Sequential Matching Problem

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

vorgelegt von
Sureshan Karichery
aus Perumbala

2004

Berichterstatter: Prof. Dr. Rainer Schrader
Prof. Dr. Michael Jünger

Tag der mündlichen prüfung: 24. May 2004

Kurzzusammenfassung

Wir stellen das Sequential Matching Problem (SMP) vor. Das SMP beschreibt die Suche einer Folge maximaler Matchings zu einer Folge von gegebenen bipartiten Graphen, die die Anzahl der gemeinsamen Kanten aufeinanderfolgender Matchings maximiert. Eine Anwendungsbeispiel ist das Problem, Arbeitern über einen gewissen Zeitraum Jobs zuzuweisen und dabei die Jobwechsel zu minimieren. Wir analysieren verschiedene algorithmische Techniken für dieses $\mathcal{NP}$-vollständige Problem. Dabei betrachten wir eine Formulierung als gemischt ganzzahliges Problem(MIP) mit einer sehr großer Zahl von Variablen. Diese Problem lösen wir mit einem Branch&Price-Verfahren, d.h einer Kombination eines Branch&Bound-Verfahrens mit einem Column-Generation-Ansatz. Hierbei verwenden wir zur Spaltenerzeugung implizites Pricing von Nichtbasisvariablen. Wir erläutern, die aus den verwendeten Branching-Regeln entstehenden Subprobleme, und gehen auf die Implementierung und die numerischen Ergebnisse ein. Schließlich analysieren wir den Spezialfall des SMP mit zwei bipartiten Graphen. Auch dieses Problem ist $\mathcal{NP}$-vollständig. Wir stellen eine Heuristik vor, die Anzahl der gemeinsamen Kanten in den Matchings schrittweise vergrößert.

Abstract

We present sequential matching problem (SMP) as the problem of finding maximum matchings in a sequence of bipartite graphs, with a strategy of making a maximum number of common edges in two consecutive matchings. One application of SMP is the problem of assigning workers to jobs in different time shifts with a goal of minimizing total number of unnecessary switches between jobs. We analyze various algorithmic techniques for this $\mathcal{NP}$-complete problem. We also analyze the Mixed Integer Programming (MIP) problem formulation with a huge number of variables and their solution by the branch and price method, a column generation scheme with branch and bound, of implicit pricing of nonbasic variables to generate new columns. We then discuss special branching rules, pricing problems, implementation issues, and computational results. Finally we analyze a simpler version of SMP with only two bipartite graphs which is still $\mathcal{NP}$-complete, and an algorithm to augment the common edges in the maximum matchings.

Acknowledgements

I would like to thank Prof. Dr. Rainer Schrader and Dr. Christoph Moll, my supervisors, for their many suggestions and constant support during this research.

It is a great pleasure to acknowledge the financial assistance I have received from Siemens AG. I am particularly pleased to thank Dr. Johannes Nierwetberg and Dr. Michael Hofmeister for their generous help.

With regards to my colleagues at the Corporate Technology, Software Engineering 6 (CT SE 6), it was a great pleasure and an utmost privilege to work in a group of so many talented individuals who shared their knowledge so freely. I had the pleasure of meeting the group at the Zentrum für Angewandte Informatik, Köln (ZAIK). They are wonderful people and the discussions with them were fruitful. Thanks to everyone.

I just would like to thank particularly, Dr. Peter Stadelmeyer, Dr. Mark Ziegelmann, Dr. Tamás Lukovszki: my office mates during the period of this research, and Mr. Krishna K Bhuwalka: for proof-reading this document. I would like to extend my gratitude to all my friends for their support and encouragement. Finally, a special thanks to friends of *sports_munich*, because they asked me to do so.

Sureshan Karichery
24 May 2004

Table of Contents

Introduction

Modern mail services use *mail processing centres* as distribution centres to consolidate and redistribute mail. These centres collect regional mail (usually delivered by trucks), sort it and send it on to other centres (usually operated by flights for international mail service). Conversely, mail coming in from other centres by flights is sorted and distributed to regional or local centres. The international mail operations are carried out at the centre called the *office of exchange*. The main function of an office of exchange is the processing and distribution of international mail within the country. With the goal of fiscal self-sufficiency, a program has been embarked upon to modernize, and in some cases radically alter the way it manages and processes the mail. Automation has been targeted for its processing systems to become ever more cost effective and efficient by replacing manual processing with mechanized solutions. This means reading, sorting, and then sequencing each mail in the order in which it will be delivered by the carrier with only a minimum of manual labor and time.

The operations of the office of exchange involve an interconnected web of components which require detailed analysis. The main aspects of a model are production planning, forecasting and simulation of the production progress within the office of exchange. Therefore the most interesting facts are throughput, processing time, staffing, buffer capacities and the splitting and merging of mail streams. An important part of planning consists of scheduling of rotation of the work force.

This work is inspired by the personnel planning of an international mail centre where the processing and distribution of international mail is undertaken. We focus on one particular aspect of the personnel scheduling problem, defined as the *sequential matching problem*. This thesis will mainly concentrate at the most general approach to the sequential matching problem. We analyze the mathematical properties and algorithmic methods for this problem. Firstly, in Chapter 1 we give a formal definition and basics of bipartite graphs, matching problem including some basic results and a brief discussion about computational complexity. The

second chapter presents the aim and motivation of the study in addition to the problem. In Chapter 3 we present the $\mathcal{NP}$-completeness proof of the problem, and a solution approach by randomized algorithm. In Chapter 4 we deals with the branch and price approach for the mixed integer programming formulation of the problem. The data structure and implementation issues are discussed in Chapter 5. Finally we conclude with a simpler version of the problem, the *2-graph sequential matching problem* which is still $\mathcal{NP}$-complete and an augmenting cycle heuristic discussed. The three appendices deal with file input formats, general graph data structures, and implementing matching algorithms.

Chapter 1

The basics

1.1 Introduction

We first introduce some notation and terminology about graphs, bipartite graphs, and matchings. Next, we show some classic results about the bipartite graphs and give a classic algorithm for solving bipartite matching problem. Even though we concentrate on *sequential matching problems* in bipartite graphs, this chapter summarizes other types of graph matching too. A brief introduction about computational complexity is also presented in the following sections. The complexity of the different graph matching problems are also analyzed. The references for this chapter are [1], [2] and [3].

1.2 Graphs, bipartite graphs, matchings and network flows

1.2.1 Definitions

A graph is a mathematical abstraction that is useful for solving many kinds of problems. A *graph* $G = (V, E)$ in its basic form is composed of *vertices* and *edges*. V is the set of vertices (also called nodes or points) and $E = V \times V$ is the set of edges (also known as arcs or lines) of graph G. The *order* (or *size*) of G is defined as the number of its vertices and it is represented as $\mid V \mid$. The number of edges represented as $\mid E \mid$. If two vertices in G, say $u, v \in V$, are connected by an edge $e \in E$, it is denoted as $e = (u, v)$ and the two vertices are said to be *adjacent* or *neighbors*. For a vertex v, the *degree* of v, $deg(v)$, is equal to the number of neighbors of v. Edges are said to be *undirected* when they have no direction, and a

graph G containing only such types of graphs is called *undirected*. When all edges have directions and therefore (u, v) and (v, u) can be distinguished, the graph is said to be *directed*. A graph is considered as *weighed* if there exist a real valued function $W : E \to \mathbb{R}$. A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A *path* between any two vertices $v_1, v_n \in V$ is a non-empty sequence of n different vertices $\{v_1, v_2, \ldots, v_n\}$ where $(v_{i-1}, v_i) \in E$, for $i = 1, 2, \ldots, n$. The *shortest path* between v_1, v_n is a path between v_1 and v_2 with least number of edges. If G is weighed graph then the *shortest path* is analogues to the path with minimum total weight of edges. A *cycle* is a path (v_1, v_1). A graph G is said to be *acyclic* when there are no cycles. A graph is *connected* if there is a path from any vertex to any other vertex. A *disconnected graph* consists of several *connected components* which are maximal connected subgraphs. Two vertices are in the same component if and only if there exist a path between them.

A *matching* in a graph $G = (V, E)$ is a subset M of the edges E such that no two edges in M share a common end vertex. Alternatively the edges M are such that, for each vertex $v \in V$ at most one edge in M is incident to v. We say that a vertex is *matched* if an edge in M is incident to v. A *maximum cardinality matching* is matching with a maximum number of edges. *i.e.*, A *maximum matching* is a matching M such that for any other matching M', $\mid M \mid \geq \mid M' \mid$. A *perfect matching* is one where all vertices are matched. The cardinality of a *perfect matching* is $\mid V \mid /2$. An edge of a graph is called *allowed* if it occurs in at least one maximum matching.

A *bipartite graph* $G(L_1, R_1, E)$ is a graph whose vertices can be partitioned into two non empty sets L_1 and R_1, and all edges go between L_1 and R_1. An alternative way of thinking about it is coloring (such that no two adjacent colors are the same) the vertices in L_1 with one color and those in R_1 with another color. Then adjacent vertices would then have different colors. A non-empty graph is bipartite if and only if its chromatic number (the minimum number of colors that can be used) is 2. A *maximal weight matchings* is matching in weighed graph G with maximum weight. A *node cover* is a set of nodes N' of G such that every edge of G has at least one node in N'. An *edge cover* is a subset of edges such that all vertices are covered by these edges. A matching in a bipartite graph $G(L_1, R_1, E)$ assigns vertices of L_1 to vertices of R_1.

A weighted, directed graph $N = (V, E)$ with two specially marked nodes, the *source*, s, and the *sink*, t, and a *capacity function*, c, that maps node pairs to real numbers, $c : V \times V \longrightarrow \mathbb{R}^+$, such that $c(u, v) > 0$) if $e = (u, v) \in E$ and $c(u, v) = 0$

if there is no edge between u and v. A *flow* on N is a real-valued function f on vertex pairs that satisfies the following properties.

1. *Skew symmetry*: $f(u,v) = -f(v,u)$
2. *Capacity constraint*: $f(u,v) \leq c(u,v)$
3. *Flow conservation*: For every vertex u except s and t, $\sum_v f(u,v) = 0$

Maximum flow problem for a capacitated network is to find the flow configuration that maximizes total amount of flow from a source to a target. The *value of the flow* $\mid f \mid$, defined as $\sum_v f(s,v)$.

A detailed description of graphs, bipartite graphs, and network flows is available in [1], [2].

1.2.2 Basic results and algorithms

This section is a review of the literature on bipartite graph matching. The problem of finding matchings and node covers are in opposite sense of each other. A matching covers the nodes of G with edges such that each node is covered by at most one edge. A node cover covers the edges of G by nodes such that each edge is covered by at least one node. The maximum cardinality of a matching is at most the minimum cardinality of a node cover, *(*i.e.), the maximum number of edges in a matching of G is always less than or equal to the minimum number of nodes in a node cover of G. In a bipartite graph the maximum cardinality of a matching is same as the minimum cardinality of a node cover, so Theorem 1.2.1 follows.

The cardinality of the smallest (minimum) node cover is called the *node cover number*, denoted by $\mu(G)$. The cardinality of the largest (maximum) matchings called the *matching number* and denoted by $\upsilon(G)$. The cardinality of the smallest (minimum) edge cover number and denoted by $\rho(G)$.

Theorem 1.2.1 (König). *If G is bipartite then $\mu(G) = \upsilon(G)$.*

Finding a node cover or finding a matching in a bipartite graph are considered as relatively simple problems. The computation of matchings in bipartite graphs are easier than finding matchings in general graphs because of the simpler structure of bipartite graphs.

An efficient algorithm for finding maximal matchings is based on constructing *augmenting paths* in graphs. Let M be a matching in a bipartite graph G. A path $P = (v_0, v_1, \ldots, v_n)$ is called *M-augmenting* if n is odd and $v_i \in M$ for each $i = 1, 2, \ldots, n$; but $v_0, v_t \notin M$.

Theorem 1.2.2 (Berge). *A matching M in a bipartite graph G is maximum if and only if G does not contain any M-augmenting path.*

Given a matching M in a graph G, by the definition of an augmenting path P, it consists of edges where every odd-numbered edge (including the first and last edge) is not in M, while every even-numbered edge is in M. Also, the first and last vertices must not be already in M. By deleting the even-numbered edges of P from M and replacing them with the odd-numbered edges of P, we enlarge the size of the matching by one edge. As per Theorem 1.2.2 we can construct maximum-cardinality matchings by searching for augmenting paths and stopping when none exist. The problem is now simplified to check the existence of an M-augmenting path with respect to a given matching M, and to find one if exists.

In case of a bipartite graph $G = (L, R, E)$, with n vertices and m edges to find a M-augmenting path, for a given matching M we do as follows [4]. In the algorithm the symbol '\' denote the set-minus.

Algorithm 1 Finding augmenting path

Orient each $e = (u, w)$ such that $(u \in L, v \in R)$ to obtain a directed graph D as follows.
if $e \in M$ **then**
  orient e as (w, u)
end if
if $e \notin M$ **then**
  orient e as (u, w)
end if
Let $L' := L \setminus M$ and $R'_1 := R \setminus M$
Determine if there exists a directed path from L'_1 to R'_1 (directed path $\iff$ M-augmenting path in G)

This method[1] of finding maximal matching has a running time complexity of $\mathcal{O}(nm/2)$. We can improve the running complexity of bipartite matching problem by reducing it as a special case of the maximum flow problem. By this we can solve the bipartite graph matching problem efficiently by use of any algorithm that solves the max-flow problem.

[1]The implementation details of this algorithm in Appendix C

Given any bipartite graph $G = (L_1, R_1, E)$, we construct a directed network $N_G = (U, A)$ where $U = V \cup \{s, t\}$ (here s and t source and target nodes) and A is set of edges consisting of three categories.

1. The edges (s, v) for all $v \in L_1$
2. The edges (u, t) for all $u \in R_1$
3. The edges $e = (v, u)$ for all $e \in E$

The edge (v, u) has the direction from v to u. Any integral maximum flow in N_G from s to t gives a maximum matching in G. The running time complexity of this method is $\mathcal{O}(\sqrt{n}m)$. The best algorithm for maximum bipartite matching, due to Hopcroft and Karp [5] and has a worst case running time complexity of $\mathcal{O}(\sqrt{n}m)$. The method is by repeatedly finding the shortest augmenting paths instead of using network flow is especially better for sparse graph. For dense graphs, the best algorithm is by Alt, Blum, Mehlhorn, and Paul [6] having a worstcase bound of $\mathcal{O}(n^{1.5}\sqrt{m/\log n})$.

The actual comparison of the algorithm when solving bipartite matching problems in practice depends on the structure of the graph and number of input classes. Some experimental results for the different bipartite matching algorithms for both sequential and parallel cases are compared on the papers [7], [8].

1.2.3 Integer programming modelling and matching polytope

For a given bipartite graph $G = (L_1, R_1, E)$ such that $V = L_1 \cup R_1$, we define a binary variable x_e, for each $e \in E$. Let $\delta(v)$ denote the set of edges incident from the node v. The problem of finding a maximum matching in G can be formulated as in Fig. 1.1. The validity of the formulation can be easily shown such that a solution to 1.1 gives maximal matching to G.

For a given set of vectors $S = \{r_i : r_i \in \mathbb{R}^n, i = 1, 2, \ldots, N\}$ the *convex hull (polytope)*, $conv(S)$ of S is defined as $\{\sum_{i=1}^{N} \lambda_i r_i : \lambda_i \in \mathbb{R}^+, \sum_{i=1}^{N} \lambda_i = 1\}$.

Let $G = (V, E)$ is any graph with a matching M. We define a vector

$$\textbf{Max} \sum_{e \in E} x_e \tag{1.2.1}$$

subject to

$$\sum_{e \in \delta(v)} x_e \leq 1, x_e \in \{0,1\}, \forall v \in V \tag{1.2.2}$$

Figure 1.1: Integer programming formulation of bipartite graph matching problem

$X_M = (X_M^e : \text{for all } e \in E)$, such that

$$X_M^e = \begin{cases} 1, & e \in M; \\ 0, & \text{otherwise.} \end{cases} \tag{1.2.3}$$

Let $\mathcal{M}$ be the *convex hull* of all vectors corresponding to matchings. (i.e.), $\mathcal{M} = conv\{X_M : X_M \text{ is a matching}\}$ and the resulting relaxation of the integral constraints 1.2.2,

$$\mathcal{P} = \{x_e : x_e \in \mathbb{R}^+, \forall e \in E, \sum_{e \in \delta(v)} x_e \leq 1, \forall v \in V\}$$

Theorem 1.2.3. *If G is bipartite, then $\mathcal{P} = \mathcal{M}$.*

A *unimodular matrix* is a real square matrix with determinant 1. A matrix A is *totally unimodular* if every square submatrix has determinant 0, 1, or -1.

Theorem 1.2.4. *Let A be totally unimodular and b an integer vector. Then vertices of polytope $P = \{x : Ax \leq b\}$ are integer vectors.*

Theorem 1.2.5. *If G is bipartite, then the constraint matrix is totally unimodular.*

Theorems 1.2.4 and 1.2.5 implies that a linear programming solution to 1.1 is an integer solution. More details can be found in [9].

1.2.4 Applications

Graphs have been proved as an effective way of representing objects, general knowledge, or information. Matching problems arise in several areas of automatic data processing, including analysis of images, artificial intelligence, and the solution of scheduling problems. In case of bipartite matching, most of them arise from the area of manpower scheduling or from personnel assignment problems. Other applications are from pattern matching or similar one to one assignments problems. A typical application of bipartite graph matching problem is as follows.

Suppose we have a set of workers and a set of machines with the information as to which worker can handle which machines. The task is to assign workers to machines in such a way that as many machines as possible are operated by a worker that can handle it. The set of workers and set of machines can be modelled as two kinds of nodes in the bipartite graph. The edges of our bipartite graph are defined by the abilities. A maximum cardinality matching gives a solution such that one worker can be assigned to at most one machine and we can assign at most one worker to one machine.

1.3 Computational complexity

The purpose of this section is to give an insight into how difficult the problem may be to solve. The search for fundamental distinctions in the tractability of problems constitute the area known as *computational complexity theory*. In other words complexity theory is part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are time (how many steps does it take to solve a problem) and space (how much memory does it take to solve a problem). The *runtime* or *space requirements* of an algorithm are expressed as a function of the problem size. The problem size measures the size, in some sense, of the input to the algorithm. Most of complexity theory deals with decision problems, because it is often considered that an arbitrary problem can always be reduced to a *decision problem*. The reference for this section is [3].

1.3.1 Definitions and results

An *instance* of a problem is specified by assigning numerical values, called data to the problem parameters. For example consider a mixed integer programming problem which is written generically as

$$\textbf{Max } \{cx + hy : Ax + Gy \leq b, x \in \mathbb{Z}_+^p, y \in R_+^n\}$$

The dimension of these matrices are as follows: c is $1 \times p$, h is $1 \times n$, A is $m \times p$, G is $m \times n$ and b is $m \times 1$, where m is any positive integer, p and n are any nonnegative integers with $p + n \leq 1$. In the case of mixed integer programming the data that specifies an instance are integers m, n and p as well as the integral matrices c, h, b, A and G with appropriate dimension. A problem consist of an infinite number of instances. The class of *polynomially solvable problems*, $\mathcal{P}$ contains all sets in which membership may be decided by an algorithm whose running time is bounded by a polynomial. The main theme of computational complexity is the inherent difference between problem known to be in $\mathcal{P}$ and others for which no polynomial time algorithm is known. A *feasibility problem* X is a pair (D, F) with $F \subseteq D$, where the elements of D is a finite binary strings. D is called *set of instances*, and F is called *set of feasible instances*. Given an instance $d \in D$, we want to determine whether $d \in F$. Given $d \in D$ the answer is *yes* or *no*. Let $X_1 = (D_1, F_1)$, $X_2 = (D_2, F_2)$ be two feasibility problems and there exist a function $g : D_1 \to D_2$ such that for every $d \in D_1$ we have $g(d) \in F_2$ if and only if $d \in F_1$. If the function g is computable in time that is polynomial in the length of encoding of d, then X_1 is said to be polynomially transformable to X_2.

Theorem 1.3.1. *If X_1 is polynomially transformable to X_2 and $X_2 \in \mathcal{P}$ then $X_1 \in \mathcal{P}$.*

There is a technique called polynomial reduction, that appears to be more general than polynomial transformation for establishing that one problem can be solved in polynomial time given that another can. We say that X_1 is polynomial reducible to X_2 if there is an algorithm for X_1 that uses an algorithm for X_2 as a subroutine and runs in polynomial time under the assumption that each call of the subroutine takes unit time. Note that transformation is a special case of reduction in which subroutine is used only once.

Theorem 1.3.2. *If X_1 is polynomially reducible to X_2 and $X_2 \in \mathcal{P}$ then $X_1 \in \mathcal{P}$*

Given a feasibility problem $X = (D, F)$, for each instance $d \in F$ we define *certificate of feasibility* Q_d as the information that can be used to check feasibility in polynomial time.

A *nondeterministic algorithm* for a feasibility problem consists of two stages. The input to the algorithm is $d \in D$. The first stage is a *guessing* stage. Here we guess a binary string Q which is then passed on to the second stage. The second stage is called checking stage, is an algorithm that works with the pair (d, Q) and may provide the output that $d \in F$. The two properties required are:

1. $d \in F$, there is a certificate Q_d such that when a pair (d, Q_d) is given to the checking stage, the algorithm gives the answer that $d \in F$.
2. $d \notin F$ and there is no output for any Q_d. Hence whenever there is a output, $d \in F$.

We measure the work done by a nondeterministic algorithm only in the checking stage and only when the checking stage is given a $d \in D$ and a certificate of feasibility. We say that, a *nondeterministic algorithm is polynomial* if for each $d \in F$ the running time in checking stage is a polynomial function of the length of the encoding of d for some Q_d for which it replies that $d \in F$. This mean that when $d \in F$, there is a short(polynomial) proof of feasibility.

The class of feasibility problem such that for each instance with $d \in F$ the answer $d \in F$ is obtained in polynomial time by some nondeterministic algorithm is called $\mathcal{NP}$.

Most important to note that is $\mathcal{NP}$ contain hardest problems too. By this we mean that there is a subset of $\mathcal{NP}$ called *$\mathcal{NP}$-complete* denoted by $\mathcal{NPC}$, such that if there exist $X \in \mathcal{NPC} \cup \mathcal{P}$, then every problem in $\mathcal{NP}$ is in $\mathcal{P}$ *i.e.*, $\mathcal{P} = \mathcal{NPC}$.

Theorem 1.3.3. *If X is $\mathcal{NP}$-complete then $\mathcal{P} = \mathcal{NP}$, if and only if $X \in \mathcal{P}$.*

Theorem 1.3.4. *If X_1 is $\mathcal{NP}$-complete and X_1 is polynomially reducible to a $\mathcal{NP}$ complete X_2 then X_2 is $\mathcal{NP}$-complete.*

So a decision problem X_1 is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and if every other problem in $\mathcal{NP}$ is reducible to it by polynomial-time algorithm which transforms instances of X_1 into instances of X_2, such that the two instances have the same truth values. As a consequence, if we had a polynomial time algorithm for X_2, we could solve all $\mathcal{NP}$ problems in polynomial time. So to prove an $\mathcal{NP}$-completeness of a problem we usually deduced from known problem which is already $\mathcal{NP}$- complete.

A *boolean expression (boolean formula)* is composed of *boolean variables*, or their negations (NOT, symbolically $\neg$), logical conjunction (AND, symbolically $\wedge$), logical disjunction (OR, symbolically $\vee$) and parentheses for grouping. A *literal* in a boolean expression is an occurrence of a variable or its negation. The boolean *satisfiability problem (SAT)* is a decision problem Φ of boolean expression to decide if there is some assignment to the variables in Φ such that it is *true* or *not*. *i.e.* The question is, for given the expression, is there some assignment of *true* and *false* values to the variables that will make the entire expression true. The *satisfiability problem*, was the first problem in $\mathcal{NP}$ shown to be $\mathcal{NP}$-complete.

Theorem 1.3.5. *The satisfiability problem is $\mathcal{NP}$-complete.*

A boolean expression is considered to be in *conjunctive normal form (CNF)* if and only if it is a single conjunction of disjunctions. Each of these disjunctions are called a *clause*.

For n variables, a collection m of disjunctive clauses of at most k literals, where a literal is a variable or a negated variable, where k is a constant, is called a *k-SAT problem*. When $k = 3$, we called it as *3-SAT* problem. *i.e.* 3-SAT is conjunctive normal form, in which every clause has exactly three distinct literals. An instance ϕ of 3-SAT with n clauses $C_1, C_2, \ldots, C_n$, with p boolean variables $\alpha_1, \alpha_1, \alpha_3, \ldots, \alpha_p$ is as follows,

$$\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_n \tag{1.3.1}$$

such that each clause being

$$C_i = (\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3}) \tag{1.3.2}$$

where α_{ij}'s being either boolean variables or negations.

Theorem 1.3.6. *3- SAT is $\mathcal{NP}$-complete.*

Every SAT problem can be transformed to 3-SAT problem. Since k-SAT (the general case) reduces to 3-SAT, and 3-SAT is known to be NP-complete, it can be used to prove that other problems are also NP-complete.

Although defined theoretically, many of these classes have practical implications. The class $\mathcal{P}$ is a very good approximation to the class of problems which can be solved quickly in practice, and if a problem is in $\mathcal{P}$ then we can prove a polynomial worst case time bound, and conversely if the polynomial time bounds, we can prove are usually small enough that the corresponding algorithms really are practical. It is believed (but so far no proof is available) that $\mathcal{NP}$-complete problems do not have polynomial-time algorithms and therefore are intractable. (i.e.), It is generally believed that $\mathcal{NPC} \neq \mathcal{NP}$. If a problem is proved $\mathcal{NP}$-complete, it is an evidence that the problems can't be solved quickly. This strongly suggests that no polynomial time algorithm exist for problems in $\mathcal{NPC}$. Some of the problems are assumed have higher complexity than some of the $\mathcal{NP}$-complete problems.

1.3.2 Solving NP-complete problems

At present, all known algorithms for NP-complete problems require time which is exponential in the problem size. It is unknown whether there are any faster algorithms. Given a NP-complete problem, what should we do? May be some algorithm performance is acceptable for small input sizes, or we can use the time limit, such that terminate the algorithm after a time limit. Use approximate algorithms for optimization problems to find a good solution, but not necessary the best (optimum) solution. Therefore, in order to solve an NP-complete problem for any non-trivial problem size, one of the following approaches is used:

- ***Approximation algorithm:*** An algorithm which quickly finds a suboptimum solution which is within a certain (known) range of the optimum one. Not all $\mathcal{NP}$-complete problems have good approximation algorithms, and for some problems finding a good approximation algorithm is enough to solve the problem itself.
- ***Probabilistic algorithm:*** An algorithm which provably yields good average runtime behavior for a given distribution of the problem instancesideally, one that assigns low probability to hard inputs.
- ***Special cases:*** An algorithm which is provably fast if the problem instances belong to a certain special case.
- ***Heuristic algorithm:*** An algorithm which works reasonably well on many cases, but for which there is no proof that it is always fast, and give the optimum solution.

Chapter 2

Problem description

Shift work is a fact of the modern society. Many critical services such as power, water, medical, police, and transportation are needed around the clock. Other than such essential services, shift work is needed for enterprises where a continuous flow of input and the production of certain goods are also needed around the clock. As a result, many workplaces operate twenty four hours a day. We discuss a constrained personnel assignment problem of dealing with the shift work schedule inside an international mail processing centre. In first part of the chapter we give a brief introduction of an international mail processing centre and the optimization problems arising from the its logistic design. Later in the chapter we discuss a mathematical formulation for the shiftwork assignment problem and the nature of this more general approach. Before discussing the details of the original problem we briefly summarizethe design and functions of a mail distributing centre. Finally we introduce the concept of *sequential matching problem* as a typical application of the personnel assignment problem in different time shifts.

