Introduction to ABACUS—A Branch-And-CUt System

Michael Jünger and Stefan Thienel*

February 1997

Abstract

The software system ABACUS is an object-oriented framework for the implementation of branch-and-cut and branch-and-price algorithms. This paper shows the basics of its application to combinatorial and mixed integer optimization problems.

Keywords: Mixed Integer Programming, Branch-and-Cut, Branch-and-Price, Software Systems

1 Introduction

ABACUS—A Branch-And-CUt System is an object oriented framework for the implementation of branch-and-cut algorithms, branch-and-price algorithms, and their combination. While classical frameworks (e.g., Cplex [3], MINTO [8], OSL [5]) focus on general mixed integer optimization problems, ABACUS is also designed for combinatorial optimization problems. In addition to the general mixed integer optimization problem [10], ABACUS has already been successfully applied to the traveling salesman problem [10], the binary cutting stock problem [10], the feedback vertex set problem [4], the multiple sequence alignment problem [9], and the betweenness problem [2]. Other applications are currently under development at various research sites.

This paper should give the reader the basic ideas of the capabilities and the usage of ABACUS, and encourage its application. A comprehensive description of the design of ABACUS can be found in [7, 10]. Further details on the use of the software systems are given in [11, 12]. For the description of branch-and-cut and branch-and-price algorithms we refer to [1, 6, 10].

Section 2 surveys the features of ABACUS. In Section 3 we show how ABACUS can be used for the implementation of a branch-and-cut or branch-and-price algorithm. More advanced parts of ABACUS are briefly sketched in Section 4.

2 Features

ABACUS is a software system written in the programming language C++ with the following features.

*Partially supported by ESPRIT LTR Project no. 20244 - ALCOM-IT. Addresses of the authors: Institut für Informatik, Universität zu Köln, Pohligstr. 1, D-50969 Köln, E-mail: {mjuenger, thienel}@informatik.uni-koeln.de

2.1 Algorithms

ABACUS provides a branch-and-bound-algorithm using linear programming relaxations. The optimization of the subproblems of the branch-and-bound tree is performed by a cutting plane and/or a column generation algorithm. The separation of cutting planes and the pricing of variables can be optionally added by the user.

2.2 Optimization Problems

Solvers for both mixed integer and combinatorial optimization problems can be implemented based on ABACUS. Combinatorial optimization problems are usually formulated with constraints and variables containing a lot of structural information, e.g., represent special subgraphs or describing a partial crew scheduling. This important information is lost as soon as the variables and constraints are converted into a matrix. Therefore, ABACUS provides an abstract representation of constraints and variables and performs the conversion to matrix form internally.

2.3 Branching

ABACUS supports classical branching on a binary or integer branching variable and on branching constraints. Moreover, ABACUS provides a simple branching scheme that should support the implementation of almost any other branching technique. Also non-binary branch-and-bound trees are possible.

2.4 Linear Programs

ABACUS contains an interface to the linear program that is independent of the LP-solver. ABACUS 1.2 supports only Cplex 2.2 [3] or newer versions. Further linear programming solvers will be supported in the future.

2.5 Pools

Any constraint and variable, either used in the initialization or dynamically generated, is stored in a pool. ABACUS provides a default pool concept that should be sufficient for many applications. However, it is possible to adapt the pools to the needs of the application.

2.6 Strategies

Various strategies for the selection of the branching variable, the elimination of constraints or variables, the frequency of the separation or pricing, the fixing of variables, etc., are implemented in ABACUS. These system strategies can be controlled with various parameters or can be replaced by problem specific strategies in a simple way.

3 First Steps

ABACUS is a collection of C++ classes. A new branch-and-cut or branch-and-price algorithm is implemented by deriving problem specific classes from some base classes. Usually, only four base classes of ABACUS are involved: VARIABLE, CONSTRAINT, MASTER, and SUB.

The class VARIABLE is the base class for any variable used in an application while any constraint has to be derived from the base class CONSTRAINT. The class MASTER contains "global" data and controls the branch-and-bound algorithm. A class derived from MASTER can, e.g., store the data of a problem instance. Finally, the class SUB represents a subproblem of the branch-and-bound tree. It implements the backbone of the cutting plane or column generation algorithm. By deriving a class from the class SUB, problem specific separation and pricing algorithms can be added.

