(115%)*	21.54	*(107%)*	20.29	*(101%)*	**20.11**
mkPktMerge	4.95	*(118%)*	4.43	*(106%)*	4.21	*(100%)*	**4.19**
sha	19.49	*(156%)*	15.83	*(126%)*	12.93	*(103%)*	**12.53**
mkSMAdapter4B	7.61	*(146%)*	6.02	*(116%)*	5.28	*(101%)*	**5.21**
raygentop	7.23	*(149%)*	5.65	*(117%)*	4.87	*(100%)*	**4.85**
boundtop	9.22	*(150%)*	7.10	*(115%)*	6.10	*(99%)*	**6.15**
or1200	17.36	*(139%)*	14.53	*(116%)*	12.55	*(100%)*	**12.52**
blob_merge	18.86	*(199%)*	12.00	*(126%)*	9.73	*(103%)*	**9.49**
mkDelayWorker32B	15.96	*(221%)*	9.21	*(127%)*	7.15	*(99%)*	**7.22**
stereovision0	10.07	*(256%)*	5.38	*(137%)*	4.01	*(102%)*	**3.94**
stereovision1	8.04	*(143%)*	6.59	*(118%)*	5.70	*(102%)*	**5.60**
LU8PEEng	289.54	*(264%)*	143.69	*(131%)*	114.32	*(104%)*	**109.78**
stereovision2	59.20	*(365%)*	29.62	*(183%)*	17.38	*(107%)*	**16.22**
bgm	75.28	*(296%)*	37.25	*(147%)*	25.34	*(100%)*	**25.41**
mcml	230.49	*(306%)*	111.28	*(148%)*	77.79	*(103%)*	**75.37**
LU32PEEng	463.65	*(425%)*	198.63	*(182%)*	110.89	*(102%)*	**108.98**
AVERAGE		***200%***		***128%***		***101%***	

Table 9: **Extended** FieldPlacer + LocalRefinement **C**ritical**P**ath**D**elay (*sorted ascendingly by* VPR SA *runtime*)

WireLength	INIT	(%-SA)	FieldPlacer	(%-SA)	LocalRef	(%-SA)	*VPR SA*
stereovision3	661	*(224%)*	472	*(160%)*	369	*(125%)*	**295**
diffeq2	9361	*(208%)*	6415	*(142%)*	4754	*(106%)*	**4507**
ch_intrinsics	4974	*(182%)*	3100	*(114%)*	2868	*(105%)*	**2730**
diffeq1	13144	*(211%)*	7999	*(128%)*	6564	*(105%)*	**6234**
mkPktMerge	22884	*(254%)*	14287	*(158%)*	9566	*(106%)*	**9019**
sha	50859	*(172%)*	30764	*(104%)*	29613	*(100%)*	**29645**
mkSMAdapter4B	50113	*(233%)*	27371	*(127%)*	21942	*(102%)*	**21482**
raygentop	52104	*(213%)*	29801	*(122%)*	25676	*(105%)*	**24511**
boundtop	68080	*(225%)*	37967	*(125%)*	31623	*(104%)*	**30321**
or1200	132830	*(232%)*	67785	*(118%)*	59573	*(104%)*	**57326**
blob_merge	275646	*(208%)*	146272	*(110%)*	133213	*(100%)*	**132557**
mkDelayWorker32B	375178	*(286%)*	192050	*(146%)*	133354	*(102%)*	**131164**
stereovision0	275808	*(340%)*	115217	*(142%)*	80164	*(99%)*	**81054**
stereovision1	416015	*(241%)*	227416	*(132%)*	174800	*(101%)*	**172899**
LU8PEEng	2203924	*(391%)*	722746	*(128%)*	571615	*(101%)*	**564193**
stereovision2	2997655	*(332%)*	1232651	*(136%)*	937581	*(104%)*	**903465**
bgm	3745187	*(439%)*	1239801	*(145%)*	853758	*(100%)*	**853630**
mcml	8912896	*(439%)*	2665026	*(131%)*	1835109	*(90%)*	**2032197**
LU32PEEng	14334144	*(511%)*	4180907	*(149%)*	3024876	*(108%)*	**2802725**
AVERAGE		***281%***		***133%***		***104%***	

Table 10: **Extended** FieldPlacer + LocalRefinement **W**ire**L**ength (*sorted ascendingly by* VPR SA *runtime*)

OverUse	INIT	(%-SA)	FieldPlacer	(%-SA)	LocalRef	(%-SA)	*VPR SA*
stereovision3	0	*(100%)*	0	*(100%)*	0	*(100%)*	**0**
diffeq2	0	*(100%)*	0	*(100%)*	0	*(100%)*	**0**
ch_intrinsics	0	*(100%)*	0	*(100%)*	0	*(100%)*	**0**
diffeq1	0	*(100%)*	0	*(100%)*	0	*(100%)*	**0**
mkPktMerge	0	*(100%)*	0	*(100%)*	0	*(100%)*	**0**
sha	5245	*(820%)*	996	*(156%)*	629	*(98%)*	**640**
mkSMAdapter4B	2118	*(3103%)*	147	*(216%)*	76	*(111%)*	**68**
raygentop	3938	*(9938%)*	194	*(490%)*	51	*(130%)*	**40**
boundtop	13473	*(3269%)*	785	*(190%)*	427	*(103%)*	**412**
or1200	23975	*(906%)*	3274	*(124%)*	3646	*(138%)*	**2645**
blob_merge	134242	*(553%)*	35304	*(146%)*	24725	*(102%)*	**24264**
mkDelayWorker32B	29818	*(767%)*	4923	*(127%)*	4776	*(123%)*	**3889**
stereovision0	71320	*(14140%)*	818	*(162%)*	488	*(97%)*	**504**
stereovision1	164118	*(1163%)*	26304	*(186%)*	11087	*(79%)*	**14114**
LU8PEEng	1664017	*(1653%)*	224126	*(223%)*	104938	*(104%)*	**100683**
stereovision2	1651868	*(1112%)*	148722	*(100%)*	130772	*(88%)*	**148495**
bgm	2980165	*(1628%)*	546759	*(299%)*	184736	*(101%)*	**183106**
mcml	7171909	*(1249%)*	1219469	*(212%)*	453068	*(79%)*	**574189**
LU32PEEng	12463143	*(1143%)*	2364584	*(217%)*	1282125	*(118%)*	**1090425**
AVERAGE		***2208%***		***176%***		***104%***	

Table 11: **Extended** FieldPlacer + LocalRefinement **O**ver**U**se (*sorted ascendingly by* VPR SA *runtime*)

	FPGA design statistics								Graph representation statistics				
Design/Code	6LUTs	# CLBs		# MULs		MEM bits	# MEMs		# I/Os		$\|V_D\|$ # Blocks	# Connections	$\|E_D\|$ # Edges	Density	Domain
bgm	38537	**2930**	(2961)	**11**	(120)	0	**0**	(80)	**289**	(2016)	3230	89115	57686	1.11%	Finance
blob merge	8067	**543**	(588)	**0**	(21)	0	**0**	(16)	**136**	(896)	679	14647	9406	4.09%	Image Processing
boundtop	3053	**233**	(234)	**0**	(8)	32768	**1**	(9)	**467**	(576)	701	5518	3327	1.36%	Ray Tracing
ch_intrinsics	425	**37**	(48)	**0**	(2)	256	**1**	(1)	**229**	(256)	267	810	560	1.58%	Memory Init
diffeq1	362	**36**	(140)	**5**	(6)	0	**0**	(4)	**258**	(448)	299	1267	661	1.48%	Mathematics
diffeq2	272	**27**	(140)	**5**	(6)	0	**0**	(4)	**162**	(448)	194	917	433	2.31%	Mathematics
LU32PEEng	86521	**7128**	(7154)	**32**	(288)	673328	**168**	(208)	**216**	(3136)	7544	220565	129453	0.45%	Mathematics
LU8PEEng	25251	**2104**	(2120)	**8**	(78)	46608	**45**	(56)	**216**	(1696)	2373	62254	36616	1.30%	Mathematics
mcml	107784	**6615**	(6745)	**30**	(276)	5210112	**159**	(180)	**69**	(3040)	6873	140593	81390	0.34%	Medical Physics
mkDelayWorker32B	5588	**447**	(1728)	**0**	(72)	532916	**43**	(48)	**1064**	(1536)	1554	11298	7178	0.59%	Packet Processing
mkPktMerge	239	**15**	(494)	**0**	(18)	7344	**15**	(16)	**467**	(832)	497	1146	614	0.50%	Packet Processing
mkSMAdapter4B	1960	**165**	(234)	**0**	(8)	4456	**5**	(9)	**400**	(576)	570	4063	2373	1.46%	Packet Processing
or1200	3075	**257**	(475)	**1**	(18)	2048	**2**	(12)	**779**	(800)	1039	7535	5048	0.94%	Soft Processor
raygentop	1884	**173**	(221)	**7**	(8)	5376	**1**	(4)	**544**	(544)	725	4300	2648	1.01%	Ray Tracing
sha	2001	**209**	(221)	**0**	(8)	0	**0**	(4)	**74**	(544)	283	4308	2793	7.00%	Cryptography
stereovision0	9567	**905**	(910)	**0**	(32)	0	**0**	(25)	**354**	(1120)	1259	11732	6223	0.79%	Computer Vision
stereovision1	9874	**889**	(1064)	**38**	(45)	0	**0**	(30)	**278**	(1216)	1205	16392	8579	1.18%	Computer Vision
stereovision2	11012	**2395**	(5504)	**213**	(231)	0	**0**	(154)	**331**	(2752)	2939	52377	26002	0.60%	Computer Vision
stereovision3	174	**13**	(20)	**0**	(0)	0	**0**	(0)	**41**	(160)	54	182	72	5.03%	Computer Vision

Table 12: Heterogeneous benchmark codes in *VPR 7.0* - **occupied** and (available) slots

Part V

How can repeated runs improve the placement?

*This chapter presents the **statistical framework** which enframes the* FieldPlacer *method to make use of repeated runs with further refinements in the graph layouting phase (**slack graph morphing**). In addition, the influence of randomized decisions can be exploited. While the force-directed graph layout is performed, the randomized decisions still influence the entire process, e. g., when inserting nodes on a finer grid 'near to' their representative on the coarser grid in the hierarchical process of* FM^3 *(more specifically* FieldFM^3*). Due to such slightly varying initial constellations, the graph can still converge to different local minima and the quality of the later embedded placement therefore still varies when performing repeated runs. However, as it was shown in Chapter 4, this influence is generally relatively small due to the multilevel approach. The framework enables the application of different (potentially mixed) target functions to meet the specific requirements in different use cases. The system can even automatically analyze after how many generated placements a 'significantly' good assignment has been found. To reduce the time of the repetitive process, it is furthermore investigated whether a repetition of the first graph layout (before performing the additional optimizations and the local refinement) already leads to good placements or if the entire process should be repeated several times.*

As a summary, this chapter delivers suggestions about how to use and improve the FieldPlacer *results in the statistical framework through repetitive runs.*

6

Repeated runs in a statistical framework

"Everything not saved will be lost."
— Nintendo 'Quit Screen' message —

Contents

6.1 THE FieldPlacer FRAMEWORK

The previous chapter presented the FieldPlacer method for heterogeneous FPGAs and showed that it can be used to create accurate placements that are comparable (in terms of different quality metrics) to those obtained by *VPR SA* but in much shorter time. The runtime advantage of the method was in fact shown to become larger with increasing input sizes.

However, an even higher quality in terms of different measures could be desired regardless of an increased placer runtime. This can especially be the case for *final* implementations that are synthesized into delivered products. It has already been shown that, self-evidently depending on the applied router, the placer runtime can be relatively small compared to the routing time. Thus, the FieldPlacer framework includes further functionalities to improve the placement at the price of a longer placer runtime. To this end, multiple placements are consecutively generated. Due to randomized assignments of node coordinates, the result may (even with the multilevel framework) vary and lead to different local optima. Anyhow, this effect should not be too immense due to the hierarchical layout procedure (see Section 5.5.2 and Section 4.2.2).

Instead of repeating the entire workflow, it could already be advantageous to repeat only the *basic* FieldPlacer (graph layouting phase for $\mathcal{G}_D^{\text{layout}}$ and embedding) while choosing the best embedding (subject to a predefined metric) and (potentially) proceed with the LocalRefinement to save the repeated and relatively high refinement runtimes (see Section 6.2). For this purpose, a *statistical framework* was implemented around the FieldPlacer which can *rate*, *backup* and *restore* the placements generated in repeated runs. Along with a predefined objective function (e. g., *minimize the critical path delay*), a *backup* of the best obtained placement is created and *restored* whenever a new placement is inferior to this one. In addition to the simple exploitation of 'randomness', the *slack graph morphing* procedure can be used to optimize for *low slack* in the system and, consequently, for a *small critical path delay* by incorporating a timing analysis to modify the graph model. This process is described in Section 6.3. Finally, combined target functions (e. g., *small critical path delay* ***and*** *small overuse*) are possible (see Section 6.4.3) and *adaptive termination criteria* in the repeated procedure are also applicable (see Section 6.6.1).

Figure 93 shows the framework's workflow which already includes the FieldPlacer and FieldOGDF from the previous chapter. It will be explained and applied in the following sections.

Remark 110. *The following sections will describe some possibilities of how the* FieldPlacer *framework* ***can*** *be used by means of examples. Several other use cases are possible, always depending on the needs of the situation.*

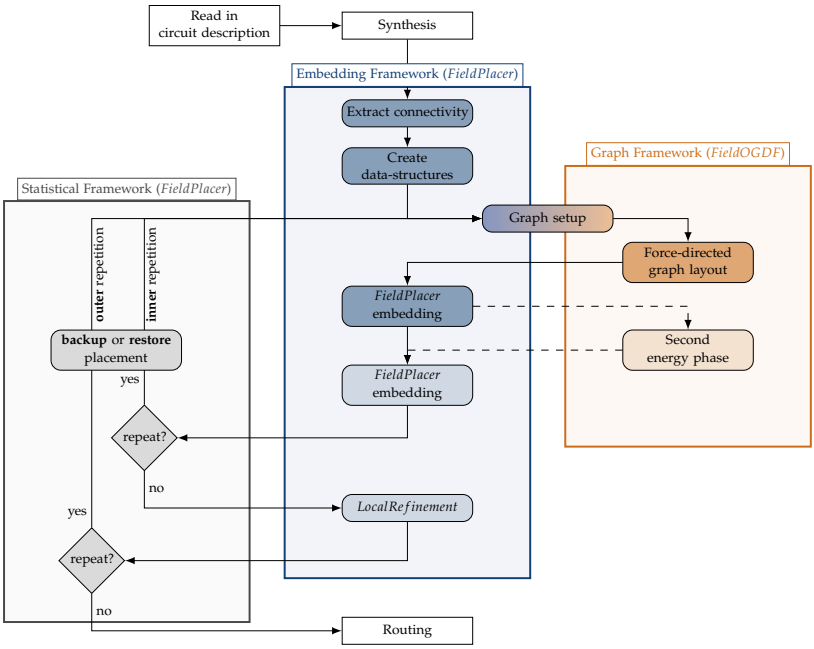


Figure 93: **Statistical framework** surrounding the `FieldPlacer`

Remark 111. *Following the results from the previous chapter, all benchmarks are conducted with the* MANHATTAN *metric for the graph layout (with* 45° *rotation) and the* DISTANCE *distribution for the CLB assignment (**without a second energy phase**).*

6.2 INNER AND OUTER REPETITIONS

Figure 93 depicts the *statistical framework* surrounding the `FieldPlacer` with the two already mentioned options for repetitions.

Assuming that a *local refinement* is principally desired to improve the quality of the placement, the first option is to perform *inner repetitions*. In this case, the *graph layout* is performed multiple times and each $\mathcal{G}_D^{\text{layout}}$ graph is embedded with the *basic* `FieldPlacer` method. Any of the presented metrics (*bounding box cost*, *wirelength*, *critical path delay*, *overuse*) can be used to rate these *basic* embeddings and if an embedding is better than the best one that has been found so far, the solution is backed and the next layout is created. This process can, for example, be repeated until a defined number of itera-

tions has been performed. After restoring the best found solution from the backup, the *local refinement* can furthermore improve this layout.

Instead of only repeating the *basic* `FieldPlacer` method in that way, the *outer repetition* option can be activated to repeat *the entire extended* `FieldPlacer` multiple times and choose the best placement (again subject to a predefined objective function). This process can either be repeated interactively by the user until a satisfying solution has been created or automatically with either a constant number of repetitions or with an adaptive termination criterion (see Section 6.6.1).

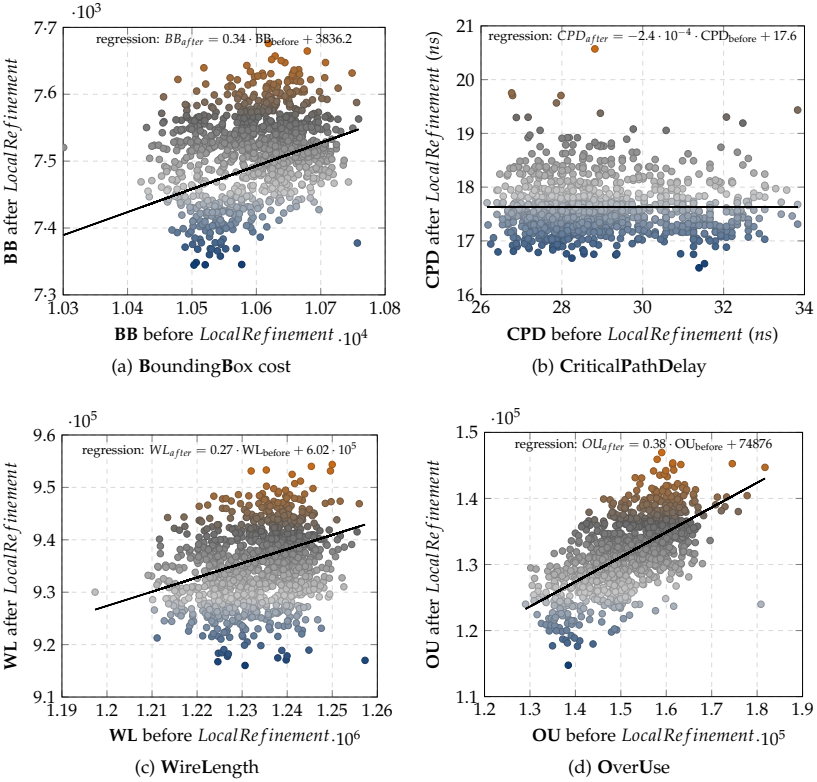
Repeating the entire *extended* `FieldPlacer` (including the *local refinement* in every repetition) is undoubtedly more time consuming than repeating only the *basic* `FieldPlacer` before a unique final *local refinement* phase. However, repeating only the *inner* part of the method would only make sense if there is a high correlation between the quality *before* and *after* the *local refinement* (concerning the chosen objective function). To investigate this, 1000 independent *extended* `FieldPlacer` layouts of the '*most average*' VPR benchmark code (`stereovision2`, see Section 5.6.3) have been created.

Due to the randomized decisions in the graph layouting phase, the quality of the results concerning the different norms varies. Moreover, the number of iterations in the spring embedder method is restricted (by using the default configuration, see Table 3) and thus the system may not have converged on all levels with the default configuration of the multilevel layout.