2.1 International mail distribution centre

The motivation of this work was the personnel planning problem which arises from the optimization problems in the distribution logistics of an international mail distribution centre. Mail centres are the locations where mail collected from specific postcode areas is concentrated for processing and onward despatch to the next node in a mail network. The scope of the original project is to design the operation of foreign mail processing to achieve lower operating costs and ensure

the quality of service. Before discussing the details of original problem we briefly summarize the design and functionalities of a mail distributing centre, and the dependent optimization problems.

2.1.1 Functioning of international mail

In this section we briefly explain how an international mail delivery system functions. The various kinds of international letters, packages or parcels are collected from different post offices or other authorized collection points around the country. This mail will be picked up by carriers on regular schedules on each work day and transported to the *international mail processing centre* or *office of exchange (OE)*. The mail collected is *processed* (by sorting it according to the size, destination, and priority) inside the office of exchange. The processed mail is despatched from the office of exchange to foreign destination ports by air, road or sea. In the case of import operations the office of exchange plays a similar role after receiving international mail from different foreign sources. From the office of exchange the imported mail is sorted and will be transported to one of the local mail centres and to its final destination. Fig. 2.1 shows an illustration.

2.1.2 Office of Exchange

As described in the above section, the office of exchange is the point in the network where all international mail processing is done, and plays an important role in mail distribution system by assisting and facilitating the operation. We mainly discuss the *export function* of the office of exchange. Export mail shall arrive at the office of exchange in a number of distinct input streams. After processing, export mail shall leave the office of exchange in a number of distinct output streams. Basically the office of exchange consists of four work areas arrival, special services, sorting and despatch.

- **Arrival Section**: Where the export mail arrives from mail centres, direct from customers via the post office network etc.
- **Special Services**: Customs processing or other clerical services.
- **Sorting Area**: In this section the items are sorted according to despatch requirements. The sorting is done manually and using machines.
- **Despatch Section**: This section handles the sorted mail to different destinations. From the despatch section mail shall be carried to any international gateway such as an airport, port, railway station or by vehicle.

Figure 2.1: An illustration: The functioning of the export mail service

An office of exchange also consists of many material handling systems, mail processing equipment, containers, storage and staging area and personnel with particular qualifications to perform manual services.

2.1.3 Optimization problems inside the office of exchange

The size and the complex functionalities of the office of exchange lead to an abundance of optimization problems. The office of exchange can be seen as a high-volume factory to be a connected network of workstations at which assigned workers process "work" that flows at certain rates through the workstation. The work may even change in the factory at any workstation according to any time-of-day profile. Workers in general are cross-trained, may work part time or full time shifts, may start work only at a designated shift starting times and may change job assignment in mid shift. One objective is to schedule the workers (and correspondingly, the work flow) in a manner that minimizes labor costs subject to variety of service level, contractual and physical constraints and predictions of workloads or service level-related measures.

The implementation of the decisions to manage and control the office of exchange, by providing the following facilities such as:

- Planning
- Scheduling
- Monitoring

Optimization problems inside the office of exchange can be categorized mainly as a *forecasting model* and as a *personnel planning problem*. Other than these, a *simulation model* allows a planning scenario to be performed to cover changes to inputs and outputs such as mail specifications, and arrival and departure schedules for the vehicles.

The forecasting model

The arrival pattern of the mail over the course of the day is highly predictable. The *forecasting model* is the basic tool, employed by both planning and monitoring, for

extrapolations and predictions of workloads or service level-related measures. Also, it provides details of vehicle arrival profiles, container and stream volumes associated with actual streams arriving at the office of exchange. The model represents the major processes in, and more relevant for decision-making.

Personnel planning

Personnel planning mainly deals with the assignment of the employees to jobs in such a way so as to use the manpower satisfying the various constraints. Like other types of resource planning for materials, machines and vehicles, human resources *planning* also involves a unique set of requirements. The personnel planning process faces problems and issues that are extremely time consuming and cost intensive. To utilize the capacity and efficiency of the office of exchange to the full extent, a shift planning method is used to distribute the human resources quickly and efficiently. Shift work usually means regularly scheduled work outside of the normal daytime working hours. We can schedule and create working hours for the employees in a flexible manner to cover the requirements. Shift planning provides the perfect view for every conceivable planning scenario, by creating time data for any number of employees at the same time, as well as for one or more days, weeks or even months. Shift times, shift location, selection and number of required employees are assigned so that personnel capacity is utilized to its maximum effect. One important part of the personnel planning is the assignment of workers to jobs in different shifts, when details of personnel requirements with their qualifications profile and designated daily shift schedules and requirement assignments are available.

We start with following assumptions,

- All working areas (arrival, sorting, and despatch) are considered as a single area.
- Personnel are cross trained, able to do different jobs.
- Each item of work is represented as a *work place* (we use the word *job* with the same meaning).
- The working day is divided into different shifts.
- Breaks are allowed inside a shift.

Working hours for each classified job shall be designated, and clearly stated at the time of planning. Each employee shall be assigned to one or more work shifts, and

some work areas may require employees in a periodic basis. Such a requirement shall also be indicated at the time of the start of a shift or in the event an employee being assigned to a new job within the work area. We concentrate on a general problem arising from scheduling of employees to various work shifts. This generalized problem formulation is presented to address objectives covering cross-training of workers, ensuring adequate levels of assignments and maximizing the efficiency of the assignment.

2.2 The problem description

We have specified a set of jobs with their start-end timings, a set of workers with certain job qualifications, and their working time schedules. The typical task is to assign workers to jobs each time to fulfill the requirements. The first constraint is from the fact that the assignment is possible only if a qualified worker is available at the time. Since the availability of workers and jobs changes from time to time a reassignment is needed on occurrence of any of the events such as a change in worker availability or job. It is possible that changes in jobs may happen for a worker in this reassignment even if the worker is available before the reassignment. A second constraint arises from the global minimization of the unnecessary switchings of workers between workplaces. The whole problem can be described as solving assignment problems in many discrete time intervals by fulfilling all the requirements and globally minimizing the number of changes between jobs for workers.

2.2.1 Mathematical modelling

In this section we describe a mathematical model for the personal scheduling problem with the described requirements. Let $\mathcal{W} = \{W_1, W_2, \ldots, W_m\}$ denotes the set of all workers, $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$, denotes the set of jobs. We say a work day starts from time $\mathbf{0}$ to $\mathbf{T}$.

Let $\mathbf{a}_i : [\mathbf{0}, \mathbf{T}] \to \{0, 1\}$ is the availability function for the worker W_i.

i.e., for every $W_i \in W$, we define,

$$\mathbf{a}_i(\mathbf{t}) = \begin{cases} 1 & \text{if worker } W_i \text{ is available at time } \mathbf{t}; \\ 0 & \text{otherwise.} \end{cases} \tag{2.2.1}$$

For each worker $W_i \in \mathcal{W}$, $\mathbf{a}_i$ provides a binary vector representing the availability profile. Similarly we define $\mathbf{b}_j : [\mathbf{0}, \mathbf{T}] \rightarrow \{0, 1\}$ for the job J_j, if it has to be done at t.

i.e., for every $J_j \in \mathcal{J}$, we define,

$$\mathbf{b}_j(\mathbf{t}) = \begin{cases} 1 & \text{if the job } J_j \text{ is has to be done at time } \mathbf{t}; \\ 0 & \text{otherwise.} \end{cases} \tag{2.2.2}$$

We define ability function $\mathbf{c} : \mathcal{W} \times \mathcal{J} \rightarrow \{0, 1\}$, such that,

$$\mathbf{c}(W_i, J_j) = \begin{cases} 1 & \text{if the worker } W_i \text{ is able to do the job } J_j \text{ ;} \\ 0 & \text{otherwise.} \end{cases} \tag{2.2.3}$$

The problem of assignment of a worker $W_i \in \mathcal{W}$ at any $\mathbf{t} \in [\mathbf{0}, \mathbf{T}]$ can be identified as mapping m_i from set of workers. we define a function, $\mathbf{m}_i(\mathbf{t}) : [\mathbf{0}, \mathbf{T}] \rightarrow \{0\} \cup \mathcal{J}$ such that,

$$\mathbf{m}_i(\mathbf{t}) = \begin{cases} J_j & \text{if the worker } W_i \in \mathcal{W} \text{ is assigned to the job } J_j \in \mathcal{J} \text{ at } \mathbf{t} \text{ ;} \\ 0 & \text{otherwise.} \end{cases} \tag{2.2.4}$$

A worker W_i can be assigned to a job J_j at time t only if he is available at t and able to do the job J_j,

i.e.

$$\mathbf{m}_i(\mathbf{t}) = J_j \quad \textit{if} \quad \mathbf{a}_i(\mathbf{t}) = 1,\ \mathbf{b}_j(\mathbf{t}) = 1,\ \text{and } \mathbf{c}(W_i, J_j) = 1 \tag{2.2.5}$$

Assuming that there is a sufficient number of workers to fulfill the jobs and no worker is idle at time t, then $\mathbf{m}_i$ can take the values depending on the following conditions,

$$\forall \quad W_i \in \mathcal{W}, \qquad \mathbf{m}_i(\mathbf{t}) = 0 \qquad \textit{if} \qquad \mathbf{a}_i(\mathbf{t}) = 0 \tag{2.2.6}$$

$$\exists \quad W_i \in \mathcal{W}, \qquad \mathbf{m}_i(\mathbf{t}) = J_j \quad \textit{if and only if} \quad \mathbf{b}_j(\mathbf{t}) = 1 \tag{2.2.7}$$

We define the number of switches for a worker $W_i \in \mathcal{W}$ as the number of jumps in the assignment function $\mathbf{m}_i$. (i.e.), The total number of switches can be evaluated

using the function $d : \mathcal{W} \to \mathbb{Z}^+$ such that,

$$d(W_i) = \max_{\epsilon>0} \left[\sum_{\mathbf{t},\mathbf{t}+\epsilon\in[\mathbf{0},\mathbf{T}]} \mathit{diff}\,(\mathbf{m}_i(\mathbf{t}+\epsilon), \mathbf{m}_i(\mathbf{t})) \right] \tag{2.2.8}$$

where for any a and b,

$$\mathit{diff}\,(a,b) = \begin{cases} 1 & a \neq b, \\ 0 & a = b. \end{cases} \tag{2.2.9}$$

The problem of assigning workers to jobs in a work day $[\mathbf{0}, \mathbf{T}]$ in an efficient way can be described as identifying the function $\mathbf{m}_i$ for each $W_i \in \mathcal{W}$, by minimizing $\sum_{W_i\in\mathcal{W}} d(W_i)$.

We defined the whole process in the continuous time interval $[\mathbf{0}, \mathbf{T}]$, while it is sufficient and more reasonable to consider the discrete time points in $[\mathbf{0}, \mathbf{T}]$ when there is a change in the values for $a_i(\mathbf{t})$ or $b_i(\mathbf{t})$. This leads to the definition of an *event* as a change in availability of a worker, or a job. For a work day, which starts from time $\mathbf{0}$ to $\mathbf{T}$, we define binary function $\mathbf{e} \to \{0, 1\}$, such that

$$\mathbf{e}(\mathbf{t}) = \begin{cases} 1 & \text{an event happening at } \mathbf{t}; \\ 0 & \text{otherwise.} \end{cases} \tag{2.2.10}$$

It is reasonable to consider from a practical point of view that the set of $\mathbf{t} \in [\mathbf{0}, \mathbf{T}]$ such that $\mathbf{e}(\mathbf{t}) = 1$ is *finite*. So we can find a time discretization $\mathbb{T}$ set of all points where $\mathbf{e} = 1$. The time discretization $\mathbb{T} = \{\mathbf{0}, \mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_\tau = \mathbf{T}\}$ such that, for each $\mathbf{t} \in \mathbb{T}$ some events are happening. Now we define an *idle time interval, t* as $[\mathbf{t}-1, \mathbf{t})$ such that $\mathbf{a}_i(t')$ and $\mathbf{b}_j(t')$ are constants for every $t \in [\mathbf{t}-1, \mathbf{t})$, $W_i \in \mathcal{W}$, $J_j \in \mathcal{J}$. Let $\mathcal{T}$ denote a set of τ such intervals,

$$\mathcal{T} = \{t | t \text{ is an idle time interval} [\mathbf{t}-1, \mathbf{t}) \text{ such that } \mathbf{t} \in \mathbb{T} - \{0\}\}$$

The possible assignments of workers to workplace at a particular time interval t can be represented as a bipartite graph G_t, for every $t \in \mathcal{T}$. Let $G_t = (L_t, R_t, E_t)$ be the bipartite graph representation, where $L_t \in \mathcal{W}$ is the set of workers available at time interval t and $R_t \in W$ is the set of jobs to do at time interval t. The edge set E_t represents the abilities of workers at t to do the jobs. The node and edges of the bipartite graph G_t can be represented using the functions $\mathbf{a}_i$, $\mathbf{b}_j$, and $\mathbf{c}$ as follows.

$$\begin{aligned} L_t &= \{W_i \in \mathcal{W} : \mathbf{a}_i(\mathbf{t}) = 1, \text{ for } \mathbf{t} \text{ in time interval } t\}, \\ R_t &= \{J_j \in \mathcal{J} : \mathbf{b}_j(\mathbf{t}) = 1, \text{ for } \mathbf{t} \text{ in time interval } t\}, \\ E_t &= \{e_{ij} = (W_i, J_i) : W_i \in L_t, J_i \in \mathcal{J}, \text{ and } c(W_i, J_i) = 1\}. \end{aligned}$$

The identification of the function $\mathbf{m}_i$ in the time interval $t = [\mathbf{t} - 1, \mathbf{t}]$, the assignment of workers to jobs can be formulated as a classical bipartite matching problem in G_t. For t we need to solve a *matching problem* corresponding to the bipartite graph G_t. *(*i.e.), Finding *maximum cardinality matching*[1] The problem of minimizing total number of switches between the assignments, the evaluation of $\sum_{W_i \in \mathcal{W}} d(W_i)$, is equivalent to the problem of global maximization of common edges between matchings M_t and M_{t+1}.

Let

$$d_t = \text{Number of different edges between } M_t \text{ and } M_{t+1}$$

i.e.

$$d_t = \mid (M_t \cup M_{t+1}) - (M_t \cap M_{t+1}) \mid$$

for $t = 1, 2, \ldots, \tau - 1$

The objective is to find feasible matchings in each of the bipartite graphs so as to minimize the shifts between the matchings.

i.e.

$$\textbf{Min} \sum_{t=1}^{\tau-1} d_t$$

The whole problem can be stated as follows,

Find maximum matchings $\{M_t : t = 1, 2, \ldots, \tau\}$ *in* $\{G_t : t = 1, 2, \ldots, \tau\}$ *such that* $\sum_{t=1}^{\tau-1} d_t$ *is minimum.*

[1]The problem can be more generalized by considering a p_t, the minimum number of assignment needed at the particular time interval. Then we say a matching M_t is feasible if $\mid M_t \mid \geq p_t$. If $p_t = m_t$ for some t, where m_t is the matching number (the cardinality of maximum matching of G_t) then problem is reduced to find the maximum matchings in each G_t.

Chapter 3

Sequential matching problem

In this thesis we present a problem that has not been dealt with in literature in the past, the *sequential matching problem* as described in Section 2.2.1. In the first part of this chapter we repeat the definition of the problem and discuss the nature of a general mathematical problem that arises from the formulation. Later on in the chapter we look as to how complex the problem is to solve and how a greedy algorithm behaves. Finally we also discuss a heuristic algorithm-based randomized scheme.

For a given bipartite graph $G_t = (L_t, R_t, E_t)$ the term *workers* is represented by the nodes in the left node set L_t and the terms *work places* or *jobs* are represented by the right node set R_t. *Abilities* of a worker to do a job are the synonym for the edges and E_t represents the collection of corresponding edges.

3.1 Sequential matching problem

Let V, W be the set of nodes and E be the set of edges of a bipartite graph $G(V, W, E)$. Define $\mathbf{G}_c = \{G_t = (L_t, R_t, E_t) : t = 1, 2, \ldots, \tau\}$ as a finite sequence of bipartite graphs where $L_t \subseteq V$, $R_t \subseteq W$ and $E_t \subseteq E$ for each $t = 1, 2, \ldots, \tau$.

Notice that each G_t is a subgraph of G. For any given matching M_t in G_t define,[1]

$$d_t = \begin{cases} \mid M_{t+1} \oplus M_t \mid & \text{for } t = 1, 2, \ldots \tau - 1; \\ 0 & t = \tau \end{cases} \tag{3.1.1}$$

i.e., d_t denote the number of different edges in two consecutive matchings. For a given finite sequence of positive integers p_t, we define a *constrained matching problem* of finding matchings M_t in G_t with at least p_t number of edges, such that $\sum_{t=1}^{\tau-1} d_t$ is minimum.

The objective is to find feasible matchings in each bipartite graph that minimize the switchings of the edges between the matchings.

$$\boldsymbol{Min} \sum_{t=1}^{\tau-1} d_t$$

Let m_t be the *matching number* (cardinality of the maximum matching) of the bipartite graph G_t. If we consider m_t instead of p_t the problem is reduced to finding the maximum matchings in each graph G_t, such that total changes between edges in matchings is minimum.

Formally we define the *sequential matching problem* (SMP) on a finite sequence of graphs $\mathbf{G}_c = \{G_t : t = 1, 2, \ldots, \tau\}$ as the problem of finding finite sequence of matchings $\mathbf{M}_c = \{M_t : t = 1, 2, \ldots, \tau\}$ where M_t is maximum in G_t such that $\sum_{t=1}^{\tau-1} d_t$ is minimum.

We use the notation $S(\mathbf{G}_c, \tau)$ for the problem where $\mathbf{G}_c$ and τ are as described.

3.2 Complexity

The purpose of this section is to gain insight into the difficulty of the problem. We give an NP-completeness proof for the problem. For a defined finite sequence of bipartite graphs $\mathbf{G}_c = \{G_t : t = 1, 2, \ldots, \tau\}$, the sequential matching problem $S(\mathbf{G}_c, \tau)$ is to find the finite sequence of matchings $\mathbf{M}_c = \{M_1, M_2, \ldots, M_\tau\}$ that minimizes different edges. For a given integer k we define the sequential matching decision problem as follows,

Sequential matching decision problem: Given a finite sequence of bipartite graphs $\mathbf{G}_c$ and an integer k is there a finite sequence of matchings $\mathbf{M}_c$ such that $\sum_{t=1}^{\tau-1} d_t \leq k$?

[1]For any two sets A and B we define the symmetric operator $\oplus$, as $A \oplus B = (A \cup B) \setminus (A \cap B)$, where $\setminus$ denotes the set minus.

Theorem 3.2.1. *Sequential matching problem is $\mathcal{NP}$-complete*

Proof. We reduce 3-SAT to SMP in two steps. Given an instance ϕ of 3-SAT, we first construct a finite sequence of bipartite graphs $\mathbf{G}_c$ that have matchings $\mathbf{M}_c$ with $\mid \mathbf{M}_c \mid \leq (2\tau - 2)$ if and only if ϕ is satisfiable. We are given an instance ϕ of 3-SAT with τ clauses $C_1, C_2, \ldots, C_\tau$, with p boolean variables $\alpha_1, \alpha_1, \alpha_3, \ldots, \alpha_p$ such that each clause is $C_t = (\alpha_{t1}, \alpha_{t2}, \alpha_{t3})$ with the α_{t1}'s being either boolean variables or negations thereof. Now we construct the sequence of graphs $\mathbf{G}_c = \{G_t : G_t = (V, W, E_t), t = 1, 2, \ldots, \tau\}$. For each clause C_t the construction of the bipartite graph $G_t = (V, W, E_t)$ is as follows:

$$V = \{v^T_{\alpha_j}, v^F_{\alpha_j}, v_c, \text{ for } j = 1, 2, \ldots, p\} \tag{3.2.1}$$

$$W = \{w_{c_{odd}}, w_{c_{even}}, w_{\alpha_j}, \text{ for } j = 1, 2, \ldots, p\} \tag{3.2.2}$$

$$E_t = E' \cup E'_t \tag{3.2.3}$$

where

$$E' = \{\{v^T_{\alpha_j}, w_{\alpha_j}\}, \{v^F_{\alpha_j}, w_{\alpha_j}\}, \text{ for all } j = 1, 2, \ldots, p\} \tag{3.2.4}$$

$$E'_t = \begin{cases} \{v_c, w_{c_{even}}\}, \{v_{\alpha_{tj}}, w_{c_{odd}}\} & \text{if } t \text{ is odd,} \\ \{v_c, w_{c_{odd}}\}, \{v_{\alpha_{tj}}, w_{c_{even}}\} & \text{otherwise.} \end{cases} \tag{3.2.5}$$

where

$$v_{\alpha_{tj}} = \begin{cases} v^T_{\alpha_j} & \text{if } \alpha_j \text{ is literal in clause } C_t, \\ v^F_{\alpha_j} & \text{if } \neg\alpha_j \text{ is literal in clause } C_t. \end{cases} \tag{3.2.6}$$

An example for four variables is depicted in Fig. 3.1. Say $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are the variables. An instance $\phi = C_1 \wedge C_2 \wedge C_3$ where,

$$C_1 = (\alpha_1 \vee \neg\alpha_2 \vee \neg\alpha_3) \tag{3.2.7}$$

$$C_2 = (\neg\alpha_1 \vee \alpha_2 \vee \neg\alpha_4) \tag{3.2.8}$$

$$C_3 = (\neg\alpha_2 \vee \neg\alpha_3 \vee \alpha_4) \tag{3.2.9}$$

We prove that a matching sequence $\mathbf{M}_c$ exists, with at least $2\tau - 2$ changes if and only if there is a truth assignment for a 3-SAT instance.

First, if there is a truth assignment, we prove that $M_1, M_2, \ldots, M_\tau$ exist with 2 changes between two consecutive graphs, and so in total $2\tau - 2$ changes. Suppose there is a truth assignment to the variables which satisfies all of the clauses such that each of the clause C_t has at least one literal $\alpha_{tj'}$ with a true value. Say $\alpha_{j'}$ is the variable corresponding to the true literal in C_t (*i.e.* $\alpha_{tj'}$ is either $\alpha_{j'}$ or $\neg\alpha_{j'}$). Now describe a matching M_t in G_t, for $t = 1, 2, \ldots, \tau$ as follows

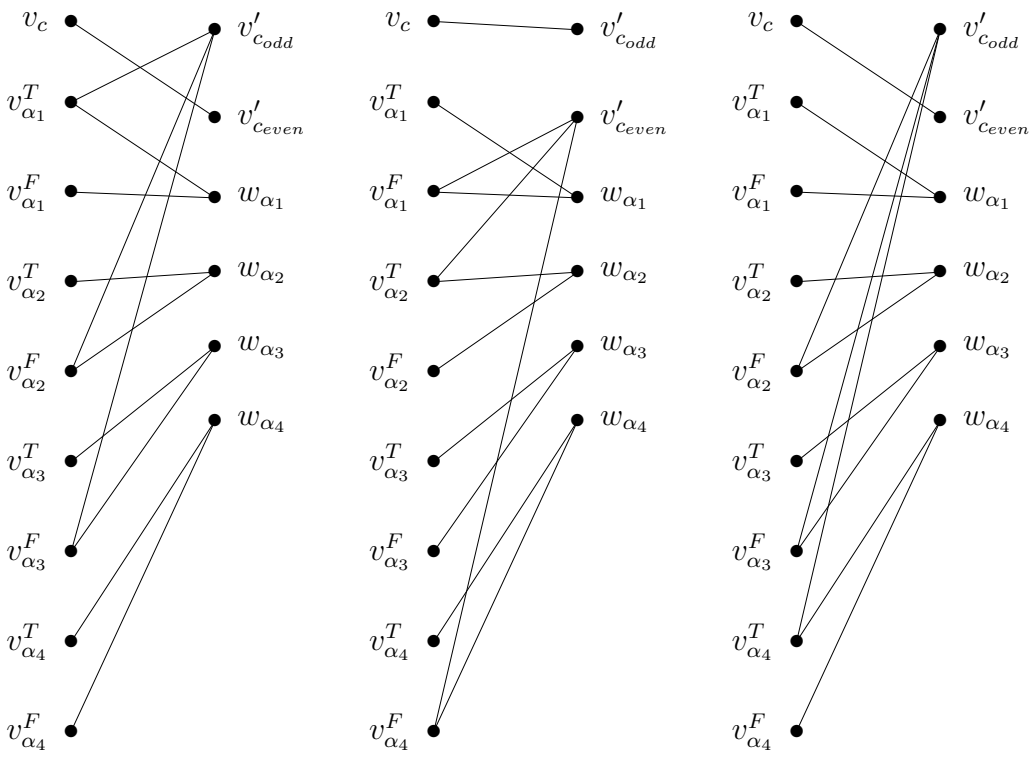

Figure 3.1: Bipartite graphs reduced from ϕ

- If t is odd,
  - Match the node with label node $w_{c_{even}}$ to v_c
  - if $\alpha_{tj'} = \alpha_{j'}$
    - Match the node with label $w_{c_{odd}}$ to $v^T_{\alpha_{j'}}$
  - else if $\alpha_{tj'} = \neg\alpha_{j'}$
    - Match the node with label $w_{c_{odd}}$ to $v^F_{\alpha_{j'}}$
- If t is even,
  - Match the node $w_{c_{odd}}$ to v_c
  - if $\alpha_{tj'} = \alpha_j{}'$
    - Match the node with label $w_{c_{even}}$ to $v^T_{\alpha_{j'}}$
  - else if $\alpha_{tj'} = \neg\alpha_{j'}$
    - Match the node with label $w_{c_{even}}$ to $v^F_{\alpha_{j'}}$
- For all $j = 1, 2, \ldots, p$

- if α_j is true
  - Match the node with label w_{α_j} to $v^F_{\alpha_j}$
- else if α_j is false
  - Match the node with label w_{α_j} to $v^T_{\alpha_j}$

It is easy to notice that each of the matchings are maximum (since all nodes W are matched), and the number of changes between M_t and M_{t+1} is *two*, for $t = 1, 2, \ldots, \tau - 1$. The total number of changes is $2\tau - 2$. To prove that $\mathbf{M}_c$ is with minimum changes, it is sufficient to show that the total changes in any maximum matchings are at least $2\tau - 2$. This is true because, from the construction of bipartite graphs, the edges incident with $w_{c_{even}}$ and $w_{c_{odd}}$ are totally different in two consecutive graphs. So at least 2 changes in edges are needed to get maximum matchings, in two consecutive graphs. This implies minimum total changes are at least $2\tau - 2$, because there are $\tau - 1$ consecutive pairs of bipartite graphs.

Conversely assuming that maximum matchings with $2\tau - 2$ changes exist. Since all the matchings are maximum, each node in W is matched for every graph. By the construction each of the w_{α_j} can match to either $v^T_{\alpha_j}$ or $v^F_{\alpha_j}$. We can find a truth assignment for an instance of 3-SAT, as follows,

- If w_{α_j} is matched to $v^T_{\alpha_j}$ take α_j is *false.*
- Else w_{α_j} is matched to $v^F_{\alpha_j}$ take α_j is *true.*

Since the minimum number of changes is $2\tau - 2$, changes can only happen in matching edges of $w_{c_{odd}}$ or $w_{c_{even}}$. If t is even, there would be at least one free node, say $v^T_{\alpha_{j'}}$ (or $v^T_{\alpha_{j'}}$) for $w_{c_{even}}$ such that $\alpha_{j'}$ (or $\neg\alpha_{j'}$) is true. If t is odd the free node would be for $w_{c_{even}}$. □

3.2.1 Vector representations

For a given bipartite graph $G = (L, R, E)$, let $\mathbf{G}_c = \{G_t : t = 1, 2, \ldots, \tau\}$ be the finite sequence of bipartite graphs such that each $G_t = (L_t, R_t, E_t)$ is a subgraph of G. Say the cardinality of E is m, we define a one to one onto mapping $f : E \to \{1, 2, \ldots, m\}$ such that an e_j edge represented by an edge number $f(e_j)$.