In general, problem specific functions are added by the definition of virtual functions in derived classes. Figure 1 shows how the problem specific classes MYCONSTRAINT, MYVARIABLE, MYMASTER, and MYSUB are derived from their base classes.

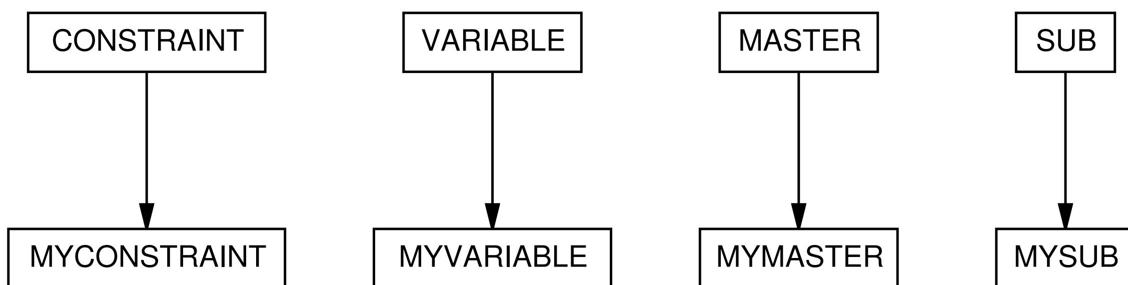

Figure 1: Deriving problem specific classes.

3.1 Constraints and Variables

The first step in the implementation of a new application is the analysis of its variable and constraint structure. We require at least one constraint class derived from the class CONSTRAINT and at least one variable class derived from the class VARIABLE. Since variables and constraints of combinatorial optimization problems contain a lot of structure, the classes derived from the classes CONSTRAINT and VARIABLE do not have to represent the support and coefficients in some sparse vector form. The transformation of constraints and variables to a matrix form is performed internally using the pure virtual functions CONSTRAINT::coeff(VARIABLE *v) and VARIABLE::coeff(CONSTRAINT *c).

We derive from the class VARIABLE the class MYVARIABLE storing the attributes specific to the variables of our application, e.g., its number, or the tail and the head of the associated edge of a graph.

```
class MYVARIABLE : public VARIABLE {
  public:
    virtual double coeff(CONSTRAINT *c);
    // other members
};
```

Then we derive the class MYCONSTRAINT from the class CONSTRAINT

```
class MYCONSTRAINT : public CONSTRAINT {
  public:
    virtual double coeff(VARIABLE *v);
    // other members
};
```

The function `CONSTRAINT::coeff(VARIABLE *v)` is a pure virtual function. Hence, we define it in the class `MYCONSTRAINT`. It returns the coefficient of variable `v` in the constraint. Usually, we need in an implementation of the function `MYCONSTRAINT::coeff(VARIABLE *v)` access to the application specific attributes of the variable `v`. Therefore, we have to cast `v` to a pointer to an object of the class `MYVARIABLE` for the computation of the coefficient of `v`.

The function `CONSTRAINT::coeff()` is used within the framework when the row format of a constraint is computed, e.g., when the linear program is set up, or a constraint is added to the linear program. When the column associated with a variable is generated, then the virtual member function `VARIABLE::coeff()` is used.

ABACUS is not restricted to a single constraint/variable pair within one application. There can be an arbitrary number of constraint and variable classes. It is only required that the coefficients of the constraint matrix can be safely computed for each constraint/variable pair.

3.2 The Master

There are three main reasons why we require a problem specific master of the optimization. First, we have to embed problem specific data members like the problem formulation. Second, the initial constraint and variable systems have to be set up. Third, the initialization of the first subproblem has to be performed, i.e., the root node of the branch-and-bound tree has to be initialized with a subproblem of the class `MYSUB`.

Therefore, a problem specific master has to be derived from the class `MASTER`:

```
class MYMASTER : public MASTER
{
  // members of the class
};
```

3.2.1 The Constructor

Usually, the input data are read from a file by the constructor or they are specified by the arguments of the constructor. The following example of a constructor for the class `MYMASTER` sets up the master of a branch-and-cut algorithm for a minimization problem.

```
MYMASTER::MYMASTER(const char *problemName) :
  MASTER(problemName, true /* cutting */, false /* pricing */, OPTSENSE::Min)
{
  // read the data from the file problemName
}
```

3.2.2 Initialization of the Constraints and Variables

The constraints and variables that are not generated dynamically, e.g., the degree constraints of the traveling salesman problem or the constraints and variables of the problem formulation of a general mixed integer optimization problem, have to be set up and inserted in pools in the function `MYMASTER::initializeOptimization()`. This virtual function is called when the optimization is started.