Remark 112. *Finding the 'right' parameters (like the maximal number of iterations for each graph representation and level) for the force-directed layout method, matching various inputs, is an art in itself. The default configuration taken from* FM^3 *works particularly well for many graphs (just as for most of the graph representations obtained from the FPGA designs in this work). However, more advanced techniques to set these parameters could be included in the future. Nevertheless, the method already combines several termination criteria by stopping, for example, either if the overall force in the system has converged or if the maximal number of iterations has been reached on each level (more details can be found in the dissertation of Hachul [81] as the termination criteria for the* $\mathcal{G}_{\mathrm{D}}^{layout}$ *layout phase have not been modified in* `FieldOGDF`*).*

Figure 94 shows scatter plots and the linear regression lines of the mentioned metrics *before* and *after* the local refinement. While a positive correlation between the measurements is principally present for the *bounding box cost*, the *wirelength* and the *overuse*, the slope of the regression line is far away from a perfect correlation (*with a slope of 1.0*). In fact, the results do *scatter* conspicuously around the regression line so that a clear trend is hardly distinguishable. The *best* correlation (*with a slope of 0.38 in this example measurements*) exists for the *overuse* metric. For this metric, a *good (small)* overuse

before the local refinement generally results in a *small* overuse *after* local refinement. For the *critical path delay*, no measurable correlation is present at all. This is due to the fact that the critical path delay is the maximal delay on *one* path in the layout and this norm is not as 'continuous' as the others are. While an improvement of the bounding box cost in the refinement phase does relatively directly lead to an improvement of the wirelength, this is not necessarily the case for the critical path delay. It depends on the *random* choice of logic blocks to be swapped in the refinement phase and on the fact, how often a pair of logic blocks is taken that can improve the critical path when being swapped. In addition, the critical path itself varies if, due to an improvement of one path, another one becomes the most critical one.



(a) **B**ounding**B**ox cost

(b) **C**ritical**P**ath**D**elay

(c) **W**ire**L**ength

(d) **O**ver**U**se

Figure 94: Correlation of quality before and after `LocalRefinement` (code: `stereovision2`)

As a summary it may be stated that the correlation between results before and after the local refinement is, though existent for several norms, not sufficiently high if results of highest quality are desired. This is especially the case for the important *critical path delay*, which rates the final *performance* of the chip design (in terms of *speed*). Thus, *outer repetitions* are applied in the following investigations.

One run of the entire VPR benchmark set with the FieldPlacer + LocalRefinement took 70.11 s on average while *VPR SA* needed 680.03 s. Thus, the *extended* FieldPlacer was (for this complete benchmark set on the test system) about 10 times faster then *VPR SA*.

For a fair comparison, the entire extended FieldPlacer method is consequently repeated 10 times to obtain better results in the following. As a result of this, the application of the *repeated extended* FieldPlacer method takes approximately the same time as the *VPR SA* approach for the complete benchmark set.

6.3 SLACK GRAPH MORPHING FOR IMPROVED CRITICAL PATH DELAY

Choosing the best placement out of several ones with *randomized* influences is one possibility to improve the final quality of the placement at the cost of a higher overall runtime. However, the FieldPlacer framework includes another mechanism to specifically improve the critical path delay involving the *timing analysis* of the entire design after each placement, which has to be performed to rate the critical path delay anyway. Sections 2.2.3 and 2.2.4 already introduced this method which finally estimates the *slack* on each path in the design. *Slack* appears if one signal has to wait for another one to proceed the signal processing. As a consequence, the critical path of a design is the path with *no slack* and *highest delay*. The slack is essentially a result of different amounts of delays on joining paths while the delays occur due to different delay types (see Section 2.2.2). While the *logic* and the *propagation delays* are independent from the placement, the *wire delay* depends on the distances between connected logic blocks and thus on the (Manhattan) *edge lengths* in the graph layout. Other influencing facts (e. g., more detailed *R*esistor-Capacitor effects) that depend on the detailed routing are not considered at that point.

The idea of the *slack graph morphing* in the FieldPlacer is as follows: A path with *high* slack finally has to 'wait' for other signals relatively *long* at the next *synchronization point*. Thus, the wires on such paths can be *elongated* without worsening the overall timing as long as the delay added by the elongation does not exceed the available slack. This opens up the possibility to spread such nodes further away from each other to let other nodes become more

close to each other. This opportunity can directly be used for paths with *small* or even *no* slack in the design. The connections on such paths could be *shrunk* to reduce the wire delay and possibly even the critical path (delay) of the design. Thus, the elongation of *un-critical* paths opens new possibilities for improvements of the *critical* ones (as the numbers of *slots* and *routing resources* on the architecture are both restricted). In this way, the system can improve the *timing* of the final layout in each repetition and an '*equilibrium slack state*' can be reached after a number of such iterations. During this process, the critical path (*not only its delay*) may even change as the optimization of one path can make another one critical.

Remark 113. *If negative slack is present in the design (e. g., due to user-defined timing constraints), all slack values are constantly shifted by the most negative slack to make all slack values* ***non-negative*** *and* ***maintain the ratios of slack to each other*** *in the system (cp. Remark 8).*

The *slack graph morphing* in the FieldPlacer framework is based on the following model. First, the *absolute slack* $\mathrm{slk}(e)$ on each connection/edge e is related to the average slack $\overline{\mathrm{slk}} = \frac{1}{|E|} \sum_{e \in E} \mathrm{slk}(e)$ in the placement in equation (61) to get a relative measure of the slack on each edge.

$$\begin{aligned} \widetilde{\mathrm{slk}}(e) &= \frac{\mathrm{slk}(e)}{\overline{\mathrm{slk}}} \\ &= \frac{\mathrm{slk}(e)}{\frac{1}{|E|} \sum_{e \in E} \mathrm{slk}(e)} \\ &= |E| \frac{\mathrm{slk}(e)}{\sum_{e \in E} \mathrm{slk}(e)} \end{aligned} \tag{61}$$

A resulting value of $\widetilde{\mathrm{slk}}(e) = 1.0$ consequently means that the slack on the edge e is just on the average of the design. A value that is *smaller* than 1.0 corresponds to a connection with lower-than-average slack (*more critical*) and connections with $\widetilde{\mathrm{slk}}(e) > 1.0$ have an over-average amount of slack (*less critical*). Using this information, *critical* edges should be *shrunk* while *uncritical* edges can be *lengthened*. The FieldPlacer framework uses the *zero-energy length* in the force-system of the spring embedder method to adjust such imbalances (see Section 4, especially Sections 4.1.2 and 4.2). By default (and therefore also in the first iteration of the repeated flow), the *zero-energy length* is set to $l^{\mathrm{zero}}(e) = 1.0$ for all edges. After every iteration, this value is updated by averaging the *actual zero-energy length* of each edge with the *new*

calculated relative slack value obtained from the new layout's timing analysis (see equation (62)).

$$l_{new}^{zero}(e) = \frac{l_{old}^{zero}(e) + \widetilde{slk}_{new}(e)}{2} \tag{62}$$

Remark 114. *For a more aggressive shrinking,* $(l_{new}^{zero}(e))^b$ *with* $b > 1$ *could be used instead of the applied case with* $b = 1$.

In each repetition, the sum of all $\widetilde{slk}(e)$ values is $|E|$. Thus, the *relative amount* of elongation of edges is exactly compensated by shrinking of other edges (see equation (63)).

$$\begin{aligned}
\sum_{e\in E} \widetilde{slk}(e) &= \sum_{e\in E} |E| \frac{slk(e)}{\sum\limits_{e\in E} slk(e)} && \text{(see equation (61))} \\
&= |E| \sum_{e\in E} \frac{slk(e)}{\sum\limits_{e\in E} slk(e)} \\
&= |E| \frac{1}{\sum\limits_{e\in E} slk(e)} \sum_{e\in E} slk(e) \\
&= |E| && (63)
\end{aligned}$$

For this part of the FieldPlacer framework in particular, the option of *removing parallel (multiple) edges* should be activated to make an accurate steering of connections' lengths possible. As already mentioned, the FM^3 method is exceptionally well suited for this purpose as it creates layouts that meet the requirements of the defined *zero-energy lengths* very precisely (see Remark 49). When multiple edges $e_1, e_2, \ldots, e_n$ have been combined to one common edge e within the FieldPlacer framework, this resulting edge e consequently gets a slack of $\widetilde{slk}(e) = \min\limits_{i=1,\ldots,n} \widetilde{slk}(e_i)$ to be as restrictive as possible.

Remark 115. *In each repetition, the presented shrinking method with equation (62) maximally halves edges' lengths that are part of the critical path (with* $\widetilde{slk}_{new} = 0.0$*) as all edges initially have a zero-energy length of* 1.0.

The following section will evaluate experimental results of the *slack graph morphing* in particular and of repeated runs in the FieldPlacer framework with various objectives in general.

6.4 BENCHMARK: REPEATED FIELDPLACER RUNS

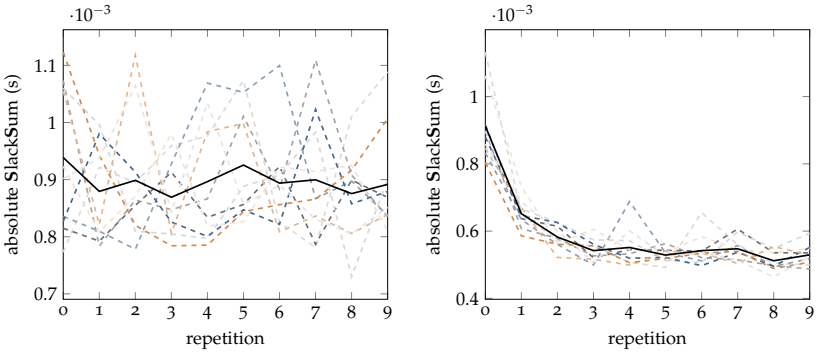
The correlations between *bounding box cost and wirelength* or *bounding box cost and critical path delay* of a design, and also the anticorrelation between *bounding box cost and overuse* (and the connection between *overuse and routability*), have already been shown in Section 5.5.6. Thus, only the two main contradicting targets *critical path delay* and *overuse* will be considered in the following.

6.4.1 *Slack graph morphing for improved critical path delay*

Due to the fact that the *critical path delay* is *the* measure of resulting speed of a circuit and as the *estimated critical path delay after placement* correlates very well with the *final critical path delay after routing* (see Figure 73), this target is often *the prevalent optimization goal*. Section 6.3 already introduced the *slack graph morphing* procedure in the `FieldPlacer` framework which modifies the desired zero-energy lengths for edges in $\mathcal{G}_D^{\text{layout}}$ in the force system of the spring embedder routine towards better timing by reduced slack on connections.

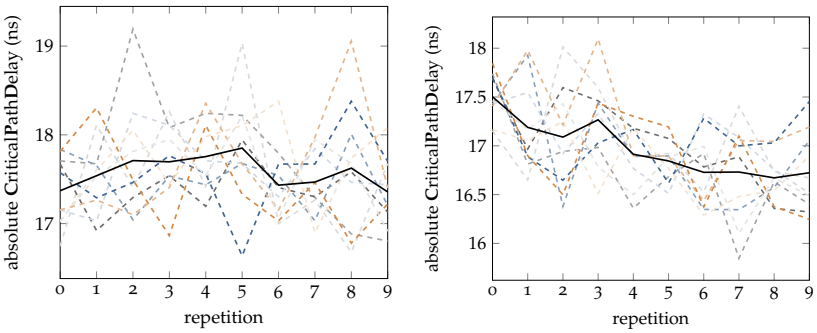
Figure 95a depicts the *slack sum* of the entire design when performing 10 (*outer*) repetitions **without** the *slack graph morphing* functionality. The experiment as a whole was repeated 10 times and the resulting slack sums are depicted as dashed 'curves' in different colors. The **black** line marks the average value of these 10 experiments for each repetition. The results obtained have no recognizable trend but are (*randomly*) sometimes slightly better and sometimes worse. The figure shows that simply choosing the best obtained solution would result in a slack sum of $7.299 \cdot 10^{-4}$ seconds while it is unpredictable in which iteration such *good* results will be achieved. On the average (the *black line*), the results remain rather constant during the repetitions.

In contrast to these results, Figure 95b shows the behavior **with** the *slack graph morphing* functionality activated. It is immediately plain to see that the overall *slack sum* tends to decrease in the repetitive process in each of the 10 experiments and, as a result, also on average. Already *after one morphing iteration* (repetition), the average slack sum is smaller ($6.518 \cdot 10^{-4}$ seconds) than the best result obtained by the previously presented 10 random runs. As early as after some very few iterations, the slack sum stagnates to a steady *equilibrium* state. Appendix A.8 visualizes the logic blocks and the slack on their interconnections for the first four repetitions of this process. The time series of graphs in this appendix thereby depicts the convergence to the steady state. After 10 iterations, the best generated placement has an average slack sum of only $4.641 \cdot 10^{-4}$ seconds.



(a) **without** slack graph morphing (b) **with** slack graph morphing

Figure 95: **S**lack**S**um in repeated runs (code: `stereovision2`)



(a) **without** slack graph morphing (b) **with** slack graph morphing

Figure 96: **C**ritical**P**ath**D**elay in repeated runs (code: `stereovision2`)

Even though a small *slack sum* in the placed design tends to facilitate a small *critical path delay*, it has been explained before that this is not necessarily a one-to-one correspondence. Figure 96 shows the results for the *critical path delay* (cp. with Figure 95). Again, **without** the *slack graph morphing* (see Figure 96a), the obtained results are randomly better or worse in each repetition. **With** activated *slack graph morphing* (see Figure 96a), the critical path delay also almost continually decreases with every repetition. However, the trend is (*as expected*) by far not as clear as the one of the *slack graph sum*. However, finally the *critical path delay* can be remarkably reduced by *affecting* the slack on connections *through the wire delay* in this method.

Figure 97 depicts the *critical path delay* for each code after the application of the *extended* FieldPlacer without repetitive runs (FieldPlacer + LocalRefinement), with 10 *outer repetitions* but *without* the slack graph morphing (...+ OUTER REP) and, finally, additionally *with* the slack graph morphing (...+ SlackGraph). Figure 98 contains the overview for the entire benchmark set including the other metrics.

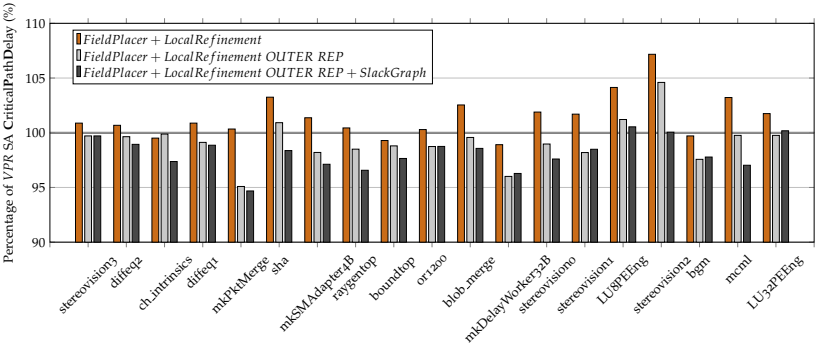
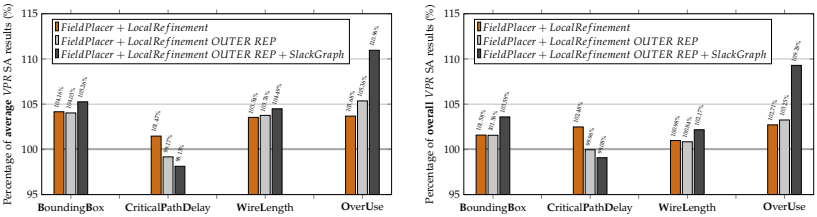


Figure 97: **C**ritical**P**ath**D**elay results for all codes (*sorted ascendingly by* VPR SA *runtime*)



(a) **Average** (b) **Overall (total)**

Figure 98: FieldPlacer + LocalRefinement + Repetitions

While the repetitive runs already contribute to improve the critical path delay (simply by choosing the placement with the smallest critical path delay out of 10 outer repetitions), the slack graph morphing improves the results even further. Although the critical path delay is reduced in that way, the counteracting overuse norm is instead increased. This effect has been discussed before in Chapter 5. Finally, the *critical path delay* of the *extended* FieldPlacer with *slack graph morphing* yields better results than *VPR SA*.

6.4.2 *Backup and restore for improved overuse*

Instead of optimizing for a smallest critical path delay, an improved routability can be desired to achieve more 'relaxed' routings with smaller channel widths or shorter routing times.

As for the critical path delay *without* slack graph morphing, *outer repetitions* can be performed while accepting a new placement if it improves the currently best one (in this case in terms of smallest *overuse*).

The final results (after 10 *outer repetitions*) for each code are shown in Figure 99. An overview with all metrics is given in Figure 100.

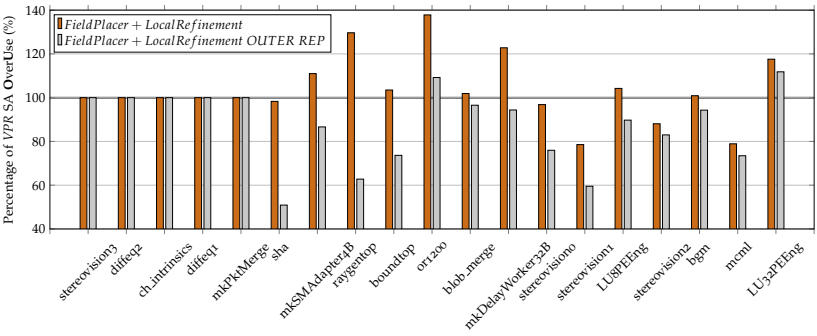
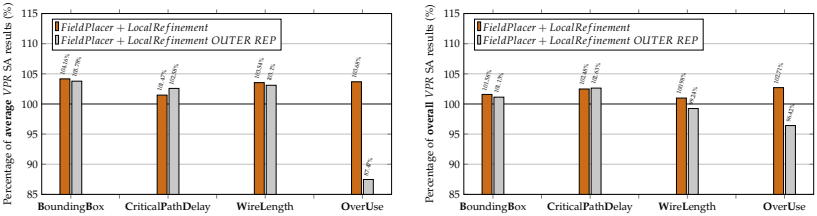


Figure 99: **O**ver**U**se results for all codes (*sorted ascendingly by* VPR SA *runtime*)



(a) **Average** (b) **Overall (total)**

Figure 100: `FieldPlacer` + `LocalRefinement` + Repetitions

As for the critical path delay, the outer repetitions improve the results in the direction of the desired target (small *overuse* in this case). On the other hand, the *critical path delay* is slightly increased by this choice. The counteracting characteristic of both metrics explains this fact once again. The *repeated*

extended FieldPlacer method can yield better results than *VPR SA* for most of all the individual codes and also in the general view.

While the previous two benchmark sets showed that the repeated runs can improve the resulting quality for *one or the other measure*, the following section will describe how *combined* target functions can be applied in repeated runs.

6.4.3 *Combined target function*

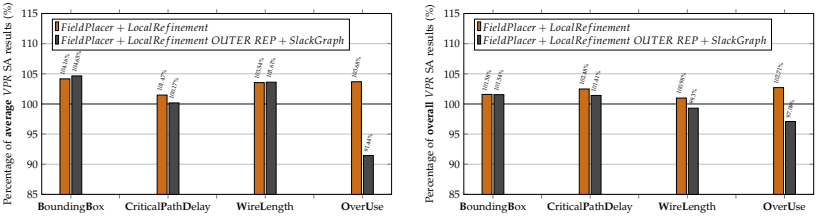
Explicitly improving the *critical path delay* in repeated runs worsens the *overuse* in the resulting layout and vice versa due to the principally contradicting characteristics of both targets. However, an 'optimization' in both directions simultaneously can often be desirable. Thus, the FieldPlacer framework includes the ability to use a combined measure to choose a layout with possibly small ***C**ritical**P**ath**D**elay* **and** small ***O**ver**U**se* at the same time. The *trade-off* can simply be steered by a function representing the *convex combination* of all desired targets.

