FAKULTÄT FÜR !NFORMATIK

Ant Colony Optimization for Tree and Hypertree Decompositions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Hammerl
Matrikelnummer 0300322

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Priv.-Doz. Dr. Nysret Musliu

Wien, 24.03.2009

_____ (Unterschrift Verfasser)

_____ (Unterschrift Betreuer)

ANT COLONY OPTIMIZATION FOR TREE AND HYPERTREE DECOMPOSITIONS

THOMAS HAMMERL

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

University of Technology
Computer Science Department
Institute of Information Systems
Database and Artificial Intelligence Group

24. März 2009

Thomas Hammerl, 24. März 2009

SUPERVISOR:
Priv.-Doz. Dr. Nysret Musliu

DECLARATION

Thomas Hammerl
Westbahnstraße 25/1/7
1070 Wien

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, 24. März 2009

Thomas Hammerl

Meinen Eltern gewidmet.

ABSTRACT

Many instances of constraint satisfaction problems can be solved efficiently if they are representable as a tree respectively generalized hypertree decomposition of small width. Unfortunately, the task of finding a decomposition of minimum width is NP-complete itself. Therefore, many heuristic and metaheuristic methods have been developed for this problem.

One metaheuristic which has not been applied yet is *ant colony optimization* (ACO). In this thesis we investigate five different variants of these ACO algorithms for the generation of tree and generalized hypertree decompositions. Furthermore, we extend these implementations with two local search methods and we compare two heuristics that guide the ACO algorithms. Moreover, we experiment with two different pheromone update strategies and we present a library called *libaco* that can be used to solve other combinatorial optimization problems as well. In order to demonstrate this we present an ACO implementation for the travelling salesman problem based on this library.

Our computational results for selected instances of the DIMACS graph coloring library and the CSP hypergraph library show that the ACO metaheuristic gives results comparable to those of other decomposition methods such as branch and bound and tabu search for many problem instances. One of the proposed algorithms was even able to improve the best known upper bound for one problem instance. Nonetheless, as the problem complexity increases other methods outperform our algorithms.

ZUSAMMENFASSUNG

Viele Instanzen von Costraint Satisfaction Problemen sind effizient lösbar wenn sie als Tree oder als Generalized Hypertree Decomposition kleiner Breite dargestellt werden können. Das Auffinden der Decomposition geringster Breite ist jedoch selbst **NP**-complete und kann daher nur mit Heuristiken und Metaheuristiken in annehmbarer Zeit gelöst werden.

Ant Colony Optimization (ACO) ist eine metaheuristische Methode die bisher noch nicht auf dieses Problem angewandt wurde. In dieser Diplomarbeit untersuchen wir fünf verschiedene Varianten von ACO Algorithmen für die Generierung von Tree und Hypertree Decompositions. Außerdem erweitern wir diese Implementierungen mit zwei lokalen Suchmethoden und vergleichen zwei Heuristiken, die den ACO Algorithmus lenken. Weiters experimentieren wir mit zwei unterschiedlichen Pheromone Update Strategien und stellen unsere C++ Bibliothek *libaco* vor mit deren Hilfe auch andere kombinatorische Optimierungsproblemen mit den in dieser Diplomarbeit beschriebenen ACO Algorithmen gelöst werden können. Um das zu demonstrieren beschreiben wir eine *libaco*-Implementierung für das Travelling Salesman Problem.

Unsere Testergebnisse für ausgewählte Beispiele der DIMACS Graph Coloring Library und der CSP Hypergraph Library zeigen, dass die ACO Metaheuristik für viele Probleminstanzen Ergebnisse liefert welche mit Ergebnissen anderer Verfahren wie beispielsweise Tabu Search und Branch & Bound vergleichbar sind. Einer der vorstellten Algorithmen konnte sogar die bisher besten bekannten Ergebnisse für eine Probleminstanz verbessern. Dessen ungeachtet liefern die ACO Algorithmen insbesondere für komplexere Probleminstanzen schlechtere Ergebnisse als andere bekannte Methoden.

DANKSAGUNGEN

Ich möchte mich an dieser Stelle bei Dr. Nysret Musliu für die Betreuung und das Korrekturlesen dieser Diplomarbeit, seine Hilfsbereitschaft und die zahlreichen wertvollen Ratschläge bedanken.

Besonderer Dank gilt meinen Eltern für eine sorgenfreie und glückliche Kindheit, die Ermöglichung meines Studiums, deren Fürsorge und die umfassende Unterstützung während der letzten 26 Jahre.

Weiters möchte ich mich bei Michael Jakl für seine Hilfe bei der Suche nach einem passenden Diplomarbeitsthema bedanken, welche schließlich zur Entstehung dieser Arbeit geführt hat.

Meinem Studienkollegen und Freund Benjamin Ferrari danke ich für die sowohl interessante als auch unterhaltsame gemeinsame Studienzeit.

Schlussendlich möchte ich noch all meinen Freunden, insbesondere meiner Freundin Claudia, für die schöne Zeit in der ich nicht mit dieser Arbeit beschäftigt war, danken.

CONTENTS

1 INTRODUCTION

1.1 THE PROBLEM

During the last couple of decades we have used computers to help us solve a variety of problems. While many of these problems can be solved in a reasonable amount of time, other problems are intractable excluding the most trivial of instances. For instance, sorting a list of numbers can be accomplished very quickly by a computer independent of the list's size.[1] On the other hand it is very computationally expensive to determine the shortest round-trip given a number of cities.[2] This problem, commonly known as the Traveling Salesman Problem (*TSP*), is proven to be **NP**-hard meaning that the time required to solve it increases very quickly with the problem size. This is not the case with problems belonging to the complexity class **P** that also contains the list-sorting problem mentioned beforehand.

NP-hard problems are often approached by approximation algorithms that give solutions that are probably suboptimal but can be computed in a reasonable amount of time. One such approximation for the *TSP* is to create a round-trip that visits the nearest not yet visited city next. This method might create a decent solution but optimality is neither guaranteed nor very likely in general.

A subset of these **NP**-hard problems, those that can be formulated as constraint satisfaction problems, can be solved efficiently by representing them as so called Tree- or Hypertree Decompositions. Each decomposition has a characteristic called *width* and each problem can be transformed to many different valid decompositions. The smaller a decomposition's width the faster the solution to the problem can be computed. Unfortunately, the problem of finding the decomposi-

1 Of course sorting a billion numbers takes longer than sorting ten. The point is that the difference is negligible in general.

2 The round-trip must visit each city exactly once returning to the same city it started from.

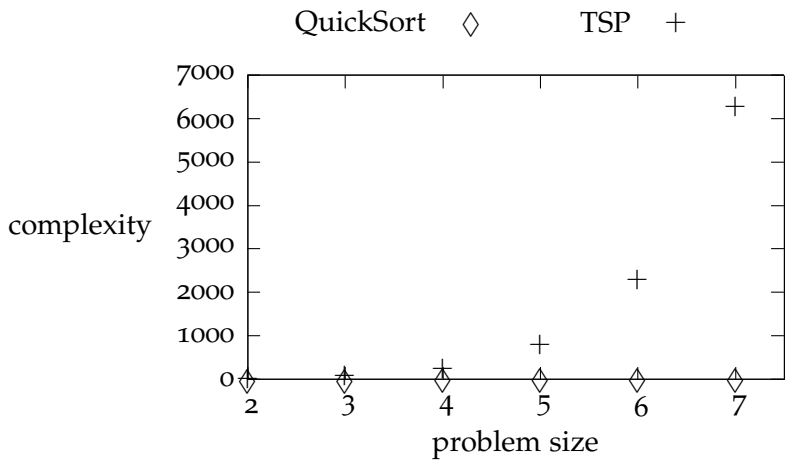


Figure 1. Comparison of the time complexities of a dynamic programming algorithm for the *TSP* ($\mathcal{O}(n^2 2^n)$) [28] and the QuickSort algorithm for the list-sorting problem ($\mathcal{O}(n \log n)$) [29].

tion having the minimum width of all valid decompositions is itself **NP**-hard [3] [27]. This is why approximation methods are necessary in order to find a decomposition having a preferably small width that then can help in solving the original problem efficiently.