Assuming f is a linear sequence such that $f(e_j) = j$ for all $e_j \in E$. For any subset of edges $E' \subseteq E$ the characteristic vector of E', $\mathbf{x}^{E'} \in \{0, 1\}^m$ is such that the j^{th}

component $x_j^{E'}$ of $\mathbf{x}^{E'}$ is as follows.

$$x_j^{E'} = \begin{cases} 1, & e_j \in E'; \\ 0, & \text{otherwise.} \end{cases} \tag{3.2.10}$$

We can define an equivalent definition for the sequential matching problem $S(\mathbf{G}_c, \tau)$ with $\mathbf{G}_c = \{G_t : t = 1, 2, \ldots, \tau\}$ such that $G_t = (L_t, R_t, E_t)$ as in Section 3.1. Let $\mathbf{x}^{E_t}$ denote the characteristic vector of the edge set E_t.

Let $\mathbf{x}^{M_t}$ be the characteristic vector of the edge set of M_t. We formulate an equivalent form for $S(\mathbf{G}_c, \tau)$using the vector notations. If "$\cdot$" denotes the dot product of the vectors then $(\mathbf{x}^{M_t} \cdot \mathbf{x}^{M_{t+1}})$ is the number of edges common in the intersection of M_t and M_{t+1}.

$$D_t = \sum_{t=1}^{\tau-1} (\mathbf{x}^{M_{t+1}} \cdot \mathbf{x}^{M_t}) \tag{3.2.11}$$

We give two different integer programming formulations for the problem in Chapter 4.

3.3 A greedy approach

Some algorithms utilized to solve assignment problems comprise a series of steps, with a set of alternative choices available at each step. A greedy algorithm opts for the choice that looks best at any decision point. The idea being that at any such point, this locally optimum choice might lead to a good solution. Greedy algorithms are generally very fast, but do not always lead to a globally optimum solution.

The idea behind our greedy method is to find the solution for the problem as follows. We start with a feasible matching for the first graph, and find feasible matchings in each consecutive step with minimum possible changes in assignment. In the upcoming section we consider the problem of finding a maximum matching rather than matchings with fixed minimum cardinality.
Initial step $(t := 1)$**:** Start with a maximum matching M_1 on the Bipartite graph G_1. $(t+1)^{th}$ **step :** If M_t is a maximum matching on a bipartite graph G_t then

find a maximum matching M_{t+1} on G_{t+1} such that

$$d_t = \mid (M_t \cup M_{t+1}) - (M_t \cap M_{t+1}) \mid$$

is minimum.

In the $(t+1)^{th}$ step of the greedy method we need to find a maximum matching in G_{t+1} with maximum edges of M_t. This can be done by finding a maximum weighted perfect matching [10] on a weighted bipartite graph formed by adding extra weights to edges of G_{t+1} which are already in M_t. Generally a globally optimum solution to the problem may be obtained by making a locally optimum (greedy) choice. Thus, at any step, the choice made by a greedy algorithm may depend on choices made up to that point, but not upon future choices or on the solutions to subproblems. Therefore, a greedy strategy proceeds in one direction making a greedy choice at each step, iteratively reducing each problem instance to a smaller one. The quality of greedy solution also depends on the initial choice for the matching in G_1.

The greedy algorithm described here does not always yield an optimum solution, even with a good initial choice. The reason is that the choice at each step is independent of solutions to the future subproblem. We construct a counter example to claim that this greedy method need not give an optimum solution.

Consider three bipartite graphs G_1, G_2, and G_3 as shown in Fig. 3.2. The problem is to find perfect matching in each bipartite graph with minimum changes in edges. Consider the matchings in 3.3 as the output of the above greedy method for the

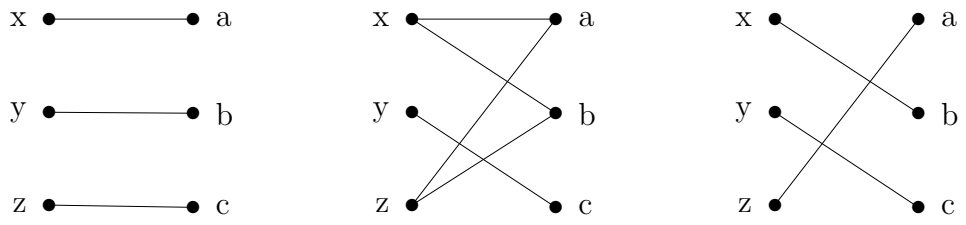

Figure 3.2: Bipartite graphs in 3 different intervals

problem in the bitartrate graphs G_1, G_2, and G_3. Now let us examine how the algorithm outputs this solution. The algorithm starts with the only maximum matching of the first bipartite graph G_1. The next step of the algorithm is to find a matching in second bipartite graph with minimum number of changes with respect to the matching of first bipartite graph. It is easy to see that in second bipartite graph there are two perfect matchings where one has two different edges

compared to the only matching of the first graph, and other has three. Naturally the greedy algorithm chooses the matching with two changes because of its local optima strategy. Same as above, the algorithm finds a perfect matching in third bipartite graph with additional two changes with respect to matching in second bipartite graph such that the total number of changes are *four*. It is easy to see

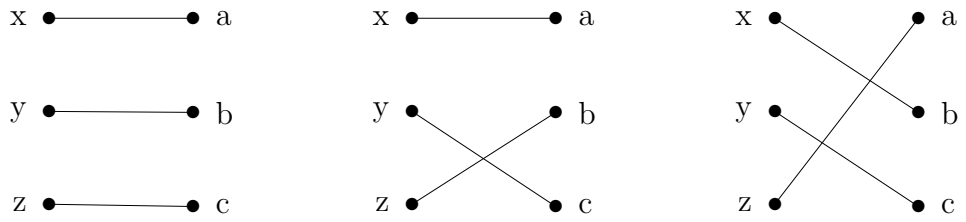

Figure 3.3: A Greedy Solution for the problem in Fig.3.2

that the optimum solution for the sequential matching problem in G_1, G_2, and G_3 is not by the above greedy method, but the matchings as in 3.4 with only *three* changes in total. The previous example shows that the greedy algorithm does not always provide an *optimum* solution to the problem, which motivates the search for a better algorithm. In the following section a randomized algorithm is discussed.

3.4 A randomized algorithm

Any algorithm that makes some *random (or pseudorandom)* choices is called a *randomized algorithm*. In this section we present an incremental randomized algorithm for the sequential matching problem with simpler assumptions. A randomized algorithm makes arbitrary choices during its execution. Here, the algorithm uses some results on finding an *allowed edge* with the concept of a random adjacency matrix.

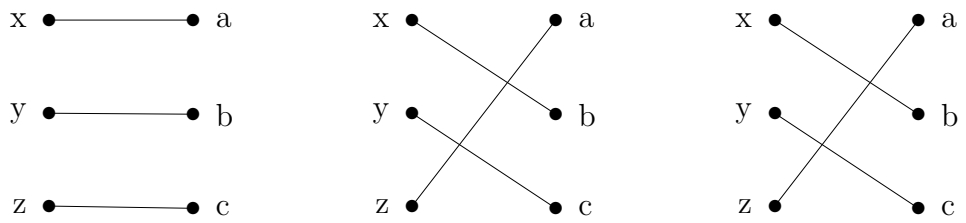

Figure 3.4: Optimum Solution for the problem in Fig.3.2

3.4.1 Simplified model

We simplify the problem by assuming that perfect matching exists in each of the bipartite graphs. Now the problem reduces to finding a perfect matching in each bipartite graph which minimizes the total number changes. Alternately,

find perfect matchings $M_1, M_2, \ldots, M_\tau$ *in bipartite graphs* $G_1, G_2, \ldots, G_\tau$ *such that* $\sum_{t=1}^{\tau-1} d_t$ *is minimum, where* $d_t =\mid M_t \oplus M_{t+1} \mid$.

Without loss of generality we can assume that the problem in section 2.2.1, of finding maximum cardinality matchings in bipartite graphs can be reduced to a problem of finding perfect matchings, if the matching number $m_t = min(\mid L_t \mid, \mid L_t \mid)$. This can be done by adding additional nodes to L_t or R_t (such that $\mid L_t \mid = \mid R_t \mid$), and connecting those added nodes to all nodes of the opposite vertex group in the bipartite graph.

3.4.2 The concept

The idea behind the algorithm is simple. To find perfect matchings with minimum changes, we look for matchings with maximum common edges. For this purpose we compute the possible edge-intersections between a set of consecutive bipartite graphs and find matchings which contain a maximum number of intersected edges. One important fact to note is that the addition of all intersected edges is not possible even if they are feasible to be in the matching. This is because some of them are not *allowed* by mean, they prohibit convergence of the local solution (a collection of edges) to a perfect matching.

3.4.3 The algorithm

Let $\tau > 0$ and $j < \tau$ be positive integers. We define the index set S_{tj} as $\tau - t$ consecutive integers starting with $j + 1$ as follows,

$$S_{tj} = \{j + 1, j + 2, \ldots, j + (\tau - t)\} \tag{3.4.1}$$

Note the cardinality, $\mid S_{tj} \mid = \tau - t$. Let I_t be the set of graphs generated by edge-intersection of bipartite graphs whose indices lie in S_{tj}. Define,

$$I_t = \{N_{tj} \mid N_{tj} = \bigcap_{k \in S_{tj}} G_k \;\; j = 1, 2, 3, \ldots, t\} \tag{3.4.2}$$

for $t = 0, 1, 2, \ldots, \tau - 1$.

Also, note that $\mid I_t \mid = t + 1$. With these assumption we define the method as in algorithm 2.

Algorithm 2 The randomized algorithm of edge-intersection method

EdgeIntersectionMethod for $\mathbf{G}_c$
input:A finite sequence of bipartite graphs $\mathbf{G}_c = \{G_t : t = 1, 2, \ldots, \tau\}$
for $t := 0$ to $\tau - 1$ **do**
  $M_t := \varnothing$
end for
for $t := 0$ to $\tau - 1$ **do**
  if $I_t \neq \varnothing$ **then**
    $G'_k := G_k$
    for $N_{tj} \in I_t$ **do**
      for $k \in S_{tj}$ **do**
        -Find a vertex $v \in G'_k$ of degree 1 if one exists
        -If such vertex does'nt exist, go to for next $k \in S_{tj}$
        -Let $e = (v, w)$ be the edge such that $degree(v) = 1$ or $degree(w) = 1$, then add e to M_k
        -Delete vertices v and w (and thus edges too) from G'_k
        $G'_k := G'_k - \{v, w\}$
      end for
      -Choose an edge e *randomly* from N_{tj}, such that $e \in G'_k$
      if e is *allowed* in each G'_k, **then** add e to each M_k
      -Delete end vertices of e (and so edges also) from each G_k.
      -Delete e from N_{tj}
    end for
  end if
end for
for $t := 1$ to τ **do**
  Find the maximum matching with existing edges from M_t
end for
return $\mathbf{M}_c$

It is easy to observe that the above algorithm gives maximum matchings M_t for G_t, for $t = 1, 2, 3, \ldots, \tau$. Assuming that all of the bipartite graphs have perfect matchings. Notice that all of the edges for which at least one end vertex has degree one should be in all perfect matchings. The other edges added to matchings are *allowed edges*, in reduced subgraphs respectively (a subgraph formed after deletion of matched nodes). By definition of *allowed edges* a perfect matching exists in each

reduced subgraph even after deletion.

3.4.4 Finding an allowed edge

Recall that an edge of a graph is called *allowed* if it occurs in at least one maximum cardinality matching.[2] The complexity of checking whether an edge is allowed or not can take as much effort as finding a maximum matching. A method of finding whether an edge, say e, is allowed or not in a graph $G = (V, E)$ is as follows,

- Let's say $e = (v, w)$ such that $v, w \in V$.
- Remove all those edges incident with v, w other than e from G. Let G' be the resultant graph.
- Solve the maximum matching problems for G and G'
- If the cardinality of maximum matchings in G and G' are equal, it implies that e is an allowed edge. Else it is not

The algorithm described in the previous section frequently need to check for an allowed edge. The following parts of this section describe a randomized method for finding *allowed edges*.

Tutte, [11] gave a good characterization of graphs that have perfect matchings. One of Tutte's innovations was to introduce the skew symmetric adjacency matrix B of the graph G, defined as follows: Associate each edge e_{ij} of G with a distinct variable x_{ij}. Then $B = B(x_{ij})$ is a $\mid V \mid \times \mid V \mid$ matrix whose entries are given by

$$B_{ij} = \begin{cases} x_{ij} & \text{if } i > j \text{ and } ij \in E, \\ -x_{ij} & \text{if } i < j \text{ and } ij \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{3.4.3}$$

Tutte observed that G has a perfect matching if and only if the determinant of $B(x_{ij})$, $det(B(x_{ij}))$, is not identical to zero; here, $det(B(x_{ij}))$ is a polynomial in the variables x_{ij}. Lovasz [12] used this observation to give an efficient randomized algorithm for the perfect matching decision problem. Choose a prime number $q = \mid V \mid^{O(1)}$, and substitute each variable x_{ij} in B by an independent random

[2] A brief discussion and general definition of an *allowed subset* are set out in Chapter 6.

number drawn from $1, 2, \ldots, q-1$. Compute the determinant of the resulting random matrix B over the field of integers modulo q. With high probability (*i.e.*, probability $\geq 1 - 1/O(\mid V \mid)$), $det(B) = 0$ *mod* q if and only if $det(B(x_{ij}))$ is not identical to zero if and only if G has a perfect matching [13]. This algorithm has two especially attractive features: it is simple, solving the decision problem by executing one "matrix operation", and is efficient, running in sequential time.

A *Monte Carlo algorithm* (a randomized algorithm that may produce incorrect results, but with bounded error probability) for finding the set of allowed edges of an arbitrary graph is described in [14]. The method for finding the set of allowed edges first constructs the Gallai-Edmonds decomposition using the randomized algorithm.

If G has a perfect matching we apply the following result of Rabin and Vazirani [14] to find (with high probability) the allowed edges.

Lemma 3.4.1. *(Rabin and Vazirani). Let G be a graph with a perfect matching, and let B be a random skew symmetric adjacency matrix of G. If* $\det(B) = 0$*, then for each index $i, 1 \leq i \leq n$, there is an index j, $1 \leq j \leq n$, such that $B_{ij} \neq 0$ and $(B^{-1})_{ji} \neq 0$; moreover, for each pair i, j satisfying this condition, the corresponding edge $v_i v_j$ is in some perfect matching of G.*

Theorem 3.4.2. *With probability at least $1-(1/n^{\Theta(1)})$, the set of allowed edges of a graph can be computed in sequential time $O(M(n))$, and in parallel time $O((logn)^2)$ using $O(M(n))$ processors.*

The implementation details of this randomized edge intersection method is described in Appendix C.

Chapter 4

A branch and price approach

4.1 Introduction [1]

Column generation is a powerful tool for solving large scale linear programming problems. Such linear programming may arise when the columns in the problem are not known in advance and a complete enumeration of all columns is not an option, or the problem is rewritten using Dantzig- Wolfe decomposition (the columns correspond to all extreme points of a certain constraint set) [18] [16]. Column generation is a natural choice in several applications, such as the well known cutting stock problem, vehicle routing and crew scheduling. In this chapter we describe a column generation approach that dynamically generates columns and also provides tighter relaxations of the underlying mixed integer optimization problem representation of the sequential matching problem.

If a linear program contains too many variables to be solved explicitly, then we initialize the linear program with a small subset of variables and compute an optimum solution of that linear program. Afterwards, we check whether the addition of a variable, which is not in the current linear program, might improve the LP solution. According to linear programming theory this can be done by the computation of the reduced costs of the variables. In a linear program of the form $min\ \{\mathbf{c}^\intercal\mathbf{x}|\mathcal{A}\mathbf{x} \leq \mathbf{b} : \mathbf{x} \geq \mathbf{0}\}$ a variable with positive reduced cost can improve the solution[2]. If no variables have positive reduced costs, then the current optimum

[1] Some of the general discussion about the branch and price method is from the references [15], [16], [17].

[2] $^\intercal$ to denote the matrix transpose.

solution also solves the original problem. The computation of the reduced costs is also called *pricing*. If a variable does not price out correctly, we add it to the linear program, re-optimize, and iterate.

In cutting plane algorithms optimum solutions are found by generating cutting planes and by adding cuts to LP relaxation. As in the case of a cutting plane algorithm, an explicit list of constraints is not required, we do not need an explicit list of variables in a column generation algorithm. We require only a method for generating variables of the original problem that do not price out correctly. Given a class of variables of a linear optimization problem, and the values of the dual variables of a basic solution, either prove that all variables of this class price out correctly or find a nonbasic variable of this class that does not price out correctly.

An algorithm that solves the general pricing problem is called an exact pricing algorithm, while a heuristic pricing algorithm may find a variable that does not price out correctly, but if it fails, it is not guaranteed that all variables of the class price out correctly. Also, the pricing problem can be formulated as an optimization problem. In this case, that variable among the variables which do not price out correctly with lowest (negative) reduced cost should be found.

The most widely used method for solving integer programs is *branch and bound*. Subproblems are created by restricting the range of the integer variables. *Branch and price* method is essentially a *branch and bound* combined with *column generation*. This method is used to solve integer programs where there are too many variables to represent the problem explicitly. Thus only the active set of variables are maintained and columns are generated as needed during the solution of the linear program. Column generation techniques are problem specific and can interact with branching decisions. The master problem discussed here is a *set partitioning* type while the *pricing problem* can be formulated as a constrained shortest path problem.

Before introducing the column generation algorithm for the *Sequential Matching Problem*, we formulate the problem as a Mixed Integer Programming (MIP) problem and then extensively reformulate it into a set partitioning type one, which could be more relevant from the branch and price point of view. In the next section we analyze the problem, and formulate the pricing problem as a set of shortest path problems. The branching scheme selected for the branch and price, convergence of the algorithm and the complexity of the pricing problem are discussed in the following sections.

4.2 Formulations

Integer programs and their associated linear relaxations encountered in applications almost always exhibit a great deal of structure. Various formulations are possible for a problem but the quality depends on the structure of the problem and ease of solving it. Computationally this also underlines the importance for strictly integral optimization techniques, as opposed to faster running real valued formulations suitable for linear programming treatment.

A traditional way of analyzing thequality of a formulation is by polytope inclusion. Most integer programming algorithms require an upper bound on the value of the objective function, and the efficiency of the algorithm is very dependent on the sharpness of the bound. An upper bound is determined by solving the linear programming relaxation. But for a column generation algorithm the best formulation is when a significantly large number of columns is compared with rows in the constraint matrix.

Firstly we introduce a "compact" MIP formulation for the sequential matching problem. In the following section we reformulate the problem into a set partitioning type one. These optimization models have certain claims to optimality, although computational constraints and the solution method decide the quality of the formulations.

4.2.1 MIP formulation of the sequential matching problem

A wide variety of problems encountered in many areas including personnel scheduling problems can be formulated as mixed integer programming (MIP) models in many different ways. The respective formulations influence the computational behavior of exact solution methods. Our aim is to develop an MIP model representation for the problem.

Consider the sequential matching problem defined in Section 3.1, $S(\mathbf{G}_c, \tau)$, where τ is the total number of bipartite graphs. A conventional way of integer formulation for the maximum matching problem in a single bipartite graph G_t is described in Section 1.2.3. The objective of that formulation is finding the maximum number edges in the matchings and constraints to make sure that the resulting set of edges forms a matching.

If we know the *matching number*, m_t for G_t the problem of finding maximum matching with m_t edges can be reduced to the integer solution of a set of constraints as in Fig. 4.1 (we use the same notations as in Section 1.2.3).

$$\begin{aligned} \sum_{e \in \delta(v)} x_e &\leq 1 \qquad \forall v \in V \\ \sum_{x_e \in E_t} x_e &\geq m_t \\ x_e &\in \{0,1\} \end{aligned}$$

Figure 4.1: An alternate formulation for the bipartite graph matching problem

We use this idea to develop a MIP formulation for the $S(\mathbf{G}_c, \tau)$. In case of $S(\mathbf{G}_c, \tau)$ the problem is with more additional constraints.

We defined $S(\mathbf{G}_c, \tau)$ as the graph theoretical problem motivated by the worker-to-job assignment in different time shifts 2.2.1. In which the collection of workers available at a time shift t is represented by the left node set L_t, and the jobs to be done at t is represented by the right node set R_t in the graph G_t, where the edges E_t are the abilities in this particular time shift.

We return to the notion of *worker, job, abilities,* and *time shifts*, because a linear programming formulation is more meaningful. Later in the column generation formulation (Section 4.3.2) it is more obvious, since each column in the binary constraint matrix represents the *life line of a worker* (*i.e.* a possible schedule for a particular worker in all time shifts). In Theorem 4.2 we prove the equivalence of the formulations. We proceed as follows.

Let $\mathcal{J} = \{1, 2, \ldots, n\}$ denote the set of job indices, $\mathcal{W} = \{1, 2, \ldots, m\}$ the set of worker indices and $\mathbb{T} = \{0, 1, 2, \ldots, \tau\}$ the time discretization such that,

$$\mathcal{T} = \{t | t \text{ is a time shift starts from } \mathbf{t}-1 \text{ ends at } \mathbf{t}, \text{ for } \mathbf{t} \in \mathbb{T} - \{0\}\}$$

and E_t is the collection of possible assignments at time shift t (*i.e.* edges in the bipartite graph G_t at t).

We define two types of binary decision variables x_{ijt} and y_{ijt}. The binary decision variable x_{ijt} assumes the value of *one* if the worker i is assigned to the job j at t. In other words the edge $e_{ij} \in M_t$ for the problem $S(\mathbf{G}_c, \tau)$. The binary decision variable y_{ijt} (defined for $t \in \{2, 3, \ldots, \tau\}$) is *one* when the worker i is assigned to

job j in two consecutive time intervals $t-1$ and t. The variable $y_{ijt} = 1$ only when the edge e_{ij} is in both M_t and M_{t+1}. More precisely,

$$x_{ijt} = \begin{cases} 1 & \text{if worker } i \text{ is assigned to job } j \text{ at time shift } t; \\ 0 & \text{otherwise.} \end{cases} \tag{4.2.1}$$

and for $t \in \{2, 3, \ldots, \tau\}$,

$$y_{ijt} = \begin{cases} 1 & \text{if worker } i \text{ is assigned job } j \text{ at time shifts } t-1 \text{ and } t; \\ 0 & \text{otherwise.} \end{cases} \tag{4.2.2}$$

An integer programming formulation for the problem of finding matchings (with cardinality at least m_t) with minimum changes in edges is as in Fig. 4.2. We denote (i, j), whether the worker i is eligible to do the job j (the edge e_{ij} in the graph formulation in Section 3.1). Solving this *compact* formulation integrally is theoretically no different from solving any integer program.

$$\textbf{\textit{Max}} \sum_{t=2}^{\tau} \sum_{(i,j)\in(E_{t-1}\cap E_t)} y_{ijt} \tag{4.2.3}$$

subject to

$$\sum_{i:(i,j)\in E_t} x_{ijt} \leq 1 \quad \forall j, \quad \forall t \tag{4.2.4}$$

$$\sum_{j:(i,j)\in E_t} x_{ijt} \leq 1 \quad \forall i, \quad \forall t \tag{4.2.5}$$

$$\sum_{i,j:(i,j)\in E_t} x_{ijt} \geq m_t \quad \forall t \tag{4.2.6}$$

$$x_{ijt-1} \geq y_{ijt} \quad \text{and} \tag{4.2.7}$$

$$x_{ijt} \geq y_{ijt} \tag{4.2.8}$$

$$\forall i, j : (i, j) \in E_{t-1} \cap E_t \quad \text{for } t = 2, 3, \ldots, \tau$$

$$x_{ijt}, y_{ijt} \in \{0, 1\} \tag{4.2.9}$$

Figure 4.2: MIP formulation

Theorem 4.2.1. *The optimum solution for the mixed integer programming problem in Fig. 4.2 gives an optimum solution for sequential matching problem in 3.1.*

Proof. The constraints 4.2.4 and 4.2.5 guarantee that a feasible solution brings out matchings in each bipartite graph. Since for each i and j, at most one x_{ijt} can assume a value of 1, which guarantees the degrees of the nodes in the corresponding bipartite graph, representing i and j is less than or equal to 1. The constraint 4.2.6 guarantees the cardinality of matchings, since m_t can be up to the matching number (the number of edges in maximum matching) of the bipartite graph at time shift t. The problem of minimizing changes in edges between matchings is equivalent to the problem of maximizing common edges. In Fig. 4.2 the variables y_{ijt} represent these common edges. So the objective function together with the constraints, 4.2.7 and 4.2.9 are to maximize common edges between two consecutive graphs. □

4.2.2 An extensive reformulation

Our intention is how to reformulate an integer program in order to build an efficient branch and price model. We propose an alternative master problem that can be quite advantageous in this situation. In the previous section we have formulated the problem in a compact way which explicitly reflects the structure and a set of coupling constraints. This type of formulation naturally leads to a solution by a *decomposition process* like that of Dantzig and Wolfe [18]. Decomposition of integer programs is done by replacing a subsystem of the constraints by a reformulation that possesses the integrality property. This problem can however also be formulated in such an extensive way, other than the decomposition method by enumeration of a subset of solutions. We make the point that this extensive formulation is not only an ideal column generation scheme, but naturally gives rise to branching rules for it.

In fact, we now formulate a master program for the *sequential matching problem* in terms of "lifeline of a worker" as variables, *(*i.e.), each column corresponds to a particular assignment pattern of workers during the time periods.

Let $\mathcal{J} = \{1, 2, \ldots, n\}$ denote the set of job indices, $\mathcal{W} = \{1, 2, \ldots, m\}$ the set of worker indices and $\mathbb{T} = \{0, 1, 2, \ldots, \tau\}$ be the time discretization such that,

$\mathcal{T} = \{t | t$ is a time shift which starts from $\mathbf{t} - 1$ and ends at $\mathbf{t}$, for $\mathbf{t} \in \mathbb{T} - \{0\}\}$

Define $\mathbf{K}_i = \{1, 2, \ldots, k_i\}$, where k_i corresponds to the total number of feasible assignments for worker i during the time intervals in $\mathcal{T}$. *(*i.e.), Assignments for a worker i can be undertaken in k_i different ways. Each of these assignments can be encoded as a binary vector A^i_k with $n\tau$ entities, for each $k = 1, 2, \ldots, k_i$, as follows,

The jt-th element of A_k^i is 1, if job j is assigned to worker i at time interval t. Let $\mathbf{A}$ be the matrix with columns A_k^i for all $i \in \mathcal{W}$, for all $k \in K_i$ with $\sum_{i \in \mathcal{W}} k_i$ columns and nt rows. Define the set of columns of $\mathbf{A}$ as $\mathcal{A} = \{A_k^i | i \in \mathcal{W}, k \in K_i\}$.

We define a binary decision variable x_{ik} for each column $A_k^i \in \mathcal{A}$, such that it assumes the value of *one* if the schedule as in A_k^i is selected for worker i.

i.e.

$$x_{ik} = \begin{cases} 1 & \text{if the assignment } k \in K_i \text{ is selected for worker } i, \\ & \textit{i.e.} \text{ colomn } (A_k^i)^\tau \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \tag{4.2.10}$$

Let $\mathbf{X} = (x_{ik} : \text{for all } i \in \mathcal{W}, k \in K_i)$. A set partitioning type formulation for the problem of finding matchings (with cardinality at least m_t) with minimum switchings is formulated as optimum selection of A_k^i which satisfies the feasibility of assignment constraints. The cost c_{ik} of each column A_k^i is defined as the number of changes in the assignments for the worker i as per the column A_k^i. This implies if $A_k^i = [a_{jt} : a_{jt} \in \{0,1\}]^\intercal$ then $c_{ik} = \sum_{t \in \mathcal{T}} \sum_{\jmath \in \mathcal{J}} (1 - a_{jt} a_{jt+1})$.