Consider n measures of cost $c_0^j, \ldots, c_{n-1}^j$ of a placement generated in repetition j. Every run ($j = 0, \ldots, r-1$) generates such a set of measures so that c_i^j denotes the placement's cost of type i in repetition j. As the different cost measures can be in very different number ranges (e. g., *critical path delay* in nanoseconds vs. *overuse*), each cost value c_i^j of a repetition ($j > 0$) is scaled by the first obtained value (c_i^0). Consequently, the relative cost measure $\tilde{c}_i^j = \frac{c_i^j}{c_i^0}$ is used to transform all cost values to a more common scale. Thus, the relative measure rates how much *better or worse* the costs in repetition j are compared to the *first* obtained layout. Due to this common scale, a convex combination can be used to provide an '*intuitive and meaningful*' trade-off between the different cost values (see equation (64)).

$$C^j = \sum_{i=0,\ldots,n-1} \lambda_i \cdot c_i^j \quad \text{with } \lambda_i \geqslant 0 \text{ and } \sum_{i=0,\ldots,n-1} \lambda_i = 1 \tag{64}$$

C^j represents the combined measure for the placement generated in run j. As an example, a benchmark set combining the two already investigated and basically contradicting measures *critical path delay* (c_0) and *overuse* (c_1) in equal shares ($\lambda_0 = \lambda_1 = 0.5$) was executed.

With the combined cost function and activated *slack graph morphing*, good layouts concerning the desired trade-off between the two measures are chosen and, as a result, both metrics are improved on the average (see Figure 101). In fact, *all cost measures* are further improved by the application of this combined target (*in total*). Figure 102 contains the resulting costs for each code and for both cost types.



(a) **Average** (b) **Overall (total)**

Figure 101: `FieldPlacer+LocalRefinement` + Repetitions (combined target)

Remark 116. *For the smallest five benchmark codes, the* ***overuse*** *is always* 0 *and therefore exactly the same for the different approaches (cp. Figures 99 and 102b).*

Remark 117. *Graph layouts ($\mathcal{G}_D^{layout}$) for all VPR benchmark codes are presented in Appendix A.7.*

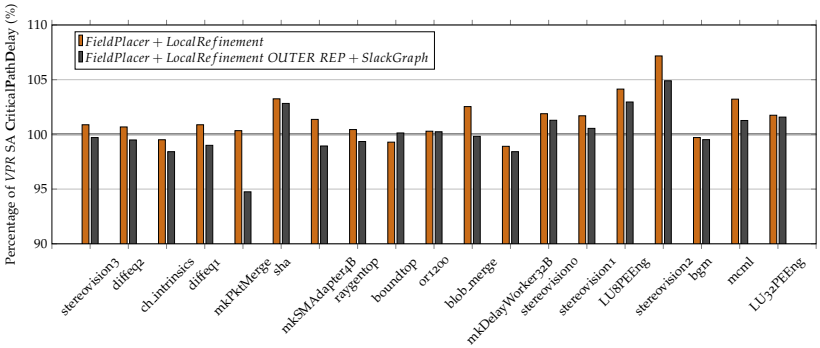
6.5 MCNC BENCHMARKS

It has been discussed in Section 2.2.5 that (*and why*) *heterogeneous* FPGAs become more and more important in the field of reconfigurable logic architectures. The `FieldPlacer` approach is specifically targeted at creating placements for such *heterogeneous* FPGA architectures. However, a comparison for a subset of the former '*classical*' and frequently applied benchmark set from the *Microelectronics Center of North Carolina* (MCNC) [188] for *general gate arrays* (not only FPGAs) is briefly presented in the following. These are 20 *large* circuits from the MCNC set which are, in addition to *VTR*, provided by the University of Toronto[1]. These codes are the basis for evaluation in their '*FPGA Place-and-Route Challenge*'[2].

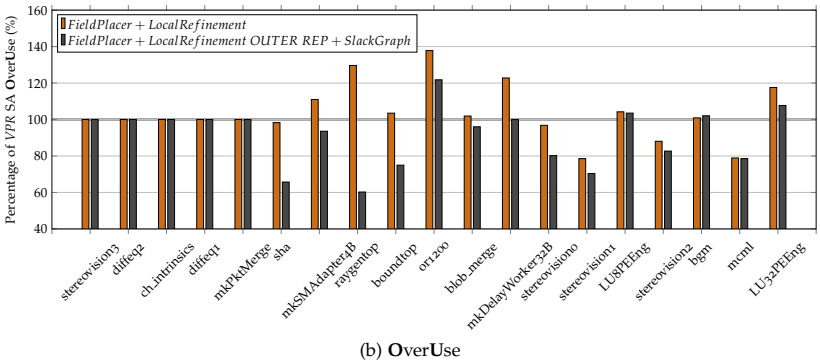
Instead of creating placements for the *heterogeneous* flagship architecture of VTR with the '*Comprehensive Architecture*' file (cp. Section 5.3), the classical *homogeneous* k4_N4_90nm architecture with four 4-LUTs per logic cluster and no hard blocks has been used for these rather 'simple' designs (see Luu et al. [130]). The `FieldPlacer` approach adapts to the architecture automatically. For example, *without special blocks*, the 5th step in the flow is skipped.

The designs in the MCNC benchmark set are relatively small. The largest example in the set is the `clma` code. After packing, it consists of just 2298 blocks (2154 'smaller' CLBs, 62 input and 82 output pins) while the largest VTR code `LU32PEEng` contains 7544 blocks (7128 CLBs, 32 MULs, 168 MEMs,

1 http://www.eecg.toronto.edu/~vaughn/vpr/download.html (accessed 15 Jul 2016)
2 http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html (accessed 16 Jul 2016)



(a) **C**ritical**P**ath**D**elay



(b) **O**ver**U**se

Figure 102: **C**ritical**P**ath**D**elay and **O**ver**U**se results for all codes (*sorted ascendingly by* VPR SA *runtime*)

114 input and 102 output pins, see Table 12) paired with almost 10 times as many nets in the circuit. Thus, the runtime of the MCNC benchmarks is still relatively small (from today's perspective). Details about the MCNC codes can, for example, be found on the referenced homepage of the '*FPGA Place-and-Route Challenge*'. Consequently, the runtime advantage of the `FieldPlacer` framework is not as remarkable as for the largest *VTR* benchmarks. Figure 103 shows the average runtime of single *extended* `FieldPlacer` runs *with* `LocalRefinement` (once again relative to the respective *VPR SA* runtimes and sorted ascendingly by these).

Like for the VTR benchmarks, the advantage of the `FieldPlacer` concerning the runtime rises with increasing input size. The relative runtime of the graph layouting is smaller for larger inputs and the time for the actual *embedding* with the `FieldPlacer` is relatively small.

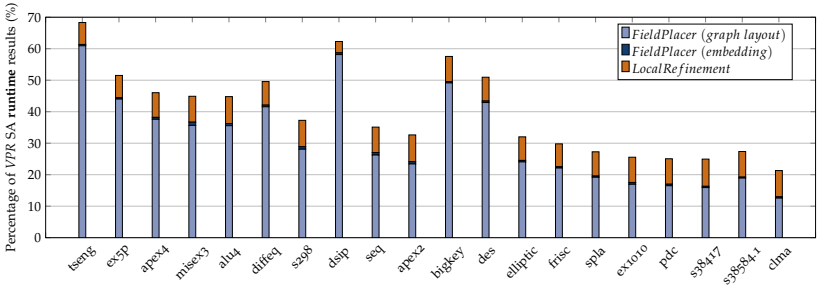


Figure 103: MCNC FieldPlacer runtime results (DISTANCE penalties, *sorted ascendingly by* VPR SA *runtime*)

Figure 105 shows that the quality concerning the different measures is improved by every 'evaluation level'. For a direct comparison, the *outer* repetitions are again performed 10 times *with activated slack graph morphing* for improved critical path delay (which pays off). The quality that has finally been achieved is comparable to the one from the *VTR benchmarks*. However, it has to be noted that the 10 FieldPlacer repetitions for this benchmark set took longer than one *VPR SA* run (due to the smaller input codes). Nevertheless, 10 repetitions are applied for the comparability to the previous sections. Section 6.6 presents *adaptive stopping criteria* to overcome the need of statically a priori defined numbers of repetitions.

Figure 104 shows the detailed *critical path delay* results for all codes in the set, once again relative to the *VPR SA* result (repeated 10 times and averaged as before).

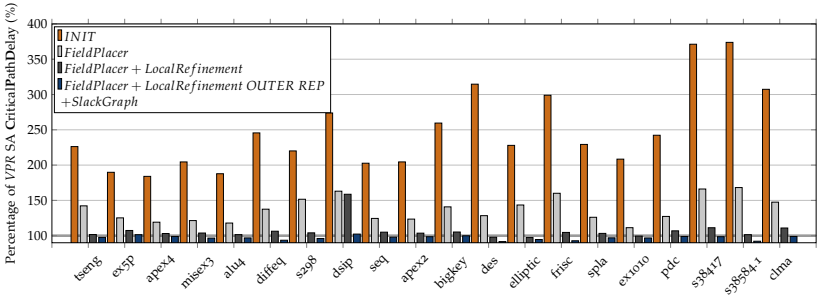
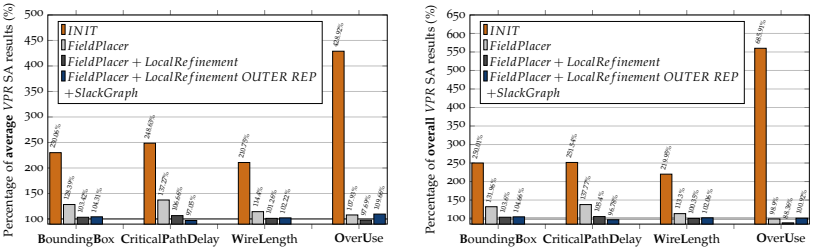


Figure 104: MCNC **C**ritical**P**ath**D**elay (DISTANCE penalties, *sorted ascendingly by* VPR SA *runtime*)



(a) **Average** (b) **Overall (total)**

Figure 105: MCNC `FieldPlacer` framework - Overview

Remark 118. *The MCNC benchmarks contain codes with much* ***more than one component*** *in the design. Thus, these codes can additionally be consulted to confirm the working behavior of the* `FieldPlacer` *in such cases.*

6.6 STATISTICS FOR SIGNIFICANTLY GOOD PLACEMENTS

In the previous sections, it was shown how repeated runs can improve the quality of a placement with different optimization targets. Both *adaptive* and *randomized* influences can therefore be taken into account to choose a '*good*' placement out of several ones.

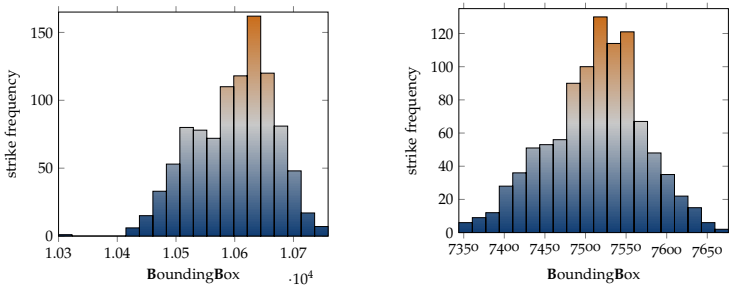
In an *interactive* case of application, a user may simply decide when a sufficiently good placement has been found and thus when to stop the placement procedure and proceed, for instance, with the routing. However, an automatic approach to find a good placement is often desirable. Even though the *extended* `FieldPlacer` method (without repetitions) leads to good results in short times, a higher quality can in many cases be preferable at the price of longer runtimes. In the former *repeated* benchmarks, the entire *extended* `FieldPlacer` was repeated 10 times to end up with runtimes that are comparable to *VPR SA* for the entire benchmark set. However, the best placement was often found after only very few iterations so that the flow could have terminated earlier.

In addition, for *different codes* and *different metrics*, the deviation of the resulting quality due to the *random influences* was *larger or smaller*. To investigate this further, the *average* `stereovision2` code was placed 1000 times with the previously defined and applied `FieldPlacer` 'standard' configuration (*DISTANCE distribution, MANHATTAN metric, 45° rotation, etc.*) but *without slack graph morphing* to reveal only the random effects in the repetitions.

Figures 106-109 show histograms for the obtained quality concerning the different metrics *before* and *after* the `LocalRefinement`.

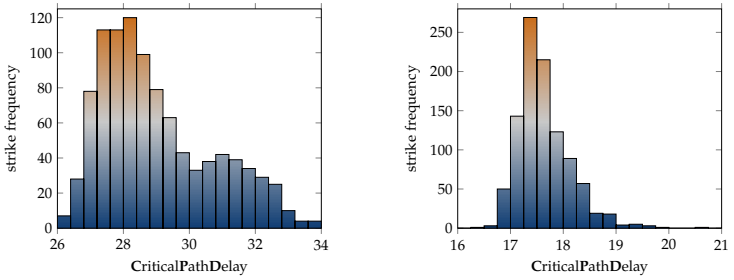
For each metric, the results tend towards a 'common state' (the *average layout*), which is based on the *similar* but not *identical* equilibrium state reached by the force-directed graph layout. Especially *after the local refinement, the picture is becoming even clearer*. Most of the placements have a similar quality as the *average placement* while *significantly better or worse* ones are rare.

In repeated runs, these rare significantly good layouts are the ones a user is interested in. As all the metrics are to be minimized, the left-end of the histograms contain the interesting layouts. To find such, it can be assumed that the layout's quality (in these cases and for these metrics) approximately follow a **normal distribution** due to the common *average state* in the force equilibrium towards which all layouts strive.



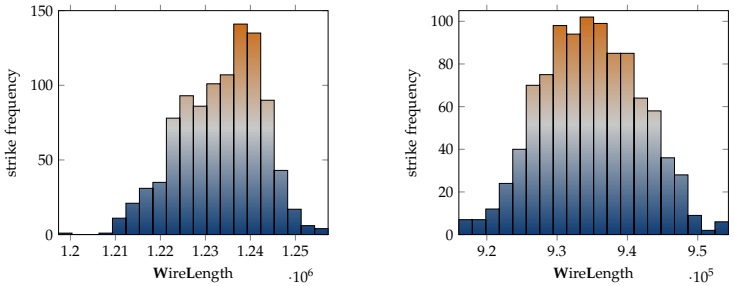
(a) before `LocalRefinement` (b) after `LocalRefinement`

Figure 106: **B**ounding**B**ox cost histogram (code: `stereovision2`, 1000 runs)



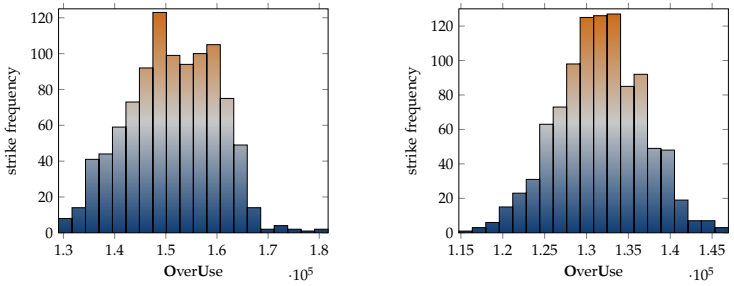
(a) before `LocalRefinement` (b) after `LocalRefinement`

Figure 107: **C**ritical**P**ath**D**elay histogram (code: `stereovision2`, 1000 runs)



(a) before `LocalRefinement` (b) after `LocalRefinement`

Figure 108: **W**ire**L**ength histogram (code: `stereovision2`, 1000 runs)



(a) before `LocalRefinement` (b) after `LocalRefinement`

Figure 109: **O**ver**U**se histogram (code: `stereovision2`, 1000 runs)

Remark 119. *The presence of the normal distribution is used in the following but is* ***not mathematically proved*** *in any sense. The assumption is based on the graphical interpretation of obtained results and on the fact that the layouts tend towards a common (****average****) equilibrium state with* ***randomized deviation****.*

The following section presents a method to define and find '*good*' layouts with *adaptive termination criteria*.

6.6.1 *Adaptive termination criteria*

Remark 120. *The following methodology is (due to the 'careless' assumption of the normal distribution)* ***not*** *intended to be mathematically (especially 'statistically') accurate or provable. It is an 'experimental' feature of the framework and several*

assumptions are made as they actually work very well in practice. The following explanations can therefore be regarded to be 'informal'.

When an optimization of the *critical path delay* with activated *slack graph morphing* is desired, the fact that the slack in the system tendentially decreases in every iteration could be used to terminate as soon as the improvement falls below a certain threshold. However, a general cost measure that follows a *normal distribution* can reach its *randomly influenced* optimum in any of the iterations. Hence, the *backup* and *restore* functionality has already been introduced to store the best placement that was found.

One question still has to be answered: How could the system determine if a '*significantly good*' placement has been found so that the system can finally *terminate* the search?

Remark 121. *By way of example, the* ***overuse*** *should be optimized by the system in the following. In general, the (possibly combined)* ***cost function value (CFV)*** *is optimized in the* `FieldPlacer` *framework.*

To find a measure for the term '*significantly good*' placement, the `FieldPlacer` framework can make use of the *confidence interval* (CI). Assuming that a *normally distributed* population is present, the value of one measurement lies in a *confidence interval* with a certain probability (see Figure 110). Cox and Hall [43] described it tangibly as follows:

'The confidence interval represents values for the population parameter for which the difference between the parameter and the observed estimate is not statistically significant at the 10% level'.

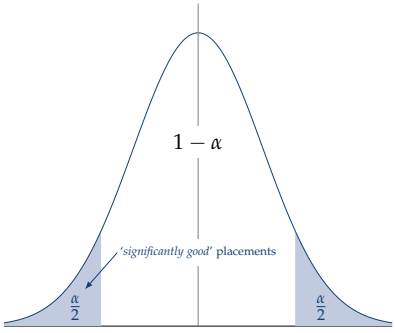


Figure 110: Confidence interval

However, the *real* distribution is not known within the process as the set of measurements only represents a *sample*. This is especially important in the beginning of the procedure when only very few placements have been

accomplished. By consequence, the *parameters* of the actual distribution have to be *estimated*. The first necessary one is the *sample mean*, which can simply be calculated by equation (65).

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{65}$$

The *sample variance* can be estimated by the *corrected sample variance* in equation (66).

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2 \tag{66}$$

In general, the *confidence interval's* size depends on the *size of the sample* n and the *corrected sample standard deviation* s (square root of the *sample variance*), precisely on $\frac{s}{\sqrt{n}}$. The smaller this ratio is, the larger is the interval. However, especially for small sample sizes, this value has to be extended to estimate a *confidence interval*. Consider a set of only two measurements with (*coincidentally*) almost identical *cost function values*. The *corrected sample standard deviation* s would be very small and the confidence interval would consequently also be very small. A *significantly good* sample would therefore often be detected only based on the fact that the sample size is too small.

To overcome this issue for normally distributed samples of small sizes, *William Sealy Gosset* developed the *t-distribution* (also called the '*Student's t-distribution*' as he originally published it under the pseudonym '*Student*'). It's application is, for example, described in the work of Fisher [62] from 1938.