By default, ABACUS provides three different pools: one for variables and two for constraints. The first constraint pool stores the constraints that are not dynamically generated and with which the first LP-relaxation of the first subproblem is initialized. The second constraint pool is empty at the beginning and is filled up with dynamically generated cutting planes. In general, ABACUS provides a more flexible pool concept, but for many applications the default pools are sufficient.

After the initial variables and constraints are generated they have to be inserted into the default pools by calling the function

```
virtual void MASTER::initializePools(
                           BUFFER<CONSTRAINT*> &constraints,
                           BUFFER<VARIABLE*>   &variables,
                           int                  varPoolSize,
                           int                  cutPoolSize,
                           bool                 dynamicCutPool = false);
```

A `BUFFER` is a generic ABACUS class that is very similar to an array. Here, `constraints` are the initial constraints, `variables` are the initial variables, `varPoolSize` is the initial size of the variable pool, and `cutPoolSize` is the initial size of the cutting plane pool. The size of the variable pool is always dynamic, i.e., this pool is increased if required. By default, the size of the cutting plane pool is fixed, but it becomes dynamic if the argument `dynamicCutPool` is `true`.

The function `initializeOptimization()` can be also used to determine a feasible solution by a heuristic such that the primal bound can be initialized.

Hence, the function `initializeOptimization()` could look as follows, under the assumption that the functions `nVar()` and `nCon()` are defined in the class `MYMASTER` and return the number of variables and the number of the constraints, respectively. In the example we initialize the size of the cut pool with `2*nCon()`.

After the pools are set up the primal bound is initialized with the value of a feasible solution returned by the function `MYMASTER::myHeuristic()`. While the initialization of the pools is mandatory, the initialization of the primal bound is optional.

```
void MYMASTER::initializeOptimization()
{
  BUFFER<VARIABLE*> variables(this, nVar());
  for (int i = 0; i < nVar(); i++)
    variables.push(new MYVARIABLE(/* arguments of constructor */));
  BUFFER<CONSTRAINT*> constraints(this, nCon());
  for (i = 0; i < nCon(); i++)
    constraints.push(new MYCONSTRAINT(/* arguments of constructor */));
  initializePools(constraints, variables, nVar(), 2*nCon());
  primalBound(myHeuristic());
```

```
}
```

3.2.3 The First Subproblem

The root of the branch-and-bound tree has to be initialized with an object of the problem specific subproblem class `MYSUB`, which is derived from the class `SUB`. This initialization must be performed by a definition of the pure virtual function `firstSub()`, which returns a pointer to the first subproblem. In the following example we assume that the constructor of the class `MYSUB` for the root node of the enumeration tree has only a pointer to the associated master as argument.

```
SUB *MYMASTER::firstSub()
{
  return new MYSUB(this);
}
```

3.3 The Subproblem

Finally, we have to derive a problem specific subproblem from the class `SUB`:

```
class MYSUB : public SUB
{
  // members of the class
};
```

Besides the constructors only two pure virtual functions of the base class `SUB` have to be defined in any application, which check if a solution of the LP-relaxation is a feasible solution of the mixed integer optimization problem, and generate the sons after a branching step, respectively. Moreover, the main functionality of the problem specific subproblem is to enhance the branch-and-bound algorithm optionally with dynamic variable and constraint generation and LP-based heuristics.

3.3.1 The Constructors

The class `SUB` has two different constructors: one for the root node of the branch-and-bound-tree and one for all other subproblems. This differentiation is required as the constraint and variable set of the root node can be initialized explicitly, whereas for the other nodes this data is copied from the father node and possibly modified by a branching rule. Therefore, we have to implement these two constructors for the class `MYSUB`.