To summarize, this model reads as follows:

$$\textbf{\textit{Min}} \sum_{i=1}^{m} \sum_{k=1}^{k_i} c_{ik} x_{ik} \tag{4.2.11}$$

subject to

$$\mathbf{AX} = \mathbf{1} \tag{4.2.12}$$

$$\sum_{k=1}^{k_i} x_{ik} = 1 \qquad \forall i, \tag{4.2.13}$$

$$x_{ik} \in \{0,1\} \tag{4.2.14}$$

Figure 4.3: A column-wise reformulation

Constraint matrix $\mathbf{A}$ is very sparse and appears as shown in Fig. 4.4

Theorem 4.2.2. *An optimum solution of the linear program in Fig. 4.3 gives an optimum solution for sequential matching problem too.*

	A_1^1	A_1^2	$\dots$	A_1^{k1}	.	.	.	.	A_m^1	A_m^2	$\dots$	A_m^{km}
11	0	0	$\dots$	0	.	.	.	.	1	0	$\dots$	0
21	1	0	$\dots$	1	.	.	.	.	0	0	$\dots$	0
$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$	.	.	.	.	$\vdots$	$\vdots$	$\dots$	$\vdots$
$m1$	0	0	$\dots$	0	.	.	.	.	0	0	$\dots$	1
.	.	.	$\dots$	.	.	.	.	.	.	.	$\dots$	.
.	.	.	$\dots$	.	.	.	.	.	.	.	$\dots$	.
.	.	.	$\dots$	.	.	.	.	.	.	.	$\dots$	.
.	.	.	$\dots$	.	.	.	.	.	.	.	$\dots$	.
1τ	0	0	$\dots$	0	.	.	.	.	0	0	$\dots$	0
2τ	1	0	$\dots$	1	.	.	.	.	0	1	$\dots$	1
$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$	.	.	.	.	$\vdots$	$\vdots$	$\dots$	$\vdots$
$m\tau$	0	0	$\dots$	0	.	.	.	.	1	0	$\dots$	0

Figure 4.4: Constraint matrix

Proof. Each $A_k^i \in \mathcal{A}$, represents a feasible assignment for i, in each time shift t. By constraint 4.2.13 only one of the columns can be chosen for each i. By the definition of A_k^i a feasible solution for 4.3 results in maximum matchings in the corresponding bipartite graphs. The objective function is to minimize the total changes for each worker assignment, which is equivalent to minimizing the changes between matchings in bipartite graphs. □

4.3 Methodology outline

In the previous section we have introduced a different reformulation for the *sequential matching problem*. However, an explanation of how to practically handle the resulting large models is still owing. Solving the linear master directly, *(*i.e.), by means of the straightforward application of, say, the simplex method is definitively out of reach. To begin with, recall that in the revised simplex method all data required per iteration is calculated directly from the original data and this tableau method which modifies the whole input data from iteration to iteration. What is more, only the pricing step needs access to nonbasic columns of the coefficient matrix when it comes to computing the reduced cost coefficients. Let the master program under consideration have the form of the linear relaxation of the problem in Section 4.3 which we denote by RMP (Relaxed Master Problem), with columns

from the set $\mathcal{A}$.

4.3.1 The restricted master program

We will work with a manageable subset of columns $\mathcal{A}^1 \subseteq \mathcal{A}$, at the worst starting with a set which contains only one primal feasible basis for RMP. Although obtaining this initial set constitutes a problem in its own right, let us assume for the moment that we are provided with such a column set. As in the literature, the master program with columns omitted is called *restricted*. At all times, the restricted master program represents all current problem information gathered from subproblem solutions, (i.e.), a subset of columns of the coefficient matrix of a linear program which proved to be useful to achieve progress in terms of the objective function value. From a technical point of view, the purpose of the restricted master program is twofold. Firstly, to combine columns/variables in an appropriate way in order to obtain a primal feasible solution, and secondly, to provide dual multipliers to be transferred to the subproblem in order to promisingly extend the current information.

4.3.2 Column generation

Since in each iteration of the simplex method exactly one basic column is exchanged for one nonbasic column, generation of columns to enter the basis is the idea behind the column generation method. Its realization is as follows. Associated with a primal optimum solution $\mathbf{X}^* \in \mathbb{R}^{|\mathcal{A}^1|}$ to RMP is and U^* is the corresponding dual optimum solution. Note again, that the optimization to obtain this solution is carried out having a (very small) subset of columns at hand but checking the optimality of $\mathbf{X}^*$ with respect to the full program requires testing nonnegativity of all reduced cost coefficients. This amounts to solving the pricing subproblem,

$$Z^* = min\{c_{ik} - U^{*\intercal} a_{ik} : i \in \mathcal{W}, k \in K_i\} \tag{4.3.1}$$

Primal methods, like column generation, maintain primal feasibility and work towards dual feasibility. It is therefore only natural to monitor the dual solution in the course of the algorithm. The dual point of view reveals a most valuable insight into the algorithms functioning. The restricted master program is first solved to obtain the optimum objective function value, and a second time with a different objective function, maximizing the sum of auxiliary variables which bound the dual variable values on the optimum face from below. We summarize the discussed linear programming column generation algorithm as follows,

1. Provide a feasible basis for restricted master program.

2. If a column $A_l \in \mathcal{A}'$ with negative reduced cost exists then go to step 3, otherwise stop.

3. Add column A_l to restricted master program, re-optimize.

4.3.3 Pricing problem

The pricing step in the simplex method is the task of pricing out the nonbasic variables, *i.e.*, determining one with a negative reduced cost coefficient (minimization assumed) which may enter the basis. In looking for such a column, we distinguish although seemingly not customary in the literature between pricing schemes and pricing rules, the former describing the set of nonbasic variables to consider, and the latter referring to the criterion according to which a column is selected from the chosen (sub)set. As a classical method of chosing from all columns the one with most negative reduced cost coefficient is an example for such a scheme/rule pair. In this sense, column generation is a pricing scheme for large scale linear programs. To each standard pricing rule there exists a column generation sibling: Instead of pricing out nonbasic variables by enumeration, e.g., the most negative reduced cost as per equation 4.3.1. In general, using this particular pricing scheme is obviously more costly than using standard pricing schemes. Despite the fact that the former cannot compete with the latter in terms of computational effort, its use is justified by extending the range of applicability of the simplex method to problem sizes impracticable to standard implementations.

4.3.4 MIP formulation of pricing problem

An integer programming formulation for the generic pricing problem is presented here. The main idea of the formulation is same as the MIP formulation for the $S(\mathbf{G}_c, \tau))$ in Section 4.2.1. The column generation subproblem can be formulated from the fact of minimizing the reduced cost as in Fig. 4.5.

The cost of the worker i for a particular schedule k is the number of total *job changes* in all time intervals. By constraint 4.3.4 and 4.3.6 the variable y_{ijt} assumes the value of 1 only when the worker i assigned to j in two consecutive time shifts. Therefore for each $i \in \mathcal{W}$, $c_{ik} = \sum_{t=2}^{\tau} \sum_{(i,j)\in(E_{t-1}\cap E_t)} y_{ijt}$ represents the common

$$\textbf{\textit{Min}} \quad \tau - \sum_{t=2}^{\tau} \sum_{(i,j)\in(E_{t-1}(E_{t-1}\cap E_t)} y_{ijt} - \sum_{t=2}^{\tau} \sum_{(i,j)\in E_t)} u_{ijt}x_{ijt} \tag{4.3.2}$$

subject to

$$\sum_{j:(i,j)\in E_t} x_{ijt} \leq 1 \quad \forall t \tag{4.3.3}$$

$$x_{ijt-1} \geq y_{ijt} \quad \text{and} \tag{4.3.4}$$

$$x_{ijt} \geq y_{ijt} \tag{4.3.5}$$

$$\forall j : (i,j) \in E_{t-1} \cap E_t \quad \text{for } t = 2, 3, \ldots, \tau$$

$$x_{ijt}, y_{ijt} \in \{0,1\} \tag{4.3.6}$$

Figure 4.5: The MIP formulation of the pricing problem.

assignments. Therefore the *cost* of a worker i to assigned for a particular schedule k is $\tau - c_{ik}$, since τ is maximum number of changes possible.

4.3.5 As shortest path problem

We say that a directed network $G = (N, A)$ with a specified source node s and a specified sink node t is *layered* if we can partition its node set N into k layers $N_1, N_2, \ldots, N_k$ so that $N_1 = \{s\}$, $N_k = \{t\}$, and for every arc $(i, j) \in A$, nodes i and j belong to adjacent layers (*i.e.*, $i \in N_l$ and $j \in N_{l+1}$ for some $1 \leq l \leq k-1$). The problem to be solved is equivalent (Theorem 4.3.1) to determining workers and the most economical way of assigning them using the available cost function. For each i, this can be represented as an acyclic directed network as follows. For each $j \in \mathcal{J}$, $t = 1, 2, \ldots, \tau - 1$, a node n_{jt} is with weight u_{jt}. The directed edges from n_{jt} to $n_{j't+1}$ if $(i, j) \in E_t$ and $(i, j') \in E_{t+1}$, the weight of the edge is 1 if $j \neq j'$ and 0 if $j = j'$. To convert the node weights to edge weight problem, we can do any of the following,

1. By replacing n_{jt} by two nodes n^1_{jt} and n^2_{jt}, and joined by an edge with weight of node n_{jt}(*i.e.* with u_{jt}).

2. By adding the node weights to all edges incident from it. This is possible since the graph is acyclic.

By introducing a source and a target node, and connecting the source node to all nodes n_{j1} and $n_{j\tau}$ to the target node, with 0-weighed edges we can transform the problem to a single source single target *shortest path problem*. For example Fig. 4.7 represent the shortest path problem for the problem in Fig. 4.6. The notations in the figure are self-explanatory, for example, the node 'w1t3' denotes the worker 1 at time shift 3, the node 'j4t3' denote the job 4 at shift 3, and the edge is between them if the worker 1 is able to do the job 4 at time shift 3.

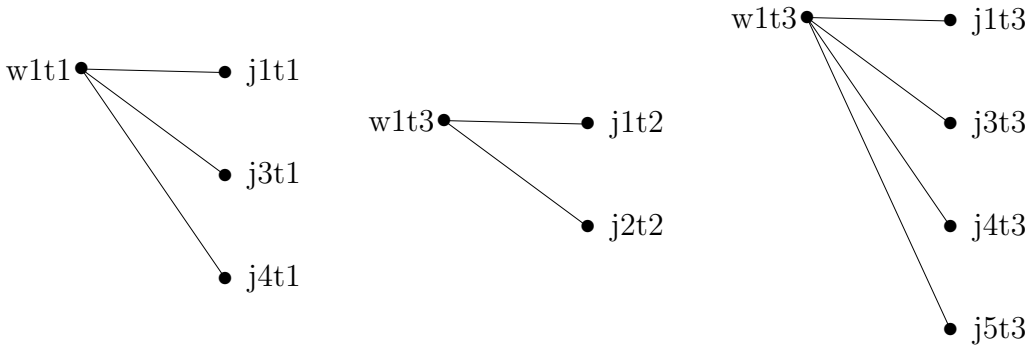

Figure 4.6: Worker abilities in 3 different time intervals

A shortest path from source node to target node gives a solution to the minimum reduced cost problem arising from the column generation method. Since the role of the pricing subproblem is to provide a column that prices out profitably or to prove that no such column exists, it is a good point to note that any column with negative reduced cost, if any, contributes to this aim. In particular, in order to keep the iteration going, there is no need to solve it exactly. With respect to the ability to choose a different pricing rule it is not even mandatory to state the pricing problem precisely the way we did. Besides the above elementary purpose of a profitable column it is useful to consider its quality with respect to the overall performance of the column generation approach.

Theorem 4.3.1. *An optimum solution to the shortest path problem 4.3.5 gives an optimum solution for the linear program in 4.3.2 too.*

Proof. Firstly we prove any path from the source node to target node of 4.3.5 gives a feasible solution to the linear program in 4.3.2. It is easy to see that any path of the above type has at least one node from the set $N_t = \{n_{jt} : j \in \mathcal{J}\}$, for all t, since N_t forms a *node cut set* (A set of nodes of a graph which, if removed (or "cut"), disconnects the graph). By taking the assignment of the worker i to job j at time shift t, we get a feasible solution for the linear program. By the construction of the problem in 4.3.5, the weights on the edges in the graph depend on both the dual prices and number of changes, and a shortest path actually minimizes the

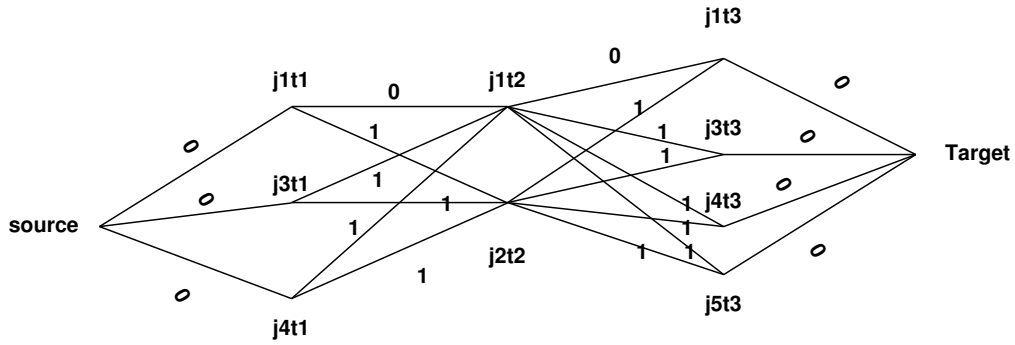

Figure 4.7: Shortest Path Problem, for worker i

changes in assignment considering the dual prices. By taking minimum over all i, it proceeds to an optimum solution of the linear program in 4.3.2. □

4.3.6 Initial basis of the restricted master program

When we use the simplex method to solve the restricted master program an initial basis is required. An initialization is possible by introduction of artificial variables, one for each constraint of the restricted master program. Usage of these variables is penalized via a large constant. The identity matrix corresponding to the artificial columns constitutes a feasible basis matrix. The penalty cost ensures that artificial variables are driven out of the basis. As soon as this happens, a feasible solution to the original is found assuming that a feasible solution exists. We limit the generation of columns that contribute to feasibility only, but whose cost may be large to be of any value in an optimum solution. An alternative initial basis may be produced by a primal heuristic, which is of course problem-specific. Here, in the case of sequential matching problem we could start with a feasible solution for extensive formulation which is obtained from the maximum matching problem of respective bipartite graphs. We used the maximum matching algorithm described

in Section 1.2.2. A greedy solution obtained by the method 3.3 is also tested because of the better initial starting point.

4.3.7 Integer solution

Linear programming-based branch and bound is a basic algorithmic technique very successful vert successful in practice as a method for solving mixed integer programs. Branch and bound is a divide and conquer approach trying to solve the original problem by splitting it into smaller problems, denoted as subproblems, for which upper and lower bounds are computed. The crucial part of a successful branch and bound algorithm is the computation of upper bounds for these subproblems. A solution of the relaxed problem gives a lower bound (minimization assumed) on the optimum objective function value of the problem it was derived from. The tighter the relaxation, the better this bound will be. But a relaxation is only useful if it can be treated at least practically efficiently by optimization algorithms. An idea for an exact algorithm to obtain integer solutions is linear programming relaxation at each node of the branch and bound tree via column generation. This method is known as the branch and price method.

We introduce some terminology concerning upper bounds (derived from solving relaxations) and lower bounds (obtained by finding feasible solutions). We call an upper bound local if it is only valid for a subproblem and global if it is a bound for the original problem. By solving a relaxation of the active problem, we obtain a local lower bound for the objective function value of the original problem. If the solution found for the relaxation happens to be feasible for the original problem (in which case it is also an optimum solution of the subproblem) and has greater objective function value than any feasible solution found so far, it is memorized and the global upper bound for the objective function value is increased accordingly.

A branch and bound algorithm maintains a list of subproblems of the original problem, which is initialized with the original problem itself. In each major iteration step the algorithm selects a subproblem from this list, computes a local upper bound for this subproblem, and tries to improve the global lower bound. If the local upper bound does not exceed the global lower bound, the active subproblem is fathomed, because its solution cannot be better than the best known feasible solution. Otherwise, we check if the optimum solution of the relaxation of the subproblem is a feasible solution of the original problem. In this case, we have solved the subproblem and thus, it is fathomed. If the local upper bound exceeds the global lower bound and no feasible solution was found for the active problem, we perform a branching step by splitting the active subproblem into a collection of new subproblems whose union of feasible solutions contains all feasible solutions

of the active subproblem. The simplest branching strategy consists of defining two new subproblems by changing the bounds of a variable.

Branching in a branch and bound environment pursues two aims: Detect integer solutions and provide good bounds so as to attest solution quality. A valid branching rule divides, desirably partitions, the solution space in such a way that the current fractional solution is excluded, integer solutions remain intact, and finiteness of the algorithm is ensured. Moreover, some general rules of thumb prove useful, such as producing branches of possibly equal size, sometimes referred to as balancing the tree. Also, important decisions should be made early in the tree. In addition, a compatible branching rule is one which prevents columns that have been branched on from being regenerated without a significant complication of the pricing problem. To achieve these we give a branching scheme, based on the following theorem. A first outline of the branch and price algorithm for SMP is given in the flowchart in Fig. 4.8.

4.3.8 Ryan and Foster branching scheme

The RyanFoster rule has been used in several papers solving crew scheduling and rostering problems, basically set partitioning type problems.

Theorem 4.3.2 (Branching Idea for Set Partitioning Problems, RYAN & FOSTER). *Given $A \in \{0,1\}^{m\times|Q'|}$ and a fractional basic solution to $A\boldsymbol{\lambda} = \mathbf{1}$* i.e.*, $\lambda \notin \{0,1\}^m$. Then $r, s \in \{1,\ldots,m\}$ exist such that*

$$0 < \sum_{q\in Q'} a_{rq}a_{sq}\lambda_q < 1. \tag{4.3.7}$$

When two such rows are identified, we obtain one branch in which these rows must be covered by the same column, *i.e.*,

$$\sum_{q\in Q'} a_{rq}a_{sq}\lambda_q = 1, \tag{4.3.8}$$

and one branch in which they must be covered by two distinct columns, *i.e.*,

$$\sum_{q\in Q'} a_{rq}a_{sq}\lambda_q = 0 \tag{4.3.9}$$

Note, that this information can be easily transferred to and obeyed by the pricing problem. The use of the above branching scheme changes the structure of the

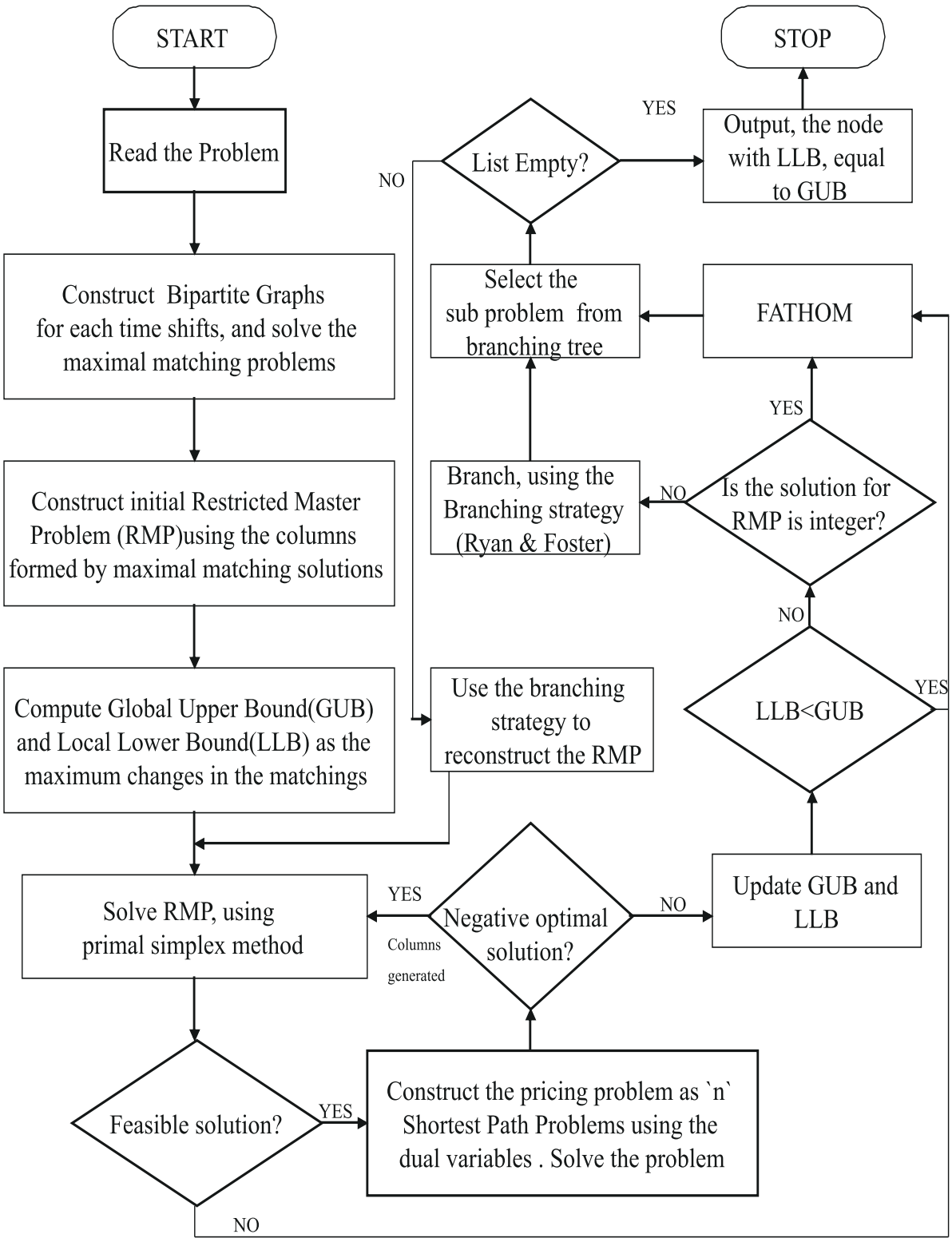

Figure 4.8: Flowchart of the branch and price algorithm for SMP

pricing problems of the corresponding subproblems. In the following section we discuss the nature of the pricing problem with two additional sets of constraints. On the left branch subproblems, the pricing problem changes to generating columns which cover the rows both r and s together. Let's say that it is equivalent to find a working schedule for worker i in such a way that both the jobs j' at t', and j'' at t'' are assigned to i. On the right branch to make these rows r and s are covered by distinct columns, we have to find a worker schedule such the either j' at t' or j'' at t'' is assigned to worker i, but not both.

We can formulate the pricing problem more precisely using the above restriction. There are 4 types of pricing problems for the subproblems of the corresponding branching nodes.

§1 A shortest path problem, as explained in Section 4.2.12, without any additional constraints.

§2 A constrained shortest path problem on a graph (S, E) with node set S, and edge set E, and has basic structure as in 4.2.12, with additional constraints. Let $L \subseteq S \times S$, and for each $(r, s) \in L$, problem is extended to find a shortest path covering either both the nodes r and s or none of them.

§3 A constrained shortest path problem on a graph (S, E) with node set S, and edge set E, and has basic structure as in 4.2.12, with additional constraints. Let $R \subseteq S \times S$, and for each $(r, s) \in R$, problem is extended to find a shortest path covering either r or s, but not both.

§4 A constrained shortest path problem on a graph (S, E) with node set S, and edge set E, and has basic structure as in 4.2.12, and having additional constraints of the type both as in 2 and 3 . *i.e.* $R, L \subseteq S \times S$, and for all $(r, s) \in R$ and $(r', s') \in L$ and find a shortest path covering either r or s, but not both, and covering both r, s or not both.

4.4 Solution methods for pricing problems

Traditional algorithm like *Dijkstra* [1] work for the shortest path problem without additional constraints, even if the weights are nonnegative since the graph is acyclic. But the so called *reaching algorithm* can solve the shortest path problem on an m-edge graph in steps of $\mathcal{O}(m)$ for an acyclic digraph. This algorithm allows paths such that edges traversed in the direction opposite to their orientation have a

negative length. No other algorithm can have better complexity because any other algorithm would have to at least examine every edge, which would itself take steps $\mathcal{O}(m)$.

Definition 4.4.1. Let us label the nodes of a graph $G = (V, E)$ by distinct numbers from 1 through n and represent them by an array *order* (*i.e.*, order(i) gives the label of node i). We say that this labelling is topological ordering of nodes if every arc joins a lower labeled node to a higher labeled node. (i.e.), for every arc $(i, j) \in E$, order(i)<order(j).

By relaxing the edges of a weighed directed acyclic graph according to a topological sort of vertices, we can compute shortest paths from single source in $\mathcal{O}(m)$ time. The algorithm starts by topological sorting of the directed acyclic graphs. If there is a path from vertex u to vertex v, then u precedes v in the topological sort. We make just one pass over the vertices in the topologically sorted order. As each vertex is processed all the edges that leave the vertex are relaxed, because of the special structure of the directed acyclic graph in Section 4.3.5 the topological ordering can be done in quite an easy way, numbering continuously starting from source node, the nodes jt, such that $j \in$ sorted($\mathcal{J}$), $t \in$ sorted(τ). It results in a topological ordering since there is no edge between the nodes of the type $j't$ and $j''t$, and from jt to $j't'$ for $t > t'$. In the following algorithm $d(p)$ denotes the distance from the source node to node p, $d(p, q)$ denotes the distance from node p to q.

Algorithm: Shortest Path 1 Topologically sort the vertices of G
2 Initialize source s, $d(s) = 0$, for all nodes p, $d(p) = M$ where M is large
3 **for** each vertex u taken in topologically sorted order
4 **do for**each vertex $v \in Adj[u]$
5 **do** $d(u) = min(d(u), d(v) + c_{ij})$

The adjacent nodes of n_{jt} are of the form n_{jt+1}. Because of this special structure of the problem we could simplify the above algorithm as in **Algorithm** 4. But the pricing problem on the branching nodes become more complicated with the additional constraints. An exact algorithm for these problems looks difficult. We next analyze the complexity of this constrained pricing problem.

Theorem 4.4.1. *The constraint shortest path problem in §2 is $\mathcal{NP}$-complete.*

Proof. We reduce 3-SAT to a shortest path problem with additional constraints as in §2 (selection of exclusive nodes from pairs). Given an instance ϕ of 3-SAT, we

Algorithm 3 Solving Shortest Path Problem

```
d(s) ← 0
for t ∈ τ do
  for j ∈ 𝒥 do
    if n_jt Exist then
      if t > 0 then
        d(n_jt) ← M
      else
        d(n_jt) ← 0
      end if
    end if
  end for
end for
for t ∈ τ do
  for j ∈ 𝒥 do
    if n_jt Exist and t not equal τ then
      u = n_jt
      for j' ∈ 𝒥 do
        if n_j't+1 Exist then
          if d(n_j't+1) > d(u) + d(u, n_j't) then
            n_j't ← n_j'
          end if
        end if
      end for
    end if
  end for
end for
```

first construct a network which has a constrained shortest path if and only if ϕ is satisfiable.

We are given an instance ϕ of 3-SAT with n clauses $C_1, C_2, \ldots, C_n$, with p boolean variables $\alpha_1, \alpha_2, \ldots, \alpha_p$, such that each clause being $C_i = (\alpha_{ik_1}, \alpha_{ik_2}, \alpha_{ik_3})$, with the α_{ik}'s being either boolean variables or negations thereof. Now constructing the node set N of the graph as:

- Add a source node v_s and a target node v_t.

- For each variable α_k adding three nodes $v^T_{\alpha_k}$, $v^F_{\alpha_k}$ and, v_{α_k}.