The t-distribution (see Figure 111) can be applied to 'correct' the measured confidence interval parameters for small sample sizes (generally for $n < 30$). The samples are in fact very often that small when using the `FieldPlacer` method. The final *two-sided* confidence interval (for *minimization* or *maximization*) is defined by equation (67).

$$\left[\overline{x} - t_{(1-\frac{\alpha}{2};n-1)} \frac{s}{\sqrt{n}} \; ; \; \overline{x} + t_{(1-\frac{\alpha}{2};n-1)} \frac{s}{\sqrt{n}} \right] \tag{67}$$

The value $t_{(1-\frac{\alpha}{2};n-1)}$ is called the $(1-\frac{\alpha}{2})$-quantile of the t-distribution with $n-1$ degrees of freedom. The smaller the sample size and the higher the confidence level $(1-\alpha)$ is, the larger is $t_{(1-\frac{\alpha}{2};n-1)}$. With respect to the optimization direction (*minimization* or *maximization*), one or the other side of the two-sided confidence interval is of interest (see Figure 110).

To finally steer the *size* of the confidence interval and, therefore, the desired quality of the result, the confidence level can be varied in the `FieldPlacer` framework. Three predefined *confidence levels* of 0.75 (*small*), 0.975 (*medium*) and 0.998 (*high*) can be chosen at runtime whereas others are easily insertable.

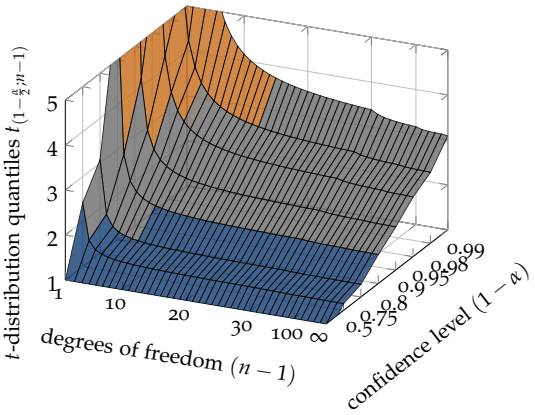


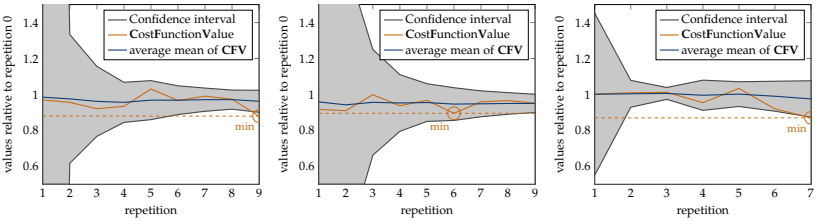
Figure 111: Interpolated $(1-\frac{\alpha}{2})$-quantiles of the t-distribution with $n-1$ degrees of freedom

Remark 122. *A t-distribution with an ever-growing sample size resembles in fact a normal distribution.*

Remark 123. *In the* `FieldPlacer` *implementation, the quantiles are approximatively taken from a respective lookup table with the depicted exact values from Figure 111.*

Figure 112 shows *three* such runs at *high* confidence level of the *average* `stereovision2` code from the *VTR* benchmark set. The first run (Figure 112a) terminates after 10 passes and reaches the optimum in the final run. The *cost function value* (*overuse* in this case) of this repetition is significantly good as it is smaller than the lower boundary of the confidence interval. The second run (Figure 112b) also terminates after 10 passes, but the optimal solution is found in repetition 6. The *backup and restore* functionality of the `FieldPlacer` framework ensures that this solution is finally recovered. The third run (Figure 112c) appreciably shows how an *outlier* (with respect to previously achieved results) enlarges the confidence interval after it continuously decreased (due to the growing n in the denominator of equation (67)) by extending the *corrected sample standard deviation* s by a factor of *four*. The high *variance* in the following repetitions keeps the interval almost constantly large by acting towards the decreasing t-quantile. The optimum is reached in repetition 7.

This strategy of automatic termination can be used to find significantly good layouts without wasting too much time and without the need to define a 'meaningful' static number of repetitions. This adds a further dimension of flexibility to the framework, which is particularly interesting for *non-*



(a) Minimum in rep 9 of 9 (b) Minimum in rep 6 of 9 (c) Minimum in rep 7 of 7

Figure 112: Adaptive termination criterion for **O**ver**U**se (code: `stereovision2`)

interactive use cases. However, the outcome still depends on the choice of the accuracy (*small, medium, high*). The confidence interval could additionally be scaled if a larger or smaller number of repetitions is desired. In any case, the involvement of the statistical parameters (*mean* and *variance*) makes the criterion adapt to the specific sample.

6.7 GRAPHICAL USER INTERFACE (GUI)

As mentioned before, the setup of the `FieldPlacer` can be conducted via a configuration header. In an *interactive* use case, a *GUI* additionally provides runtime access to all mentioned options. Figure 113 shows how this GUI extends the original *VPR GUI*. The `FieldPlacer` GUI only appears if the added `FieldPlacer` button in the *VPR* GUI is pressed. Otherwise, the original workflow can be continued with the `Proceed` button. Choosing the `FieldPlacer` option opens a new window including the `FieldPlacer` method's controls (*on the right*) and a summary of the last 10 `FieldPlacer` repetitions or runs (*on the left*). After potentially varying the options, the new setup can instantly be used to find improved solutions with the `OPTIMIZE` button. Repeated runs can be continuously repeated in that way. While the line diagrams indicate the obtained quality metrics of every iteration, the bars represent the (potentially combined) *cost function value* (CFV) while the blue bar highlights the backed best solution found so far. Whenever the user decides to use the obtained solution, the surrounding workflow can be continued with the `PROCEED` button.

The basic configurations of the `FieldPlacer` applies *DISTANCE* penalties and uses the *MANHATTAN* metric with 45° rotation of the layout and the *slack graph morphing*, always with *removed parallel edges*. The *second energy phase* is *not activated* by default. The different **standard setups** (light blue buttons at the bottom) perform the following `FieldPlacer` setups:

- *ULTRAFAST*: *basic* FieldPlacer
- *FAST*: *extended* FieldPlacer (with LocalRefinement)
- *NORMAL*: *extended* FieldPlacer (with LocalRefinement) and automatically repeated runs at confidence level *medium* ($1 - \alpha = 0.95$)
- *ACCURATE*: *extended* FieldPlacer (with LocalRefinement) and automatically repeated runs at confidence level *high* ($1 - \alpha = 0.998$)

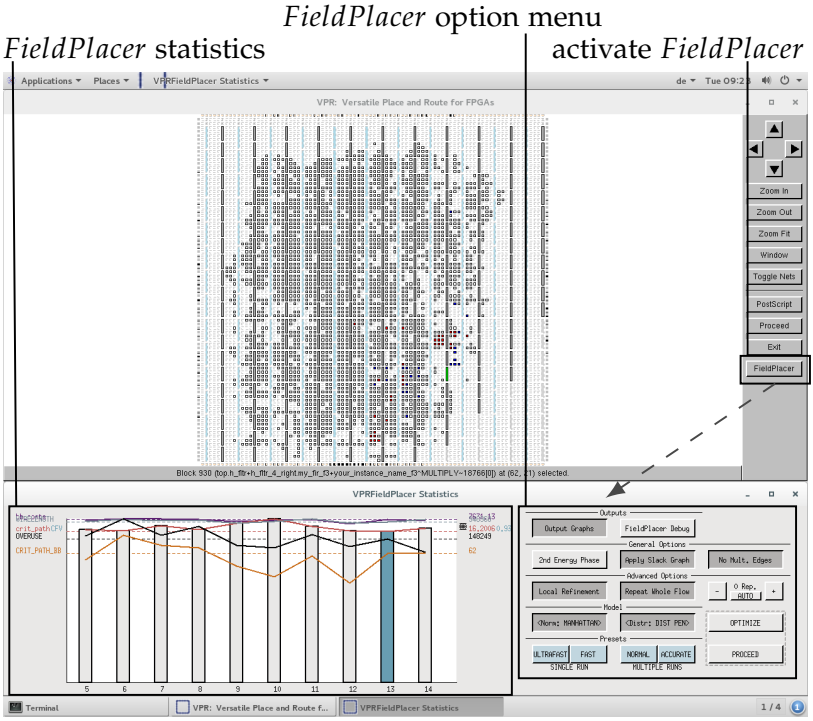


Figure 113: The FieldPlacer GUI

In addition to the mentioned 'methodological' options, a general *command line debugging mode* and the *graph debugger/tracer* (see Section 5.9.2) can be activated.

Remark 124. *Graph layouts for all VPR benchmark codes are presented in Appendix A.9 (automatically created by the graph debugger/tracer).*

Part VI

What does "the bigger picture" look like?

What has been achieved? This chapter briefly discusses the entire framework taking the benchmark results into account and summarizing what the overall goal of the presented method is. The FieldPlacer framework should neither be a black box nor a static FPGA design tool plugin. Instead, it could open a field of research to other researchers, e. g., from the domain of graph drawing.

My personal wish is that the framework introduces an easy-to-use 'scientific playground' in a field which is (normally) not readily accessible for researchers from other domains in order to finally contribute their own ideas in the future.

7

Discussion

"Phantasie ist wichtiger als Wissen. Wissen ist begrenzt, Phantasie aber umfasst die ganze Welt."
— Albert Einstein —

(Imagination is more important than knowledge. For knowledge is limited, whereas imagination encircles the world.)

Contents

7.1 RÉSUMÉ

This work introduced the FieldPlacer framework, a flexible, fast and unconstrained force-directed placement method for heterogeneous reconfigurable logic architectures.

It has been shown that force-directed methods can offer great advantages compared with traditional simulated annealing based approaches even though they also introduce further challenges. In this context, the theoretical foundations of the general placement problem have been discussed while the main focus has been on the special task of FPGA placement. In order to provide methodological bases for the developed framework, state-of-the-art techniques from the field of force-directed graph layouting were intensively investigated and compared culminating in the application and extension of the FM^3 algorithm.

FM^3, just as it is, already combines multiple desired characteristics of a force-directed layouting approach. While the *multilevel* functionality makes solutions relatively independent of an initial random assignment of the nodes, the multipole approach in turn speeds up the layouting process remarkably. In combination, force-directed graph layouts can be *created quickly* to form the basic sketch of the actual embedding on the integer grid of the reconfigurable logic architecture. The involvement of repulsive forces in the spring embedder simulation of FM^3 allows for an *entirely unconstrained* realization of the graph layout. This is a rather unique feature of the presented force-directed method as it allows *free* movements of all node types in the layouting phase (as the compared simulating annealing approach does). This results in layouts which are not influenced by a priori fixed nodes (e. g., surrounding I/O blocks). Furthermore, *all heterogeneous block types* of a design are included in the global layouting procedure to reach a force equilibrium state which considers all such types simultaneously.

For high *flexibility*, the FieldPlacer framework is able to perform the entire embedding into the integer grid of the chip architecture with *various configurable objectives*, e. g., small critical path delay or improved routability. A corresponding norm to quantify the routability of a placement has consequently been developed for the FieldPlacer method. In addition to that, how different optimization goals correlate to each other and how different extensions, like the application of the *Manhattan distance* in the force model right from the outset, can positively influence the layout by matching the architecture's properties has been investigated extensively. In the embedding phase of the FieldPlacer, additional methods from the field of graph drawing have been integrated, e. g., the *barycenter heuristic to reduce wirelengths and crossings* between I/O pins and logic blocks.

Even though the **basic** FieldPlacer embedding is *exceptionally fast*, it was emphasized that a local refinement can greatly improve the quality of the placement while preserving the desired properties of the layout obtained by each specific embedding strategy. Including these and other techniques, the **extended** FieldPlacer is able to create placements of *good quality* (at a comparable level to a state-of-the-art method) with freely definable objectives. However, the FieldPlacer method is, at the same time, 10 times faster than the compared state-of-the-art placer which is based on simulated annealing. More importantly, the runtime advantage of the FieldPlacer continuously increases with the size of the input design - the larger the design, the bigger the benefit! Thus, the advantage will grow in the future with yet larger inputs.

If a placement of *highest quality* (concerning the user-defined objective) is desired, the assignment on the chip can be further improved by **repeated** FieldPlacer runs. In combination with the *slack graph morphing* functionality, the critical path delay can be adaptively reduced in successive iterations. In general, variances in the quality of placements that result from randomly influenced decisions in the graph layouting procedure can also be exploited in the *statistical framework* of the FieldPlacer. Finally, objective functions combining different goals can be used to improve the quality of placements concerning several, even principally contradicting, goals at the same time.

Throughout the entire work, many benchmark results have been included in order to give suggestions for a meaningful usage the framework.

In summary, the FieldPlacer framework should provide an intuitive access to the field of chip placement for future researches and for researchers from different fields to *bridge the gaps* between FPGA development, general algorithm development and graph drawing.

7.2 COMPARISON & OUTLOOK

The work of Chang et al. [33] has already been mentioned in Section 5.1.1. In their comparison of different placement methods, they came to the conclusion that simulated annealing is able to generate good placements for *small* designs while considering multiple objectives simultaneously. However, *analytical placement* was described to be *efficient and scalable* for *large circuits* with *high quality* results and *multiple objectives*. The results obtained in this work support this thesis and provide a placer of such a kind for *heterogeneous* FPGA architectures.

It has additionally been stated how the runtime advantage of the FieldPlacer could further be increased in the future. As mentioned in Remark 63, the repulsive force calculation can, e. g., even be speeded up by

massively parallel architectures like GPUs. This has also been proposed for simulated annealing in Chong et al. [37]. However, GPU usage in the FieldPlacer could have huge advantages compared to many available simulated annealing approaches since necessary synchronization in simulated annealing makes it generally difficult to obtain an *effective* parallelization of the method on many cores (cp. Remark 67). On the contrary, parallel architectures will be able to improve FieldPlacer compilation time remarkably. As a result, many repeated runs could be performed in shorter times to generate even better placements in the statistical framework of the FieldPlacer. Due to the simple exchangeability of the force-directed layouting method, other approaches and different implementations, like the one of Gronemann, can directly be integrated in the future, e. g., to reduce the runtime of a single run (see Section 5.7). The local refinement approach could also be enhanced in the future. In addition, other stopping criteria for repeated runs are possible.

Furthermore, FM^3 has always been used in its default configuration in this work. Further adaptive features could help to guarantee, for example, that the system has 'completely' converged. Actually, the predefined constant maximal numbers of iterations on the multiple levels of FM^3's graph representation have frequently been exploited, especially for the finest level. In addition, it will be investigated if a smaller $p < 4$ in the p-term multipole expansion can speed up the calculation remarkably and keep the accuracy sufficiently high at the same time.

While Mak and Li [133] presented an approach that was able to create better placements than the current VPR placer at that time in slightly shorter runtimes (concerning both net and critical path *delay*), the FieldPlacer approach can generate improved assignments, too, especially if *repeated runs* are applied. However, the FieldPlacer approach can also create comparable placements much faster than VPR's placer. The flexibility of the framework offers different runtime-quality trade-offs for different situations, e. g., a single (**extended**) FieldPlacer run for rapid prototyping or multiple repeated runs for final products. It certainly also depends on the applied router, how much time to spend for the placement phase. An improvement of overuse in the placement through repeated runs could reduce the routing time remarkably so that the additional time that was spend in the placement can be more than offset.

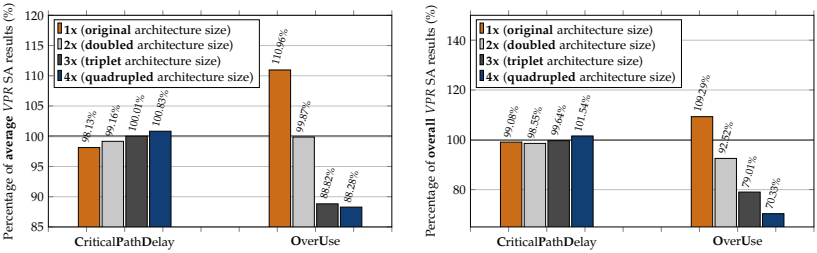
Even though Upadhyay [184] presented some comparable ideas for homogeneous FPGAs in 2015, his approach needs a priori placed and fixed I/O blocks. This was achieved by running the whole simulated annealing placement of VPR in advance. The placements that were generated by his approach for *homogeneous* FPGAs have been comparable to VPR's results at the cost of 11% higher runtime (not including the preceding VPR placement). Instead, the FieldPlacer generates a *free* (*unconstrained*) placement

for all block types through the inclusion of repulsive forces in its analytical placement while being remarkably faster than VPR's placer. However, a **basic** FieldPlacer run could be used in Upadhyay's method in order to obtain a good initial I/O assignment much faster than by applying simulated annealing in advance.

With version 7.0, the simulation of the energy consumption of implemented designs has been added to *VPR* (see Luu et al. [130]). Thus, strategies for *energy* optimization based on an adapted graph layouting or slot assignment should be included into the FieldPlacer framework in the future. It has to be investigated how the graph layout can influence the energy consumption in a positive manner. In addition, *thermal* considerations could be considered in the FieldPlacer framework.

7.3 A FINAL TEST CASE

Finally, further benchmarks brought to light that the FieldPlacer method performs particularly well in situations where the number of available slots on the architecture is much larger than the number of necessary ones. In all presented benchmarks in this work, the *simulated* architecture has been kept '*as small as possible and as large as necessary*' (*VPR*'s default strategy). In contrast to that, the size of the underlying simulated architecture has (for test purposes) been *oversized* in the following. More precisely, it was first estimated which architecture size would be necessary to make a placement possible (*by the use of the bisection method in VPR*). Then, this size was *doubled*, *tripled* and *quadrupled* to simulate an oversized architecture. Figure 114 shows the gathered average and overall quality measures for the entire VPR benchmark set. The results have been obtained by the **extended** FieldPlacer method with 10 repeated runs and activated *slack graph morphing*.



(a) **Average** (b) **Overall (total)**

Figure 114: Oversized architecture results

The available free slots have been exploited by the FieldPlacer's DISTANCE distribution to reduce the estimated *overuse* on the routing architecture, what in turn facilitates a subsequent routing. By imitating the force equilibrium obtained from the force-directed layout along with the *slack graph morphing functionality*, the final layouts tend to combine a *drastically reduced overuse* with a still *small critical path delay*. In general, the average critical path delays are at a level similar to that of the VPR results.

Remark 125. *In contrast to these results obtained with the DISTANCE distribution of the* FieldPlacer*, the critical path delay would be remarkably smaller when using the CENTER distribution instead (while the overuse would be accordingly larger).*

Such situations with large chips are indeed realistic as the FPGA architecture is generally much larger than the design itself. Instead of forcing all elements into one corner or the center of the chip, the DISTANCE distribution of the FieldPlacer spreads the logic blocks over the entire chip with respect to the arrangement in the force-directed layout. The reduction of overuse should also have a positive influence on the thermal distribution on the chip. Finally, these are promising results for such and other future researches with the FieldPlacer framework.

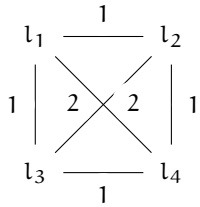
Part VII

Anything else?