An approximation method that has not been applied to the problem of finding decompositions of small width is Ant Colony Optimization (*ACO*). *ACO* has been applied successfully to many other **NP**-hard problems as the Traveling Salesman Problem [18], Network Routing [7] and the Car Sequencing Problem [21]. The main goal of this thesis was to examine different strategies of applying *ACO* for the generation of Tree- and Hypertree Decompositions and to evaluate their performance.

1.1.1 *Constraint Satisfaction Problems*

Informally speaking, a constraint satisfaction problem (*CSP*) consists of a set of variables each having a set of possible values that is also known as the variable's domain. Additionally, a number of constraints eliminate certain combinations of variables. For instance, a constraint might enforce that variable x and variable y must not have the same

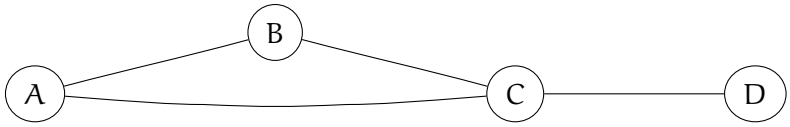


Figure 2. Instance of the Graph Coloring Problem.

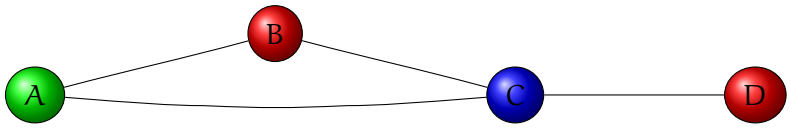


Figure 3. A valid 3-coloring of the graph in Figure 2.

value. Solving the *CSP* implies to find an assignment for each variable that does not conflict with any of the given constraints.

A constraint satisfaction problem that is very popular in the literature is the Graph Coloring Problem (*GCP*). It is the problem of coloring the vertices of a given graph in such a way that no two vertices connected by an edge share the same color.

Example 1.1. Figure 2 shows an instance of the *GCP*. The task is now to find a valid coloring just using the colors red, green, and blue.

Formulating this and any other *GCP* as a constraint satisfaction problem is quite straightforward:

- (Variables) The variables are given by the vertices of the graph: A, B, C, and D.
- (Domains) Each variable can be assigned one of the colors red, green, and blue. Thus, all variables share the same Domain.
- (Constraints) The constraints are given by the graph's edges. An edge represents the constraint that the connected vertices must not share the same color. Hence, the following constraints exist: $A \neq B$, $A \neq C$, $B \neq C$, and $C \neq D$.

One naive approach to this problem might be to try out all possible combinations of variable assignments and see which ones are valid. There are d^n possible combinations in general where d is the number

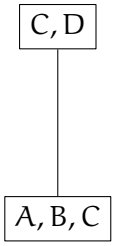


Figure 4. Tree Decomposition of Constraint Graph in Figure 2.

of available colors and n is the number of vertices. Accordingly, in this case there are 3^4 (81) possible combinations which is probably an acceptable number to check for validity. The drawback of this approach is that this number increases exponentially with the number of vertices. Given a graph with only twenty vertices we would already have to check 3^{20} (nearly 3.49 billion) possible variable assignments. Using a technique called Tree Decomposition we can reduce these numbers drastically.

1.1.2 *Tree Decompositions*

Informally speaking, a tree decomposition is a tree of subtrees of the corresponding graph.[3] Each vertex in the tree decomposition is labeled with vertices that build a subtree in the corresponding graph. Besides, a tree decomposition must satisfy these three conditions:

1. Every vertex of the corresponding graph must appear in at least one vertex of the tree decomposition.
2. If two vertices are connected by an edge in the corresponding graph, then there must be at least one vertex in the tree decomposition that contains them both.
3. If a vertex of the corresponding graph appears in multiple vertices of the tree decomposition, then these vertices must build a subtree in the tree decomposition.

3 A more formal definition of tree decompositions can be found on Page 16 of this thesis.

A	B	C	C	D
red	green	blue	blue	red
green	red	blue	blue	green
red	blue	green	green	red
blue	red	green	green	blue
green	blue	red	red	green
blue	green	red	red	blue

Table 1. Solutions to the subproblems

The tree decomposition illustrated in Figure 4 satisfies all of these conditions for the constraint graph in Figure 2.

If we want to solve the graph coloring problem based on this tree decomposition, we can start out by solving the subproblems given by each vertex in the tree decomposition. Using our naive approach of trying out all possible combinations of variable assignments we generate 3^3 (27) different solution candidates for the vertex containing A, B, and C. Because of the constraints $A \neq B$, $A \neq C$, and $B \neq C$ only six of them are valid. For the subproblem containing the vertices C and D we generate 3^2 (9) solution candidates and rule out three of them because of the constraint $C \neq D$.

We can now get all solutions to the whole problem by joining the subproblem solutions. Therefore, we will take a look at the variables both subproblems have in common. In this case, that is the variable C. Each solution for the subproblem A, B, C is joined with the solutions for the subproblem C, D sharing the same color for the vertex C. As can be seen in Table 1, there are two such solutions to C, D for every solution to A, B, C. Consequently, there are twelve solutions to the whole graph coloring problem.

We had to generate 36 combinations of variable assignments in order to determine these twelve solutions compared to the 81 combinations we had to generate without the tree decomposition. This difference increases very quickly with the size of the graph coloring problem and constraint satisfaction problems in general.

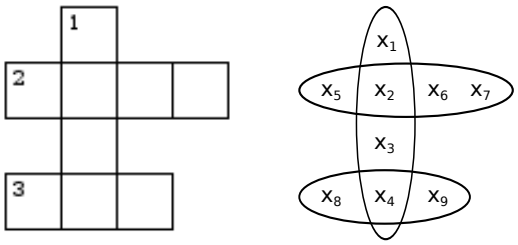


Figure 5. A crossword puzzle and its corresponding hypergraph.

The smaller the subproblems in the tree decomposition the more efficient we can solve a particular problem. This is why we are interested in finding tree decompositions of small *width*. This width is defined as the size of the tree decomposition's biggest subproblem minus one. Consequently, the width of the tree decomposition in Figure 4 equals 2.

1.1.3 *Hypertree Decompositions*

The same principle of decomposition of graphs can be applied to problems that can be respresented as hypergraphs. A hypergraph is a generalization of a graph. While an edge is only defined as a connection between exactly two vertices, a hyperedge can contain any number of vertices.

A crossword puzzle can be represented as a hypergraph very intuitively. The words to fill in can be considered the hypergraph's hyperedges, while the single character boxes correspond to the hypergraph's vertices. This analogy should become clear when looking at the crossword puzzle and the hypergraph in Figure 5.

More detailed and formal explanations of hypergraphs and hypertree decomposistions are given in Section 2.1 respectively in Section 2.4. For the understanding of this introduction it is sufficient to know that the same principle of decomposition of graphs presented in Section 1.1.2 can be applied to hypergraphs as well.

1.1.4 *Elimination Orderings*

One way of creating a tree decomposition for a given constraint graph is by applying the algorithm *Vertex Elimination* given in Algorithm 1. The algorithm takes as input the constraint graph and some ordering of the graph's vertices. The width of the resulting tree decomposition depends entirely on this elimination ordering.