- For each literal $v_{\alpha_{ik}}$ of C_i in ϕ add the node $v_{\alpha_{ik}}$.

- Add additional nodes as $v_{C_1}, v_{C_2}, \ldots, v_{C_{n-1}}$ for each clause $C_1, C_2, \ldots, C_{n-1}$.

The edge set E is defined as follows

- The source node is connected to both the node $v^T_{\alpha_1}$ and $v^F_{\alpha_1}$.

- For each k from 1 to p, add the edges connecting $v^T_{\alpha_k}$ and $v^F_{\alpha_k}$ to v_{α_k}, and for each k from 1 to $p-1$ add the edges connecting from v_{α_k} to $v^T_{\alpha_{k+1}}$ and $v^F_{\alpha_{k+1}}$.

- Edges connecting v_{α_p} to the nodes correspond to the literals $v_{\alpha_{11}}$, $v_{\alpha_{12}}$ and $v_{\alpha_{13}}$of C_1.

- For each clause C_i for $i = 1$ to $n-1$, edges connecting the nodes correspond to the literals $v_{\alpha_{i1}}$, $v_{\alpha_{i2}}$ and $v_{\alpha_{i3}}$ are to v_{C_i}, and for each variable index k from 1 to $p-1$ add edges connecting from $v^T_{\alpha_k+1}$ to v_{α_k}.

- Connect the nodes $v_{\alpha_{n1}}$, $v_{\alpha_{n2}}$ and $v_{\alpha_{n3}}$ to the target node v_t.

Now $L \in N \times N$ is defined as follows. For each variable α_k, the pair $(v^T_{\alpha_k}, v_{\alpha_k})$is in L if α_{ik} is a negation of the variable α_k in C_i, else the pair $(v^F_{\alpha_k}, v_{\alpha_k})$ is in L.

An example for four variables is depicted in Fig. 4.9. Say $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ be the variables. An instance $\Phi = C_1 \wedge C_2 \wedge C_3$ where,

$$C_1 = (\alpha_1 \vee \neg\alpha_2 \vee \neg\alpha_3) \tag{4.4.1}$$
$$C_2 = (\neg\alpha_1 \vee \alpha_2 \vee \neg\alpha_4) \tag{4.4.2}$$
$$C_3 = (\neg\alpha_2 \vee \neg\alpha_3 \vee \alpha_4) \tag{4.4.3}$$

The graph deduced from Φ has nodes $v^T_{\alpha_k}$, $v^F_{\alpha_k}$ v_{α_k} for $k = 1$ to 4, and $v_{\alpha_{ik}}$, v_{C_i} for $i = 1$ to 3. The constraint set $L = \{(v^T_{\alpha_1}, v_{\alpha_{21}}), (v^T_{\alpha_2}, v_{\alpha_{12}}), (v^T_{\alpha_2}, v_{\alpha_{32}}), (v^T_{\alpha_3}, v_{\alpha_{13}}),$ $(v^T_{\alpha_3}, v_{\alpha_{33}}), (v^T_{\alpha_4}, v_{\alpha_{24}}), (v^F_{\alpha_1}, v_{\alpha_{11}}), (v^F_{\alpha_2}, v_{\alpha_{22}}), (v^F_{\alpha_4}, v_{\alpha_{34}})\}$

We prove that a shortest path P exists from v_s to v_t, and only if there is a truth assignment for a 3-SAT instance.

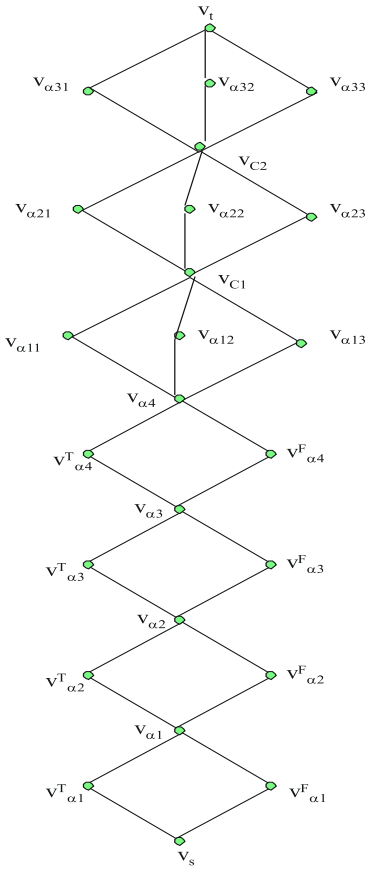

Figure 4.9: The shortest path problem on graph which deduced from Φ

Firstly, if there is a truth assignment, we prove there is a shortest path of length $2(p+n)$ and satisfying the constraints. Suppose there is truth assignment to the variables which satisfies all of the clauses. So each of the clause C_i has at least one literal α_{ik} with true value. Let's say α_k is the variable corresponding to the true literal in C_i (*i.e.* $\alpha_{ij'}$ is either or $\neg\alpha'_j$). Now describe a path P from v_s to v_t as follows.

$$P = (v_s, v^X_{\alpha_1}, v_{\alpha_1}, v^X_{\alpha_2} v_{\alpha_2}, \ldots, v^X_{\alpha_p}, v_{\alpha_p}, v_{\alpha_{1k}}, v_{C_1}, v_{\alpha_{1k}}, v_{C_1}, \ldots, v_{\alpha_{nk}}, v_t)$$

where

$$v^X_{\alpha_k} = \begin{cases} v^T_{\alpha_k} & \text{the variable } \alpha_k \text{ is } \textit{true} \text{ in the truth assignment,} \\ v^F_{\alpha_k} & \text{otherwise.} \end{cases} \tag{4.4.4}$$

It is easy to claim that P is a shortest path, since every shortest path from v_s to v_t has the length of $2(p+n)$, which is equal to the length of P. The path P also

satisfies the condition that it passes through only one of those nodes in the pairs of the constraint set, *i.e.* if the pair $(v^T_{\alpha_k}, v_{\alpha_{ik}}) \in L$, then both the nodes can't be in the shortest path. This is true because if the node $v^T_{\alpha_k}$ is in shortest path, which happens only if the variable α_k is *true* (by 4.4.5) and so $\alpha_{ki} = \neg\alpha_k$ is *false*. This implies that the variable $v_{\alpha_{ki}}$ cannot be in the shortest path. The same is true for the pairs of the type $(v^F_{\alpha_k}, v_{\alpha_{ki}}) \in L$.

Conversely assuming that a shortest path P exists which satisfies the constraints. By taking a truth assignment as follows,

$$\alpha_k = \begin{cases} false & \text{if the path } P \text{ passes through } v^T_{\alpha_k}, \\ true & \text{if the path } P \text{ passes through } v^F_{\alpha_k},. \end{cases} \tag{4.4.5}$$

We claim that the above truth assignment satisfies the 3-SAT. Since the path has length of $2(p+n)$ it has exactly one node from $v_{\alpha_{ik}}$ for each i. The literal corresponds to the node α_{ik} (either α_k or $\neg\alpha_k$) satisfies the truth assignment, since if $\alpha_{ik} = \alpha_k$ then α_k has to be true in the truth assignment. Since $v_{\alpha_{ik}}$ is in the shortest path P, the node $v^F_{\alpha_k}$ cannot be in P which implies α_k is *true*. □

Since the complexity of the *pricing problem* turns out to be $\mathcal{NP}$-complete, we look for new branching rules. Branching on individual variables is attractive since its implementation is quite simple within the bounded-variable simplex algorithm that is normally embedded in a MIP solver. However this type of branching can create an embedded enumeration tree which can lead to excessive enumeration. We propose another branching technique that has the potential to yield significant computational improvements for the *pricing problem*.

This new branching rule can be deduced for the branch and price scheme, using the following observation.

Lemma 4.4.2. *If X^* is a fractional solution for the LP in column generation form then*

either,

1. Two columns, say k_1 and k_2, exist which represent the different work schedules for the same worker, and having a shared job J^1_t and one different job J^2_{t+1}. (In another words, for a fractional solution, there exist two paths in corresponding layered network formulation of the pricing problem, which shares a node and one different one, which are in adjacent layers)

or,

2. Two workers, say W_1 and W_2, exist, of different types and having a shared job in a fixed time interval, and each of them has at least one different job.

Proof. Argument 1 is true since x_{k_1} is fractional, at least one more fractional x_{k_2} should exist because of partitioning constraint 4.2.12. Both of them represent different work schedules for the same worker. The columns for the fractional x_{k_1} and x_{k_2} represent two paths in the graph in *pricing problem*. In this case if the paths share a node then condition 1 holds. If neither of the two paths share a node, *i.e.* if they are disjoint, the condition 2 follows. Since problems are of the set partitioning type, a job exists at a particular time shift shared by two different workers, and at least one different job. □

From the above result we can deduce the branching scheme as follows.

If condition 1 is satisfied, it is easy to find the one node which is shared by the paths and not sharing

- (1L) On the left branch (just like in Ryan and Foster), generate columns which contain both the nodes of the type in 1.
- (1R) But on the right branch, generate columns with only one node among them

If condition 2 is satisfied, the branching strategy changes slightly. Let's say the workers W_1 and W_2 share a job J_t

- (2L) On the left branch generate columns with W_2 not assigned to J_t
- (2R) On the right branch W_1 is not assigned to j_t.

The proposed branching technique has the potential to yield significant computational improvement if better approximation algorithm can be developed for the pricing problem on the branching nodes. Since some of the pricing problems on the nodes of branching tree are still $\mathcal{NP}$-complete. Branching on individual variables is quite simple to implement, however this type of branching technique can lead to excessive enumeration because of the unbalanced enumeration tree.

4.5 Conclusion

In this chapter we proposed a branch and price procedure, a column generation scheme with branch and bound for solving the sequential matching problem. We have also discussed different MIP formulations for the sequential matching problem $S(G_c, \tau)$. The formulation with a huge number of variables is used to develop a column generation scheme. The pricing problem for the general case can be formulated as a shortest path problem in a layered network and solved by an algorithm of $\mathcal{O}(|E|)$. The special branching rules lead to $\mathcal{NP}$-complete pricing problems.

Chapter 5

Implementing branch and price method

5.1 Introduction

In this chapter we describe in detail the implementation of our branch and price approach to the sequential matching problem, making a few general remarks on implementing column generation codes as we proceed. To begin with we briefly explain the details of the implementation and consequently we analyze results of the column generation implementation. We should emphasize that the code described here is a research prototype, and not a production implementation. It is first and foremost intended for ease of testing and evaluating the potential of our approach. All experiments were performed on a 333 MHz Pentium II PC with 128 MB core memory running Windows 2000. We used the **CPLEX 6.5** and **SOPLEX 1.0**, the callable libraries to solve the linear and integer programs. For ease of implementing the branch and price scheme we used **ABACUS A B**ranch **A**nd **CU**t System, a column generation framework. Compilation with the **Visual C++ 6.0**, is invoked "*Maximum Speed*" optimization. The program outputs different assignments for workers to workplaces in each time shift. We have tested the code with the randomly generated data, as explained in Section 5.3.12. Firstly we describe the design and architecture of ABACUS as in its Reference Manual [19]. Next the implementation details of the branch and price approach for the sequential matching problem, and the numerical results are presented in the following section.

5.2 ABACUS A Branch And Cut System[1]

In this section we explain briefly the architecture of ABACUS, and follow with the details of the implementation and computational results. ABACUS - A Branch And Cut, is a framework for the implementation of branch and bound algorithms using linear programming relaxations that can be complemented with the dynamic generation of columns or cutting planes (linear programming based branch and bound algorithm). This system allows us to concentrate merely on the problem-specific parts such as column generation, the cutting plane (for branch and cut) and the different heuristic needed for column generation. Moreover, ABACUS provides a variety of general algorithmic concepts, e.g., enumeration and branching strategies, from which we can choose the best alternative for the application. Finally, ABACUS provides many basic data structures and useful tools for the implementation of such algorithms. ABACUS has been designed in such a way that we can use it both for general mixed integer optimization problems and for combinatorial optimization problems. It unifies cutting plane and column generation within one algorithm framework. Briefly ABACUS is a collection of abstract data structures and algorithms which are met by object oriented programming as a collection of C++ classes.

From the point of view of a user wishing to implement a linear programming-based branch and bound-based algorithm, ABACUS provides a small system of base classes from which the application specific classes can be derived. In virtual functions ABACUS provides default implementations, which are redefined if required, e.g., the branching strategy. An application based on ABACUS can be refined step by step. Then this branch and bound algorithm can be enhanced by the dynamic generation variables, primal heuristics, or the implementation of new branching or enumeration strategies. Default strategies are available for numerous parts of the branch and bound algorithm, which can be controlled via a parameter file. If none of the system strategies meets the requirements of the application, the default strategy can simply be replaced by the redefinition of a virtual function in a derived class.

The inheritance graph of ABACUS has been designed as a tree with a single exception where it uses multiple inheritance. The following sections give a survey of the the most important application base classes of ABACUS from a column generation point of view. From these different classes we have derived the classes for the

[1]The contents of this section is extracted from the Reference Manual of ABACUS version 2.0. [19], [20]

column generation implementation of SMP. Other important type of classes are in the group of *Pure Kernel* which is usually invisible to users, deals mainly with branch and bound algorithm, the solution of linear programs, and the management of constraints and variables. This group covers classes that are required for the implementation of the kernel of ABACUS but are usually of no direct importance for the user of the framework. There are other classes providing basic data structures and tools which can optionally be used for the implementation of an application.

The following application base classes are mainly involved in the derivation process for the implementation for the SMP.

5.2.1 The Master

The class `ABA_MASTER` is one of the central classes of the framework. It controls the optimization process and stores global data structures for the optimization. For each new application a class has to be derived from the class `ABA_MASTER`. The class `ABA_MASTER` also provides default implementations of pools for the storage of constraints and variables. Some virtual functions are also defined in this class with different enumeration strategies which are required in a branch and bound framework.

5.2.2 The Subproblem

The class `ABA_SUB` represents a subproblem of the implicit enumeration, *i.e.*, a node of the branch and bound tree. The subproblem optimization is performed by the solution of linear programming relaxations. Usually, most run time is spent within the member functions of this class. Also, from the class SUB a new class has to be derived for each new application. By redefining virtual functions in the derived class problem specific algorithms such as cutting plane or column generation can be embedded.

5.2.3 The Constraints and Variables

ABACUS provides some default concepts for the representation of constraints and variables. However, it might still be necessary, for a new application, for special

classes to have to be derived from the classes `ABA_CONSTRAINT` and `ABA_VARIABLE`, which then implement application-specific methods and storage formats.

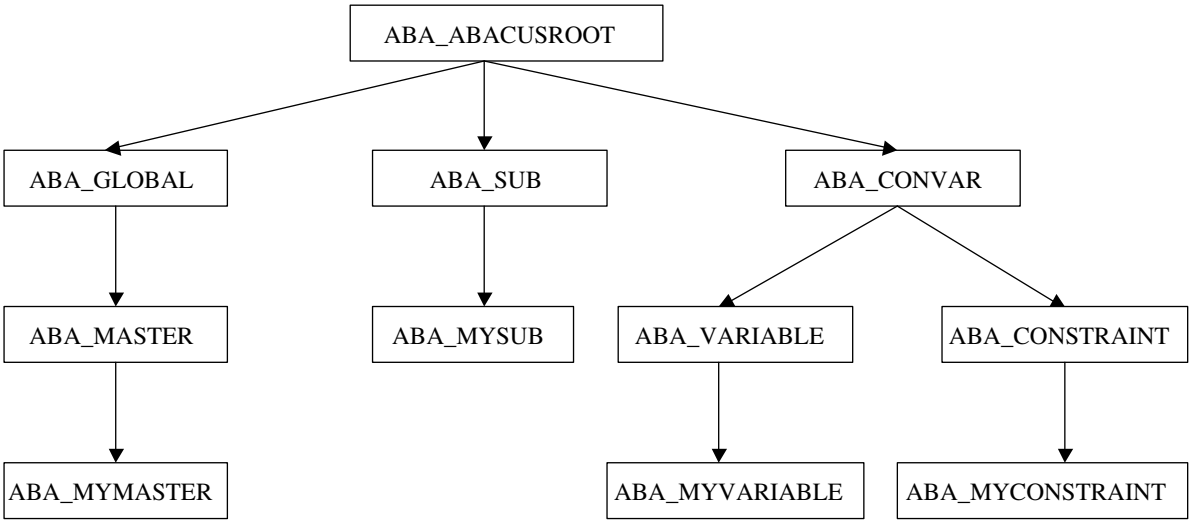

Figure 5.1: Problem specific classes in Abacus

5.3 Sequential Matching Problem: Implementation details

In this section we describe the details of the implementation of the branch and price approach to the Sequential Matching Problem (SMP), which uses ABACUS. In order to allow for easy testing of our code, we generated a file input/output system. Outside the branch and price scheme, the whole project includes codes including for the random generation of problem and the graph algorithms which are needed for some cases of pricing problem solution. Most options are introduced throughout the chapter, otherwise they are self-explanatory.

5.3.1 Restricted Master Problem

We introduced in Chapter 4 a column generation formulation of the sequential matching problem, with variables as a possible *lifeline of a worker*. The objective function we actually implemented in our column generation code is minimization

of the total changes in jobs. *i.e.*, for this particular objective, the cost coefficients range in integers for some practical instances, because it is the number of changes in jobs during a schedule. Note also, that the cost can simply be calculated by straightforward addition. Section 4.3.2 in Chapter 4 outlines the master iteration of our branch and price model which we present in more detail below. We always initialize the restricted master program with an all artificial basis as a solution of the initial matching problem and an identity matrix. This is the default initial solution. This can be changed by redefining the program settings. The artificial variable penalty M can be modified via $-M$. In the early stages of the algorithm, this penalty cost strongly influences the dual variable values. Therefore, this value should be carefully chosen. In our experience, very large penalties amplify the aforementioned effect; too small penalties fail to penalize sufficiently.

5.3.2 Subproblem Solution and Column Management

The pricing subproblem is the most frequently executed essential component of a column generation code. Each call should therefore be as effective as possible, or in other words, the computation timeinvested should pay off to the greatest possible extent. As explained as in Chapter 4 the pricing problem on the root node is a shortest path problem. But the problem becomes more complicated with additional branching strategy which is introduced in Chapter 4. This becomes clear when we recall that it is neither mandatory to add a most profitable column to the restricted master program nor are we restricted to adding only one column at a time, while this is precisely what happens if the pricing problem is solved exactly, in alternation with the re-optimization of the restricted master.

5.3.3 Column Pool

We manage our column pool and subproblem optimization in standard ways. The generated columns with negative cost which are not added to the restricted master program are not simply rejected, but are stored in a *column pool*, if the reduced cost is significantly smaller. The size of the column pool can set to a predefined value. This concept can be implemented using ABACUS. Before using other methods to find columns to be entered we check whether the pool contains columns with significantly lower value of reduced cost. These may become active in later iterations. However, such columns are only added when in the same iteration at least one negative reduced cost column is added from the column pool, since cycling may

occur. We try to keep a pool of high quality. When no columns are delivered from the pool we use the constraint shortest path algorithms to generate columns.

5.3.4 Implementation

In this section we discuss the details of the class structure and inheritance for Sequential Matching Problem implementations, which are derived mostly from ABACUS classes. We discus only the main important problem-specific classes `SMP`, `SUBSMP`, `SMPVAR`, `SETPARCON`, `SMPINSTANCE`, `BRANCHRULE_RF`, `BRANCHRULE_SMP` and `SPP`.

5.3.5 The class for master problem: `SMP`

The class `SMP` is the central class of the implementation. It is derived from the class `ABA_MASTER`. This class mainly deals with problem-specific variables and functions for the master problem in the branch and price algorithm.

```
class SMP:public ABA_MASTER{
    public:
        SMP(const char* problemName);
        virtual~SMP();
        virtual ABA_SUB* firstSub();
        virtual void output();
        virtual void initializeParameters();
        virtual int enumerationStrategy(ABA_SUB*s1,ABA_SUB*s2);

        int nJobs()const;
        int nWorkers()const;
        int nTimeShifts()const;
        int nAbleJobs(int worker, int shift)const;
        int jobAtT(int worker, int shift, int jobindex)const;
        int nMaxRows()const;
        int nMaxConstraints;
        int  nChanges(int columnNumber);
        int initMatch(int worker, int shift);

        double firstMatchings(ABA_BUFFER<ABA_BUFFER<int>*> &matchings);
        void updateBestSolution(double value,
```

```
                                        ABA_BUFFER<ABA_COLUMN*>&columns);
        void genIntegerProgram(const char*fileName);
        bool checkFarleysCriterion()const;
        bool poolPricing()const;
        bool stopAfterPoolPricing()const;
        void countShortestPathProblem();
        void newIntegerProgramCols(int nNewCols);
        void newPoolPriceCols(int nNewCols);
        void newSPPCols(int nNewCols);
        void DisplayProblem();
private:
        virtual void initializeOptimization();
        int leftFirstSearch(ABA_SUB* s1,ABA_SUB* s2);
        SMPINSTANCE* instance_;
        ABA_BUFFER<ABA_COLUMN*>*bestColumns_;
        bool showSolution_;
        bool genIP_;
        bool checkFarleysCriterion_;
        bool poolPricing_;
        bool stopAfterPoolPricing_;
        int nShortestPathProblem_;
        int nSPPCols_;
        int nPoolPriceCols_;
        SMP(const SMP&rhs);
        const SMP&operator= (const SMP&rhs);
};
```

The class `SMP` derived from the class `ABA_MASTER` has a constructor such as

`SMP::SMP(const char* problemName):ABA_MASTER(problemName,false,true, ABA_OPTSENSE::Min)` {...}, where it initializes `SMP` as column generation, and problem data is read. The arguments are explained as follows.

`problemName`, the name of the problem being solved.

`cutting` is false, since no cutting planes are generated.

`pricing` is true, since inactive variables are priced in, and the function `ABA_SUB::pricing` is redefined.

`optSense`, the sense of the optimization is `ABA_OPTSENSE::Min`.

In order to initialize the upper bound we apply the `firstMatchings(...)` a matching algorithm for each graph and the solution as initial feasible solution to RMP. The variables of this solution are enlarged by the variables induced by the identity matrix, to make the linear programs feasible, even after some branching steps. Otherwise, the pricing problem could not be solved. With this variable set the variable pool and the first subproblem are initialized. The constraints are given of the type of set partitioning. Since the method is branch and price, with column generation, we do not require a cut pool, provided by ABACUS. As expected, our computational experience shows that the pricing problem can be solved faster in subproblems that do not have any side constraints as explained in Chapter 4.

5.3.6 The class for subproblem: `SUBSMP`

The class `SUBSMP` is managing the subproblem in the column generation method. It is derived from the class `ABA_SUB`. It mainly deals with subproblem-specific functions and variables. New subproblems are generated according to the branching. We embedded the branching rules within the mixed integer programming, but the corresponding linear programming solved by CPLEX and SOPLEX. The current implementation of the code first carries out a Ryan and Foster branching scheme, the default branching rule. The SMP branching rule is also implemented, and the corresponding pricing problem solution technique implemented.

```
class SUBSMP:public ABA_SUB {
    public:
        SUBSMP(ABA_MASTER* master,ABA_SUB* father,
                                        ABA_BRANCHRULE* branchRule);
        SUBSMP(ABA_MASTER* master);
        virtual~SUBSMP();
        virtual bool feasible();
        virtual ABA_SUB *generateSon(ABA_BRANCHRULE* rule);
        virtual int improve(double& primalValue);
        virtual int pricing();
        SMP* smp();
        int branchingConstraints(int&con0,int&origCon0,
                                            int&con1,int&origCon1);
        int addToPoolAndBuffer(ABA_BUFFER<SMPVAR*>&newCols);
        void printMatrix(ostream&out);
        int tailSosCon(int i)const;
        int headSosCon(int i)const;
        int nSosCon()const;
```

```
        BRANCHRULE\_RF* branchRule();
        ABA_BUFFER<int> *remove;
    private:
        virtual void activate();
        virtual void deactivate();
        virtual ABA_LP::METHOD chooseLpMethod(int nVarRemoved,int nConRemoved,
        int nVarAdded,int nConAdded);
        virtual initMakeFeas(ABA_BUFFER<ABA_INFEASCON*>&infeasCons,
                        ABA_BUFFER<ABA_VARIABLE*>&newVars,
                        ABA_POOL<ABA_VARIABLE,ABA_CONSTRAINT>**pool);
        virtual int generateBranchRules(ABA_BUFFER<ABA_BRANCHRULE*>&rules);
        ABA_SET* conSets_;
        ABA_ARRAY<bool> *removedCon_;
        int nRemovedCon_;
        ABA_ARRAY<int> *tailSosCon_;
        ABA_ARRAY<int> *headSosCon_;
        int nSosCon_;
        ABA_ARRAY<bool>* inSosCon_;
        SUBSMP(const SUBSMP&rhs);
         const SUBSMP&operator= (const SUBSMP&rhs);
};
```

The stopping criterion for each subproblem is when there is no new column generated for the subproblems.

5.3.7 The class for variables: `SMPVAR`

The variables are represented by the class `SMPVAR` which is derived from the ABACUS class `COLVAR`, to redefine some virtual functions. A variable of the SMP is valid if it satisfies all branching rules. Therefore, we redefine the function `valid( )`. We also redefine the virtual function `redCost( )` for efficiency reasons.

```
class SMPVAR:public ABA_COLVAR { public:
    SMPVAR(ABA_MASTER* master,bool dynamic,double obj,
                int nnz,ABA_ARRAY<int>&support,ABA_ARRAY<double>&coeff);
    SMPVAR(ABA_MASTER*master,bool dynamic,double obj,ABA_SPARVEC&vector);
    ......
};
```

5.3.8 The class for constraints: SETPARCON

The class SETPARCON is for the representation of these set partitioning constraints from the class NUMCON. The constraint matrix for the sequential matching problem is defined through the variables. So each constraint has a unique representation given by its number.

```
class SETPARCON:public ABA_NUMCON {
    SETPARCON(ABA_MASTER*master,int id);
    ......
};
```

Other than special classes for the master, the subproblem, the variables, and the constraints, we implemented problem-specific classes for the representation of an instance of the sequential matching problem, for the solution of the constrained shortest path problem in the pricing phase, and for the different branching rules which we discussed as in Chapter 4, including Ryan and Foster.

5.3.9 The class for problem instance: SMPINSTANCE

SMPINSTANCE is the class to manage an instance of the problem. It is derived from the class ABA_ABACUSROOT. It mainly deals with the variables regarding an instance of sequential matching problem. The class SMPINSTANCE has a constructor with a filename as argument, from which the problem data is read.

```
class SMPINSTANCE:public
ABA_ABACUSROOT { public:
        SMPINSTANCE(ABA_MASTER* master,const char* fileName);
        .....
};
```

5.3.10 The branching rules: The classes BRANCHRULE_RF and BRANCHRULE_SMP

We have implemented two kinds of branching rule as explained in Chapter 4. The first one is, as described, the branching rule of Ryan and Foster for set partitioning

problems. We derived the class BRANCHRULE_RF from BRANCHRULE, an abstract base class for all branching rules within the ABACUS framework. The most important member function of this class is the function extract(), which defines a pure virtual function of the base class BRANCHRULE. This function modifies a subproblem according to the branching rule. It removes invalid variables and removes the redundant constraint in the left subproblem. In the right subproblem the respective constraint is added to the pricing problem.

The second branching rule, described is BRANCHRULE_SMP is also derived from the abstract base class BRANCHRULE.

5.3.11 The pricing problems: SPP

The root node pricing problem of the sequential matching problem is a *layered shortest path problem* that can have additional side constraints. The class SPP provides data structures and solution methods for the *layered shortest path problem* that can have additional constraints derived from the different branching strategy. This class contains different algorithms for the pricing problems in left and right branches, by applying the different branching strategy. As per in Chapter 4 while using Ryan and Foster method, the pricing problems on the branches turns up to $\mathcal{NP}$-Complete problems of finding shortest paths with restriction on the pairs of nodes. We attempted to solve these problems using the heuristic algorithms and found computationally inefficient problems become complex when going down through the branching tree.