Appendix

A.1 A detailed simple example for the QAP model

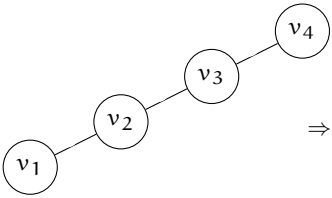
Distance-matrix $\mathcal{D}$ with Manhattan distance



$$\Rightarrow \mathcal{D} = \begin{array}{c} \\ l_1 \\ l_2 \\ l_3 \\ l_4 \end{array} \begin{array}{c} \begin{array}{cccc} l_1 & l_2 & l_3 & l_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \\ 1 & 2 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix} \end{array} \tag{68}$$

Figure 115: The Manhattan distances between four locations

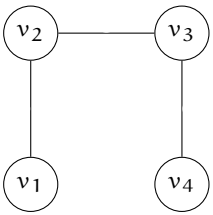
Connection-matrix $\mathcal{V}$



$$\Rightarrow \mathcal{V} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{69}$$

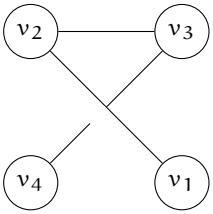
Figure 116: The connection-matrix of the graph's nodes

ASSIGNMENT-MATRIX $\mathcal{X}$



$$\Rightarrow \mathcal{X}_1 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} l_1 & l_2 & l_3 & l_4 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{70}$$

Figure 117: Exemplary optimal assignment of the graph



$$\Rightarrow \mathcal{X}_2 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} l_1 & l_2 & l_3 & l_4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{71}$$

Figure 118: Exemplary not optimal assignment of the graph

LOCATIONS' CONNECTION-MATRIX

$$\begin{aligned} \mathcal{V}_1^{loc} = \mathcal{X}_1^T \cdot \mathcal{V} \cdot \mathcal{X}_1 &= \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{aligned} \tag{72}$$

$$\mathcal{V}_2^{loc} = \mathcal{X}_2^T \cdot \mathcal{V} \cdot \mathcal{X}_2 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \tag{73}$$

THE COST CALCULATION

$$\begin{aligned} \tilde{c}_1 &= \frac{1}{2} \cdot \sum_{i=1}^{4} diag_i \left(\mathcal{X}_1^T \cdot \mathcal{V} \cdot \mathcal{X}_1 \cdot \mathcal{D} \right) \\ &= \frac{1}{2} \cdot \sum_{i=1}^{4} diag_i \left(\mathcal{V}_1^{loc} \cdot \mathcal{D} \right) \\ &= \frac{1}{2} \cdot \sum_{i=1}^{4} diag_i \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \\ 1 & 2 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix} \\ &= \frac{1}{2} \cdot \sum_{i=1}^{4} diag_i \begin{pmatrix} \textcircled{2} & 2 & 2 & 2 \\ 2 & \textcircled{2} & 2 & 2 \\ 0 & 1 & \textcircled{1} & 2 \\ 1 & 0 & 2 & \textcircled{1} \end{pmatrix} \\ &= \frac{1}{2} * (2+2+1+1) \\ &= \frac{1}{2} * 6 = 3 \end{aligned} \tag{74}$$

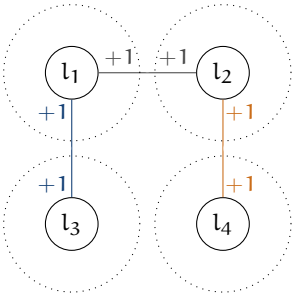
The assignment $\mathcal{X}_1$ of the graph causes costs of $\frac{1}{2} * (2+2+1+1) = 0.5 * 6 = 3$.

$$
\begin{aligned}
\tilde{c}_2 &= \frac{1}{2} \cdot \sum_{i=1}^{4} \mathrm{diag}_i \left(X_2^T \cdot \mathcal{V} \cdot X_2 \cdot \mathcal{D} \right) \\
&= \frac{1}{2} \cdot \sum_{i=1}^{4} \mathrm{diag}_i \left(\mathcal{V}_2^{loc} \cdot \mathcal{D} \right) \\
&= \frac{1}{2} \cdot \sum_{i=1}^{4} \mathrm{diag}_i \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \\ 1 & 2 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix} \\
&= \frac{1}{2} \cdot \sum_{i=1}^{4} \mathrm{diag}_i \begin{pmatrix} \textcircled{2} & 1 & 1 & 0 \\ 1 & \textcircled{2} & 0 & 1 \\ 3 & 1 & \textcircled{3} & 1 \\ 1 & 3 & 1 & \textcircled{3} \end{pmatrix} \\
&= \frac{1}{2} * (2+2+3+3) \\
&= \frac{1}{2} * 10 = 5
\end{aligned} \tag{75}
$$

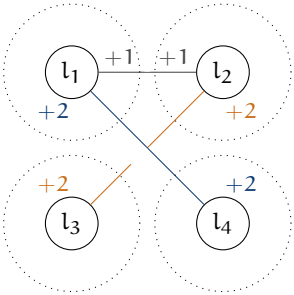
The assignment X_2 of the graph causes costs of $\frac{1}{2} * (3+3+2+2) = \frac{1}{2} * 10 = 5$.

ORIGINS OF THE COSTS

Cost origins:
l_1 causes costs of 2,
l_2 causes costs of 2,
l_3 causes costs of 1,
l_4 causes costs of 1.

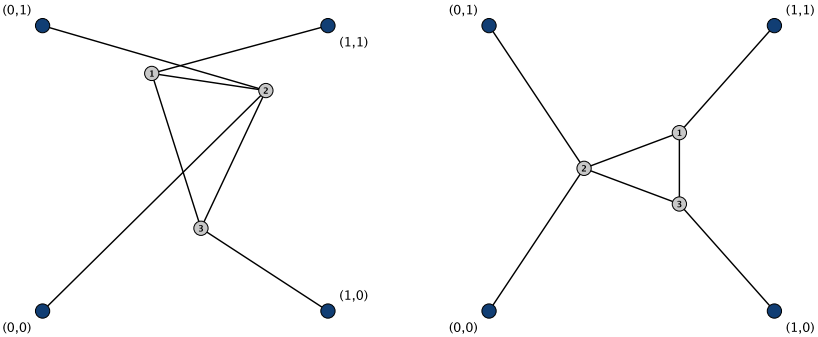


Cost origins:
l_1 causes costs of 3,
l_2 causes costs of 3,
l_3 causes costs of 2,
l_4 causes costs of 2.



A.2 A SIMPLE EXAMPLE FOR THE CALCULATION OF A TUTTE EMBEDDING

Consider the graph shown in Figure 119a. The vectors and the matrix of the linear equation systems are shown in equation (76), the Tutte embedding with the solutions x and y is visualized in Figure 119b.



(a) Random assignment of free nodes

(b) Tutte embedding

Figure 119: Tutte embedding - four fixed (blue) and three free (gray) nodes

$$
\begin{aligned}
A &= \begin{pmatrix} 3 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 3 \end{pmatrix}, b^x = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, b^y = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \\
x &= \begin{pmatrix} 0.667 \\ 0.333 \\ 0.667 \end{pmatrix}, y = \begin{pmatrix} 0.375 \\ 0.5 \\ 0.625 \end{pmatrix}
\end{aligned} \tag{76}
$$

A.3 FORCE-DIRECTED LAYOUT BY FRUCHTERMAN & REINGOLD OR FM^3

Figure 120 shows a random assignment of the nodes in the 'Crack' graph to a rectangular canvas. Figures 121a and 121b depict the resulting outcomes of the basic Fruchterman & Reingold approach and the result of FM^3 which is more than 20 faster due to the approximation of the repulsive forces. The close-ups verify that the results of both methods are, though not identical, very similar to each other.

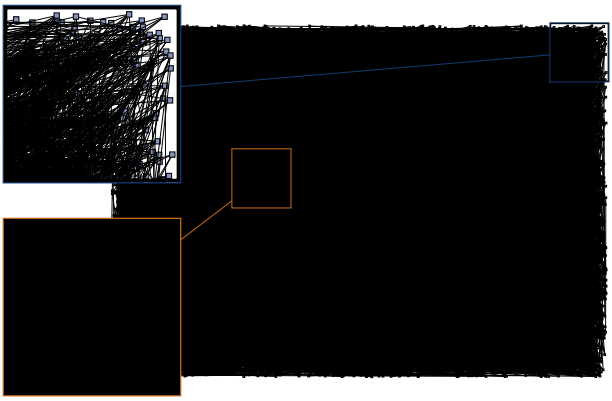


Figure 120: Random layout of 'Crack' Graph

(a) Fruchterman & Reingold (b) FM^3

Figure 121: 'Crack' graph layouts

A.4 GRAPH-THEORETICAL DISTANCE

An example for all graph-theoretical distances in a graph (to node 11) is shown in Figure 122.

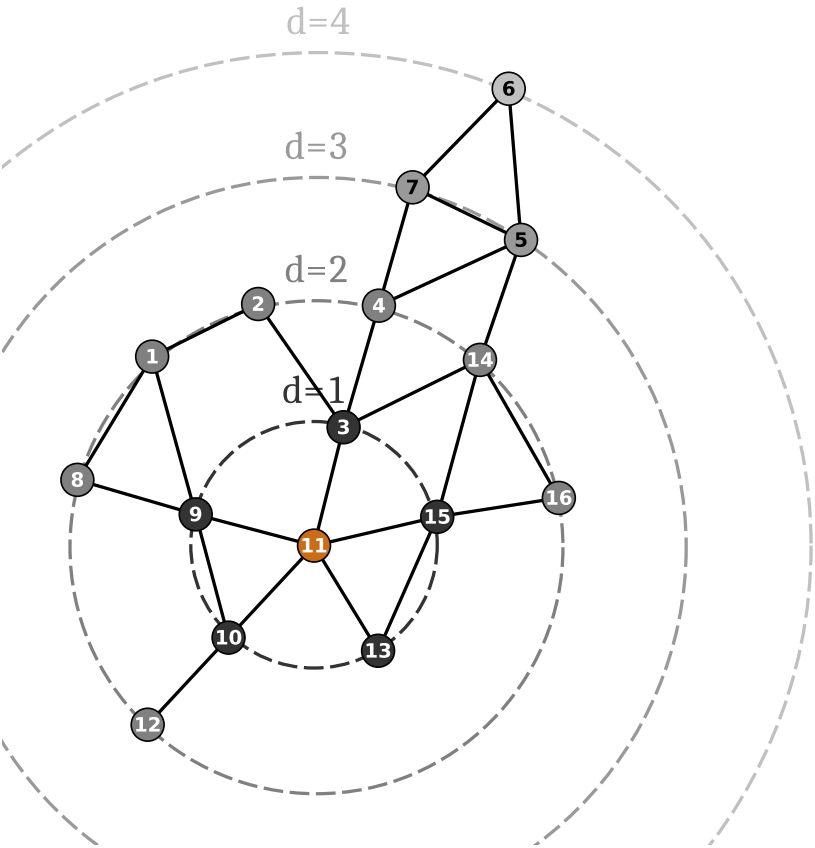
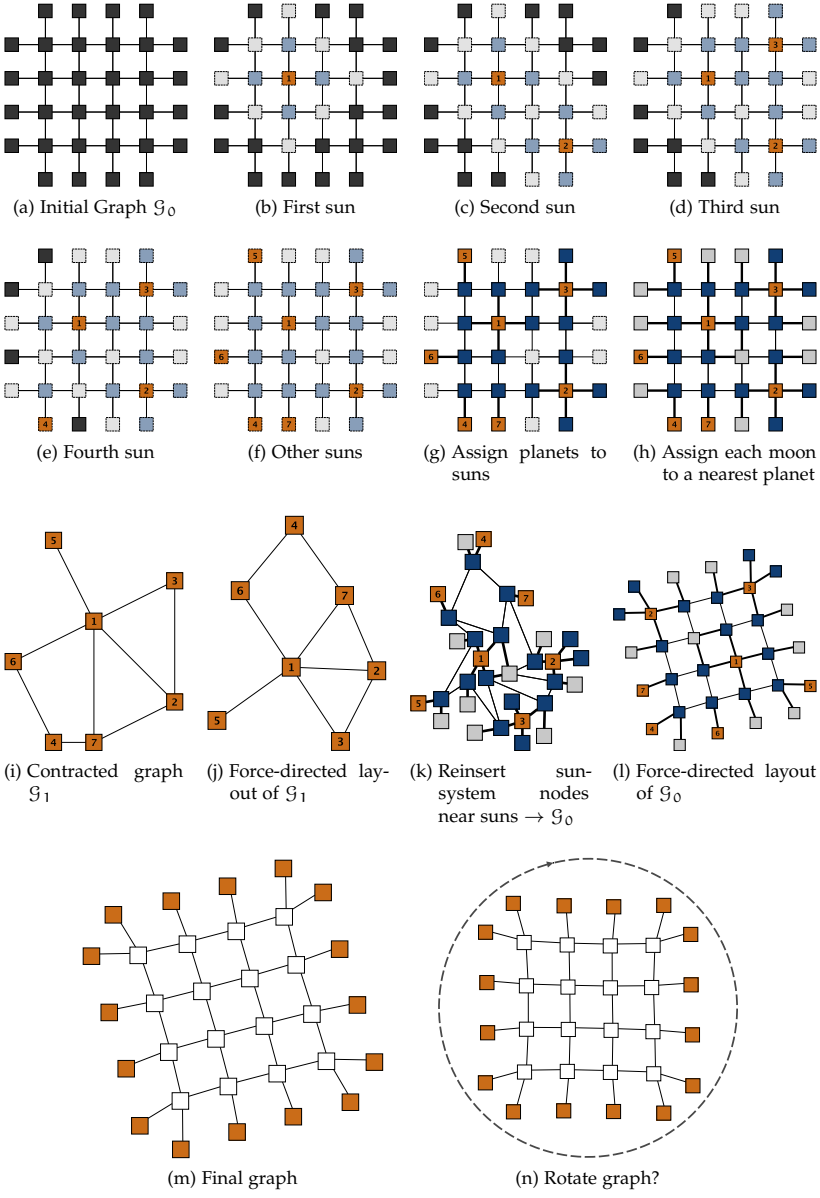


Figure 122: Graph-theoretical distance to node 11

A.5 MULTILEVEL CONSTRUCTION & APPLICATION

Figure 123 shows a detailed example of how one multilevel graph is constructed and how its coarsened representation is used to perform a coarse-grained layout of the sun systems before layouting on the finer level.



(a) Initial Graph $\mathcal{G}_0$ (b) First sun (c) Second sun (d) Third sun

(e) Fourth sun (f) Other suns (g) Assign planets to suns (h) Assign each moon to a nearest planet

(i) Contracted graph $\mathcal{G}_1$ (j) Force-directed layout of $\mathcal{G}_1$ (k) Reinsert sun-system nodes near suns $\rightarrow \mathcal{G}_0$ (l) Force-directed layout of $\mathcal{G}_0$

(m) Final graph (n) Rotate graph?

Figure 123: Multilevel construction and application

A.6 VPR DEFAULT CONFIGURATION

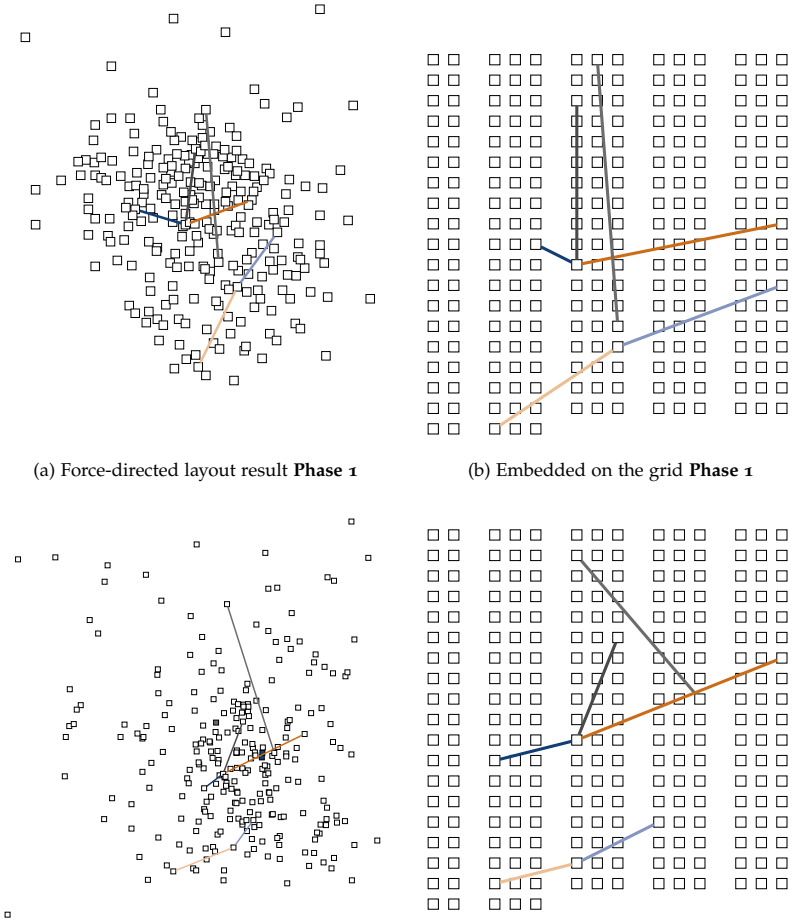
Listing 2 shows the defult configuration of VPR that is used for all benchmarks in this work.

```
PackerOpts.allow_early_exit: FALSE
PackerOpts.allow_unrelated_clustering: TRUE
PackerOpts.alpha_clustering: 0.750000
PackerOpts.aspect: 1.000000
PackerOpts.beta_clustering: 0.900000
PackerOpts.block_delay: 0.000000
PackerOpts.cluster_seed_type: TIMING
PackerOpts.connection_driven: TRUE
PackerOpts.global_clocks: TRUE
PackerOpts.hill_climbing_flag: FALSE
PackerOpts.inter_cluster_net_delay: 1.000000
PackerOpts.intra_cluster_net_delay: 0.000000
PackerOpts.recompute_timing_after: 32767
PackerOpts.sweep_hanging_nets_and_inputs: TRUE
PackerOpts.timing_driven: TRUE

PlacerOpts.place_freq: PLACE_ONCE
PlacerOpts.place_algorithm: PATH_TIMING_DRIVEN_PLACE
PlacerOpts.pad_loc_type: FREE
PlacerOpts.place_cost_exp: 1.000000
PlacerOpts.inner_loop_recompute_divider: 0
PlacerOpts.recompute_crit_iter: 1
PlacerOpts.timing_tradeoff: 0.500000
PlacerOpts.td_place_exp_first: 1.000000
PlacerOpts.td_place_exp_last: 8.000000
PlaceOpts.seed: 1
AnnealSched.type: AUTO_SCHED
AnnealSched.inner_num: 1.000000

RouterOpts.route_type: DETAILED
RouterOpts.router_algorithm: TIMING_DRIVEN
RouterOpts.base_cost_type: DELAY_NORMALIZED
RouterOpts.fixed_channel_width: NO_FIXED_CHANNEL_WIDTH
RouterOpts.acc_fac: 1.000000
RouterOpts.bb_factor: 3
RouterOpts.bend_cost: 0.000000
RouterOpts.first_iter_pres_fac: 0.500000
RouterOpts.initial_pres_fac: 0.500000
RouterOpts.pres_fac_mult: 1.300000
RouterOpts.max_router_iterations: 50
RouterOpts.astar_fac: 1.200000
RouterOpts.criticality_exp: 1.000000
RouterOpts.max_criticality: 0.990000

RoutingArch.directionality: UNI_DIRECTIONAL
RoutingArch.switch_block_type: WILTON
RoutingArch.Fs: 3
```

Listing 2: VPR 7.0 default configuration

A.7 SECOND ENERGY PHASE EXAMPLES

Figure 124 and Figure 125 show two additional examples of vectors before and after grid embedding in the second energy phase.



(a) Force-directed layout result **Phase 1**

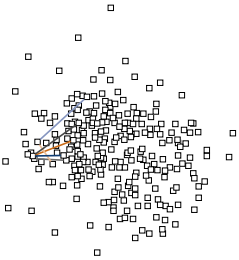
(b) Embedded on the grid **Phase 1**

(c) Force-directed layout result **Phase 2**

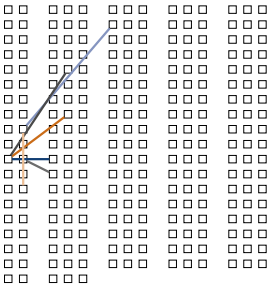
(d) Embedded on the grid **Phase 2**

Figure 124: Displacement in first and second energy phase **test 3** (code: `or1200`)

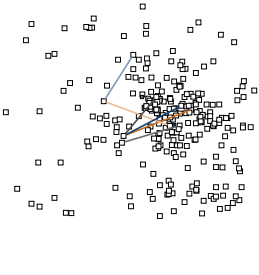
Figure 125 shows that the displacement after embedding can become relatively large when the energy layout without repulsive forces wastes much space.