The numbers in the following example of a root node constructor give the amount of additional memory that should be allocated for dynamically generated constraints and variables. Good estimations of these values reduce memory consumption and run time for reallocation. However, an incorrect estimation cannot cause a run error since ABACUS automatically performs reallocations for all internal memory.

```
MYSUB::MYSUB(MYMASTER *master) :
  SUB(master,
      50.0, // additional space for constraints
      0.0,  // additional space for variables
```

```
        100.0 // additional space for nonzero coefficients
      )
  { }
```

With further optional arguments of the root node constructor it is also possibly to initialize explicitly the first solved linear program. If these arguments are omitted as in our example the linear program is initialized with the constraints and variables stored in the pools (see Section 3.2.2).

While there are some alternatives for the implementation of the root node constructor, the constructor of the non-root nodes has usually the same form for all applications, but might be augmented with some problem specific initializations.

```
  MYSUB::MYSUB(MASTER *master, SUB *father, BRANCHRULE *branchRule) :
    SUB(master, father, branchRule)
  { }
```

The class BRANCHRULE defines the modification of the father subproblem in order to generate the sons. As long as standard branching is applied, the user does not have to care about this class.

In the function MYMASTER::firstSub() the root node constructor of the class MYSUB has to be called (see Section 3.2.3). The constructor for non-root nodes must be applied in the function MYSUB::generateSon() (see Section 3.3.3).

3.3.2 The Feasibility Check

After the LP-relaxation is solved, ABACUS has to check if the optimum LP-solution is a feasible solution of our optimization problem. Therefore, we have to define the pure virtual function feasible() in the class MYSUB, which should return true if the LP-solution is a feasible solution of the optimization problem, and false otherwise:

```
  bool MYSUB::feasible()
  {
    if (/* LP-solution is feasible */) return true;
    else                               return false;
  }
```

If all constraints of the integer programming formulation are present in the LP-relaxation, then the LP-solution is feasible if all discrete variables have integer values. This check can be performed by calling the member function SUB::integerFeasible():

```
  bool MYSUB::feasible()
  {
    return integerFeasible();
  }
```

If the LP-solution is feasible and its value is better than the primal bound, then ABACUS automatically updates the primal bound.

3.3.3 The Generation of the Sons

Like the pure virtual function `firstSub()` of the class `MASTER`, which generates the root node of the branch-and-bound tree, we need a function generating a son of a subproblem. This function is required as the nodes of the branch-and-bound tree have to be identified with a problem specific subproblem of the class `MYSUB`. This is performed by the pure virtual function `SUB::generateSon()`, which calls the constructor for a non-root node of the class `MYSUB` and returns a pointer to the newly generated subproblem. If the constructor for non-root nodes of the class `MYSUB` has the same arguments as the corresponding constructor of the base class `SUB`, then the function `generateSon()` can have the following form:

```
SUB *MYSUB::generateSon(BRANCHRULE *rule)
{
  return new MYSUB(master_, this, rule);
}
```

This function is automatically called during a branching process. If the built-in branching strategies are used, we do not have to care about the generation of the branching rule `rule`.

3.3.4 A Branch-and-Bound Algorithm

The two constructors, the function `feasible()`, and the function `generateSon()` must be implemented for the subproblem class `MYSUB` of every application. As soon as these functions are available, a branch-and-bound algorithm can be performed. All other functions of the class `MYSUB` that we are going to explain now, are optional in order to improve the performance of the implementation.

3.3.5 The Separation

By redefinig the virtual function `SUB::separate()` problem specific cutting planes can be generated.

```
int MYSUB::separate()
{
  // perform separation and return number of new constraints
}
```

We distinguish between the separation from scratch and the separation from a constraint pool. Newly generated constraints have to be added by the function `addCons()`. Constraints generated in earlier iterations that have been become inactive in the meantime might still be contained in the cut pool. These constraints can be regenerated by calling the function `constraintPoolSeparation()`, which adds the constraints to the buffer without an explicit call of the function `addCons()`.

A very simple separation strategy is implemented in the following example of the function `separate()`. Only if the pool separation fails, we generate new cuts from scratch. The function `mySeparate()` performs here the application specific separation. If more cuts are added with the function `addCons()` than there is space in the internal buffer for cutting planes, then the redundant cuts are discarded. The function `addCons()` returns the number of actually added cuts.

```
int MYSUB::separate()
{
```

```
  int nCuts = constraintPoolSeparation();
  if (!nCuts) {
    BUFFER<CONSTRAINT*> newCuts(master_, /* size of the buffer */);
    nCuts = mySeparate(newCuts);
    if (nCuts) nCuts = addCons(newCuts);
  }
  return nCuts;
}
```

If not all constraints of the integer programming formulation are active, and all discrete variables have integer values, then the solution of a separation problem might be required to check the feasibility of the LP-solution. In order to avoid a redundant call of the same separation algorithm later in the function `separate()`, constraints can be already added here by the function `addCons()`.