Algorithm 1: Vertex Elimination [47] [48]

Input: an elimination ordering $\sigma = (v_1, \ldots, v_n)$
Input: a (constraint) graph $G = (V, E)$
Output: a tree decomposition $\langle T, \chi \rangle$ for G
Initially $B = \emptyset$, $A = \emptyset$
foreach *vertex* v_i **do**
  introduce an empty bucket B_{v_i}, $\chi(B_{v_i}) := \emptyset$
for $i = 1$ **to** n **do**
  $\chi(B_{v_i}) = \{v_i\} \cup N(v_i)$
  Introduce and edge between all non adjacent vertices in $N(v_i)$
  Let v_j be the next vertex in $N(v_i)$ following v_i in σ
  $A = A \cup (B_{v_i}, B_{v_j})$
  Remove v_i from G
return $\langle (B, A), \chi \rangle$, *where* $B = \{B_{v_1}, \ldots, B_{v_n}\}$

For example, an elimination ordering for the constraint graph in Figure 2 might be $\langle C, D, A, B \rangle$. This elimination ordering results in a tree decomposition having just one vertex containing all vertices of the constraint graph because C, the first vertex in the elimination ordering, is connected with all other vertices in the constraint graph. Hence, the resulting tree decomposition has a width of three[4]. A smaller tree decomposition of width two can be obtained by applying the vertex elimination algorithm with the elimination ordering $\langle D, B, A, C \rangle$[5]. Both of these tree decompositions are illustrated in Figure 6.

It is guaranteed that there is a so-called *optimal elimination ordering* that yields the tree decomposition with the minimum width of all

4 Remember that a tree decomposition's width is defined as the size of its biggest subproblem minus one.

5 In this case every elimination ordering not starting with vertex C would result in a tree decomposition of width two.

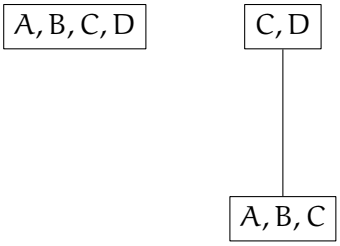


Figure 6. Two different tree decompositions of the graph in Figure 2.

valid tree decompositions for the given constraint graph.[6] This is why elimination orderings are a very popular search space for the detection of tree decompositions of small width. Unfortunately, there are $n!$ different elimination orderings where n is the number of vertices in the constraint graph. For that reason not only exact methods but also many approximation algorithms have been applied to the problem of finding tree decompositions of small width. They are discussed in Section 3.1 of this thesis.

1.1.5 *Ant Colony Optimization*

Ant algorithms are a class of algorithms that use real-life ants as a role model in order to solve computational problems. Real ants are capable of finding the shortest path between their nest and a food source by communicating with each other only through pheromones they deposit on their way. The more pheromone there is in a certain direction the more likely the ant will choose this direction. When an ant finds some food it will return to the nest. Consequently, the first ant that finds a food source will also return to the nest first and will also deposit pheromone on the way back. After some time the system converges to this short path and all ants will be very likely to follow this path while few ants will explore other paths that might be even shorter. An experiment called the *Double Bridge Experiment* that demonstrates this behaviour is discussed in Section 4.1.1.

Ant algorithms imitate this behaviour with a certain number of virtual ants constructing solutions on a so-called *construction graph*.

6 This minimum width is also called the constraint graph's treewidth.

Every edge in the *construction graph* is assigned an initial amount of pheromone in the *pheromone matrix*. After the ants have constructed the solutions each solution is evaluated. The better the solution the more pheromone the corresponding ant may deposit on the edges it traversed during the construction of the solution. This ensures that the ants will be more likely to choose these edges in the next iteration of the algorithm. Optionally, the constructed solutions can be improved by a *local search* procedure. Algorithm 2 describes the high-level, general structure of all *ACO* algorithms.

Algorithm 2: High Level ACO algorithm [16]

```
while termination condition not met do
    construct solutions
    apply local search (optional)
    update pheromones
```

A reasonable termination condition might be a certain amount of time, a fixed number of iterations or a fixed number of iterations without improvement of the best solution found so far.

The different *ACO* variants differ only in the way solutions are constructed and the pheromone matrix is updated. For instance, the variant *Rank-Based Ant System* only allows the best ants of every iteration to deposit pheromone while the variant *Max-Min Ant System* introduces lower and upper bounds on the pheromone values.

The details on how solutions are constructed by the ants and how the pheromone matrix is updated are discussed in Chapter 4 of this thesis.

1.2 GOALS OF THIS THESIS

The main goals of this thesis are:

- Application and evaluation of different existing variants of ACO algorithms to the following problems:
  - Finding tree decompositions of small width.
  - Finding generalized hypertree decompositions of small width.

- Experimental comparison of these variants of ACO algorithms for tree and hypertree decompositions.
- Hybridization of the best ACO algorithm with existing local search methods for this problem.
- Comparison of the results achieved by the best ACO algorithm with results achieved by other state of the art decomposition methods.

1.3 MAIN RESULTS

The main results of this thesis are:

- We implement the following variants of ACO algorithms known from the literature and apply them to the problem of finding tree decompositions and generalized hypertree decompositions of small width:
  - Simple Ant System [14] [17]
  - Elitist Ant System [14] [17]
  - Rank-Based Ant System [6]
  - Max-Min Ant System [52] [53]
  - Ant Colony System [15]

  Our computational results reveal that Ant Colony System and Max-Min Ant System perform slightly better than the other variants. Further, we discovered that the pheromone trails play a minor role in the search for a good solution. This is different from the results that were reported for the ACO implementations for the travelling salesman problem [16].
- We propose two different pheromone update strategies. One that lets the ants deposit the same amount of pheromone on all edges belonging to the constructed solution and another that tries to differentiate between each of the edges based on their utility. The executed experiments lead us to the conclusion that the latter gives slightly better results.

- We implement two stagnation measures that indicate the degree of diversity of the solutions constructed by the ants. These measures can be used as termination criteria or to reinitialize the algorithm when no new areas of the search space are explored.

- We implement two constructive heuristics that can be incorporated alternatively into every ACO variant as a guiding function:
  - Min-Degree
  - Min-Fill

  The computational results suggest that the ACO algorithms give better results using min-fill as a guiding heuristic. Nonetheless, due to the fact that the min-degree heuristic is more time-efficient we used it for all other experiments.

- We studied the combination of ACO with two existing local search methods:
  - Hill Climbing
  - Iterated Local Search [41]

  In our experimental studies the combination of Ant Colony System with an iterated local search was able to find better solutions than the combination of Ant Colony System with hill climbing for almost all problem instances.

- We compare the results achieved by Ant Colony System for 62 DIMACS graph coloring instances with the results of other state of the art heuristic and exact algorithms. The ACO algorithm gives comparable or even better solutions for some instances than other decomposition methods such as Tabu Search or Maximum Cardinality Search. The hybrid algorithm that incorporates an iterated local search into Ant Colony System was even able to improve the best known upper bound for the problem instance *homer.col*. Nevertheless, especially bigger problem instances cause the ACO algorithm to generate tree decompositions of considerably greater width than other approaches such as genetic algorithms.

- We extend our ACO decomposition methods with a set cover algorithm and apply the Ant Colony System variant to 19 selected

instances from the CSP Hypergraph Library [22]. The results are comparable to the results of some other hypertree decomposition methods but inferior for most of the instances compared to the best upper bounds known from the literature.

- We present the command line program *acotreewidth* that was used to obtain the computational results given in this thesis. Further, we present a library called *libaco* that was used to implement the *acotreewidth* program and that can be used to solve other combinatorial optimization problems as well. In order to demonstrate this we implement a command line program called *acotsp* based on *libaco* that constructs tours for instances of the travelling salesman problem.

1.4 FURTHER ORGANIZATION OF THIS THESIS

Chapter 2 defines the basic terminology used in this thesis and gives an overview on some fields of knowledge relevant to Tree- and Hypertree Decompositions. In Chapter 3 exact and heuristic methods that have already been applied to the problem of generating tree- and hypertree decompositions of small width are presented. Chapter 4 is all about the *Ant Colony Optimization* metaheuristic, its roots in nature and the differences between the variants. Furthermore, various real-world problems are described that already have been solved using *ACO*. In Chapter 5 we present our approach of applying *ACO* to the problem of Tree- and Hypertree Decomposition. We present different strategies of updating the pheromone matrix, measures to determine algorithm stagnation and local search algorithms in order to improve the solutions constructed by the ants. Chapter 6 documents all implementation artefacts that were created in the course of this thesis. There is a focus on the *libaco* library and how to write client code for it in order to solve other combinatorial optimization problems. In Chapter 7 we give the computational results we obtained by applying our implementation to examples taken from popular benchmark libraries. Finally, Chapter 8 concludes and describes future work.