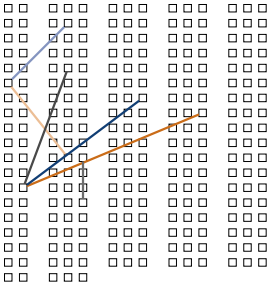
(a) Force-directed layout result **Phase 1**



(b) Embedded on the grid **Phase 1**

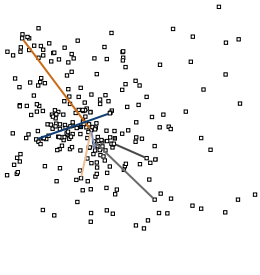


(c) Force-directed layout result **Phase 2**

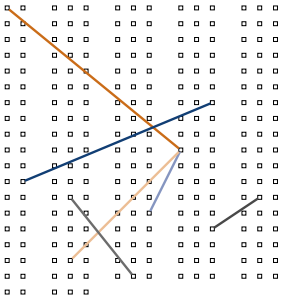


(d) Embedded on the grid **Phase 2**

Figure 125: Displacement in first and second energy phase **test 4** (code: `or1200`)



(a) Force-directed layout result **Phase 2**

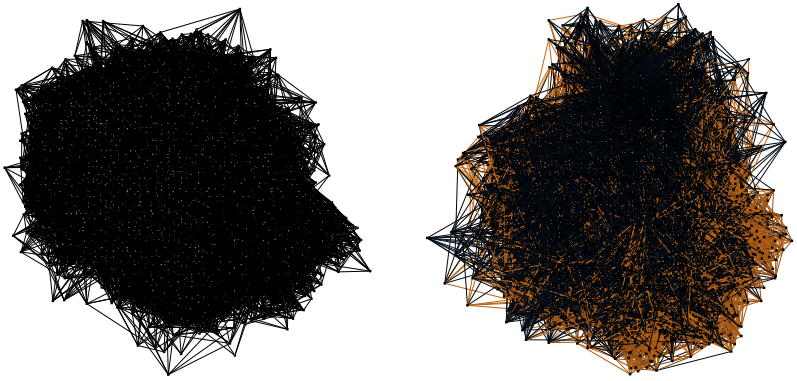


(b) Embedded on the grid **Phase 2**

Figure 126: Displacement in second energy phase **test 2** (code: `or1200`)

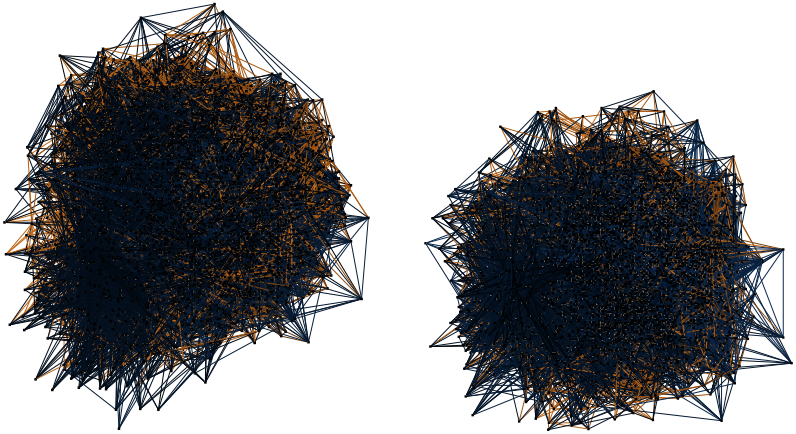
A.8 SLACK GRAPH MORPHING

Figure 127 depicts the logic blocks (*without I/O*) of the `stereovision2` code and their interconnections in *four* repetitions of the slack graph morphing. Connections with *low* slack are *orange* while those with *high slack* are *blue* (*average* ones are *black*). The slack (and consequently the *critical path delay*) tends to decrease in the successive iterations.



(a) Slack: $5.495 * 10^{-4}$ CPD: 32.611 ns

(b) Slack: $3.236 * 10^{-4}$ CPD: 23.277 ns



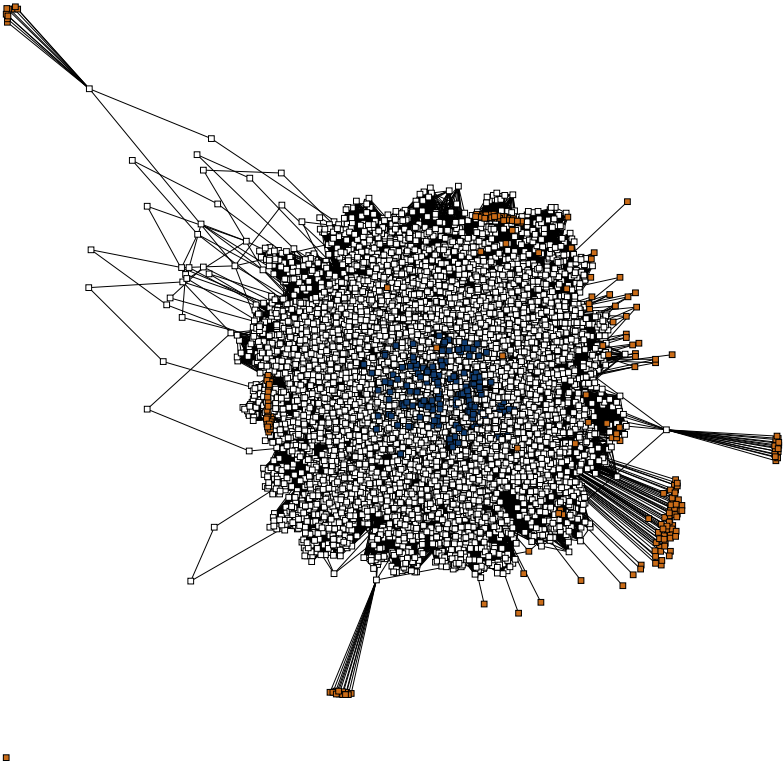
(c) Slack: $2.734 * 10^{-4}$ CPD: 20.886 ns

(d) Slack: $2.672 * 10^{-4}$ CPD: 20.617 ns

Figure 127: Slack graph morphing (code: `stereovision2`)

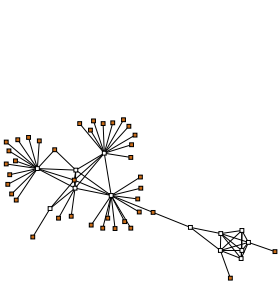
A.9 ENERGY LAYOUT GALLERY

Figures 128-131 show examples for generated energy graphs $\mathcal{G}_D^{\text{layout}}$.

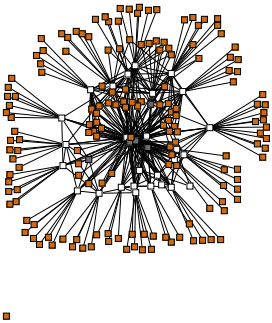


(a) LU32PEEng

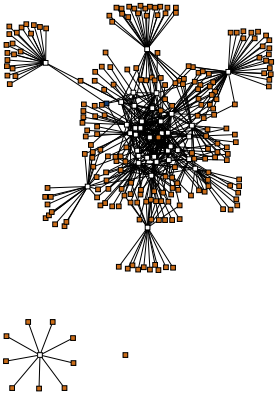
Figure 128: Energy layout gallery 1



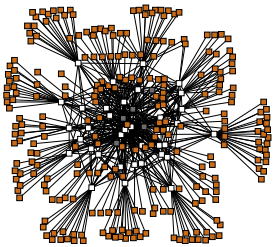
(a) stereovision3



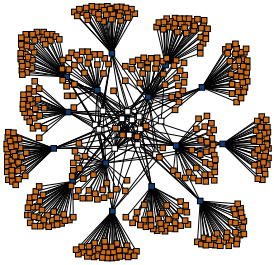
(b) diffeq2



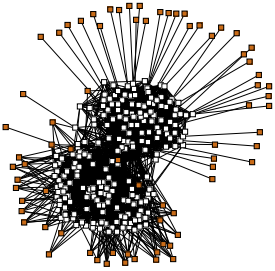
(c) ch_intrinsics



(d) diffeq1

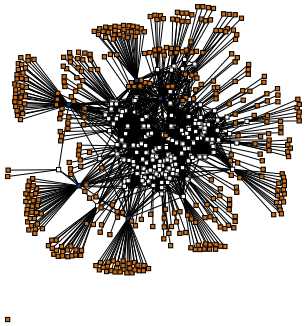


(e) mkPktMerge

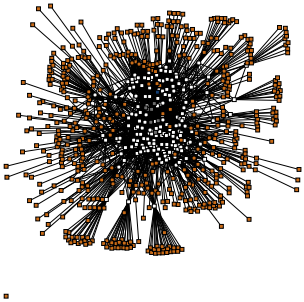


(f) sha

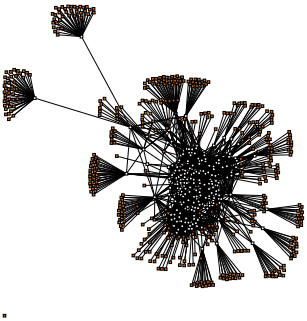
Figure 129: Energy layout gallery 2



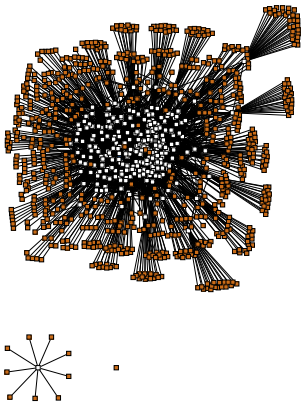
(a) `mkSMAdapter4B`



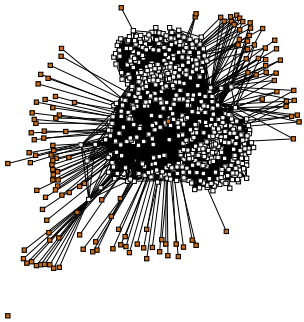
(b) `raygentop`



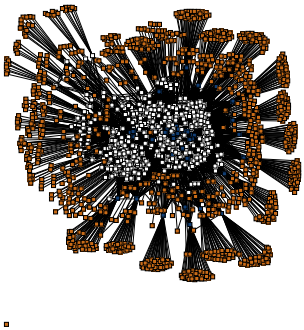
(c) `boundtop`



(d) `or1200`

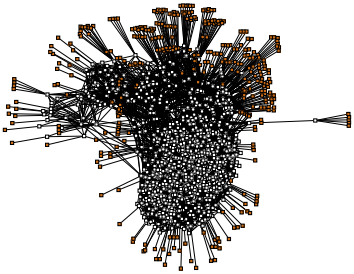


(e) `blob_merge`

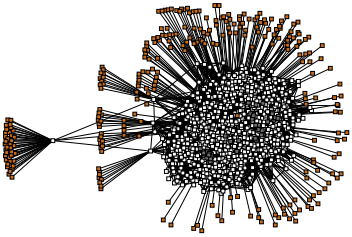


(f) `mkDelayWorker32B`

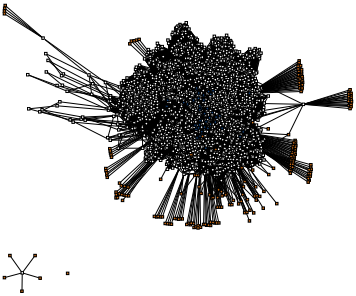
Figure 130: Energy layout gallery 3



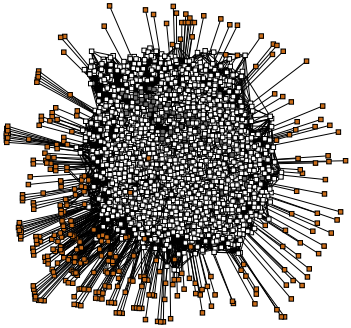
(a) `stereovision0`



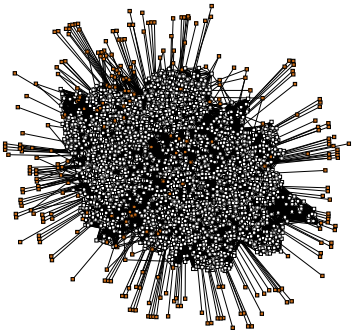
(b) `stereovision1`



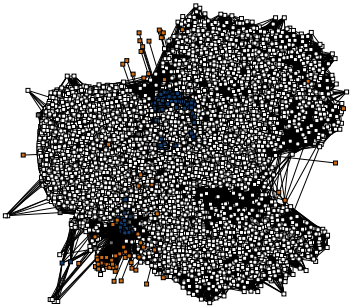
(c) `LU8PEEng`



(d) `stereovision2`



(e) `bgm`



(f) `mcml`

Figure 131: Energy layout gallery 4

REFERENCES

[1] IEEE graphic symbols for logic functions (includes IEEE std 91a-1991 supplement, and IEEE std 91-1984). *IEEE Std 91a-1991 IEEE Std 91-1984*, page 160, 1991. doi: 10.1109/IEEESTD.1991.81068.

[2] Warren P. Adams and Terri A. Johnson. Improved linear programming-based lower bounds for the quadratic assignment problem. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–77, 1994.

[3] Elias Ahmed and Jonathan Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, March 2004. ISSN 1063-8210. doi: 10.1109/TVLSI.2004.824300.

[4] Srinivas Aluru, John Gustafson, Gurpur M. Prabhu, and Fatih E. Sevilgen. Distribution-independent hierarchical algorithms for the N-body problem. *The Journal of Supercomputing*, 12(4):303–323. ISSN 1573-0484. doi: 10.1023/A:1008047806690. URL http://dx.doi.org/10.1023/A:1008047806690.

[5] Srinivas Aluru, Gurpur M. Prabhu, and John Gustafson. Truly distribution-independent algorithms for the N-body problem. In *Supercomputing '94., Proceedings*, pages 420–428, Nov 1994. doi: 10.1109/SUPERC.1994.344305.

[6] Miguel F. Anjos and Jean-Bernard Lasserre, editors. *Handbook on semidefinite, conic and polynomial optimization*, volume 166 of *International series in operations research & management science*. Springer, New York, 2012. ISBN 978-1-461-40768-3. URL http://opac.inria.fr/record=b1133560.

[7] Miguel F. Anjos and Frauke Liers. Global approaches for facility layout and VLSI floorplanning. In Miguel F. Anjos and Jean B. Lasserre, editors, *Handbook on Semidefinite, Conic and Polynomial Optimization*,

volume 166 of *International Series in Operations Research & Management Science*, pages 849–877. Springer US, 2012. ISBN 978-1-4614-0768-3. doi: 10.1007/978-1-4614-0769-0_29. URL http://dx.doi.org/10.1007/978-1-4614-0769-0_29.

[8] Miguel F. Anjos and Frauke Liers. Global approaches for facility layout and VLSI floorplanning. In Miguel F. Anjos and Jean B. Lasserre, editors, *Handbook on Semidefinite, Conic and Polynomial Optimization*, volume 166 of *International Series in Operations Research & Management Science*, pages 849–877. Springer US, 2012. ISBN 978-1-4614-0768-3. doi: 10.1007/978-1-4614-0769-0_29. URL http://dx.doi.org/10.1007/978-1-4614-0769-0_29.

[9] Kurt M. Anstreicher. Eigenvalue bounds versus semidefinite relaxations for the quadratic assignment problem. *SIAM Journal on Optimization*, 11(1):254–265, 2000. doi: 10.1137/S1052623499354904. URL http://dx.doi.org/10.1137/S1052623499354904.

[10] Kurt M. Anstreicher and Nathan W. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. *Mathematical Programming*, 89(3):341–357. ISSN 1436-4646. doi: 10.1007/PL00011402. URL http://dx.doi.org/10.1007/PL00011402.

[11] Andrew W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. and Stat. Comput*, 6(1):85–103, 1985.

[12] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007. ISBN 0691129932, 9780691129938.

[13] Pawan K. Aurora and Shashank K. Mehta. New facets of the QAP-Polytope. *Submitted to Operations Research Letters*, 2014. URL http://arxiv.org/abs/1409.0667v2.

[14] Mutlu Avci and Serhan Yamacli. An improved elmore delay model for VLSI interconnects. *Mathematical and Computer Modelling*, 51(7-8):908 – 914, 2010. ISSN 0895-7177. doi: http://dx.doi.org/10.1016/j.mcm.2009.08.024. URL http://www.sciencedirect.com/science/article/pii/S0895717709002866. 2008 International Workshop on Scientific Computing in Electronics Engineering (WSCEE 2008).

[15] Christoph Bartoschek, Stephan Held, Jens Maßberg, Dieter Rautenbach, and Jens Vygen. The repeater tree construction problem. *In-*

formation Processing Letters, 110(24):1079 – 1083, 2010. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/j.ipl.2010.08.016. URL http://www.sciencedirect.com/science/article/pii/S0020019010002747.

[16] Martin Beckman and Tjalling C. Koopmans. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

[17] Saifallah Benjaafar. Modeling and analysis of congestion in the design of facility layouts. *Manage. Sci.*, 48(5):679–704, May 2002. ISSN 0025-1909. doi: 10.1287/mnsc.48.5.679.7800. URL http://dx.doi.org/10.1287/mnsc.48.5.679.7800.

[18] Vaughn Betz and Jonathan Rose. Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size. In *Custom Integrated Circuits Conference, 1997., Proceedings of the IEEE 1997*, pages 551–554, May 1997. doi: 10.1109/CICC.1997.606687.

[19] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, FPL '97, pages 213–222, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63465-7. URL http://dl.acm.org/citation.cfm?id=647924.738755.

[20] Vaughn Betz and Jonathan Rose. FPGA routing architecture: Segmentation and buffering to optimize speed and density. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 59–68, New York, NY, USA, 1999. ACM. ISBN 1-58113-088-0. doi: 10.1145/296399.296428. URL http://doi.acm.org/10.1145/296399.296428.

[21] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792384601.

[22] Lilian Bossuet, Guy Gogniat, Jean-Philippe Diguet, and Jean-Luc Philippe. *System-on-Chip for Real-Time Applications*, chapter A Modeling Method for Reconfigurable Architectures, pages 170–179. Springer US, Boston, MA, 2003. ISBN 978-1-4615-0351-4. doi: 10.1007/978-1-4615-0351-4_16. URL http://dx.doi.org/10.1007/978-1-4615-0351-4_16.

[23] Robert Brayton and Alan Mishchenko. *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, chapter ABC: An Academic Industrial-Strength Verification Tool, pages 24–40. Springer Berlin Heidelberg, Berlin, Heidelberg,

2010. ISBN 978-3-642-14295-6. doi: 10.1007/978-3-642-14295-6_5. URL http://dx.doi.org/10.1007/978-3-642-14295-6_5.

[24] Nathan W. Brixius and Kurt M. Anstreicher. *17. The Steinberg Wiring Problem*, chapter 17, pages 293–307. doi: 10.1137/1.9780898718805.ch17. URL http://epubs.siam.org/doi/abs/10.1137/1.9780898718805.ch17.

[25] Stephen Brown, Jonathan Rose, and Zvonko G. Vranesic. A detailed router for field-programmable gate arrays. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 382–385, Nov 1990. doi: 10.1109/ICCAD.1990.129931.