But when the new branching strategy as per Section 4.3.7 of Chapter 4 is used the problem turns out to be less complicated. In this case the class SPP contain an algorithm for solving this kind of special problems.

The Fig. 5.2 shows the important class inheritance of SMP.

We have many implementation specific classes, but a detailed description here is superfluous.

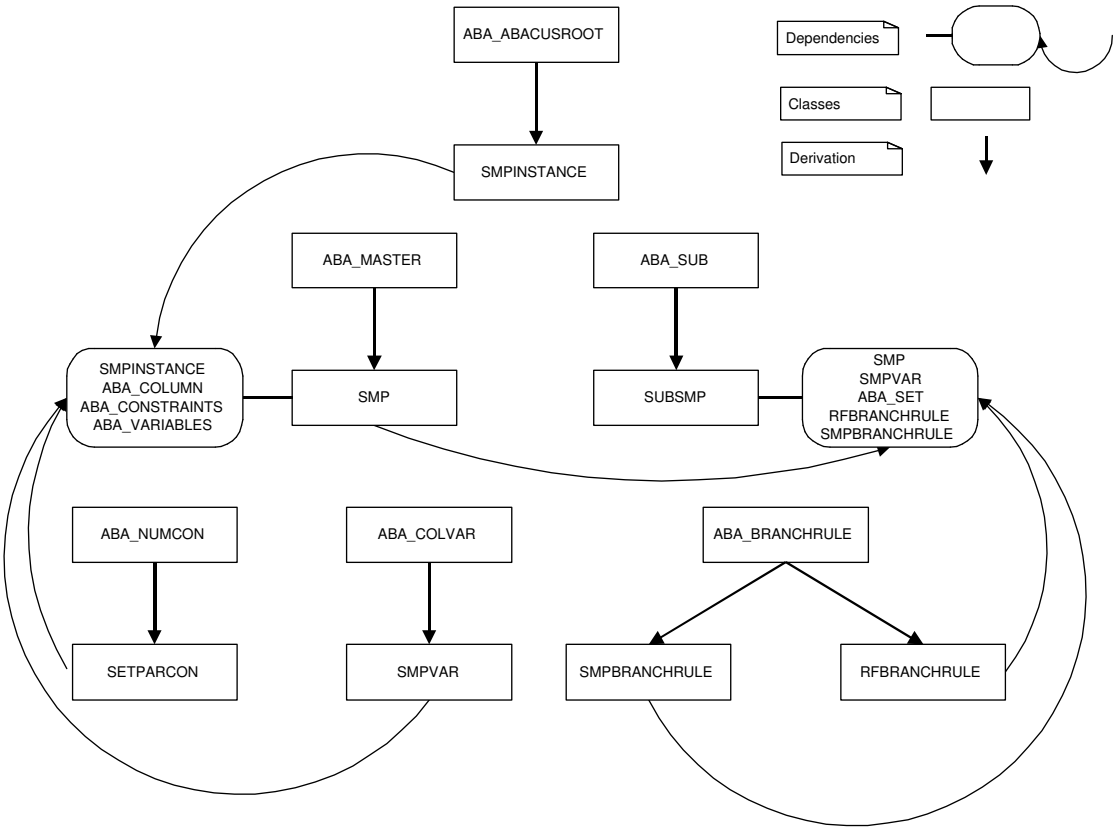

Figure 5.2: Class structures, derivation and dependencies in `SMP`

5.3.12 Problem generation

Let us conclude this section with a few methods to generate random problems by using the random variables from common distributions. We use a standard method of generating random variables on a distribution. Usually there is more than one method to generate a pre-specified random variant. The relative merits of different generation methods are compared based on their accuracy (theoretical and numerical), execution speed, ease of implementation (coding effort and subroutine support), portability, memory requirement, and interaction with variance reduction techniques. Some random problems are generated using the following assumptions.

- The discrete time intervals are uniform during a day.
- The number of workers and jobs are fixed.

- The number of jobs allowed/able to be done by a worker on a time shift is normally distributed with a predefined mean and deviation.
- The jobs for a worker at a time shift are randomly selected (uniform distributed, without repetition).

A random, problem-generating algorithm is implemented separately, which can output a problem instance with specific input parameters. For a given number of workers and jobs, and assuming normally distributed jobs with mean μ and standard deviation σ.

To generate a uniformly distributed random number between any two integers `lownumber` and `upnumber`, we implemented a small routine `SMPRANDOM::urandom(int lownumber, int upnumber)`, which basically uses the `(int)rand()`, a pseudo random number generating function (between `0` and `RAND_MAX`) provided by VC++ 6.0. To generate a non-repeating random sequence we provid a subroutine `void SMPRANDOM::randomseq(int lownumber, int endnumber, int n, int* select)`. The idea of the algorithm is to generate indices from 1 to n of an array A randomly, using `SMPRANDOM::urandom(1, n)`, and output $A[i]$. We iterate the process with $A[i] \leftarrow A[n]$, $n \leftarrow n-1$ until all the numbers are generated.

The function `int SMPRANDOM::nrandom(int max_jobs, float mu, float sigma)` generates an identically distributed normal random numbers with mean `mu` and variance `sigma`, and the maximum number of `max_jobs`. The algorithm is based on *polar method to generate two identically-distributed normal random variables*, which we briefly describe below.

$N(0,1)$ - Normal distribution with mean 0 and standard deviation 1.

$U(0,1)$ - Uniform distribution from 0 to 1.

Algorithm 4 Generating two $N(0,1)$ random variables

STEP 1: Generate X_1 and X_2 two independently and uniformly-distributed random numbers from $U(0,1)$.
STEP 2: Let $Y_1 = 2X_1 - 1$, $Y_2 = 2X_2 - 1$, $S = Y_1^2 + Y_2^2$
STEP 3: If $S > 1$, return to **STEP 1**
STEP 4: Generate 2 independent standard normal random numbers

$$Z_1 = \sqrt{\frac{-2\log(S)}{S}}Y_1 \qquad Z_2 = \sqrt{\frac{-2\log(S)}{S}}Y_2$$

This algorithm does indeed produce two independently-distributed $N(0,1)$ random variables Z_1 and Z_2. The polar coordinates (R, θ) of (Y_1, Y_2) such that $R = \sqrt{S}$ is uniformly distributed in $U(0,1)$ and $\theta = \tan^{-1}(V2/V1)$ is uniformly distributed in $U(0, 2\pi)$. By Box-Muller transformations,

$$X = \sqrt{-2log(X_1)}\cos(2\pi X_2) \tag{5.3.0}$$

and

$$Y = \sqrt{-2log(X_1)}\sin(2\pi X_2) \tag{5.3.0}$$

are independently and normal distributed random variables for X_1, X_2 are independently distributed in $U(0,1)$, since

$$f(x,y) = \frac{1}{\sqrt{2\pi}}\exp(-\frac{x^2}{2})\frac{1}{\sqrt{2\pi}}\exp(-\frac{y^2}{2}) \tag{5.3.0}$$

such that

$$\begin{aligned}\mathbf{P}(X \le x_1, Y \le y_1) &= \frac{1}{2\pi}\int_{-\infty}^{x_1}\int_{-\infty}^{y_1}\exp(-\frac{x^2+y^2}{2})dxdy \\ &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x_1}\exp(-\frac{x^2}{2})dx\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{y_1}\exp(-\frac{y^2}{2})dy\end{aligned}$$

For proof see Knuth [21].

Since R is independent of θ, by substituting back into the Box-Muller transformation, we get,

$$X = \sqrt{-2\log(U)}\frac{Y_1}{\sqrt{R}} = Y_1\sqrt{\frac{-2log(R)}{R}} \tag{5.3.-2}$$

and

$$Y = \sqrt{-2\log(U)}\frac{Y_2}{\sqrt{R}} = Y_2\sqrt{\frac{-2log(R)}{R}} \tag{5.3.-2}$$

On average, this method requires $4/\pi = 1.273$ iterations [21].

5.4 Computational results

We report in this section computational results obtained with our column generation approach for sequential matching problem. All computational experiments

Problem Size	Constraints	Columns	Optimum Solution	CPU Time(in sec)
(10,10,3)	40	120	10	2.594
(20,20,3)	80	420	8	5.277
(40,40,3)	160	1680	3	51.985
(60,60,3)	240	4860	5	301.333
(80,80,3)	320	5040	2	710.542
(100,100,3)	400	8500	3	1597.03

are performed on the randomly generated problem as described in the previous section.. In all experiments a *steepest edge dual simplex algorithm* was used for solving the LP relaxations. The solvers default branching scheme we implemented is Ryan and Foster 4.3.8, and the solution of corresponding subproblems are solved by mixed integer programming method (An exact solution method for the corresponding constrained shortest path problem is still owing). The new branching scheme using the Lemma 4.4.2 also implemented to get a faster solution.

We first performed computational experiments with various randomly generated problem instances. The table in 5.4 gives some examples where the contents are explained as,

- **Problem Size:** The triplet `(nW,nJ,nT)` represents the number of workers, jobs, time shifts respectively.
- **Constraints:** The number of constraints in the final problem.
- **Columns:** The number of columns in the final problem.
- **Optimum Solution:** The optimum solution (the number of changes in final matchings or the optimum assignments).
- **CPU Time:** The total run time for each of the instances.

5.5 Conclusion

We have presented many classes in the branch and price implementation for the column generation formulation for the sequential matching problem. Computationally the method is effective for medium-size problem instances. To be able to

solve larger instances, it is necessary to have better branching schemes and pricing algorithms. Furthermore, the performance of the algorithm may be improved if better pricing heuristics can be developed for the pricing problem in some of the classes of subproblems. Also, the computational experiments indicate that solving pricing problems is likely to be even more challenging.

Chapter 6

2-Graph Problem

6.1 Introduction

In this chapter we describe a simplified version of the sequential matching problem, by considering only two bipartite graphs. The problem is to find the maximum cardinality matchings in these two graphs with minimum changes in edges. It is equivalent to maximum assignment of the workers to jobs in two different shifts in such a way as to minimize the unnecessary changes. In the next sections we analyze the problem, $\mathcal{NP}$- completeness, an augmenting cycle algorithm with the main idea of growing the common edges and conclude with some computational results.

6.2 Problem definition

The sequential matching problem $S(G, H)$ for two bipartite graphs G and H to minimize the changes between the edges in the maximum cardinality matchings can be defined as follows.

Definition: Given two bipartite graphs $G = (L_1, R_1, E_1)$, $H = (L_2, R_2, E_2)$ where L_1, R_1, L_2, R_2 subsets of V, denote the node sets and $E_1, E_2 \subseteq E$ denote the edge sets. The problem $S(G, H)$ is to find two maximum cardinality matchings M_1 in G and M_2 in H such that, $\mid M_1 \cap M_2 \mid$ is maximum.

Definition: For any graph $G = (V, E)$ we say a subset of edges $\mathcal{A}$ is *allowed* with respect to G if a maximum cardinality matching M exists in G such that $\mathcal{A} \subseteq M$. *(*i.e.), any subset $\mathcal{A}$ of M is called *allowed* in G with respect to M.

Property 6.2.1. *The sequential matching problem $S(G, H)$ for two bipartite graphs in G and H, is a simpler case of the problem of finding a maximum cardinality edge set $\mathcal{A} \subseteq E$, such that $\mathcal{A}$ is allowed in both G and H with respect some maximum matching M_1 and M_2 in respective matchings.*

This is because the problem S(G,H) can be deduced from the problem of finding the maximum cardinality $\mathcal{A}_{int} \subseteq E_1 \cap E_2$ such that $\mathcal{A}_{int}$ is allowed in both G and H with respect some maximum cardinality matchings M_1 and M_2.

6.3 Complexity analysis

Instance: Two bipartite graphs G and H on a node set V. Let k be an integer.

Question: Are there two matchings g in G and h in H, such that

1. g and h are of maximum cardinality.
2. There are at least k common edges in g and h.

We show that an instance $S(G, H)$ of sequential matching problem for two graphs can be polynomially transformed to an instance of 3-SAT. We state the result as Theorem 6.3.1.

Theorem 6.3.1. *Sequential Matching Problem with 2 bipartite graphs is $\mathcal{NP}$-complete.*

Proof. The reduction is from 3-SAT to sequential matching problem for two bipartite graphs. Given an instance ϕ of 3-SAT, we first construct two bipartite graphs G and H having maximum cardinality matchings g and h with at least k common edges, if and only if ϕ is satisfiable.

We are given an instance ϕ of 3-SAT with k clauses $C_1, C_2, \ldots, C_k$, with p boolean variables $\alpha_1, \alpha_1, \alpha_3, \ldots, \alpha_p$, such that each clause being $C_i = (\alpha_{i1}, \alpha_{i2}, \alpha_{i3})$, with

the α_{ij}'s being either boolean variables or negations thereof. Now construct the graphs $G = (L_1, R_1, E_1)$ and $H = (L_2, R_2, E_2)$ as follows: (L_1, L_2 represent left node sets, R_1, R_2 represent the right node sets).

§1 For each literal we have an edge and two nodes in both graphs. Say for each i, the literals α_{ij} represented by nodes $v^1_{ij} \in L_1$, $w^1_{ij} \in R_1$, $v^2_{ij} \in L_2$ and $w^2_{ij} \in R_2$, and connecting them by edges $e^1_{ij} \in E_1$ and $e^2_{ij} \in E_2$.

§2 In graph G, for each variable we need to add edges to get a false/true switching behavior for the literals belonging to a common variable. So for each literal of ϕ such that it is non-negated variable α_j we add the nodes $\{v^1_{i_1j}, v^1_{i_2j}, \ldots, v^1_{i_qj}\}$ to L_1, and $\{w^1_{i_1j}, w^1_{i_2j}, \ldots, w^1_{i_qj}\} \in R_1$,. Also the nodes $\{v^1_{i^1j}, v^1_{i^2j}, \ldots, v^1_{i^rj}\}$ are added to L_1 and $\{w^1_{i^1j}, w^1_{i^2j}, \ldots, w^1_{i^rj}\}$ to R_1, to represent the literals of ϕ which are negated variable $\neg\alpha_j$. By adding two new nodes $v^1_{\alpha_j}$ to L_1 and $w^1_{\alpha_j}$ to R_1, we introduce new edges such that, $\{e^1_{i_1j}, e^1_{i_2j}, \ldots, e^1_{i_qj}\} \in E_1$ and $\{e^1_{i^1j}, e^1_{i^2j}, \ldots, e^1_{i^rj}\} \in E_1$ as follows,

- Add edges e_{i_rj}, connecting $v^1_{i_1j}$ to $w^1_{i_2j}$, e_{i_2j}; connecting $v^1_{i_2j}$ to $w^1_{i_3j}$ etc. But the edge e_{i_qj} connecting $v^1_{i_qj}$ to $w^1_{\alpha_j}$.
- Add edge e_{i^1j} connecting $v^1_{i^1j}$ to $w^1_{\alpha_j}$.
- Add edges e_{i^2j}, connecting $w^1_{i^1j}$ to $v^1_{i^2j}$; e_{i^3j} connecting $w^1_{i^2j}$ to $v^1_{i^3j}$ etc. But the edge e_{i^rj} is connecting $w^1_{i^rj}$ to $v^1_{\alpha_j}$.
- Add edge e_{α_j} connecting $v^1_{\alpha_j}$ to $w^1_{i_1j}$

The construction for a non-negated variable α_j is illustrated as in Fig. 6.1.

§3 In H for each clause C_i, add two nodes each in L_2 and R_2 and 6 edges to E_2, such a way that any maximum matching of subgraph induced by nodes of C_i contain exactly one literal edge from E_2. *i.e.*, for each i, (for the clause C_i) adding the nodes $v^2_{ia}, v^2_{ib} \in L_2$ and connecting to all the nodes $w^2_{ij} \in R_2$. Similarly add nodes $w^2_{ia}, w^2_{ib} \in R_2$ and connecting to all the nodes $v^2_{ij} \in L_2$. Fig. 6.2 illustrates this construction for the clause C_i.

We prove that maximum matchings g and h exist, with at least k common edges if and only if there is a truth assignment for a 3-SAT instance.

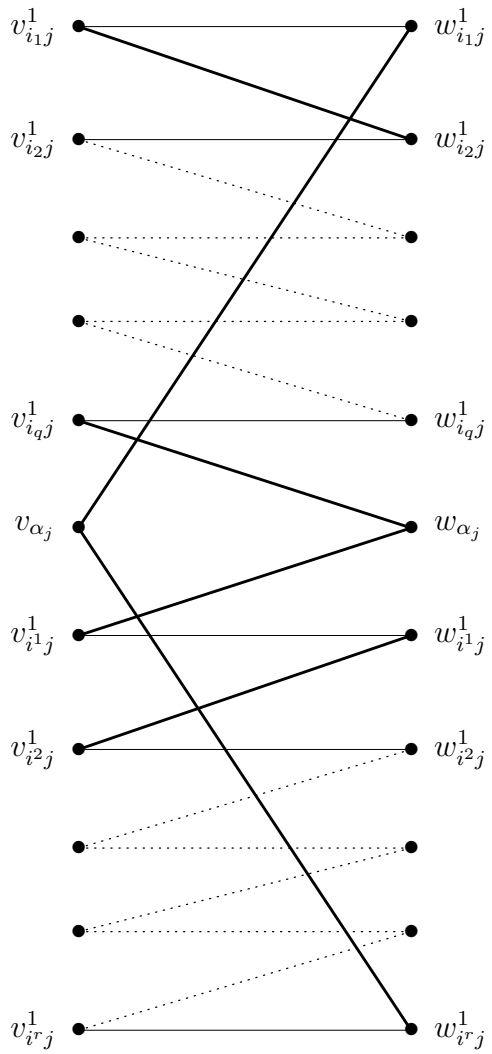

Figure 6.1: Subgraph of G induced by the variable α_j

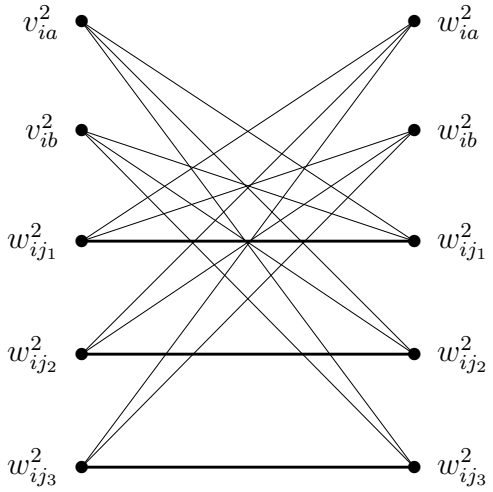

Figure 6.2: Subgraph of H induced by the clause C_i

Assuming that maximum matchings g and h exist with at least k common edges, we can find a truth assignment for the corresponding 3-SAT problem. Since both matchings are maximum, notice that all the nodes in L_1 have to be matched in G (since the matching number $G = \mid L_1 \mid$). By the construction, the common edges possible are from $e^1_{ij} \in E_1$ and $e^2_{ij} \in E_2$, which represents the literals. That matching h is maximum implies that, for each i there exists exactly one edge of the form $e^2_{ij'}$ is in h (the construction §1) for some variable α'_j. Taking a truth assignment for each α'_j such that the literal corresponds the edge $e_{ij'}$ is *true*, satisfies each of the clause C_i. By the construction of G it is clear that we can have such a truth assignment, since a maximum matching in G cannot have both the edges of the literal α_j and $\neg\alpha_j$, since both these edges are incident with the same node. This implies that such a truth assignment also satisfies the instance ϕ.

Conversely, suppose there is a truth assignment to the variables which satisfies instance ϕ such that each of the clauses C_i has at least one literal $\alpha_{ij'}$ with true value. Let's say $\alpha_{j'}$ is the variable corresponding to the true literal in C_i (*i.e.*, $\alpha_{ij'}$ is either $\alpha_{j'}$ or $\neg\alpha_{j'}$). Now describe a matching g in G such that it contains edges $e^1_{ij'}$ and a matching h in H such that $e^2_{ij'}$ in h. Since both α_j and α'_j cannot be true we can add edges to g such that it becomes a maximum matching in G. Since every maximum matching of H contains exactly one edge from the edges representing the clause C_i, surely h can also be extended to a maximum matching. So both g and h have exactly k common edges. □

6.4 Solution method

In this section we are generating heuristic procedures to examine how the solution can be better attained for this special case of the sequential matching problem with two graphs. First we look at a *weighted edge method* by solving the maximum weighted maximum cardinality matchings on weighted graphs. Let G and H be the bipartite graphs, we construct two weighted bipartite graphs G' and H' from G and H such that they have an extra weight for the common edges. The idea of the algorithm is to find maximum weighted maximum cardinality matchings M_1, M_2 in weighted graphs G' and H' respectively. The algorithm 5 describes the procedure precisely.

Algorithm 5 Weighted edge method: Solving the 2-Graph problem

Find the edges which are common in both graphs.
Add an extra weight (say 1) to these edges in both graphs.
Solve maximum weighted maximum cardinality matching problem in both graphs.

Even though the basic idea of the algorithm is to maximize common edges in the graphs, the algorithm need not give an optimum solution for the problem. This is because the same weighted common edges need not occur in both maximum cardinality matchings simultaneously. The following counter example illustrates this fact.

Consider two bipartite graphs G and H as shown in Fig.6.3. The problem is to find maximum cardinality matching in each bipartite graphs with minimum changes in edges. The common edges in both graphs are represented by the thick edges. Let

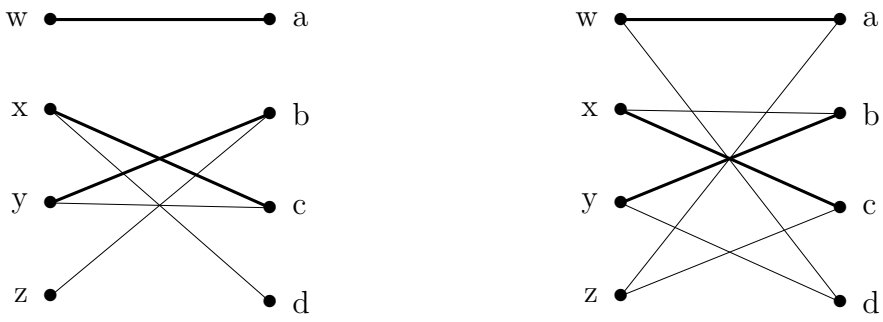

Figure 6.3: Bipartite graphs G and H

us examine the solution for this problem, as shown in Fig. 6.4, which is given by

algorithm 5. Since in this method the attempt is made to maximize the number of common edges locally in each bipartite graph, the algorithm results in matchings which have the greatest number of weighted edges (thick edges). But it fails to make these edges common in both matchings. So the edges in each matchings are still different, not giving an optimum solution. Compare the solution with the following optimum solution.

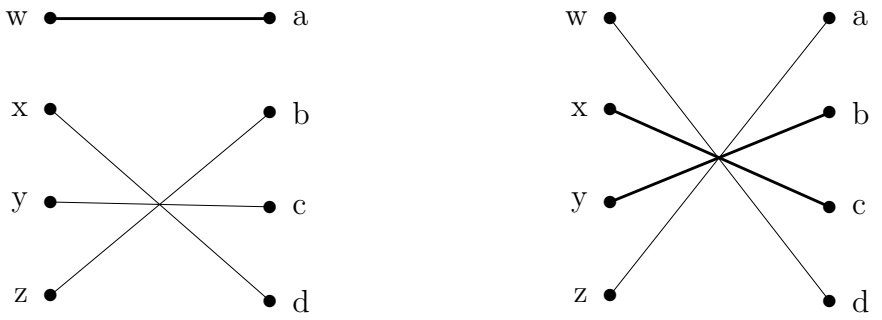

Figure 6.4: Matchings in G and H output by the algorithm 5

We can find an optimum solution for the problem with a single change in total, as shown in Fig. 6.5. This shows that the algorithm does not give an *optimum*

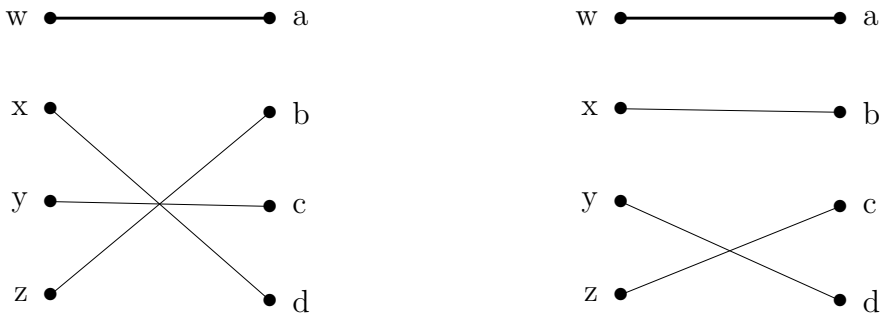

Figure 6.5: Optimum matchings for G and H

solution in this problem, and motivates the search for a better method.

6.5 Augmenting cycle method

Definition 6.5.1 (Alternating Paths and Cycles). Let $G = (V, E)$ be a graph, We refer to a path P in the graph as an *alternating path* with respect to a matching

M if every consecutive pair of arcs in the path contains one matched and one unmatched arc. We refer to an alternating path as an *even alternating path* if it contains an even number of arcs and an *odd alternating path* if it contains an odd number of arcs.

Definition 6.5.2 (Alternating Cycle). An *alternating cycle* C with respect to M is an alternating path that starts and ends with same node.

Note that for a bipartite graph, the *alternating cycle* is of even length.

Definition 6.5.3 (Augmenting Paths). We refer to an odd alternating path P with respect to a matching M as an *augmenting path* if the first nodes in the path are unmatched.

Let $\oplus$ denote the augment operator as defined in Chapter 1. *i.e.*, for any two sets $A \oplus B = (A \cup B) - (A \cap B)$.

Property 6.5.1. *If G is a bipartite graph and C is an alternating cycle with respect to a matching M, then a matching $M' = M \oplus C$ exists with the same cardinality of M.*

When M is a maximum cardinality matching and if an alternating cycle exists as above, then another matching exists, which is again of maximum cardinality. Using this property we sketch an algorithm which has basic steps as follows.

The algorithm starts with two bipartite graphs G and H, with maximum cardinality matchings M_1 in G, and M_2 in H. The idea of the algorithm is to find a better matching M_1' (or M_2') which has more common edges with respect to M_2 (or M_1). This updating for M_1 can be done by adding an extra weight to those edges of G which exists in M_2, and by finding a positive weighted alternating cycle C_1 in G. A similar updating for M_2 can be done by adding extra weight to the edges of H that are in M_1. This updating can be done in a special order, until there is no improvement in the common edges. We implement the above idea into the algorithm 6 by alternatively augmenting the common edges between the matchings in G and H. For $i = 1, 2$ define $\mathcal{W}_i : E_i \to \mathbb{R}$, the weight function from E_i to $\mathbb{R}$ the set of real numbers. Also for any $E_i' \subseteq E_i$, we define $\mathcal{W}_i(E_i') = \sum_{e_k \in E_i'} \mathcal{W}(e_k)$.