In the following example of the function `feasible()` the separation is even performed if there are discrete variables with fractional values such that the separation routine does not have to be called a second time in the function `separate()`.

```
bool MYSUB::feasible()
{
  BUFFER<CONSTRAINT*> newCuts(master_, /* size of the buffer */);
  bool feasible;

  if (integerFeasible()) feasible = true;
  else                   feasible = false;
  int nSep = mySeparate(newCuts);
  if (nSep) {
    feasible = false;
    addCons(newCuts);
  }
  return feasible;
}
```

3.3.6 Pricing out Inactive Variables

The dynamic generation of variables is performed very similarly to the separation of cutting planes. Here, the virtual function `SUB::pricing()` has to be redefined. We illustrate the redefinition of the function `pricing()` by an example that is an analogon to the example given for the function `separate()`.

```
int MYSUB::pricing()
{
  int nNewVars = variablePoolSeparation();
  if (!nNewVars) {
    BUFFER<VARIABLE*> newVariables(master_, /* size of the buffer */);
    nNewVars = myPricing(newVariables);
    if (nNewVars) nNewVars = addVars(newVariables);
```

```
  }
  return nNewVars;
}
```

3.3.7 Heuristics

After the LP-relaxation has been solved in the subproblem optimization, ABACUS calls the virtual function `SUB::improve()`. Again, the default implementation does nothing, but in a redefinition in the derived class `MYSUB` application specific heuristics can be inserted:

```
int MYSUB::improve(double &primalValue)
{
  // perform heuristic, store its value in primalValue
  // return 1 for better solution, 0 otherwise
}
```

3.4 Starting the Optimization

After the problem specific classes are defined as discussed in the previous sections, the optimization can be performed with the following main program. We suppose that the only parameter of the constructor of class `MYMASTER` is the name of the input file.

```
#include "mymaster.h"

int main(int argc, char **argv)
{
  MYMASTER master(argv[1]);

  MYMASTER::STATUS status = master.optimize();
  if (status == MASTER::Optimal) return 0;
  else                           return 1;
}
```

4 Advanced Steps

Section 3 shows how easily a branch-and-cut or branch-and-price algorithm can be implemented with ABACUS. However, ABACUS contains more features that we briefly want to sketch in this section.

4.1 Other Pools

For a better control of the pool separation it can be advantageous to have more pools than ABACUS provides in its default pool concept (e.g., a pool for every constraint type). In this case each pool should become a member of the class `MYMASTER`. The pool in which a constraint is later inserted is an optional additional argument of the function `SUB::addCons()`.

4.2 Compression of Constraints and Variables

Sometimes constraints and variables can be stored in a "compressed" format using a very small amount of memory. However, such a format can be very inefficient for the repeated computation of coefficients in row/column format by the function `CONSTRAINT::coeff()` and `VARIABLE::coeff()`, respectively. Therefore, ABACUS supports the management of a second "expanded" format for fast repeated coefficient computation.

4.3 Enumeration Strategy

The enumeration strategies best-first search, depth-first search, breadth-first search and a diving strategy are implemented in ABACUS. Problem specific strategies can be added be redefining a virtual function comparing two subproblems.

4.4 Selection of the Branching Variable

The default branching variable selection strategy can be changed by the redefinition of the virtual function

```
int SUB::selectBranchingVariable(int &variable);
```

in a class derived from the class `SUB`. If a branching variable is found it has to be stored in the argument `variable` and the function should return 0. If the function fails to find a branching variable, it should return 1. Then, the subproblem is automatically fathomed.

4.5 Other Branching Strategies

For the implementation of different branching strategies we have introduced the concept of branching rules in the class `BRANCHRULE`. A branching rule defines the modifications of a subproblem in order to generate one of its sons. The virtual function

```
int SUB::generateBranchRules(BUFFER<BRANCHRULE*> &rules);
```

returns 0 if it can generate branching rules and stores for each subproblem that should be generated, a branching rule in the buffer `rules`. If no branching rules can be generated this function returns 1 and the subproblem is fathomed.