[26] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. The Springer International Series in Engineering and Computer Science. Springer US, 1992. ISBN 9780792392484. URL https://books.google.de/books?id=8s4M-qYOWZIC.

[27] Sally A. Browning. *The tree machine, a highly concurrent computing environment*. PhD thesis, Dept. of Computer Science, CIT, 1980. URL http://resolver.caltech.edu/CaltechCSTR:3760-tr-80.

[28] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. QAPLIB – a quadratic assignment problem library. *J. of Global Optimization*, 10(4): 391–403, June 1997. ISSN 0925-5001. doi: 10.1023/A:1008293323270. URL http://dx.doi.org/10.1023/A:1008293323270.

[29] Rainer E. Burkard, Eranda Çela, Panos M. Pardalos, and Leonidas S. Pitsoulis. The quadratic assignment problem, 1998.

[30] Tony Chan, Jason Cong, and Kenton Sze. Multilevel generalized force-directed method for circuit placement. In *Proceedings of the 2005 International Symposium on Physical Design*, ISPD '05, pages 185–192, New York, NY, USA, 2005. ACM. ISBN 1-59593-021-3. doi: 10.1145/1055137.1055177. URL http://doi.acm.org/10.1145/1055137.1055177.

[31] Yao-Wen Chang, S. Thakur, Kai Zhu, and D. F. Wong. A new global routing algorithm for FPGAs. In *Computer-Aided Design, 1994., IEEE/ACM International Conference on*, pages 356–361, Nov 1994. doi: 10.1109/ICCAD.1994.629817.

[32] Yao-Wen Chang, Martin D. F. Wong, and Chak-Kuen Wong. Universal switch modules for FPGA design. *ACM Trans. Design Automation of Electronic Systems*, 1:80–101, 1996.

[33] Yao-Wen Chang, Zhe-Wei Jiang, and Tung-Chieh Chen. Essential Issues in Analytical Placement Algorithms. *IPSJ Transactions on System Lsi Design Methodology*, 2:145–166, 2009. doi: 10.2197/ipsjtsldm.2.145.

[34] Chia-I Chen, Bau-Cheng Lee, and Juinn-Dar Huang. Architectural exploration of 3D FPGAs towards a better balance between area and delay. In *2011 Design, Automation Test in Europe*, pages 1–4, March 2011. doi: 10.1109/DATE.2011.5763290.

[35] Tung-Chieh Chen, Zhe-Wei Jiang, Tien-Chang Hsu, Hsin-Chen Chen, and Yao-Wen Chang. NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1228–1240, July 2008. ISSN 0278-0070. doi: 10.1109/TCAD.2008.923063.

[36] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (OGDF), 2014. URL http://e-archive.informatik.uni-koeln.de/704/.

[37] Alexander Choong, Rami Beidas, and Jianwen Zhu. Parallelizing simulated annealing-based placement using GPGPU. In *2010 International Conference on Field Programmable Logic and Applications*, pages 31–34, Aug 2010. doi: 10.1109/FPL.2010.17.

[38] Nicos Christofides and M. Gerrard. A graph theoretic analysis of bounds for the quadratic assignment problem. In P. Hansen, editor, *Annals of Discrete Mathematics (11)Studies on Graphs and Discrete Programming*, volume 59 of *North-Holland Mathematics Studies*, pages 61 – 68. North-Holland, 1981. doi: http://dx.doi.org/10.1016/S0304-0208(08)73458-3. URL http://www.sciencedirect.com/science/article/pii/S0304020808734583.

[39] Jens Clausen, Stefan E. Karisch, Michael Perregaard, and Franz Rendl. On the applicability of lower bounds for solving rectilinear quadratic assignment problems in parallel. *Computational Optimization and Applications*, 10(2):127–147. ISSN 1573-2894. doi: 10.1023/A:1018308718386. URL http://dx.doi.org/10.1023/A:1018308718386.

[40] Peter Buitenkant (Consultant). Overcoming erase/write-endurance limitations in eeproms. Technical report, Compuserve, September 2000. URL http://m.eet.com/media/1149460/22945-47235.pdf.

[41] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, mar 1990. ISSN 0747-7171. doi: 10.1016/S0747-7171(08)80013-2. URL http://dx.doi.org/10.1016/S0747-7171(08)80013-2.

[42] Georges-Henri Cottet and Petros D. Koumoutsakos. Fast multipole methods for three-dimensional n-body problems. In *Vortex Methods*, pages 284–300. Cambridge University Press, 2000. ISBN 9780511526442. URL http://dx.doi.org/10.1017/CBO9780511526442.011. Cambridge Books Online.

[43] David R. Cox and David V. Hinkley. *Theoretical Statistics*. Springer US, Boston, MA, 1974. ISBN 978-0-412-12420-4. doi: 10.1007/978-1-4899-2887-0. URL http://dx.doi.org/10.1007/978-1-4899-2887-0.

[44] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, oct 1996. ISSN 0730-0301. doi: 10.1145/234535.234538. URL http://doi.acm.org/10.1145/234535.234538. *preliminary version published at the 'The Weizmann Institute' in 1989*.

[45] Gilberto de Miranda, Henrique P. L. Luna, Geraldo R. Mateus, and Ricardo P. M. Ferreira. A performance guarantee heuristic for electronic components placement problems including thermal effects. *Computers & Operations Research*, 32(11):2937 – 2957, 2005. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/j.cor.2004.04.014. URL http://www.sciencedirect.com/science/article/pii/S0305054804000875.

[46] André DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 69–78, New York, NY, USA, 1999. ACM. ISBN 1-58113-088-0. doi: 10.1145/296399.296431. URL http://doi.acm.org/10.1145/296399.296431.

[47] James Demmel, Ioana Dumitriu, and Olga Holtz. Fast linear algebra is stable. *Numerische Mathematik*, 108(1):59–91, 2007. ISSN 0945-3245. doi: 10.1007/s00211-007-0114-x. URL http://dx.doi.org/10.1007/s00211-007-0114-x.

[48] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0945-3245. doi: 10.1007/BF01386390. URL http://dx.doi.org/10.1007/BF01386390.

[49] Alfred E. Dunlop and Brian W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4(1):92–98, January 1985. ISSN 0278-0070. doi: 10.1109/TCAD.1985.1270101.

[50] Peter Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984.

[51] Peter Eades and Patrick Garvan. Drawing stressed planar graphs in three dimensions. In *In*, pages 212–223. Springer, 1995.

[52] Peter Eades and Sue Whitesides. Drawing graphs in two layers. *Theoretical Computer Science*, 131(2):361 – 374, 1994. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(94)90179-1. URL http://www.sciencedirect.com/science/article/pii/0304397594901791.

[53] C.S. Edwards. The derivation of a greedy approximator for the koopmans-beckmann quadratic assignment problem. *Proceedings of the 77-th Combinatorial Programming Conference (CPP77)*, pages 55–86, 1977.

[54] C.S. Edwards. A branch and bound algorithm for the Koopmans-Beckmann quadratic assignment problem. In V.J. Rayward-Smith, editor, *Combinatorial Optimization II*, volume 13 of *Mathematical Programming Studies*, pages 35–52. Springer Berlin Heidelberg, 1980. ISBN 978-3-642-00803-0. doi: 10.1007/BFb0120905. URL http://dx.doi.org/10.1007/BFb0120905.

[55] W. C. Elmore. The transient response of damped linear networks with particular regard to wideband amplifiers. *Journal of Applied Physics*, 19(1):55–63, 1948. doi: http://dx.doi.org/10.1063/1.1697872. URL http://scitation.aip.org/content/aip/journal/jap/19/1/10.1063/1.1697872.

[56] Marina A. Epelman. Introduction to semidefinite programming (SDP). University Lecture, 2007. URL http://users.math.msu.edu/users/markiwen/Teaching/MTH995/Papers/SDP_notes_Marina_Epelman_UM.pdf.

[57] Güneş Erdoğan and Barbaros Tansel. A note on a polynomial time solvable case of the quadratic assignment problem. *Discrete Optimization*, 3 (4):382 – 384, 2006. ISSN 1572-5286. doi: http://dx.doi.org/10.1016/j.disopt.2006.04.001. URL http://www.sciencedirect.com/science/article/pii/S1572528606000429.

[58] Güneş Erdoğan and Barbaros Tansel. A branch-and-cut algorithm for quadratic assignment problems based on linearizations. *Computers & Operations Research*, 34(4):1085 – 1106, 2007. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/j.cor.2005.05.027. URL http://www.sciencedirect.com/science/article/pii/S0305054805001760.

[59] Hongbing Fan, Yu-Liang Wu, and Catherine L. Zhou. Augmented disjoint switch boxes for FPGAs. In *Proceedings of the 4th International Symposium on Information and Communication Technologies*, WISICT '05, pages 129–134. Trinity College Dublin, 2005. ISBN 1-59593-169-4. URL http://dl.acm.org/citation.cfm?id=1071752.1071778.

[60] Tom Feist. Vivado Design Suite (wp416). Technical report, Xilinx, Inc., jun 2012.

[61] Gerd Finke, Rainer E. Burkard, and Franz Rendl. Quadratic assignment problems. *Annals of Discrete Mathematics*, 31:61–82, 1987.

[62] Ronald Aylmer Sir Fisher. *Statistical methods for research workers*. Edinburgh Oliver and Boyd, 7th ed., rev. and enl edition, 1938. ISBN 0050021702. URL http://openlibrary.org/books/OL181269M. 'Sources used for data and methods': p. 341-344; Bibliography: p. 345-352.

[63] Jon Frankle. Iterative and adaptive slack allocation for performance-driven layout and FPGA routing. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC '92, pages 536–542, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-89791-516-X. URL http://dl.acm.org/citation.cfm?id=113938.149626.

[64] Robert M. Freund. Introduction to Semidefinite Programming (SDP), in Dimitris Bertsimas course 6.251J Introduction to Mathematical Programming. University Lecture, Massachusetts Institute of Technology: MIT OpenCourseWare, License: Creative Commons BY-NC-SA, 2009. URL http://ocw.mit.edu.

[65] Alan M. Frieze and Joseph Yadegar. On the quadratic assignment problem. *Discrete Applied Mathematics*, 5(1):89 – 98, 1983. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/0166-218X(83)90018-5. URL http://www.sciencedirect.com/science/article/pii/0166218X83900185.

[66] Yaniv Frishman and Ayellet Tal. Multi-level graph layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, Nov 2007. ISSN 1077-2626. doi: 10.1109/TVCG.2007.70580.

[67] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, nov 1991. ISSN 0038-0644. doi: 10.1002/spe.4380211102. URL http://dx.doi.org/10.1002/spe.4380211102.

[68] Pawel Gajer and Stephen Kobourov. Grip: Graph drawing with intelligent placement. *Journal of Graph Algorithms and Applications*, 6:2002, 2002.

[69] Pawel Gajer, Michael T. Goodrich, and Stephen G. Kobourov. *Graph Drawing: 8th International Symposium, GD 2000 Colonial Williamsburg, VA, USA, September 20–23, 2000 Proceedings*, chapter A Multi-dimensional Approach to Force-Directed Layouts of Large Graphs, pages 211–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44541-8. doi: 10.1007/3-540-44541-2_20. URL http://dx.doi.org/10.1007/3-540-44541-2_20.

[70] Michel Gendreau and Jean-Yves Potvin. *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2nd edition, 2010. ISBN 1441916636, 9781441916631.

[71] P. C. Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *SIAM J. Appl. Math.*, 10:305–313, 1962.

[72] Fred Glover. Improved linear integer programming formulations of nonlinear integer problems. *Management Science*, 22(4):455–460, 1975. URL http://www.scopus.com/inward/record.url?eid=2-s2.0-0016619730&partnerID=40&md5=9f05acecca2b736bdc8808fe4f284d30. cited By 263.

[73] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. ISBN 079239965X.

[74] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C. Hart. *Graph Drawing: 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008. Revised Papers*, chapter Rapid Multipole Graph Drawing on the GPU, pages 90–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00219-9. doi: 10.1007/978-3-642-00219-9_10. URL http://dx.doi.org/10.1007/978-3-642-00219-9_10.

[75] Teofilo F. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics (Chapman & Hall/Crc Computer & Information Science Series)*. Chapman & Hall/CRC, 2007. ISBN 1584885505.

[76] Ananth Grama, Vivek Sarin, and Ahmed Sameh. Improving error bounds for multipole-based treecodes. In *High Performance Computing, 1998. HIPC '98. 5th International Conference On*, pages 73–80, Dec 1998. doi: 10.1109/HIPC.1998.737973.

[77] Vincent Granville, Mirko Křivánek, and Jean-Paul Rasson. Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):652–656, 1994. ISSN 0162-8828. doi: http://doi.ieeecomputersociety.org/10.1109/34.295910.

[78] David J. Greaves and Myoung-Jin Nam. Synthesis of glue logic, transactors, multiplexors and serialisors from protocol specifications. In *Specification Design Languages (FDL 2010), 2010 Forum on*, pages 1–7, Sept 2010. doi: 10.1049/ic.2010.0148.

[79] Leslie Frederick Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. PhD thesis, New Haven, CT, USA, 1987. AAI8727216.

[80] Leslie Frederick Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 135(2):280 – 292, 1997. ISSN 0021-9991. doi: http://dx.doi.org/10.1006/jcph.1997.5706. URL http://www.sciencedirect.com/science/article/pii/S0021999197957065.

[81] Stefan Hachul. *A potential field based multilevel algorithm for drawing large graphs*. PhD thesis, University of Cologne, 2005. URL http://kups.ub.uni-koeln.de/volltexte/2005/1409/index.html.

[82] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Proceedings of the 12th International Conference on Graph Drawing*, GD'04, pages 285–295, Berlin, Heidelberg, 2004. Springer-Verlag. ISBN 3-540-24528-6, 978-3-540-24528-5. doi: 10.1007/978-3-540-31843-9_29. URL http://dx.doi.org/10.1007/978-3-540-31843-9_29.

[83] Stefan Hachul and Michael Jünger. Large-graph layout with the fast multipole multilevel method. Technical report, 2005. URL http://e-archive.informatik.uni-koeln.de/509/.

[84] Stefan Hachul and Michael Jünger. Large-graph layout algorithms at work: An experimental study. *Journal of Graph Algorithms and Applications*, 11(21):345–369, 2007. URL http://e-archive.informatik.uni-koeln.de/510/.

[85] Scott W. Hadley, Franz Rendl, and Henry Wolkowicz. Bounds for the quadratic assignment problem using continuous optimization techniques, 1990.

[86] Scott W. Hadley, Franz Rendl, and Henry Wolkowicz. A new lower bound via projection for the quadratic assignment problem. *Mathematics of Operations Research*, 17:727–739, 1992.

[87] Malay Haldar, Anshuman Nayak, Alok N. Choudhary, and Prithviraj Banerjee. Parallel algorithms for FPGA placement. In *GLVLSI*, 2000.

[88] David Harel and Yehuda Koren. A fast multi-scale method for drawing large graphs (full version). In *Journal of Graph Algorithms and Applications*, pages 183–196. Springer-Verlag, 2000.

[89] David Harel and Yehuda Koren. *Graph Drawing: 10th International Symposium, GD 2002 Irvine, CA, USA, August 26–28, 2002 Revised Papers*, chapter Graph Drawing by High-Dimensional Embedding, pages 207–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-36151-0. doi: 10.1007/3-540-36151-0_20. URL http://dx.doi.org/10.1007/3-540-36151-0_20.

[90] John Harrison, Ted Kubaska, Shane Story, Microprocessor Software Labs, and Intel Corporation. The computation of transcendental functions on the ia-64 architecture. *Intel Technology Journal*, 4:234–251, 1999.

[91] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

[92] Lougee R. Heimer. The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47, 2003.

[93] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM. ISBN 0-89791-816-9. doi: 10.1145/224170.224228. URL http://doi.acm.org/10.1145/224170.224228.

[94] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, July 2000. ISSN 0018-9162. doi: 10.1109/2.869367. URL http://dx.doi.org/10.1109/2.869367.

[95] Michael Himsolt. GML: A portable Graph File Format. Technical report, Universität Passau, 94030 Passau, Germany, 1999. URL http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html.

[96] Roger W. Hockney and James W. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, Inc., Bristol, PA, USA, 1988. ISBN 0-85274-392-0.

[97] Bo Hu. Timing-driven placement for heterogeneous field programmable gate array. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 383–388, Nov 2006. doi: 10.1109/ICCAD.2006.320062.

[98] Bo Hu, Yue Zeng, and Malgorzata Marek-Sadowska. mFAR: Fixed-points-addition-based VLSI placement algorithm. In *Proceedings of the 2005 International Symposium on Physical Design*, ISPD '05, pages 239–241, New York, NY, USA, 2005. ACM. ISBN 1-59593-021-3. doi: 10.1145/1055137.1055189. URL http://doi.acm.org/10.1145/1055137.1055189.

[99] Tao Huang. *Continuous Optimization Methods for the Quadratic Assignment Problem*. BiblioBazaar, 2011. ISBN 9781243461346. URL https://books.google.de/books?id=UrSXpwAACAAJ.

[100] Mohamed Saifullah Hussin and Thomas Stützle. Hierarchical iterated local search for the quadratic assignment problem. In Maria J. Blesa, Christian Blum, Luca Di Gaspero, Andrea Roli, Michael Sampels, and Andrea Schaerf, editors, *Hybrid Metaheuristics*, volume 5818 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2009. ISBN 978-3-642-04917-0. URL http://dblp.uni-trier.de/db/conf/hm/hm2009.html#HussinS09.

[101] Peter Jamieson and Jonathan Rose. A Verilog RTL synthesis tool for heterogeneous FPGAs. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 305–310, Aug 2005. doi: 10.1109/FPL.2005.1515739.

[102] Stephen Jang, Billy Chan, Kevin Chung, and Alan Mishchenko. Wiremap: FPGA technology mapping for improved routability and enhanced LUT merging. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2): 14:1–14:24, June 2009. ISSN 1936-7406. doi: 10.1145/1534916.1534924. URL http://doi.acm.org/10.1145/1534916.1534924.

[103] Michael Jünger and Volker Kaibel. A basic study of the QAP-polytope. Technical report, Institut für Informatik, Universität zu Köln, Pohligstrasse 1, 50969 Köln, Germany, 1996.

[104] Michael Jünger and Volker Kaibel. On the SQAP-polytope. *SIAM J. on Optimization*, 11(2):444–463, feb 2000. ISSN 1052-6234. doi: 10.1137/S1052623496310576. URL http://dx.doi.org/10.1137/S1052623496310576.

[105] Michael Jünger and Volker Kaibel. The QAP-polytope and the star transformation. *Discrete Appl. Math.*, 111(3):283–306, aug 2001. ISSN 0166-218X. doi: 10.1016/S0166-218X(00)00272-9. URL http://dx.doi.org/10.1016/S0166-218X(00)00272-9.

[106] Michael Jünger and Petra Mutzel. *Exact and heuristic algorithms for 2-layer straightline crossing minimization*, pages 337–348. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. ISBN 978-3-540-49351-8. doi: 10.1007/BFb0021817. URL http://dx.doi.org/10.1007/BFb0021817.

[107] Stefan E. Karisch and Franz Rendl. Lower bounds for the quadratic assignment problem via triangle decompositions. *Mathematical Programming*, 71(2):137–151. ISSN 1436-4646. doi: 10.1007/BF01585995. URL http://dx.doi.org/10.1007/BF01585995.

[108] L. Kaufman and F. Broeckx. An algorithm for the quadratic assignment problem using bender's decomposition. *European Journal of Operational Research*, 2(3):207 – 211, 1978. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/0377-2217(78)90095-4. URL http://www.sciencedirect.com/science/article/pii/0377221778900954.