2

PRELIMINARIES

2.1 GRAPHS AND HYPERGRAPHS

Definition 2.1. (Undirected Graph [43]). An undirected graph is an ordered pair $G = (V, E)$ with the following properties:

1. The first component, V, is a finite, non-empty set. The elements of V are called the vertices of G.
2. The second component, E, is a finite set of sets. Each element of E is a set that is comprised of exactly two (distinct) vertices. The elements of E are called the edges of G.

Definition 2.2. (Neighbourhood). Let $G = (V, E)$ be an undirected graph. The neighbourhood $N(v)$ of a vertex $v \in V$ is the set $\{w \mid \{v, w\} \in E\}$.

Definition 2.3. (Path). Let $G = (V, E)$ be an undirected graph. A sequence $\langle\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \ldots, \{v_{k-1}, v_k\}\rangle$ is a *path* of G between $v_1, v_k \in V$.

Definition 2.4. (Simplicial Vertex [33]). A simplicial vertex of G is a vertex of which the neighbourhood induces a clique.

Definition 2.5. (Connected Graph). G is *conntected* iff for any $v_i, v_j \in V$ there exists a path between v_i and v_j.

Definition 2.6. (Acyclic Graph). G is *acyclic* iff there is no path in G that starts and ends at the same vertex $v \in V$.

Definition 2.7. (Tree). A *tree* is a connected, undirected, acyclic graph.

Definition 2.8. (Hypergraph [42]). A hypergraph is a structure $\mathcal{H} = (V, H)$ that consists of vertices $V = \{v_1, \ldots, v_n\}$ and a set of subsets of these vertices $H = \{h_1, \ldots, h_m\}, h_i \subseteq V$, called hyperedges. Without loss of generality we assume that each vertex is contained in at least

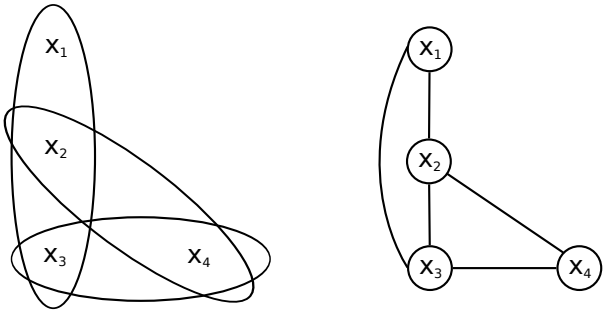


Figure 7. A hypergraph and its corresponding primal graph.

one hyperedge. Hyperedges differ from edges of regular graphs in that they may be defined over more than two vertices. Note that every regular graph may be regarded as a hypergraph whose hyperedges connect two vertices.

Definition 2.9. (Primal Graph, Gaifman Graph [9]). Let $\mathcal{H} = (V, H)$ be a hypergraph. The *Gaifman graph* or *primal graph* of $\mathcal{H}$, denoted $G^*(\mathcal{H})$, is a graph obtained from $\mathcal{H}$ as follows:

1. $G^*(\mathcal{H})$ owns the same set of vertices as $\mathcal{H}$.
2. Two vertices v_i and v_j are connected by an edge in $G^*(\mathcal{H})$ iff v_i and v_j appear together within a hyperedge of $\mathcal{H}$.

2.2 CONSTRAINT SATISFACTION PROBLEMS

Definition 2.10. (Constraint Satisfaction Problem [55]). A Constraint Satisfaction Problem *CSP* is a triple $\langle X, D, C \rangle$ where...

- X is a finite set of *variables* $\{x_1, x_2, \ldots, x_n\}$.
- D is a function which maps every variable in X to a set of objects of arbitrary type: $D : X \rightarrow$ finite set of objects (of any type). We shall take D_{x_i} as the set of objects mapped from x_i by D. We call these objects possible values of x_i and the set D_{x_i} the *domain* of x_i.

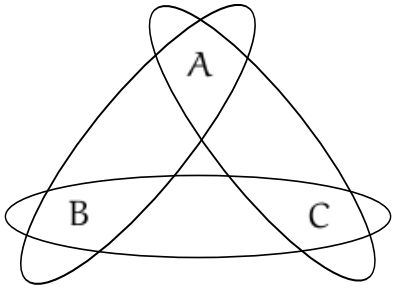


Figure 8. Constraint hypergraph of Formula 2.1.

- C is a finite (possibly empty) set of constraints on an arbitrary subset of variables in X. In other words, C is a set of sets of compound labels. $C_{x_1,x_2,\ldots,x_k}$ restricts the set of compound labels that $x_1, x_2, \ldots,$ and x_k can take simultaneously.

In Section 1.1.1 we have used the Graph Coloring Problem as an introductory example to constraint satisfaction. Now we want to take a look at another constraint satisfaction problem called the *Boolean Satisfiability Problem*.

Definition 2.11. (Boolean Satisfiability Problem). The Boolean Satisfiability Problem (*SAT*) is the decision problem of answering the question whether a given Boolean expression in conjunctive normal form (*CNF*) is satisfiable. A Boolean expression is satisfiable iff each of its variables can be assigned a logical value in such a way that the whole formula is true.

Definition 2.12. (Conjunctive Normal Form). A Boolean expression is said to be in conjunctive normal form (*CNF*) iff it consists only of conjunctions of clauses, where a clause is a disjunction of literals.

Example 2.13. Given the following Boolean formula:

$$(A \vee B) \wedge (\neg A \vee \neg C) \wedge (\neg B \vee C) \tag{2.1}$$

Figure 8 illustrates the constraint hypergraph for this problem instance that can be forumlated as a constraint satisfaction problem as follows:

$$
\begin{aligned}
\text{Variables:} \quad & X = \{A, B, C\} \\
\text{Domains:} \quad & D = \{D_A, D_B, D_C\} \\
& \forall D_i \in D : D_i = \{t, f\} \\
\text{Constraints:} \quad & C = \{C_{A,B}, C_{A,C}, C_{B,C}\} \\
& C_{A,B} = \langle\{A, B\}, \{(t,t), (t,f), (f,t)\}\rangle \\
& C_{A,C} = \langle\{A, C\}, \{(f,f), (f,t), (t,f)\}\rangle \\
& C_{B,C} = \langle\{B, C\}, \{(f,t), (f,f), (t,t)\}\rangle \\
\text{Solutions:} \quad & A = t, B = f, C = f \\
& A = f, B = t, C = t
\end{aligned}
$$

2.3 TREE DECOMPOSITIONS

Definition 2.14. (Tree Decomposition [46]). Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a tuple (T, χ), where T is a tree, s.t. $T = (N, E)$, and χ is a function, s.t. $\chi : N \to 2^V$, which satisfies all the following conditions:

1. $\bigcup_{t \in N} \chi(t) = V$
2. for all $\{v, w\} \in E$ there exists a $t \in N$ that $v \in \chi(t)$ and $w \in \chi(t)$
3. for all $i, j, t \in N$ if t is on the path from i to j in T, then $\chi(i) \cap \chi(j) \subseteq \chi(t)$

Definition 2.15. (Width of a Tree Decomposition of a Graph [33]). The *width* of a *tree decomposition* $\langle \chi, T \rangle$, where $T = (N, E)$, is $\max_{t \in N} | \chi(t) - 1 |$.

Definition 2.16. (Treewidth of a Graph). The *treewidth* $tw(G)$ of a graph G is the minimum width over all feasible tree decompositions of G.

2.4 HYPERTREE DECOMPOSITIONS

Definition 2.17. (Hypertree for a Hypergraph [19]) A *hypertree for a hypergraph* $\mathcal{H} = (V, H)$ is a triple $\langle T, \chi, \lambda \rangle$, where T is a rooted tree (N, E) and χ and λ are labelling function which associate with each node $n \in T$ a set of vertices $\chi(n) \subseteq V$ and a set of edges $\lambda(n) \subseteq H$.