The initial input matchings play an important role in this algorithm. Supposing we start with two disjoint matchings in non-disjoint graphs there is no further improvement in the solution, unless we consider the zero weighted alternating cycles. Therefore the results of the algorithm can be improved by starting with

Algorithm 6 Augmenting Cycle Method (ACM): Solving the 2-Graph problem

AugmentingCycleMethod(G_1, G_2, M_1, M_2)
input: Bipartite graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, W_2)$, with matchings M_1 and M_2.
$i := 1$,$f_1 :=$ `NOTFOUND`, $f_2 :=$ `NOTFOUND`
while $f_1 \neq$ `FOUND` **or** $f_2 \neq$ `FOUND` **do**
  for $e_k \in M_{3-i}$ **do**
    if $e_k \in G_i$ **then**
      if $e_k \notin M_i$ **then**
        $\mathcal{W}_i(e_k) := 1$
      else
        $\mathcal{W}_i(e_k) := -1$
      end if
    end if
  end for
  $\mathcal{C}_i :=$ *NegWeighedAlternatingCycle*$(G_i,\ M_i)$
  if $C_i \neq \emptyset$ **then**
    Update $M_i := M_i \oplus \mathcal{C}_i$ Comment: Augment $|M_i \cap M_{3-i}|$
    $f_1 :=$ `NOTFOUND` , $f_2 :=$ `NOTFOUND`
  else
    $f_i :=$ `FOUND`
  end if
  $i \leftarrow 3 - i$
end while
output: Matchings M_1 and M_2

a *good* solution, probably an output given by algorithm 5. The run time of the algorithms can be as bad as $O(n^4)$.

Now we address the subproblem of finding a negative alternate cycle in the bipartite graph $G_i = (L_i, R_i, E_i)$ with respect to the matching M_i. The algorithm 7 output a negative alternate cycle if one exists. In the following section we briefly discuss the procedures in the algorithm.

If we look back at the construction of an alternating cycle we may notice that one of the neighbours of a vertex $v_i \in L_i$ in the cycle is always its matched node in R_i. We can thus simplify the search technique by ignoring one level of nodes (say right) and going directly from the left level of vertices to the new left level of vertices. Obviously this corresponds to searching in a directed graph $D = (N_d, E_d)$ constructed from the G_i based on the matching M_i. For $v_1, v_2 \in N_d$, the edge

Algorithm 7 Negative weighted alternating cycle in G w.r.t. M

NegWeighedAlternatingCycle(G, M)
input: A weighted bipartite graph $G = (L, R, E)$ with a matching M.
$D = (N_d, E_d)$, new empty directed weighted graph
for each matched node $v_i \in L$ in M **do**
  $N_d := N_d \cup v_i$
end for
for each matched node $v_i \in L$ in M **do**
  $w_i := match(v_i)$
  for all neighbours $v_j \in L$ of w_i **do**
    new edge d_{ij} from v_i to v_j
    $\mathcal{W}(d_{ij}) := \mathcal{W}(e(v_i, w_i)) + \mathcal{W}(e(w_i, v_j))$
    $E := E \cup \{d_{ij}\}$
  end for
end for
cycle := FloydWarshallNegativeCycle(D)
altcycle :=$\emptyset$
if *cycle* $\neq \emptyset$ **then**
  for v_i in *cycle* **do**
    $w_i := match(v_i)$
    altcycle := *altcycle* $\cup\{v_i, w_i\}$
  end for
end if
return *altcycle*

$e(v_1, v_2) \in E_d$ exists only if v_2 is a neighbour of the matched node $w_i \in R$ of v_1. So the construction leads to a reduced-size directed graph with $|V_i|$ nodes and $|E_i| - |M_i|$ edges. Also we distribute the weights of the edges to the new graph such that, for an edge $d_{ij} \in E_d$ $\mathcal{W}(d_{ij}) = \mathcal{W}(e(v_i, w_i)) + \mathcal{W}(e(w_i, v_j))$. A negative alternate cycle in G_i, if one exists, can be constructed from a negative cycle in D. The issue of finding a negative cycle in D, if one exists, can be handled by a modification of the *Floyd-Warshall* algorithm to find all pairs shortest path in directed/undirected network. *Floyd-Warshall* algorithm (algorithm 8) to find all pairs shortest paths is described in the next section and it is the modification to find a negative cycle in weighted graph [1].

6.5.1 *The Floyd-Warshall* negative cycle algorithm

In the *Floyd-Warshall* all pairs shortest path algorithm, for any pair of nodes (v_i, v_j) we obtain a matrix of shortest path distance $d[i, j]$ within $\mathcal{O}(n^3)$ computations. It uses a dynamic-programming methodology and uses triple operations cleverly. The algorithm is based on inductive arguments developed by an application of a dynamic programming technique. Let $d^k[i, j]$ represent the length of a shortest path from node v_i and v_j, subject to using the nodes $v_1, v_2, \ldots, v_{k-1}$ as internal nodes.

Clearly $d^{n+1}[i, j]$ represents the actual shortest path distance from node v_i to v_j. The *Floyd-Warshall* algorithm first computes $d^1[i, j]$ for all node pairs to v_i, and v_j. Using the $d^1[i, j]$ then computes $d^2[i, j]$ for all node pairs i and j. It repeats this process until we obtain $d^{n+1}[i, j]$ for all node pairs v_i and v_j when it terminates. Given $d^k[i, j]$, the algorithm computes $d^{k+1}[i, j]$ using the following property.

Property 6.5.2. $d^{k+1}[i, j] = min\{d^k[i, j], d^k[i, k] + d^k[k, j]$.

This property is valid for the following reason. A shortest path that uses only the nodes $1, 2, \ldots, k$ as internal nodes either (1) does not pass though node k, in which case $d^{k+1}[i, j] = d^k[i, j]$ or, (2) does not pass through node k, in which case $d^{k+1}[i, j] = d^k[i, k] + d^k[k, j]$. Therefore $d^{k+1}[i, j] = min\{d^k[i, j], d^k[i, k] + d^k[k, j]$.

The algorithm uses predecessor indices, $pred[i, j]$ for each node pair $[v_i, v_j]$. The index $pred[i, j]$ denotes the last node prior to node v_j in the tentative shortest path from node v_i to node v_j. The algorithm maintains the invariant property that when $d[i, j]$ is finite, the network contains a path from node v_i to node v_j of length $d[i, j]$. Using the predecessor indices we can obtain this path say P from node v_k to v_l as follows. We backtrack along the path P starting at node l. Then g is the node prior to node l in P. Similarly $h = pred[k, g]$ is the node prior to node g in P, and so on. We repeat this process until we reach the node k.

Now we describe the *Floyd Warshall* algorithm as the method for detecting the presence of a negative-weight cycle, and backtrack to find the negative cycle. In the final output of the *Floyd Warshall* algorithm, all entries along the main matrix should be 0. If any of them is negative, then that node is part of a negative weight cycle, since there is a path from that node to itself with a negative weight. This can be proved as follows.

We show that whenever the network contains a negative cycle, then during the computation we will eventually satisfy the condition $d[i, i] < 0$ for some i. Suppose

Algorithm 8 Floyd-Warshall: Negative Cycle Algorithm

```
FloydWarshallNegativeCycle(D)
input: Directed weighted graph D(N, E)
for all node pairs (v_i, v_j) ∈ E × E do
   d[i, j] := 0
   pred[i, j] := 0
end for
for all nodes v_i ∈ E do
   d[i, i] := 0
end for
for each edge e_ij ∈ E do
   d[i, j] := c_ij
   pred[i, j] := i
end for
cyclefound:= false
for each k := 1 to n and not cyclefound do
   for all node pairs (v_i, v_j) ∈ E × E do
      if d[i, j] > f[i, k] + d[k, j] then
         d[i, j] := d[i, k] + d[k, j]
         pred[i, j] := pred[k, j]
         if (d[i, i] < 0) then
            cyclefound := true
            k := i
         end if
      end if
   end for
end for
comment: Backtracking the negative cycle using predecessor indices
C := ∅
if cyclefound then
   l := pred[k, k]
   C := {v_k, v_l}
   while (v_k ≠ v_l) do
      l := pred[l, k]
      C := C ∪ v_l
   end while
end if
return C_i
```

that there is a negative weight cycle containing nodes $v_1, \ldots, v_i$, where v_i is the highest number node. When we update the row for v_i in the matrix, one of the diagonal values for the nodes will definitely become negative. In other words, there will be a way such that we can pivot on at least one node in the cycle, and get back to the original starting point with a cost less than 0 (which was the cost to remain at that node and thus minimum in the graph with no negative-weight cycles). Also, one may perform the algorithm twice, using the output of the first as the starting point for the second. If any of the cells are updated in the second pass of the algorithm, then there must be a negative weight cycle, since a lower cost will be obtained by going through that cycle multiple times, as determined by the second pass of the algorithm.

In the *Floyd-Warshall* algorithm we detect the presence of a negative cycle simply by checking the condition $d[i, i] < 0$ whenever we update $d[i, i]$ for some node v_i. Using the predecessor graph maintained by the algorithm we can backtrack this negative cycle.

6.5.2 Complexity

For the bipartite graph $G_i = (L_i, R_i, E_i)$ say $n = min(|L_i|, |R_i|)$. Then the directed graph constructed has at most n nodes. The *Floyd Warshall* algorithm clearly perform n major iterations, one for each k and within each iteration, it performs $\mathcal{O}(1)$ computations for each node pair. Consequently it runs in $\mathcal{O}(n^3)$. The complexity of negative weighted alternating cycle algorithm 7 is same as that of *FloydWarshall* algorithm, because the construction of the directed graph is of $\mathcal{O}(|E_i|)$. We note that a matching in G_i can have no more than $n = min(|L_i|, |R_i|)$ edges. So in the algorithm *AugCycleMethod* the main iteration will run until there is no alternate negative cycle in the matchings. Since each augmentation increases the cardinality of matchings, we can have at most n stages. Since in each stage we need to find a negative alternative cycle, the *AugCycleMethod* has a complexity of $\mathcal{O}(n^4)$. In the following section we briefly explain the details of the implementation of this approach and consequently we analyze results.

6.5.3 Data structures and implementation issues

In this section we briefly describe the particulars of the implementations of *augmenting cycle method* (ACM). The implementation works with the two partitions

L_i and R_i of node sets for separate graphs. All graphs use the same initial matching algorithm for maximum cardinality matchings.

It is first and foremost intended for the ease of testing and of evaluating the potential of our approach. The program outputs maximum cardinality matchings in each bipartite graph with a significantly higher number of common edges. We have tested the code with the randomly generated graphs, as explained in Chapter 5. They were designed to be representative of sequential matching problems that might arise in practice.

As mentioned before, the graphs used to test the programs are variations of random bipartite graphs. A detailed description of the random problem generation is given in Chapter 5. In the graphs each vertex has an expected number of neighbors. The actual number is obtained by simulating a Poisson random variable which in turn approximates to the binomial random variable in a real random graph. The vertices in L_i and R_i are divided into $n/2$ vertices each. The neighbors for a vertex v_i are chosen randomly but uniquely from vertices R_i. This expected number of neighbors which are generated randomly according to Poisson or normal distribution can be modified with given options for changing parameters. The number of edges can also vary using different seeds for the pseudorandom number generator (which was `random()`). In all the generated graphs $|L_i| = |R_i|$, and in the dense graph resulting maximum matchings were always just below perfect. For the testing of the augmenting cycle method we use the graphs perfect matchings. The class `RandomProblem` contains the functions for generating random problems by this method.

We make few general remarks about the data structures implemented to represent the bipartite graphs. The two basic data structures for any graphs are adjacency matrices and adjacency lists. Adjacency matrices require much space but can represent better accessible entries. In the case of bipartite graphs the size of the adjacency matrix reduces significantly, since the node sets are partitioned in such a way that there are no edges for the nodes in the partition. So each row corresponds to a vertex $v_j \in L_i$, and each column to a vertex $w_k \in R_i$, with a nonzero entry in $M[j,k]$ if and only if vertex v_j is adjacent to node w_k. An alternate way, especially for sparse bipartite graphs, is to represent an adjacency list. Adjacency lists are typically used to represent the incidence structure of a graph. To do this each $v_j \in L_i$ we represent a list of nodes in R_i which are adjacent to v_j. This method saves memory but accessibility of edges reduces. In the class `BipartiteGraph` we use both representations and one can be chosen as required.

Further characteristics of the experiments are as follows: At the end of each run the

solution is checked for consistency and maximality. Run times were measured with the system call `clock()` and using the `clock_t` structure. As mentioned before, the graphs used to test the programs are variations of random bipartite graphs with dense/sparse amount of edges. Run times reported exclude input, checking, and output time.

Some important classes in the implementation of the *AugCycleMethod* are described in Appendix B.

6.5.4 Computational results

The computational experiments have been run with the randomly generated test cases. The procedures are implemented with C++ language, compilation with the Visual C++ 6.0 and the test runs are executed on a personal computer with 450 MHz Pentium II processor with 128 MB core memory running Windows 2000. All the compilations are invoked "Maximum Speed" optimization. Firstly we tested how the *AugCycleMethod* behaves. The procedure terminates when there are no common edges between two consecutive graphs. Before analyzing these results, let us stress the fact that the procedure we propose is heuristic.

The test is run for problem sizes (*i.e.*, the number of nodes) 300. The results given in table 6.6, 6.7 report the run times, operation counts, and growth rates of common edges for different graphs. It can be seen from the Fig. 6.6 how the *AugCycleMethod* increases the common edges, for an input of randomly generated dense bipartite graphs of size 300 nodes. This can be explained by the fact that the algorithm significantly increases the common edges between the matchings, and the growth rate in common edges is uniform.

6.5.5 Comparison with optimum solution

The *augmenting cycle algorithm* doesn't give an optimum solution always but a comparatively better solution in a significantly lower run time. Table 6.5.5 and Fig. 6.8 illustrate the comparison between the solution and run time obtained by *augmenting cycle algorithm* and the optimum solution. The mixed integer programming formulation of the sequential matching problem in Chapter 4, used for finding the optimum solution by using *mixed integer optimizer of CPLEX*.

Iteration	Commonedges
0	23
40	64
60	73
80	104
100	125
140	165
180	205
190	215
199	222

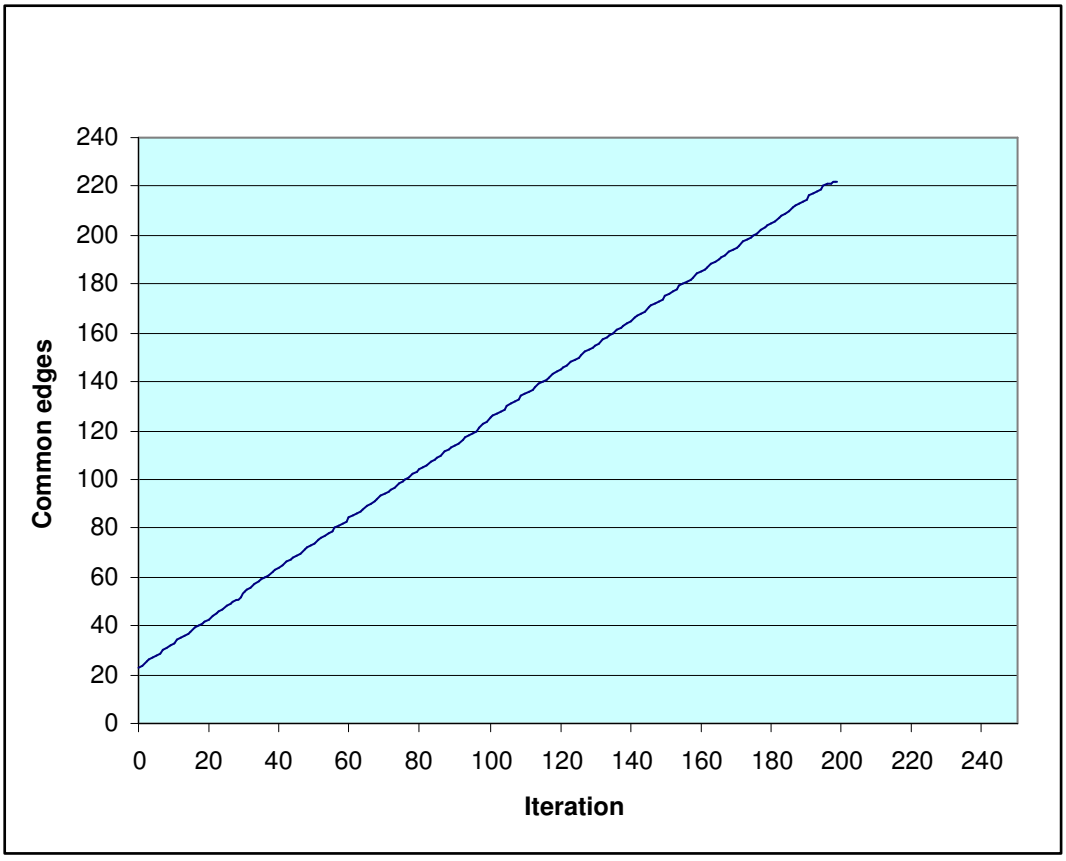

Figure 6.6: The common edges incremented by the *AugmetingCycleMethod*

Left Nodes	Common edges	Run time(in Sec)
10	7	0.01
50	38	0.23
100	94	3.21
150	143	14.26
200	191	44.22
250	242	116.52
300	282	240.78
350	337	496.59
400	387	903.18
450	442	1510.32

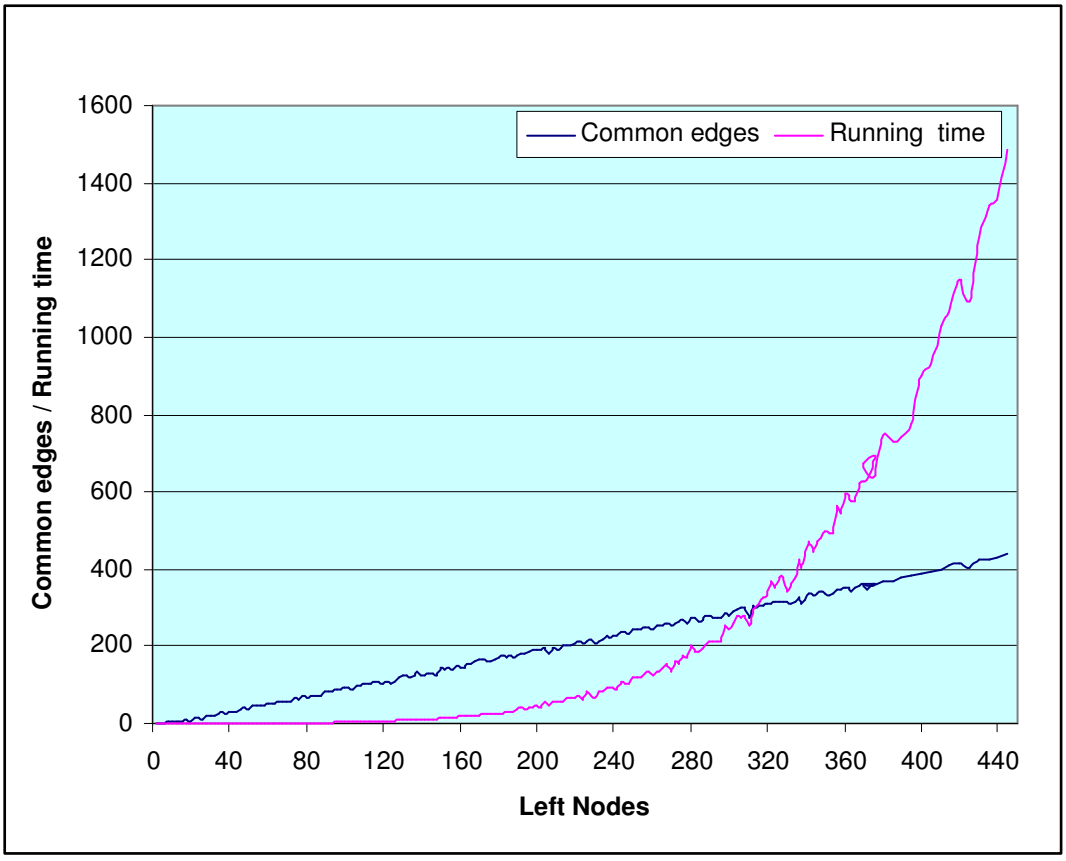

Figure 6.7: The common edges/run time versus the number of left nodes by the *AugmetingCycleMethod*

Left Nodes	CE-ACM	RT-ACM	CE-CPLEX	RT-CPLEX
10	5	0.02	5	0.51
20	14	0.06	20	0.24
40	94	0.34	40	1.68
60	44	0.79	58	4.73
80	73	2.09	80	24.54
100	93	4.69	100	76.97
120	107	8.00	118	154.41
140	134	14.36	140	434.15
160	156	21.05	159	730.23

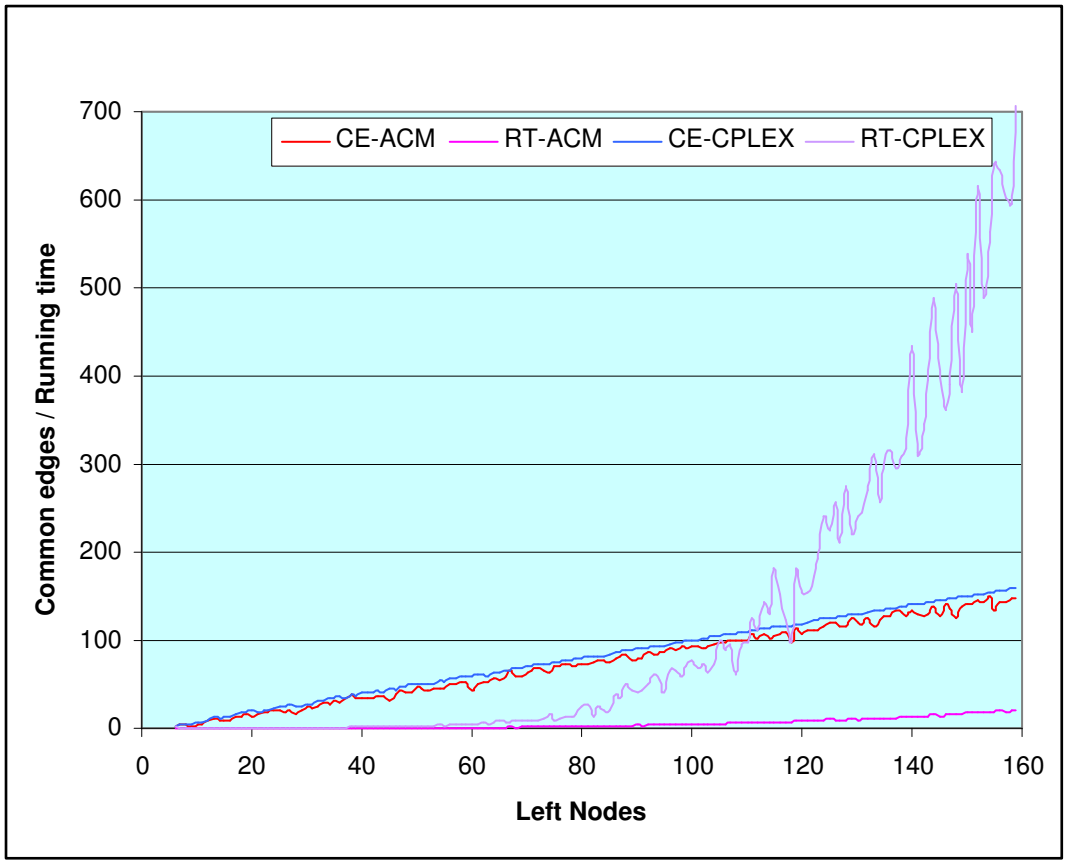

Figure 6.8: The comparison with optimum solution and solution obtained by *AugmetingCycleMethod*

1

Algorithm 9 Augmenting Cycle Method (ACM) for general problem $S(\mathbf{G}_c, \tau)$.

***AugmentingCycleMethod*($S(\mathbf{G}_c, \tau)$, $\mathbf{M}_c, \tau'$)**
input: Bipartite graphs $\mathbf{G}_c$, with matchings $\mathbf{M}_c$.
for $t := 1$ to $t := \tau$ **do**
  $f_t :=$ `NOTFOUND`
end for
$t := \tau'$
while $f_{t'} \neq$ `FOUND` for any t' among 1 to τ **do**
  if $t = \tau$ **then**
    $t := 1$
  end if
  for $e_k \in M_{t+1}$ **do**
    if $e_k \in G_t$ **then**
      if $e_k \notin M_t$ **then**
        $\mathcal{W}_t(e_k) := 1$
      else
        $\mathcal{W}_t(e_k) := -1$
      end if
    end if
  end for
  $\mathcal{C}_t :=$ *NegWeighedAlternatingCycle*(G_t, M_t)
  if $C_t \neq \emptyset$ **then**
    Update $M_t := M_t \oplus \mathcal{C}_t$
    $f_{t'} :=$ `NOTFOUND` $\forall t'$ from 1 to τ
  else
    $f_t :=$ `FOUND`
  end if
  $t \leftarrow t + 1$
end while
output: Matchings M_1 and M_2

We performed computational experiments with various sized problem instances (number of nodes), and comparison of the CPU time of the solution method. See Section 6.5.2 to notice that the algorithm has a worst case complexity of $\mathcal{O}(n^4)$. The algorithm checked for randomly generated problem of left node size varies from 2 to 220. Fig. 6.7 illustrates the algorithm in the terms of run time complexity, one can clearly notice that the run time increases significantly with the size of an instance. Other computational results show that the algorithm runs faster for a sparse bipartite graph, but obviously with fewer common edges.

6.6 Generalization of ACM for $S(\mathbf{G}_c, \tau)$

We can generalize the augmenting cycle method for the general sequential matching problem, $S(\mathbf{G}_c, \tau)$. The most important step to notice is the direction of augmenting. In the 2-graph case we increase the common edge alternativelys in both graphs. The generalized algorithm 9 augment the common edges between consecutive matching cyclicly, with initial starting graph $G_{\tau'}$, where $\mathbf{M}_c = \{M_t : t = 1, 2, \ldots, \tau\}$ denotes a sequence of initial maximum matchings in $\mathbf{G}_c$.

6.7 Conclusion

To conclude this chapter let us say a few words about solution methods that are suggested to solve the problem. Preliminary computational results are promising and indicate that good and feasible solutions may be found quickly with the procedure. When the procedure is applied to the general sequential matching problem, the solution mainly depends on the value of starting point τ', and the direction of augmentation. We tried to tune the algorithm in different directions in the next steps, but the default steps of the same direction produced the best results in many cases.

Appendix A

Input file formats

To produce an efficient implementation we followed different data structures and corresponding input file formats. The method is to standardize the method by which information is retrieved from a file. This is done by creating a well-defined data structure that contains easily searched information about the data, the graph and then separating that information from the data, and storing it in the memory.

In the following section we illustrate different file formats, with an example of the sequential matching problem arising from bipartite graphs in three time intervals.

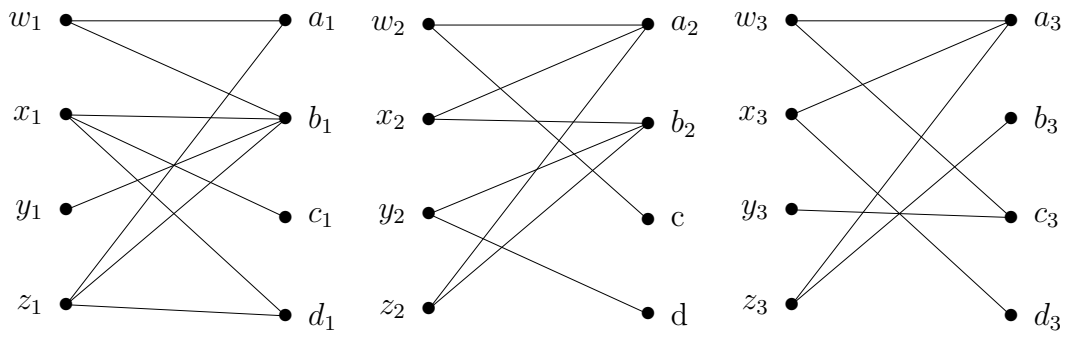

Figure 9: A sequential matching problem with 3 bipartite graphs

SGR format

The native file format for representing graphs consists of several lines and each entry is separated by a space. The '# ' characters in the first column are ignored, and can be used as comment lines. The first two non-comment entries contain the number of nodes and number of edges. Consecutively the file has been divided into a node section and edge section. In the node section each line contains the letter n to indicate node and is followed by the name of the node. The nodes are ordered and numbered according to their position in the node list of the graph. In the edge section each line starts with an indictor letter e and is followed by the names of the source the target edges. If the graph is weighted the weight of the corresponding edges follows the target name.