[109] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970. ISSN 1538-7305. doi: 10.1002/j.1538-7305.1970.tb01770.x. URL http://dx.doi.org/10.1002/j.1538-7305.1970.tb01770.x.

[110] Myung-Chul Kim, Jin Hu, Dong-Jin Lee, and Igor L. Markov. A simplr method for routability-driven placement. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 67–73, Piscataway, NJ, USA, 2011. IEEE Press. ISBN 978-1-4577-1398-9. URL http://dl.acm.org/citation.cfm?id=2132325.2132346.

[111] Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. ISSN 0036-8075. doi: 10.1126/science.220.4598.671. URL http://science.sciencemag.org/content/220/4598/671.

[112] Jürgen M. Kleinhans, Georg Sigl, Frank M. Johannes, and Kurt J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(3):356–365, Mar 1991. ISSN 0278-0070. doi: 10.1109/43.67789.

[113] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974.

[114] Yehuda Koren, Liran Carmel, and David Harel. Drawing huge graphs by algebraic multigrid optimization. *Multiscale Modeling & Simulation*, 1(4):645–673, 2003. doi: 10.1137/S154034590241370X. URL http://dx.doi.org/10.1137/S154034590241370X.

[115] M. Kováč. Solving quadratic assignment problem in parallel using local search with simulated annealing elements. In *Digital Technologies (DT), 2013 International Conference on*, pages 18–20, May 2013. doi: 10.1109/DT.2013.6566279.

[116] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM. ISBN 1-59593-292-5. doi: 10.1145/1117201.1117205. URL http://doi.acm.org/10.1145/1117201.1117205.

[117] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, Feb 2007. ISSN 0278-0070. doi: 10.1109/TCAD.2006.884574.

[118] Viktor M. Kureichik, Boris K. Lebedev, and Oleg B. Lebedev. A simulated evolution-based solution of the placement problem. *Journal of Computer and Systems Sciences International*, 46(4):578–589, 2007. ISSN 1555-6530. doi: 10.1134/S1064230707040089. URL http://dx.doi.org/10.1134/S1064230707040089.

[119] Young-Su Kwon, Payam Lajevardi, Anantha P. Chandrakasan, and Donald E. Troxel. A 3-D FPGA wire resource prediction model validated using a 3-D placement and routing tool. In *in Proc. of SLIP '05*, pages 65–72. ACM, 2005.

[120] Jimmy Lam and Jean-Marc Delosme. Performance of a new annealing schedule. In *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pages 306–311, June 1988. doi: 10.1109/DAC.1988.14775.

[121] Eugene L. Lawler. The quadratic assignment problem. *Management Science*, 9(4):586–599, 1963. doi: 10.1287/mnsc.9.4.586. URL http://dx.doi.org/10.1287/mnsc.9.4.586.

[122] Chin-Yang Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, Sept 1961. ISSN 0367-9950. doi: 10.1109/TEC.1961.5219222.

[123] Yuh-Sheng Lee and Allen C.-H. Wu. A performance and routability-driven router for FPGAs considering path delays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(2):179–185, Feb 1997. ISSN 0278-0070. doi: 10.1109/43.573832.

[124] Yong Li, Panos M. Pardalos, K. G. Ramakrishnan, and Mauricio G. C. Resende. Lower bounds for the quadratic assignment problem. *Annals of Operations Research*, 50(1):387–410. ISSN 1572-9338. doi: 10.1007/BF02085649. URL http://dx.doi.org/10.1007/BF02085649.

[125] Chih liang Eric Cheng. RISA: Accurate and efficient placement routability modeling. In *Computer-Aided Design, 1994., IEEE/ACM International Conference on*, pages 690–695, Nov 1994. doi: 10.1109/ICCAD.1994.629897.

[126] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.*, 21(2):498–516, apr 1973. ISSN 0030-364X. doi: 10.1287/opre.21.2.498. URL http://dx.doi.org/10.1287/opre.21.2.498.

[127] Tzu-Hen Lin, P. Banerjee, and Y. W. Chang. An efficient and effective analytical placer for FPGAs. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–6, May 2013.

[128] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter Hahn, and Tania Querido. An analytical survey for the quadratic assignment problem. *EUROPEAN JOURNAL OF OPERATIONAL RESEARCH*, pages 657–690, 2007.

[129] Adrian Ludwin and Vaughn Betz. Efficient and deterministic parallel placement for FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 16(3):22:1–22:23, jun 2011. ISSN 1084-4309. doi: 10.1145/1970353.1970355. URL http://doi.acm.org/10.1145/1970353.1970355.

[130] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose,

and Vaughn Betz. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2):6:1–6:30, jul 2014. ISSN 1936-7406. doi: 10.1145/2617593. URL http://doi.acm.org/10.1145/2617593.

[131] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. Timing-driven partitioning-based placement for island style FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):395–406, March 2005. ISSN 0278-0070. doi: 10.1109/TCAD.2004.842812.

[132] Mirko Maischberger. COIN-OR METSlib, a metaheuristics framework in modern C++. http://www.coin-or.org/metslib/docs/releases/0.5.2/metslib-tr.pdf, April 2011.

[133] Wai-Kei Mak and Hao Li. Placement for modern FPGAs. In *Conference, Emerging Information Technology 2005.*, pages 4 pp.–, Aug 2005. doi: 10.1109/EITC.2005.1544354.

[134] Stephen Lim (Product Marketing Manager). Expect a breakthrough advantage in next- generation FPGAs. Technical report, Altera Corporation, Jun 2015.

[135] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for FPGAs. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, FPGA '00, pages 203–213, New York, NY, USA, 2000. ACM. ISBN 1-58113-193-3. doi: 10.1145/329166.329208. URL http://doi.acm.org/10.1145/329166.329208.

[136] Alexander (Sandy) Marquardt, Vaughn Betz, and Jonathan Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 37–46, New York, NY, USA, 1999. ACM. ISBN 1-58113-088-0. doi: 10.1145/296399.296426. URL http://doi.acm.org/10.1145/296399.296426.

[137] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009. ISSN 0740-7475. doi: 10.1109/MDT.2009.83.

[138] Larry McMurchie and Carl Ebeling. Pathfinder: A negotiation-based performance-driven router for FPGAs. In *Field-Programmable Gate Arrays, 1995. FPGA '95. Proceedings of the Third International ACM Symposium on*, pages 111–117, 1995. doi: 10.1109/FPGA.1995.242049.

[139] Donald A. McQuarrie. *Statistical Mechanics*. University Science Books, 2000. ISBN 97818913891 53. URL https://books.google.de/books?id=itcpPnDnJM0C.

[140] Wim Meeus, Kristof van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012. ISSN 1572-8080. doi: 10.1007/s10617-012-9096-8. URL http://dx.doi.org/10.1007/s10617-012-9096-8.

[141] Michael D. Moffitt. MaizeRouter: Engineering an effective global router. In *2008 Asia and South Pacific Design Automation Conference*, pages 226–231, March 2008. doi: 10.1109/ASPDAC.2008.4483946.

[142] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in FPGA placement and routing. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, FPGA '01, pages 29–36, New York, NY, USA, 2001. ACM. ISBN 1-58113-341-3. doi: 10.1145/360276.360294. URL http://doi.acm.org/10.1145/360276.360294.

[143] Ravi Nair, C. Leonard Berman, Peter S. Hauge, and Ellen J. Yoffa. Generation of performance constraints for layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(8):860–874, Aug 1989. ISSN 0278-0070. doi: 10.1109/43.31546.

[144] William C. Naylor, Ross Donelly, and Lu Sha. Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer, oct 2001. URL http://www.google.com/patents/US6301693. US Patent 6,301,693.

[145] Christopher E. Nugent, Thomas E. Vollmann, and John Ruml. An experimental comparison of techniques for the assignment of facilities to locations. *Operations Research*, 16(1):150–173, 1968. doi: 10.1287/opre.16.1.150. URL http://dx.doi.org/10.1287/opre.16.1.150.

[146] Andreas Paffenholz. Faces of Birkhoff polytopes. *Electr. J. Comb.*, 22 (1):P1.67, 2015. URL http://www.combinatorics.org/ojs/index.php/eljc/article/view/v22i1p67.

[147] Igor Pak. Four questions on Birkhoff polytope. *Annals of Combinatorics*, 4(1):83–90, 2000. ISSN 0219-3094. doi: 10.1007/PL00001277. URL http://dx.doi.org/10.1007/PL0000127.

[148] Michael Palczewski. Plane parallel A* maze router and its application to FPGAs. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC '92, pages 691–697, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-89791-516-X. URL http://dl.acm.org/citation.cfm?id=113938.149679.

[149] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982. ISBN 0-13-152462-3.

[150] Panos M. Pardalos, Franz Rendl, and Henry Wolkowicz. The quadratic assignment problem: A survey and recent developments. In *In Proceedings of the DIMACS Workshop on Quadratic Assignment Problems, volume 16 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–42. American Mathematical Society, 1994.

[151] Kara K. W. Poon, Andy Yan, and Steven J. E. Wilton. A Flexible Power Model for FPGAs. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 312–321, London, UK, 2002. Springer-Verlag. ISBN 3540441085. URL http://portal.acm.org/citation.cfm?id=740248.

[152] Neil R. Quinn and Melvin A. Breuer. A force-directed component placement procedure for printed circuit boards. *IEEE Transactions on Circuits and Systems*, 26(6):377–388, Jun 1979. ISSN 0098-4094. doi: 10.1109/TCS.1979.1084652.

[153] Sanguthevar Rajasekaran. On the convergence time of simulated annealing. Technical Report MS-CIS-90-89/GRASP LAB 242, University of Pennsylvania. Philadelphia (PA US), 1990. URL http://opac.inria.fr/record=b1048187.

[154] K. G. Ramakrishnan, Mauricio G. C. Resende, and Panos M. Pardalos. A branch and bound algorithm for the quadratic assignment problem using a lower bound based on linear programming. In *In C. Floudas and P.M. Pardalos, editors, State of the Art in Global Optimization: Computational Methods and Applications*, pages 57–73. Kluwer Academic Publishers, 1995.

[155] Srilata Raman, C. L. Liu, and Larry G. Jones. Timing-constrained FPGA placement: A force-directed formulation and its performance evaluation. *VLSI Design*, 4(4):345–355, 1996. doi: 10.1155/1996/53238. URL http://dx.doi.org/10.1155/1996/53238.

[156] Deepak Rautela and Rajendra Katti. Design and implementation of FPGA router for efficient utilization of heterogeneous routing resources. In *IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05)*, pages 232–237, May 2005. doi: 10.1109/ISVLSI.2005.26.

[157] Franz Rendl and Renata Sotirov. Bounds for the quadratic assignment problem using the bundle method. *Mathematical Programming*, 109(2): 505–524, 2006. ISSN 1436-4646. doi: 10.1007/s10107-006-0038-8. URL http://dx.doi.org/10.1007/s10107-006-0038-8.

[158] Franz Rendl and Henry Wolkowicz. Applications of parametric programming and eigenvalue maximization to the quadratic assignment problem. *Mathematical Programming*, 53(1):63–78. ISSN 1436-4646. doi: 10.1007/BF01585694. URL http://dx.doi.org/10.1007/BF01585694.

[159] Jonathan Rose and Stephen Brown. Flexibility of interconnection structures for field-programmable gate arrays. *Solid-State Circuits, IEEE Journal of*, 26(3):277–282, 1991.

[160] Jonathan Rose and Dwight Hill. Architectural and physical design challenges for one-million gate FPGAs and beyond. In *Proceedings of the 1997 ACM Fifth International Symposium on Field-programmable Gate Arrays*, FPGA '97, pages 129–132, New York, NY, USA, 1997. ACM. ISBN 0-89791-801-0. doi: 10.1145/258305.258324. URL http://doi.acm.org/10.1145/258305.258324.

[161] Jorge Rubinstein, Paul Penfield, and Mark A. Horowitz. Signal delay in rc tree networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(3):202–211, July 1983. ISSN 0278-0070. doi: 10.1109/TCAD.1983.1270037.

[162] Sartaj Sahni and Teofilo F. Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, 1976. doi: 10.1145/321958.321975. URL http://doi.acm.org/10.1145/321958.321975.

[163] Gamal Abd El-Nasser A. Said, Abeer M. Mahmoud, and El-Sayed M. El-Horbaty. A Comparative Study of Meta-heuristic Algorithms for Solving Quadratic Assignment Problem. *International Journal of Advanced Computer Science and Applications(IJACSA)*, 5(1), 2014. URL http://ijacsa.thesai.org/.

[164] Alexander Schrijver. *Combinatorial optimization : Polyhedra and Efficiency*. Algorithms and combinatorics. Springer-Verlag, Berlin, Hei-

delberg, New York, N.Y., et al., 2003. ISBN 3-540-44389-4. URL http://opac.inria.fr/record=b1124844.

[165] Navaratnasothie Selvakkumaran, Abhishek Ranjan, Salil Raje, and George Karypis. Multi-resource aware partitioning algorithms for FPGAs with heterogeneous resources. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 741–746, July 2004.

[166] S. I. Sergeev. Improved lower bounds for the quadratic assignment problem. *Automation and Remote Control*, 65(11):1733–1746. ISSN 1608-3032. doi: 10.1023/B:AURC.0000047888.76717.7a. URL http://dx.doi.org/10.1023/B:AURC.0000047888.76717.7a.

[167] Akshay Sharma, Scott Hauck, and Carl Ebeling. Architecture-adaptive routability-driven placement for FPGAs. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 427–432, Aug 2005. doi: 10.1109/FPL.2005.1515759.

[168] Georg Sigl, Konrad Doll, and Frank M. Johannes. Analytical placement: A linear or a quadratic objective function? In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 427–432, New York, NY, USA, 1991. ACM. ISBN 0-89791-395-7. doi: 10.1145/127601.127707. URL http://doi.acm.org/10.1145/127601.127707.

[169] Jadranka Skorin-Kapov. Tabu search applied to the quadratic assignment problem. *ORSA Journal on Computing*, 2(1):33–45, 1990. doi: 10.1287/ijoc.2.1.33. URL http://dx.doi.org/10.1287/ijoc.2.1.33.

[170] Richard Smith. Working draft, standard for programming language C++, rev. N4140. Technical report, Google Inc, 2014. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf.

[171] Renata Sotirov. SDP relaxations for some combinatorial optimization problems. In Miguel F. Anjos and Jean B. Lasserre, editors, *Handbook on Semidefinite, Conic and Polynomial Optimization*, volume 166 of *International Series in Operations Research & Management Science*, pages 795–819. Springer US, 2012. ISBN 978-1-4614-0768-3. doi: 10.1007/978-1-4614-0769-0_27. URL http://dx.doi.org/10.1007/978-1-4614-0769-0_27.

[172] Peter Spindler, Ulf Schlichtmann, and Frank M. Johannes. Kraftwerk2 - a fast force-directed quadratic placement approach using an accurate net model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(8):1398–1411, Aug 2008. ISSN 0278-0070. doi: 10.1109/TCAD.2008.925783.

[173] Leon Steinberg. The backboard wiring problem: A placement algorithm. *SIAM Review*, 3(1):37–50, 1961. URL http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&id=SIREAD000003000001000037000001&idtype=cvips&gifs=yes.

[174] Thomas Stützle and Marco Dorigo. Local search and metaheuristics for the quadratic assignment problem. Technical Report AIDA-01-01, FG Intellektik, FB Informatik, TU Darmstadt, Germany, 2001.

[175] William Swartz and Carl Sechen. New algorithms for the placement and routing of macro cells. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 336–339, Nov 1990. doi: 10.1109/ICCAD.1990.129918.

[176] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, release 70930. http://www.eecs.berkeley.edu/~alanmi/abc/. URL http://www.eecs.berkeley.edu/~{}alanmi/abc.

[177] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, jun 1987. ISSN 0097-5397. doi: 10.1137/0216030. URL http://dx.doi.org/10.1137/0216030.

[178] Danesh Tavana, Wilson Yee, Steve Young, and Bradly Fawcett. Logic block and routing considerations for a new SRAM-based FPGA architecture. In *Custom Integrated Circuits Conference, 1995., Proceedings of the IEEE 1995*, pages 511–514, May 1995. doi: 10.1109/CICC.1995.518235.

[179] Russell G. Tessier. *Fast Place and Route Approaches for FPGAs*. PhD thesis, Cambridge, MA, USA, 1999. AAI0800664.

[180] William T. Tutte. Convex representations of graphs. *Proceedings of the London Mathematical Society*, 10:304–320, 1960. URL http://www.ams.org/mathscinet-getitem?mr=0114774.

[181] William T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 13:743–767, 1963. URL http://www.ams.org/mathscinet-getitem?mr=28:1610.

[182] Vaishali Udar and Sanjeev Sharma. Analysis of place and route algorithm for field programmable gate array (FPGA). In *Information Communication Technologies (ICT), 2013 IEEE Conference on*, pages 116–119, April 2013. doi: 10.1109/CICT.2013.6558073.

[183] Jeffrey D Ullman. *Computational Aspects of VLSI*. W. H. Freeman & Co., New York, NY, USA, 1984. ISBN 071678145X.

[184] Satya Prakash Upadhyay. Wirelength-driven analytical placement for FPGA. Master's thesis, 2015. URL http://hdl.handle.net/11299/174729.

[185] Natarajan Viswanathan, Min Pan, and Chris Chu. FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. In *2007 Asia and South Pacific Design Automation Conference*, pages 135–140, Jan 2007. doi: 10.1109/ASPDAC.2007.357975.

[186] Kristofer Vorwerk, Andrew Kennings, and Anthony Vannelli. Engineering details of a stable force-directed placer. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 573–580. IEEE Computer Society, 2004.

[187] Chris Walshaw. *Graph Drawing: 8th International Symposium, GD 2000 Colonial Williamsburg, VA, USA, September 20–23, 2000 Proceedings*, chapter A Multilevel Algorithm for Force-Directed Graph Drawing, pages 171–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44541-8. doi: 10.1007/3-540-44541-2_17. URL http://dx.doi.org/10.1007/3-540-44541-2_17.

[188] Saeyang Yang. Logic synthesis and optimization benchmarks user guide version 3.0, 1991.

[189] Habib Youssef and Eugene Shragowitz. Timing constraints for correct performance. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 24–27, Nov 1990. doi: 10.1109/ICCAD.1990.129830.

[190] Chi-Wai Yu, Wayne Luk, Steven J. E. Wilton, and Philip H. W. Leong. Routing optimization for hybrid FPGAs. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 419–422, Dec 2009. doi: 10.1109/FPT.2009.5377695.

[191] Wenwei Zha and Peter Athanas. An FPGA router for alternative reconfiguration flows. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 163–171, May 2013. doi: 10.1109/IPDPSW.2013.221.

[192] Qing Zhao, Stefan E. Karisch, Franz Rendl, and Henry Wolkowicz. Semidefinite programming relaxations for the quadratic assignment

problem. *Journal of Combinatorial Optimization*, 2(1):71–109. ISSN 1573-2886. doi: 10.1023/A:1009795911987. URL http://dx.doi.org/10.1023/A:1009795911987.

[193] Dian Zhou, Franco P. Preparata, and Sung-Mo Kang. Interconnection delay in very high-speed VLSI. In *Computer Design: VLSI in Computers and Processors, 1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on*, pages 52–55, Oct 1988. doi: 10.1109/ICCD.1988.25658.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`http://code.google.com/p/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

Final Version as of March 21, 2017 (`classicthesis` version 1.0).

Erklärung

Ich versichere, dass ich die von mir vorgelegte Dissertation selbständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie - abgesehen von unten angegeben Teilpublikationen - noch nicht veröffentlicht worden ist sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen der Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Dr. Michael Jünger betreut worden.

Teilpublikationen liegen nicht vor.

Dustin Feld *Köln, den 29.08.2016*