Definition 2.18. (Generalized Hypertree Decomposition [19]). A hypertree $\langle T, \chi, \lambda \rangle$ for a hypergraph $\mathcal{H} = (V, E)$ is a *generalized hypertree decomposition* of $\mathcal{H}$, if it satisfies the following conditions:

1. for each hyperedge $e \in E$ there exists $t \in N$ such that $e \subseteq \chi(t)$. (t covers e)
2. for each vertex $v \in V$, the set $\{t \in N \mid v \in \chi(t)\}$ induces a (connected) subtree of T.
3. for each $t \in N$, $\chi(t) \subseteq (\bigcup_{e \in \lambda(t)} e)$

Definition 2.19. (Hypertree Decomposition [19]) A *Hypertree Decomposition* is a Generalized Hypertree Decomposition that satisfies this additional condition:

4. for each node $n \in N$, $\bigcup \lambda(n) \cap \chi(T_n) \subseteq \chi(n)$, where T_n denotes the subtree of T rooted at n; that is, each vertex v that occurs in some edge of the edge label and in the vertex label of n or some node below, must already occur in the vertex label of n.

Definition 2.20. (Width of a Hypertree Decomposition [19]). The *width of a hypertree decomposition* $\langle T, \chi, \lambda \rangle$ is given by $\max_{t \in N} \mid \lambda(t) \mid$, i.e., the largest size of some edge label.

Definition 2.21. (Hypertree Width). The hypertree width $hw(\mathcal{H})$ of a hypergraph $\mathcal{H}$ is the minimum width over all its feasible hypertree decompositions.

2.5 DECOMPOSITION METHODS

There are several known algorithms for the construction of Tree and Hypertree Decompositions. In this thesis we will focus on methods that are based on *elimination orderings*.[1] As already mentioned in Section 1.1.4 an elimination ordering is a permutation of the constraint graph's vertices.

1 Nevertheless, Section 3.4.2 discusses a method called *Hypergraph Partitioning* that is not based on elimination orderings.

Definition 2.22. (Elimination Ordering [33]) Given a graph $G = (V, E)$, an *elimination ordering* for G is an ordering $\sigma = (v_1, \ldots, v_n)$ of the vertices in V.

Elimination Orderings can serve as a search space for the treewidth of a graph because for every graph there is an ordering that produces a tree decomposition having minimum width. Gogate and Dechter [24] showed that instead of searching the space of all possible elimination orderings it is even sufficient to search a subset of these orderings named the treewidth elimination set.

Definition 2.23. (Treewidth Elimination Set [24]). Let P be the set of all possible orderings $\sigma = (v_1, v_2, \ldots, v_n)$ of vertices of G constructed in the following manner. Select an arbitrary vertex and place it at position 1. For $i = 2$ to n, if there exists a vertex v such that $v \notin N(v_{i-1})$, make it simplicial and remove it from G. Otherwise, select an arbitrary vertex v and remove it from G. Place v at position i. P is called the *treewidth elimination set* of G.

Both of the algorithms discussed next take as input an elimination ordering and return a tree decomposition for the given constraint graph. In order to obtain a *hypertree decomposition* both algorithms can be extended by an algorithm solving the *set cover problem* described in Section 2.6.

2.5.1 *Vertex Elimination*

The *vertex elimination* algorithm creates a tree decomposition by eliminating a sequence of vertices from the given constraint graph one after the other.[2] Eliminating a vertex v from the graph is achieved by performing the following steps.

1. Create a node t in the tree decomposition where $\chi(t) = \{v\} \cup N(v)$.
2. Node t will be connected to the node that is created when the next vertex in $N(v)$ is eliminated.

2 If a constraint hypergraph is given the algorithm operates on its corresponding primal graph.

3. Introduce an edge between all neighbouring vertices of v if there does not already exist one.

4. Remove vertex v from the graph.

This is done until the tree decomposition satisfies all conditions given in Definition 2.14. The complete algorithm can be found on Page 7 of this thesis.

Example 2.24. Given the constraint graph in Figure 9 and the elimination ordering $\sigma = (x_4, x_1, x_3, x_2, x_5)$ we want to create a tree decomposition by applying the vertex elimination algorithm.

The Figures 10 to 12 illustrate different states of the constraint graph and the tree decomposition as the algorithm progresses. A grey vertex implies that it is being eliminated while a dashed edge is being inserted because of the elimination.[3]

(Figure 10) The first vertex in the elimination ordering is x_4. Therefore, a node of the tree decomposition is created containing x_4 itself and its neighbours x_2 and x_3. This node is going to be connected with the node that will be created when x_3 is eliminated because x_3 is the first vertex to be eliminated among the neighbours of x_4. In the constraint graph we connect the neighbours of x_4 and remove x_4 itself. Afterwards we can proceed with the second vertex in the elimination ordering.

(Figure 11) Again, we create a node in the tree decomposition containing the vertex to be eliminated and its neighbours. This node is also going to be connected to the node that will be created when x_3 is eliminated. This time it is not necessary to add any edges to the constraint graph since the neighbours of x_1 are already connected.

(Figure 12) Finally we eliminate vertex x_3. As announced before, we connect the node that is created in the tree decomposition with the other two nodes that were created during the previous two iterations. Now, as the tree decomposition satisfies all conditions given in Defintion 2.14, there is no need to eliminate the remaining vertices x_2 and x_5. We have generated a valid tree decomposition (having width two) for the constraint graph in Figure 9.

3 Remember that all neighbouring vertices of the vertex to be eliminated are connected before its elimination.

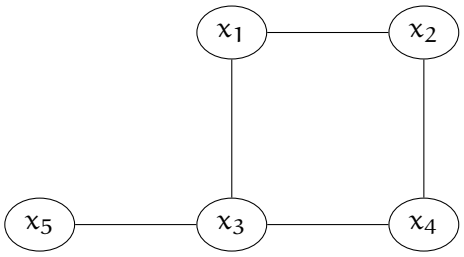


Figure 9. Constraint Graph.

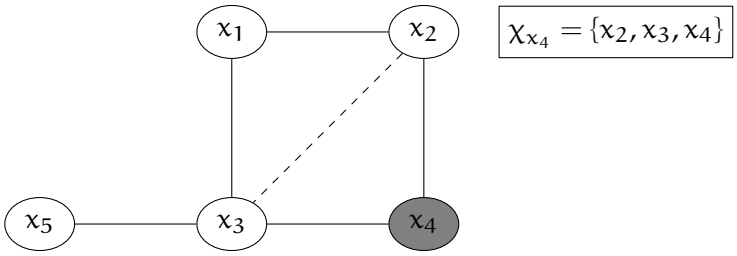


Figure 10. Elimination of vertex x_4.

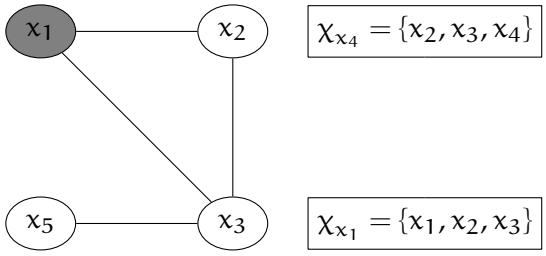


Figure 11. Elimination of vertex x_1.

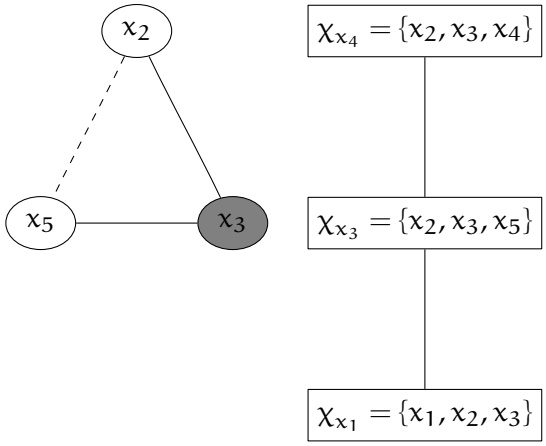


Figure 12. Elimination of vertex x_3.