4.6 Calling ABACUS Recursively

The separation or pricing problem in a branch-and-bound algorithm can be again a mixed integer optimization problem. In this case, it might be appropriate to solve this problem also with an application of ABACUS. Due to its object oriented design there can be an arbitrary number of ABACUS optimizers, i.e., objects of classes derived from the class `MASTER`.

4.7 Fixing and Setting Variables by Logical Implications

ABACUS performs fixing and setting variables by reduced cost criteria automatically. Fixing and setting variables by logical implications can be performed by the redefinition of a virtual function.

4.8 Parameters

A configuration file allows the modification of system parameters for

- the enumeration strategy,
- the quality of the solution,
- the height of the enumeration tree,
- the cpu time limit,
- the elapsed time limit,
- the integrality assumption of the objective function,
- the tailing off control,
- the delayed branching,
- the amount of output written to the standard output device and the log file,
- the initialization of the primal bound,
- the frequency of pricing in branch-and-cut-and-price algorithms,
- the frequency of cutting plane generation,
- the fixing and setting of variables by reduced cost,
- the output of the linear programs,
- the pricing strategies of the LP-solver Cplex,
- the amount of output of the LP-solver Cplex,
- the number of buffered and added constraints and variables,
- the iteration limit of the cutting plane algorithm,
- the elimination of fixed and set variables,
- the reoptimization of new root nodes of the remaining branch-and-bound tree,
- the elimination of variables and constraints.

Moreover, ABACUS provides an easy to use concept for the implementation of application parameter files.

4.9 Further Tuning

Although the default strategies of ABACUS proved good performance in various branch-and-cut and branch-and-price algorithms further tuning might be required for special applications. Since most ABACUS functions are virtual functions a substitution by a problem specific implementation can be performed easily.

5 Availability

ABACUS can be obtained from:

`http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus.html`

References

[1] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for huge integer programs. Technical report, Georgia Institute of Technolgy, 1995.

[2] Thomas Christof, Michael Jünger, John Kececioglu, Petra Mutzel, and Gerhard Reinelt. A branch-and-cut approach to physical mapping with end-probes. In *First Annual International Conference on Computational Molecular Biology*. ACM, 1997.

[3] Cplex. *Using the Cplex Callable Library*. Cplex Optimization, Inc, 1995.

[4] Meinrad Funke and Gerhard Reinelt. A polyhedral approach to the feedback vertex set. In W.H. Cunningham, S.T. McCormick, and M. Queyranne, editors, *Integer Programming and Combinatorial Optimization (5th International IPCO Conference, Vancouver, Canada, June 1996, Proceedings)*, volume 1084 of *Lecture Notes in Computer Science*, pages 445–459. Springer Verlag, 1996.

[5] IBM Corporation. *Optimization Subroutine Library - Guide and Reference, Release 2.1*, 1995.

[6] Michael Jünger, Gerhardt Reinelt, and Stefan Thienel. Practical problem solving with cutting plane algorithms in combinatorial optimization. In Willian Cook, Lázló Lovász, and Paul Seymour, editors, *Combinatorial Optimization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 111–152. American Mathematical Society, 1995.

[7] Michael Jünger and Stefan Thienel. The design of the branch-and-cut system ABACUS. Technical report, Universität zu Köln, 1997.

[8] G.L. Nemhauser, M.W.P. Savelsbergh, and G.C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[9] Knut Reinert, Hans-Peter Lenhof, Petra Mutzel, Kurt Mehlhorn, and John D. Kececioglou. A branch-and-cut algorithm for multiple sequence alignment. In *First Annual International Conference on Computational Molecular Biology*. ACM, 1997.

[10] Stefan Thienel. *ABACUS—A Branch-And-CUt System*. PhD thesis, Universität zu Köln, 1995.

[11] Stefan Thienel. ABACUS 1.2: User's guide and reference manual. Technical report, Universität zu Köln, 1996. http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus/manual.ps.gz/.

[12] Stefan Thienel. A simple TSP-solver. Technical report, Universität zu Köln, 1996. http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus/abacus_tutorial.ps.gz.