A finite sequence of graphs can be represented using the above method by distinguishing them by special symbol '$', and followed by the corresponding graph number. We used the file extension **SGR** (**S**equence of **GR**aphs) to represent the text file of representing this sequence of graphs. One main advantage of this is when the data is required for a special graph number, it is found by using the easily searched information of graph number which points to the data for the graph itself. The benefits are that the data can be placed anywhere in the file. The **SGR** file in the following example represents the problem in the Fig. 9.

```
#The text file SMPFORM.SGR
$0
8 9
#node format n name x y
#edge format e node1 node2 matched
n 0
n 1
n 2
n 3
n 4
n 5
n 6
n 7
e 0 5
e 0 4
e 1 7
e 1 6
e 1 5
e 2 5
e 3 5
e 3 4
e 3 7
$1
8 8
#node format n name x y
#edge format e node1 node2 matched
n 0
n 1
n 2
n 3
n 4
n 5
n 6
n 7
e 0 4
e 0 6
e 1 5
```

```
e 1 4
e 2 5
e 2 7
e 3 4
e 3 5
$2
8 7
#node format n name x y
#edge format e node1 node2 matched
n 0
n 1
n 2
n 3
n 4
n 5
n 6
n 7
e 0 4
e 0 6
e 1 7
e 1 4
e 2 6
e 3 5
e 3 4
```

CGP format

While observing the column generation formulation we can notice that the sequence of graphs is viewed in a different way, and not like the adjacency list data structure of native graph problems. Also, the data structures for the column generation method are such that generating/adding the columns is made easy. So we used a different problem file format for input of the problem to make implementation more efficient.

A sequential matching problem can be also seen as a different point of view, as in column generation formulation. The number of assignments for each worker, and the assignments in each time shift, rather than considering the whole graph at a particular time shift. In the **CGP** (**C**olumn **G**eneration **P**roblem) extension file we represent the problem in such a way that the data structure can access the problem more easily. The file is organized such that for each worker there is a section representing the jobs that they are able to do in different time shifts, and these sections are arranged in the same order as the worker indices. The first row of a section is the total number, the second row in a section is for job indices for the worker say, i, at time shift, t, is followed by the total job numbers. In each section these rows are ordered in the time index. The **CGP** representation of the graphical representation of a sequential matching problem in figure 9, is as follows.

```
#The text file smpform.cgp
4 4 3
#Worker 0
2
1 0
2
```

```
0 2
2
0 2
#Worker 1
3
3 2 1
2
1 0
2
3 0
#Worker 3
1
1
2
1 3
1
2
#Worker 4
3
1 0 3
2
0 1
2
1 0
```

MPS format

As another format we use the standard **MPS** format for the mixed integer programming problem. The **MPS** format(developed originally by IBM) is a standard file input for resolving a problem instance. For a detailed description of the representation of an MIP instance in an **MPS** format see the CPLEX manual [22].

Every **MPS** file has at least the three sections:

1. `ROWS`
2. `COLUMNS`
3. `RHS`

The `ROWS` section lists the row names, starting with L means ¡ , E means =, ¿ represented by G, or not constrained represented by N. The `COLUMNS` section lists each nonzero element of the matrix, preceded by the column name and row name in which it appears. The `RHS` section lists the elements of the right-hand side. One must also give a column name to the right-hand side. A `BOUNDS` section (optional) allows one to supply simple upper and lower bounds on variables. A `RANGES` section (optional) allows one to supply upper and lower limits on constraints. Consider a graphical representation of a sequential matching problem as in Fig. 9, the following text file *smpform.mps* represents the problem in .mps format. Note that the graphs

in the example have perfect matching and so the matching number of each graph is four.

The mixed integer linear programming formulation of the problem as described in Section 4.2.1 can be used to find an optimum solution using an MIP solver such as CPLEX or SOPLEX. To construct the MIP instance for the corresponding SMP instance we need to intersect the edge sets of two consecutive edge sets, since some variables (y_{ijt}).

```
#The text file smpform.mps
NAME smpformulation
ROWS
 N obj
 L c_0_0
 L c_1_0
 L c_2_0
 L c_3_0
 L c_4_0
 L c_5_0
 L c_6_0
 L c_7_0
 G c_0
 G c_0_4_0p
 G c_0_4_0n
 G c_1_5_0p
 G c_1_5_0n
 G c_2_5_0p
 G c_2_5_0n
 G c_3_5_0p
 G c_3_5_0n
 G c_3_4_0p
 G c_3_4_0n
 L c_0_1
 L c_1_1
 L c_2_1
 L c_3_1
 L c_4_1
 L c_5_1
 L c_6_1
 L c_7_1
 G c_1
 G c_0_4_1p
 G c_0_4_1n
 G c_0_6_1p
 G c_0_6_1n
 G c_1_4_1p
 G c_1_4_1n
 G c_3_4_1p
 G c_3_4_1n
 G c_3_5_1p
 G c_3_5_1n
 L c_0_2
 L c_1_2
 L c_2_2
 L c_3_2
 L c_4_2
 L c_5_2
 L c_6_2
 L c_7_2
 G c_2
COLUMNS
  x_0_5_0  c_0_0  1
  x_0_5_0  c_5_0  1
  x_0_5_0  c_0  1
  x_0_4_0  c_0_0  1
  x_0_4_0  c_4_0  1
  x_0_4_0  c_0  1
  x_0_4_0  c_0_4_0n  1
  mark_0_4_0  'MARKER'  'INTORG'
  y_0_4_0  obj  -1  c_0_4_0p  -1
  y_0_4_0  c_0_4_0n  -1
  mark_0_4_0_X  'MARKER'  'INTEND'
  x_1_7_0  c_1_0  1
  x_1_7_0  c_7_0  1
  x_2_5_0  c_0  1
  x_2_5_0  c_2_5_0n  1
  mark_2_5_0  'MARKER'  'INTORG'
  y_2_5_0  obj  -1  c_2_5_0p  -1
  y_2_5_0  c_2_5_0n  -1
  mark_2_5_0_X  'MARKER'  'INTEND'
  x_3_5_0  c_3_0  1
  x_3_5_0  c_5_0  1
  x_3_5_0  c_0  1
  x_3_5_0  c_3_5_0n  1
  mark_3_5_0  'MARKER'  'INTORG'
  y_3_5_0  obj  -1  c_3_5_0p  -1
  y_3_5_0  c_3_5_0n  -1
  mark_3_5_0_X  'MARKER'  'INTEND'
  x_3_4_0  c_3_0  1
  x_3_4_0  c_4_0  1
  x_3_4_0  c_0  1
  x_3_4_0  c_3_4_0n  1
  mark_3_4_0  'MARKER'  'INTORG'
  y_3_4_0  obj  -1  c_3_4_0p  -1
  y_3_4_0  c_3_4_0n  -1
  mark_3_4_0_X  'MARKER'  'INTEND'
  x_3_7_0  c_3_0  1
  x_3_7_0  c_7_0  1
  x_3_7_0  c_0  1
  x_0_4_1  c_0_1  1
  x_0_4_1  c_4_1  1
  x_0_4_1  c_1  1
  x_0_4_1  c_0_4_0p  1
  x_0_4_1  c_0_4_1n  1
  mark_0_4_1  'MARKER'  'INTORG'
  y_0_4_1  obj  -1  c_0_4_1p  -1
  y_0_4_1  c_0_4_1n  -1
  mark_0_4_1_X  'MARKER'  'INTEND'
  x_0_6_1  c_0_1  1
  x_0_6_1  c_6_1  1
  x_0_6_1  c_1  1
  x_0_6_1  c_0_6_1n  1
  mark_0_6_1  'MARKER'  'INTORG'
  y_0_6_1  obj  -1  c_0_6_1p  -1
  y_0_6_1  c_0_6_1n  -1
  mark_0_6_1_X  'MARKER'  'INTEND'
  x_1_5_1  c_1_1  1
  x_1_5_1  c_5_1  1
  x_1_5_1  c_1  1
  x_1_5_1  c_1_5_0p  1
  x_1_4_1  c_1_1  1
  x_1_4_1  c_4_1  1
  x_1_4_1  c_1  1
  x_1_4_1  c_1_4_1n  1
  mark_1_4_1  'MARKER'  'INTORG'
  y_1_4_1  obj  -1  c_1_4_1p  -1
  y_1_4_1  c_1_4_1n  -1
  mark_1_4_1_X  'MARKER'  'INTEND'
  x_2_5_1  c_2_1  1
  x_2_5_1  c_5_1  1
  x_2_5_1  c_1  1
  x_2_5_1  c_2_5_0p  1
  x_2_7_1  c_2_1  1
  x_2_7_1  c_7_1  1
  x_2_7_1  c_1  1
  x_3_4_1  c_3_1  1
  x_3_4_1  c_4_1  1
  x_3_4_1  c_1  1
  x_3_4_1  c_3_4_0p  1
  y_3_5_1  c_3_5_1n  -1
  mark_3_5_1_X  'MARKER'  'INTEND'
  x_0_4_2  c_0_2  1
  x_0_4_2  c_4_2  1
  x_0_4_2  c_2  1
  x_0_4_2  c_0_4_1p  1
  x_0_6_2  c_0_2  1
  x_0_6_2  c_6_2  1
  x_0_6_2  c_2  1
  x_0_6_2  c_0_6_1p  1
  x_1_7_2  c_1_2  1
  x_1_7_2  c_7_2  1
  x_1_7_2  c_2  1
  x_1_4_2  c_1_2  1
  x_1_4_2  c_4_2  1
  x_1_4_2  c_2  1
  x_1_4_2  c_1_4_1p  1
  x_2_6_2  c_2_2  1
  x_2_6_2  c_6_2  1
  x_2_6_2  c_2  1
  x_3_5_2  c_3_2  1
  x_3_5_2  c_5_2  1
  x_3_5_2  c_2  1
  x_3_5_2  c_3_5_1p  1
  x_3_4_2  c_3_2  1
  x_3_4_2  c_4_2  1
  x_3_4_2  c_2  1
  x_3_4_2  c_3_4_1p  1
RHS
  rhs  c_0_0  1
  rhs  c_1_0  1
  rhs  c_2_0  1
  rhs  c_3_0  1
  rhs  c_4_0  1
  rhs  c_5_0  1
  rhs  c_6_0  1
  rhs  c_7_0  1
  rhs  c_0  4
  rhs  c_0_4_0p  0
  rhs  c_0_4_0n  0
  rhs  c_1_5_0p  0
  rhs  c_1_5_0n  0
  rhs  c_2_5_0p  0
  rhs  c_2_5_0n  0
  rhs  c_3_5_0p  0
  rhs  c_3_5_0n  0
  rhs  c_3_4_0p  0
  rhs  c_3_4_0n  0
  rhs  c_0_1  1
  rhs  c_1_1  1
  rhs  c_2_1  1
  rhs  c_3_1  1
  rhs  c_4_1  1
  rhs  c_5_1  1
  rhs  c_6_1  1
  rhs  c_7_1  1
  rhs  c_1  4
  rhs  c_0_4_1p  0
  rhs  c_0_4_1n  0
  rhs  c_0_6_1p  0
  rhs  c_0_6_1n  0
  rhs  c_1_4_1p  0
  rhs  c_1_4_1n  0
  rhs  c_3_4_1p  0
  rhs  c_3_4_1n  0
```

```
 x_1_7_0  c_0  1                 x_3_4_1  c_3_4_1n  1                rhs  c_3_5_1p  0
 x_1_6_0  c_1_0  1               mark_3_4_1  'MARKER'  'INTORG'       rhs  c_3_5_1n  0
 x_1_6_0  c_6_0  1               y_3_4_1  obj  -1  c_3_4_1p  -1       rhs  c_0_2  1
 x_1_6_0  c_0  1                 y_3_4_1  c_3_4_1n  -1               rhs  c_1_2  1
 x_1_5_0  c_1_0  1               mark_3_4_1_X  'MARKER'  'INTEND'     rhs  c_2_2  1
 x_1_5_0  c_5_0  1               x_3_5_1  c_3_1  1                    rhs  c_3_2  1
 x_1_5_0  c_0  1                 x_3_5_1  c_5_1  1                    rhs  c_4_2  1
 x_1_5_0  c_1_5_0n  1            x_3_5_1  c_1  1                      rhs  c_5_2  1
 mark_1_5_0  'MARKER'  'INTORG'  x_3_5_1  c_3_5_0p  1                 rhs  c_6_2  1
 y_1_5_0  obj  -1  c_1_5_0p  -1  x_3_5_1  c_3_5_1n  1                 rhs  c_7_2  1
 y_1_5_0  c_1_5_0n  -1  ->       mark_3_5_1  'MARKER'  'INTORG'       rhs  c_2  4
                                 y_3_5_1  obj  -1  c_3_5_1p  -1 ->   ENDATA
```

The random problem implementation

We have implemented a code to generate a random problem using the rules of Section 5.3.12. A dynamic link library is generated using the implementation such that each algorithm implementation can easily access the random generation classes. Also easy conversion to each file format is possible with a proper function call. The following `RandomProblem` is the most important class for the random problem generation.

```
class RandomProblem
{
private:
    int nW, nJ, nT;
    int mu, sigma;
    bool onlypmatching;

    int *nTotalAbleJobs;
    int **nNumberOfAbleJobs;
    int ***Ablejob;

    int **nCommonJobs;
    int ***CommonJob;
public:
    RandomProblem(int nWorkers,int nJobs,int nTimeShifts, int dense);
    void WritetoSGR(char* filename);
    void WritetoCGP(char* filename);
    void WritetoMPS(char* filename, int mn, bool integer);

    void InterSection();
    int URandom(int lownumber, int number);
    int NRandom1(int number);
    int PRandom(int number)
```

```
    int NRandom2(int number);
    int NRandom3(int number, int dense);

    void RandomSeq(int vnumber, int selectednumber, int*&select);
    int Sum(int*&iarray, int arraylength);
    bool CheckPerfect1(int t);
    float  Rec_determinant(float *Mat, int Dimension);
    ~RandomProblem();
}
```

The constructor has the following arguments,

- `nWorkers`: Number of workers (or cardinality of left node sets).
- `njobs`: Number of jobs (or cardinality of right node sets).
- `nTimeShifts`: Number of time shifts (or total number of graphs).
- `dense`: The parameter controls the density the edges in the bipartite graphs. It can be from 1 to 100, where 1 denotes the most dense graph, while larger values outputs sparser graphs.

Appendix B

Augmenting cycle method: Implementation details

This appendix deals with the implementation issues of the *augmenting cycle method* in Chapter 5. A brief discussion about implementing graph algorithms is given in Section 6.5.3. Here we present various classes in the implementation, but only the important ones.

The class `Node`

The class `Node` is usedto represent nodes of a general graph. The basic node operations are implemented in this class.

```
class Node{
private:
    char* cname;
    int  iname;
    float weight;
public:
    int index, label, number;
    Node();
    void Setcname(char* name);
    void Setiname(int name);
    char* Getcname();
    int  Getiname();
    void init();
    bool operator == (Node &rhs);
    void Setweight(float w);
```

```
    float Getweight();
    ~Node();
};
```

The class BEdge

The class BEdge is used to represent an edge of a graph. The basic edge operation are implemented in this class.

```
class BEdge {
private:
    Node source;
    Node target;
    int iname;
    float weight;
    bool matched;
public:
    BEdge();
    void InitBEdge(int name, float w, Node S, Node T);
    void Setiname(int name);
    void Setweight(float w);
    void  SetSource(Node& s);
    void SetTarget(Node& t);
    int Getiname();
    float Getweight();
    Node  GetSource();
    Node  GetTarget();
    void SetMatched(bool flag);
    bool GetMatched();
    bool operator == (BEdge &rhs);
    ~BEdge();

};
```

The class Directed Graph

The class Directed Graph is used to represent a directed graph. It constructs a directed graph with a given bipartite graph and a matching. The modified *Floyd Warshall* algorithm to find a negative cycle is implemented in this class. The function outputs a negative cycle in the weighted graph if one exists. We used a modification of the *Floyd-Warshall* algorithm to find the all pairs shortest path in a directed/undirected network.

```
class DirectedGraph{
private:
    int nodecount;
    Node *node;
    int edgecount;
    BEdge *edge;
public:
    DirectedGraph();
    DirectedGraph(BGraph&B, BMatching&M);
    bool FloydWarshallCycle(int& cyclestart, int&cyclelength,
                                              int*& negcycle);
    void    DirectedGraphDisplay(int gn);
    void DOutput(ostream&out, int gn);
    ~DirectedGraph();
};
```

The class BipariteGrpah

This is one of the main classes of the implementation used to represent a bipartite graph. The edges are represented by adjacency list. The class also contains modules for solving the maximum cardinality matching problem and other utility functions.

```
class BipartiteGraph {
protected:
    int leftnodecount, rightnodecount, edgecount, maxmatchcount;
    Node* leftnode;
    Node* rightnode;
    BEdge* edge;
    BMatching M;
public:
    BipartiteGraph();
    BipartiteGraph(const char* infilename);
    int LeftNodeCount();
    int RightNodeCount();
    int EdgeCount();
    BEdge GetEdge(int edgeindex);
    int  GetEdgeIndex(int intname);
    int CheckEdgeExist(BEdge&checkedge);
    BMatching GetBMatching();
    void WeightsUpdate1(BMatching&matching,int&commonedgecount);
    void WeightsUpdate2(BMatching&matching1,BMatching&matching2,
```

```
                                                int&commonedgecount);
    Cycle FindCycle(int j);
    void MatchingUpdate(int& cyclestart,
                            int &cl, int*& negcycle,ofstream&out);
    void MatchImproved();
    void BGraphDisplay(int num);
    void GetNeighbours(Node& thisnode,bool left,
                            int neighbourscount,Node*&neighbour);
    int CountNeighbours(Node& thisnode, bool left);
    bool CheckEdgeExist(BEdge checkedge);
    void Augument();//Augmenting with Cycle and Matching
    Node GetNode(int i, bool left);
    BEdge  GetEdge2(int edgename);
    int CommonEdgeCount(BipartiteGraph& target);
    void SetMatching(BMatching&M1);
    ostream&operator>>(ostream&out);
    void Output(ostream&out,int num);
    void MatchingSolve(int verbose);
};
```

The class BMatching

This class represents a matching in the bipartite graph. The edges are represented by an adjacency list. The class also contains modules for solving the maximum cardinality matching problem and other utility functions.

```
class BMatching {
private:
    BEdge* matchedge;
    int maxmatchcouNt;
    bool matched;
public:
    BMatching();
    BMatching(BMatching& BM);
    void CopyMatching(BMatching& BM);
    void SetMatchNumber(int matchcount);
    BEdge  GetMatchBEdge(int matchindex);
    void  SetMatchBEdge(int matchindex,BEdge  match);
    void SetMaxmatchCount(int mmc);
    int   GetMaxmatchCount();
    bool InsertEdge(BEdge&insedge);
    void DiplayMatch(int gn);
```

```
    bool EdgeExist(int edgeiname);
    int CheckEdgeExist(BEdge&checkedge);
    void SetMatchedgeWeight(int matchindex, float w);
    bool DeleteEdge(BEdge deledge);
    Node GetNeibhbour(Node s, bool left);
    int GetNeighbour(int s, bool left);
    void Output(ostream&out,int gn);
    ~BMatching();
};
```

The class `AugCycleMethod`

The class `AugCycleMethod` constructs an instance of the *sequential matching problem*. While initializing it solves the maximum cardinality matching problem in the bipartite graphs to input initial matching. The initial matching algorithm is the standard augmenting path algorithm. We tested with a faster algorithm using the network model which is implemented using TURBO (See Appendix C, and Section 1).

```
class AugCycleMethod {
private:
    BGraph *G;
    int commonedgecount;
    ofstream output;
public:
    AugCycleMethod();
    void InitializeGraphs(char* arg1,int gn);
    void SolveProblem(ofstream&output, int gn);
    void Display(int gn);
    void CommonEdgeCount(int count);
    int GetCommonEdgeCount();
    void SetCommonEdgeCount(int count);
    void ConvertFromTurbo(const char* sourcefile,
                          const char* targetfile);
    void FileReadFromSerialFile(const char* seriesfile,
                          int gindex, const char* outputfile);
    ~AugCycleMethod();
};
```

Appendix C

Matching and randomized algorithm implementation

This appendix deals with our implementation issues of a general matching algorithms in 1.2.2 (network and augmenting), and the randomized algorithm in Section 2.

Randomized Algorithm classes

We use TURBO: graph data structures and algorithms, for the implementation of the randomized method in Chapter 3. We briefly present the important classes for the implementation, but most options are self-explanatory.

The class `MyNode`

The class `MyNode` is derived from the TURBO class `Hnode`, to represent the nodes of a bipartite graph.

```
class MyNode : public Hnode,
               public obj_ind {
public:
  int name;
  int x;
  int y;
  static int MyNode::init_index();
  inline MyNode() : name(0), x(0), y(0) {
```

```
  }
}
```

The class MyEdge

The class MyEdge is derived from the TURBO classHedge, to represent the edges of a bipartite graph.

```
 class MyEdge : public Hedge { public:
  int matched;
  int selected;
  int intersection;
  color line_color ;
  void print(char* = "");
  inline MyEdge() : matched(0) {
  }
  bool compare(MyEdge b1); };
```

The class MyGraph

The class MyGraph is derived from the TurboGraph, where it is taken from the template provided by TURBO.

```
typedef hgraph<MyNode,MyEdge> TurboGraph;
```

```
const class MyGraph : public TurboGraph { protected:
 int* randommatrix;
 float* randommatrix_inv;
 int numberofnodes,  determinant, new_numberofnodes;
 int halfnodes;
 int *degree;
public:
    int *degree;
    GeneralMatrix* Matrix;
    int matched;
    int* matchedname;
    inline MyGraph(): TurboGraph(undirected){}
    void initmatrix(int value);
    void MakeRandomMatrix(int seed);
    void AssignMatrix();
```

```
    bool CheckMatched();
    bool CheckDeleted(int i_ind, int j_ind);
    Reduce_Degree(int st_vertex_name, int end_vertex_name)
    void MatrixReduce(int i_ind, int j_ind);
    void displaymatrix();
    void InverseRandomMatrix();
    void writematrix_file(const char* matrixfilename);
    float Determinant();
    bool allowed(int s, int t);
    void MakeInterZero();
    void Unselect();
    void MatchOne();
    void IntersectionMatch();
};
```

The other important subroutine is

`void IntersectionOfGraphs(int start_ind, int end_ind)`

which used to evaluate the intersection as bipartite graphs as in 3.4.3, with given `start_ind` and `end_ind`.

Matching algorithm Implementation

Here we present two main classes from the augment matching algorithm for a bipartite graph. The main idea of the algorithm is described in 1.

The class `BGraph`

The most important class of the method derived from `BipartiteGraph`, which is described in the Appendix B.

```
class BGraph: public BipartiteGraph {
protected:
    int NumberofWorkers, NumberofJobs;
    Node *workers, *jobs;
    bool *adj;
    int *mate, *exposed;
public:
```

```
    BGraph();
    void initializefromfile(const char* infilename);
    void initdisplay();
    void MatchingSolve(int verbose);
    void augument(int vertexnumber);
    void resultdisplay();
    void displaycurrentmatching(int itnumber);
    ~BGraph();

};
```

The class `DirectedGraph`

This class is for the construction of the directed graph, which arises in the subproblem of finding alternate paths in the augmenting method.

```
class DirectedGraph {
private:
    int nodecount;
    Node *node;
    int edgecount;
    BEdge *edge;
public:
    DirectedGraph();
    DirectedGraph(BGraph&B, BMatching&M);
    void    DirectedGraphDisplay(int gn);
    void DOutput(ostream&out, int gn);
    ~DirectedGraph();
};
```

A faster method for the bipartite graph matching problem (ref. 1.2.2), has been implemented using the network flow algorithm provided by TURBO.

Bibliography

[1] Ahuja R. K., Magnanti T. L., and Orlin J. B. *Networks flows.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[2] K. Papadimitriou, C. H. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[3] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and sons, Chichester, 1988.

[4] E. de Klerk. Bipartite matchings.

[5] R. M Hopcroft, J. E. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 4:225–231, 1973.

[6] Alt H., Blum N., Mehlhorn K., and Paul M. Computing a maximum cardinality matching in a bipartite graph in time $\mathcal{O}(n^{1.5}\sqrt{m/\log n})$. *Inform. Process. Lett.*, 37:237–240, 1991.

[7] J. C. Setubal. New experimental results for bipartite matching. Technical report, Relatóoorio Técnico DCC, 1992.

[8] W. Derigs, U. Meier. Implementing goldberg's maxflow algorithm a computational investigation. *Methods and Models of Operations Research*, 33:383–403, 1989.

[9] A. Schrijver. *Theory of Linear and Integer Programming.* JohnWiley & Sons, Chichester, 1986.

[10] M Lovász, L. Plummer. *Matching Theory.* Akadémiai Kiadó, Budapest, Hungary, 1986.

[11] T. Tutte. The factorization of linear graphs. *J. London Math. Soc.*, 22:107–111, 1947.

[12] L Lovász. On determinants, matchings and random algorithms. *Fundamentals of Computation Theory*, pages 565–574, 1979.

[13] J. Cheriyan. Randomized $o(m(\mid v \mid))$ algorithms for problems in matching theory. *SIAM J. Comput.*, 26:1635–1655, 1997.

[14] U. V. Rabin, M. O. Vazirani. An $\mathcal{O}(\sqrt{\mid v \mid} \mid e \mid)$ algorithm to find maximum matching in general graphs through randomization. *J. Algorithms*, 10:105–113, 1989.

[15] Lübbecke Marco. *Engine scheduling by column generation.* PhD thesis, Technischen Universität Braunschweig, Braunschweig, 2001.

[16] Barnhart C., Johnson E.L., Nemhauser G.L., Savelsbergh M.W.F., and Vance P.H. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, May-June 1998.

[17] Chvátal. *Linear Programming.* W.H. Freeman and Company, New York, 1983.

[18] P. Dantzig, G. B. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101111, 1960.

[19] Universität zu Köln, Köln. *ABACUS A Branch And CUt System, Version 2.0, User's Guide and Reference manual*, 1996.

[20] S. Thienel. *ABACUS A Branch And CUt System.* PhD thesis, Universität zu Köln, Köln, 1995.

[21] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms, Third Edition.* Addison-Wesley, Massachusetts, 1997.

[22] ILOG, Gentily Cedex. *ILOG CPLEX 6.5 Reference Manual*, 1999.

Lebenslauf

Persönliche Daten

Name:	Sureshan Karichery
Adresse:	Oberölkofenerstr. 4, 81671, München
Geburtsdatum:	25. Mai 1974
Geburtsort:	Perumbala, Kerala, Indien
Familienstand:	ledig
Staatsangehörigkeit:	indisch

Ausbildung/Wehrdienst/Studium[1]

1980-90	SSLC	Government High School Udma
1990-92	Pre-Degree	Calicut University
1992-96	B.Sc. Mathematik	Calicut University
1996-98	M.Sc. Mathematik	Cochin University of Science and Technology
1999-01	M.Tech. in IMSC	Indian Institute of Technology Madras
2000-01	M.Tech. Thesis	Technische Universität Kaiserslautern/ITWM
2001-04	Doctorand	Universität zu Köln/Siemens AG

[1] SSLC: Secondary School Leaving Examination
B.Sc.: Bachelor of Science
M.Sc. Master of Science
M.Tech.: Master of Technology
IMSC: Industrial Mathematics and Scientific Computing
ITWM: Institut für Techno und Wirtschaftsmathematik

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 noch nicht veröffentlicht worden ist sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen dieser Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Professor Dr. R. Schrader betreut worden.