2.5.2 *Bucket Elimination*

In contrast to the vertex elimination algorithm the *bucket elimination* algorithm takes a constraint *hyper*graph as input. At first it creates an empty bucket B_{x_i} for every vertex x_i. Then it takes a look at every hyperedge and fills the bucket of the vertex that is eliminated first among all vertices in the hyperedge with all the vertices contained in the hyperedge. After that we take a look at the bucket of the first vertex in the elimination ordering and search for the vertex that is eliminated next among all vertices in that bucket. The bucket of this vertex is filled with the vertices of the other bucket excluding the vertex the bucket "belongs" to and both buckets are connected by an edge. Then we go on and do the same for all other vertices according to the elimination ordering.

Example 2.25. Given the elimination ordering $\sigma = (x_2, x_1, x_3, x_4)$ we want to obtain a tree decomposition for the hypergraph in Figure 13 using the bucket elimination algorithm.

Among all vertices in edge h_1 vertex x_2 is the first one being eliminated so we fill its bucket with all vertices contained in h_1. Vertex x_2 is also eliminated before x_4 so we can also add all vertices within h_2 to the bucket of x_2. Finally we look at edge h_3 and find out that x_3 is

Algorithm 3: Bucket Elimination [9] [39]

Input: a (constraint) hypergraph $\mathcal{H} = (V, H)$
Input: an elimination ordering $\sigma = (v_1, \ldots, v_n)$ of the vertices in V
Output: a tree decomposition $\langle T, \chi \rangle$ for $\mathcal{H}$
Initially $B = \emptyset$, $E = \emptyset$
foreach *vertex* v_i **do**
  introduce an empty bucket B_{v_i}, $\chi(B_{v_i}) := \emptyset$
foreach *hyperedge* $h \in \mathcal{H}$ **do**
  Let $v \in h$ be the minimum vertex of h according to σ
  $\chi(B_v) = \chi(B_v) \cup$ vertices(h)
for $i = 1$ **to** n **do**
  Let $A = \chi(B_{v_i}) - \{v_i\}$
  Let $v_j \in A$ be the next vertex in A following v_i in σ
  $\chi(B_{v_j}) = \chi(B_{v_j}) \cup A$
  $E = E \cup (B_{v_i}, B_{v_j})$
return $\langle (B, E), \chi \rangle$, *where* $B = \{B_{v_1}, \ldots, B_{v_n}\}$

eliminated before x_4 and therefore add these two vertices to the bucket of x_3. Figure 13 illustrates the state of the buckets after these steps.

Vertex x_2 is the first vertex in the elimination ordering so we examine the content of bucket B_{x_2} excluding x_2. Among the vertices $\{x_1, x_3, x_4\}$ vertex x_1 is next in the elimination ordering. That is why we introduce an edge between bucket B_{x_2} and bucket B_{x_1} which additionally is filled with the vertices x_1, x_3 and x_4. These steps are repeated for all vertices in the elimination ordering which results in the tree decomposition pictured in Figure 14.

2.6 SET COVER PROBLEM

Both, vertex elimination and bucket elimination, return a tree decomposition for a given hypergraph and must be extended in order to obtain a (generalized) hypertree decomposition.

A tree decomposition of a hypergraph can be transformed into a generalized hypertree decomposition by assigning appropriate λ labels to the tree decomposition's nodes. Therefore a set of hyperedges must be found for each node in the tree decomposition that covers all of the

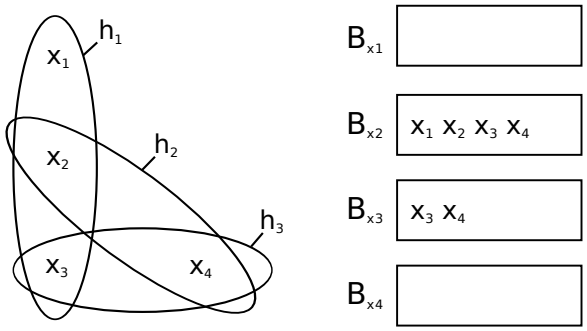


Figure 13. Bucket Elimination. Step 1.

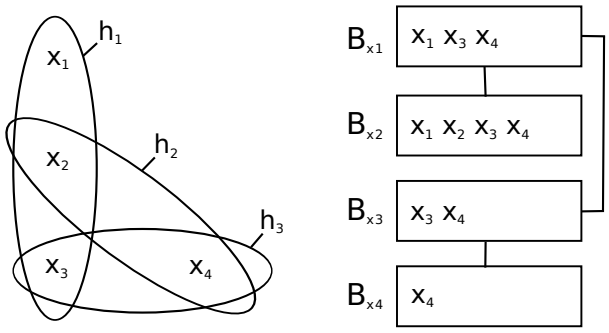


Figure 14. Bucket Elimination. Step 2.

vertices contained in χ. That is the equivalent of solving the *set cover problem* for each node of the tree decomposition.

Definition 2.26. (Set Cover Problem [20]). Given a collection $\mathcal{F}$ of subsets of $S = \{1, \ldots, n\}$, *set cover* is the problem of selecting as few subsets of $\mathcal{F}$ as possible such that their union covers S.

In the case of our special problem $\mathcal{F}$ is the collection of hyperedges of the constraint hypergraph and S is the set of vertices contained in the χ collection of a certain node in the tree decomposition.

One greedy algorithm to solve the set cover problem is to choose the subset that contains the largest number of uncovered elements at each stage. Ties are broken randomly.

Example 2.27. Given the tree decomposition in Figure 14 we can now apply this greedy algorithm in order to obtain a generalized hypertree decomposition.

The node B_{x_1} contains the vertices x_1, x_3 and x_4. The hyperedge h_3 covers the vertices x_3 and x_4 and is therefore the hyperedge covering the most vertices among all hyperedges. That is the reason why we add h_3 to the λ collection of B_{x_1}. With x_1 there is still one vertex left in the χ collection of B_{x_1} that is not covered by h_3 so we need to add another hyperedge to λ. This is going to be h_1 since it is the only edge covering x_1. Now all vertices in χ are covered by hyperedges in λ. Thus, we can proceed by applying the same algorithm to the other nodes of the tree decomposition. Finally we obtain the hypertree decomposition illustrated in Figure 15.

2.7 SOLVING CSPS FROM TREE DECOMPOSITIONS

Acyclic Solving [9] is an algorithm that can be used to solve a constraint satisfaction problem efficiently based on its tree decomposition. It achieves that by generating solutions for each node in the tree decomposition in a bottom-up fashion. At first the algorithm generates solutions for the leaf nodes which are then joined with the solutions of their ancestors in the tree. When the root node is reached and if its solution set is not empty, the solutions for the whole constraint satisfaction problem can be obtained by a top-bottom procedure. If the

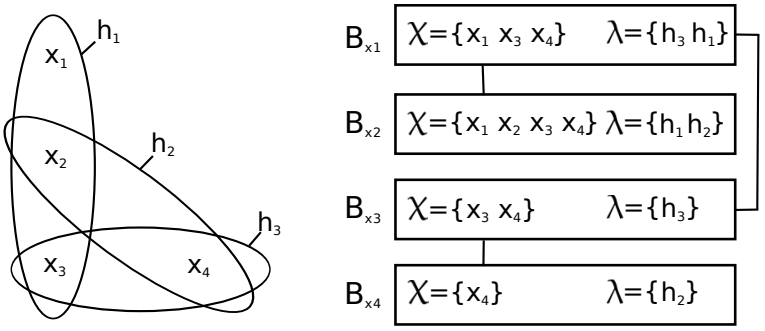


Figure 15. Hypertree Decomposition

root node's solution set is empty then it is guaranteed that there does not exist any solution.

3

STATE OF THE ART

Many different algorithms have already been applied to the **NP**-hard problem of deciding whether a given (hyper)graph has a (hyper)tree decomposition with a maximum width of k. This chapter gives an overview of some of these approaches and categorizes them into heuristic, exact and metaheuristic methods. Heuristic and metaheuristic methods give good results in a reasonable amount of time while exact methods determine the (hyper)treewidth of the given (hyper)graph with the drawback of higher computational costs.

3.1 UPPER BOUND HEURISTICS

The following three heuristics are very simple polynomial time algorithms. Decompositions of small width can be obtained very quickly with all of them. Nevertheless, the methods discussed in Section 3.3 and Section 3.4 yield better results by incorporating these heuristics for example for the generation of initial solutions that are then improved by a local search.

3.1.1 *Min-Degree*

The min-degree heuristic eliminates the vertex having minimum degree at first. From the graph that results from this elimination the vertex having minimum degree is eliminated next. This is repeated until every vertex has been eliminated. If multiple vertices share an equal degree then one of these vertices is chosen at random.

3.1.2 *Min-Fill*

The min-fill heuristic eliminates the vertex next that will cause the least amount of edges to be added to the graph. Ties are broken randomly.

3.1.3 *Maximum Cardinality Search (MCS)*

Maximum Cardinality Search, proposed by Tarjan and Yannakakis [54], chooses the first vertex in the elimination ordering randomly. After that, the vertex that is connected to the most vertices already selected by MCS (once again, ties are broken randomly) is added to the elimination ordering. Notice that MCS in contrast to min-degree and min-fill does not eliminate any vertices from the graph during the construction of the elimination ordering.

3.2 LOWER BOUND HEURISTICS

The following heuristics return a number x that is guaranteed to be a lower bound for the given graph's treewidth t meaning that $x \leqslant t$. These heuristics are used by the exact methods discussed in Section 3.3 in order to narrow the search space. For a number of other existing lower bound heuristics take a look at [5].

3.2.1 *Minor-Min-Width*

Minor-Min-Width is a heuristic that was proposed by Gogate and Dechter [24]. It searches for the minimum degree vertex v contained in the graph and the minimum degree vertex among its neighbours $w \in N(v)$. The degree of v is remembered as lb[1] if the degree of v is greater than lb before we contract the edge between v and w. An edge is contracted by merging the vertices connected by the edge and therefore removing the edge itself. This is repeated until no vertices remain in the graph. Finally, lb is returned as the lower bound for the graph's treewidth.

1 lb is initialized to zero.

3.2.2 γ_R

Ramachandramurthi [44] introduced the lower bound heuristic γ_R that looks at every pair of non-adjacent vertices (v_i, v_j) in the graph G and determines the maximum of their degrees (δ_i, δ_j):

$$\gamma_R(G) = \begin{cases} n-1 & \text{if G is a complete graph,} \\ \min_{(v_i,v_j)\notin E}\{\max\{\delta_i, \delta_j\}\} & otherwise \end{cases} \tag{3.1}$$

Ramachandramurthi proved that the minimum of all numbers determined this way represents a lower bound for the treewidth of the graph.

3.3 EXACT METHODS

Exact (also known as complete) algorithms are proven to deliver optimal solutions theoretically. In practice they often run out of memory or take an unacceptable amount of time to solve the problem instance they are confronted with due to the enormous size of the search space. In many cases the reason for the high memory consumption is that the algorithm needs to keep track of the (partial) solutions generated during the search. Most exact methods try to address these issues by identifying areas of the search space that cannot contain a solution that is better than the best solution found so far. This section discusses by means of A* and other branch and bound algorithms present in the literature how this is achieved for the problem of finding (hyper)tree decompositions of small width.

3.3.1 *Branch and Bound Algorithms*

Branch and Bound is a general method that can be used to solve optimization problems exactly while narrowing the search space with the help of various pruning techniques. Land and Doig [37] first proposed the method for linear programming in 1960.

The branching part of the algorithm divides the set of all feasible solutions S into two or more smaller subsets S_1, S_2,... whose union covers S. This branching is performed recursively on all those subsets

resulting in a tree structure similar to the structure of the search tree in Figure 16.

During the search certain branches of the tree a pruned due to the bounding part of the algorithm. Starting at the root of the tree a lower and an upper bound is calculated for each vertex in the tree and the minimum of all upper bounds is stored in a dedicated global variable m. If the search gets to a vertex whose lower bound is greater than the global minimum upper bound m then all solutions in this branch of the search tree can be discarded. A branch can also be pruned if the algorithm reaches a vertex having a lower bound equal to its upper bound because it then has already found the optimal solution within its branch. When the algorithm is done all vertices either have been pruned or their lower bound equals their upper bound meaning that it contains an optimal solution.

QuickBB

Gogate and Dechter presented a branch and bound algorithm called QuickBB in [24]. QuickBB introduced a new lower bound heuristic called minor-min-width (see Section 3.2.1) and the treewidth elimination set (see Definition 2.23 on Page 18). Min-fill was used as an upper bound heuristic for the bounding part of the algorithm. Additionally QuickBB uses several pruning techniques such as the *simplicial vertex rule* and the *almost simplicial vertex rule* due to Koster et al. [36].

QuickBB was applied to randomly generated graphs as well as on DIMACS benchmarks and bayesian networks. Gogate and Dechter reported that QuickBB was "consistently better than complete algorithms like QuickTree [49] in terms of cpu time" which was the best existing complete algorithm prior to QuickBB.

BB-tw

Bachoore and Bodlaender proposed their own branch and bound algorithm called BB-tw [4] in 2006. Although QuickBB and BB-tw were worked on independently both algorithms share some ideas especially in regard to search space pruning. For example, BB-tw makes also use of the simplical vertex rule and the strongly almost simplicial rule.[2] On

2 In [4] these rules are referred to as Pruning Rule 5.

the other hand while QuickBB searches the treewidth elimination set BB-tw searches all possible elimination orderings. What is special about the search tree is that the vertices are arranged in each level of the tree due to their sequence in the elimination ordering for finding the best upper bound. This way the branch and bound algorithm visits less vertices since it probably will find a decent upper bound right at the start and therefore will be able to prune more branches whose lower bound is greater than this upper bound.

Bachoore and Bodlaender experimented with various heuristics and combinations of pruning rules. They concluded that the ordering of the pruning rules has a significant impact on the running time of the algorithm. Further, they reported that BB-tw is efficient for graphs having either a very small or very large treewidth[3].

3.3.2 *A* Algorithms*

An A* algorithm is a special kind of branch and bound algorithm. One such algorithm for the computation of a graph's treewidth called A*-tw was proposed by Schafhauser in [48]. The algorithm is of course also applicable to the primal graph of a given hypergraph and can be extended by the set cover algorithm described in Section 2.7 in order to obtain a hypergraph decomposition.

In general, A* algorithms are graph search algorithms that find the least-cost path from a given initial vertex to one of possibly multiple goal vertices. This is accomplished by maintaining a priority queue of vertices. The priority $f(x)$ of a vertex x is calculated according to the following equation:

$$f(x) = g(x) + h(x)$$

Where

$g(x)$ is the cost of the path from the initial vertex to x,

$h(x)$ is a heuristic (estimated) value of the cost of reaching a goal vertex from x. This heuristic must be *admissible* what means that it must not overestimate the cost of reaching a goal vertex and

3 Very large meaning that the treewidth is close to the number of vertices contained in the graph.

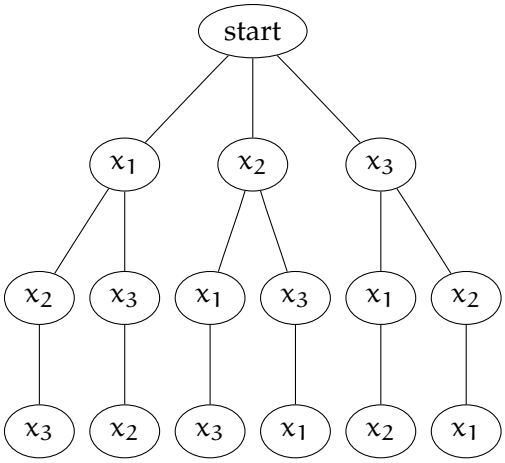


Figure 16. Search Tree for a graph containing three vertices.

$f(x)$ is the sum of $g(x)$ and $h(x)$.

In the beginning only the initial vertex s is put into the priority queue whereas $g(s) = 0$ (since it is the starting point) and $h(s)$ is some admissible heuristic value.[4]

At each iteration of the A*-algorithm the vertex having the lowest $f(x)$-value is removed from the priority queue. This vertex is added to the so-called closed set and all its neighbours not present in the closed set are evaluated meaning that they are added to the priority queue after their $g(x)$, $h(x)$ and $f(x)$ values are calculated. This process is repeated until a goal vertex is reached. When this happens it is guaranteed that the algorithm has found the least-cost path.

The A*-tw algorithm applies these principles on trees similar to the one illustrated in Figure 16. Starting out at the root node all possible elimination orderings can be constructed by traversing the edges down to the leaves of the tree. A*-tw additionally does apply some pruning and reduction rules that narrow the search space but are not mentioned here for the purpose of simplification. These details can be looked up in Chapter 5.1 of [48].

4 For example, $h(s)$ could be the air-line distance between s and the goal if the problem is about finding the geographically shortest way between two places through a network of streets. This heuristic would be guaranteed not to overestimate the real cost.

At first the algorithm computes an upper and a lower bound on the treewidth of the graph. Search states having a value $f(x)$ that exceeds the computed upper bound will be ignored because it is impossible to find the treewidth of the graph in that area of the search tree. The maximum number returned by the minor-min-width and the minor-γ_R heuristics is used as the heuristic value $h(x)$. The value $g(x)$ is set to the width of the decomposition that is constructed using the partial elimination ordering represented by the search state. In contrast to the general case where $f(x)$ is the sum of $g(x)$ and $h(x)$, $f(x)$ is set to the maximum of $g(x)$ and $h(x)$ by A*-tw.

A*-tw was applied on selected DIMACS graphs and could determine the treewidth of the graph miles1000 which could not be fixed before. Additionally it was able to significantly improve the lower bound for the graph DSJC125.5. On the other hand A*-tw could not return the treewidth for myciel5 and queen7_7 whereas QuickBB and BB-tw could.

3.4 METAHEURISTIC METHODS

Metaheuristics are a very general class of algorithms that are applicable to a wide variety of problems. Such an algorithm tries to continually find and improve feasible solutions to a given (often combinatorial) optimization problem with the guidance of an underlying problem-specific heuristic – hence the name. Usually a metaheuristic runs in iterations and in each iteration one or more solutions are generated using the knowledge about the search space acquired during the previous iterations.[5] A termination condition for a metaheuristic algorithm can be everything from a time limit to a number of iterations without improvement of the best solution found.

3.4.1 *Genetic Algorithms*

Genetic Algorithms were introduced by John H. Holland in 1975. The idea behind them is to imitate the biological principle of evolution by selection, recombination and mutation of a set of initial candidate solutions — the *population* — which is either created randomly or heuris-

5 Metaheuristics that generate multiple solutions per iteration are called *population-based*.

Figure 17. Position-based crossover operator (left: parents, right: offsprings).

tically. The quality of these solutions is evaluated in every iteration by a so-called *fitness function*. Better solutions are more likely to advance to the next iteration and are also more likely to be recombined with other solutions to produce *offsprings*. Additionally some of the solutions are slightly altered (mutated) in order to cause the search to explore other areas of the search space. For this all to work out the solution properties have to be encoded as *genes* which, when put together, form a *chromosome*.

Musliu and Schafhauser developed the genetic algorithms GA-tw and GA-ghw, based on a genetic method for the decomposition of bayesian networks proposed by Larrañaga et al. [38], for the creation of tree respectively generalized hypertree decompositions of small width in [42]. Both algorithms take a hypergraph and several control parameters (population size, mutation rate, crossover rate,...) as input. The initial population is generated randomly whereas each individual in the population is an elimination ordering. Those individuals are evaluated by the fitness function which is the width of the resulting tree decomposition for GA-tw whereas GA-ghw computes the width of the resulting generalized hypertree decomposition[6].

The individuals going to be kept for the next iteration are selected using *tournament selection*. This selection technique randomly chooses a certain number of individuals and the best one among them (the one having the smallest width) is selected. This is repeated until enough individuals have been selected for the next iteration.

Musliu and Schafhauser implemented many different crossover operators that differ in the way two individuals are recombined and compared them. Position-based crossover (POS) illustrated in Figure 17 turned out to "achieve the best average width". POS chooses a set of

6 GA-ghw uses the greedy set cover heuristic discussed in Section 2.6 to obtain a generalised hypertree decomposition.

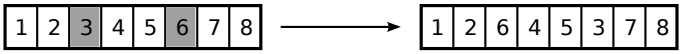


Figure 18. Exchange mutation operator (example taken from [42]).

positions in the elimination orderings and exchanges those elements between the parents. The vertices that are missing after this exchange are reinserted in the order of the other parent.

One mutation operator that was implemented in [42] among many others is the exchange mutation operator exemplified in Figure 18. The operator simply exchanges to arbitrarily chosen vertices in the elimination ordering.

GA-tw was able to improve the best upper bounds on the treewidth known at the time for twelve graphs among the 62 graphs of the DIMACS benchmark library.

GA-ghw could improve the known upper bounds on the hypertreewidth of eleven hypergraphs among the 25 hypergraphs from the *CSP Hypergraph Library* from Ganzow et al. [22].

3.4.2 *Hypergraph Partitioning*

Unlike the other methods discussed in this chapter hypergraph partitioning does *not* search for a good elimination ordering. Instead, it tries to subdivide the given hypergraph into small, loosely coupled components.

Every time the hypergraph is partitioned as in Figure 19 a node is created in the hypertree decomposition either containing the hyperedges that were "cut" or the vertices that are covered by these hyperedges. Accordingly, for the hypergraph in Figure 19 a node would be created either having a λ-label containing the hyperedges h_1 and h_3 or having a χ-label containing the vertices x_1, x_3, x_4 and x_6. This is repeated recursively for all resulting subcomponents of the hypergraph. Afterwards the χ respectively the λ labels are added to the nodes of the hypertree decomposition depending on whether the vertices or the hyperedges have been added during the partitioning. Finally, the nodes are con-

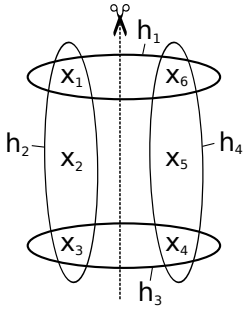


Figure 19. A hypergraph being partitioned into two components.

nected in a way that results in a generalized hypertree decomposition. A detailed description of this process can be found in [34].

Dermaku et al. have experimented with several hypergraph partitioning algorithms in [13] and [12] that differ in the way they partition the hypergraph. They reported that the algorithm based on the HMETIS library [30] (a popular hypergraph partitioning library) performed better than the ones based on Tabu Search and the Fiduccia-Mattheyses algorithm. Furthermore, it was reported that heuristics based on tree decomposotions give slightly better results than algorithms based on hypergraph partitioning. On the other hand hypergraph partitioning is very effective and time-efficient for the generation of hypertree decompositions of large hypergraphs. This is why they were able to improve the best known upper bounds of many large hypergraphs in the CSP hypergraph library.

3.4.3 *Tabu Search*

Tabu Search is a local search technique that was proposed by Glover [23] in 1989. A local search algorithm tries to improve an initial solution (generated randomly or heuristically) by looking at neighbourhood solutions. A solution's neighbourhood is defined by some kind systematic modification of the solution. One such modification might be the swapping of two solution elements. The best solution in the neighbourhood is selected and its neighbourhood is evaluated next.[7] What

7 This step is known as a *move* in the solution space.

is special about tabu search is that it remembers a certain number of previous moves and adds them to a so-called tabu-list. For example, if an element has been swapped in the previous five moves, the element must not be swapped again. This shall prevent the algorithm from moving in circles in the solution space.

Clautiaux et al. presented a tabu search approach for the generation of tree decompositions in [8]. They reported that their "results actually improve on the previous best results for treewidth problems in 53% of the cases" [8] applying their tabu search algorithm to the DIMACS library. Some of their results could be further improved by Gogate and Dechter later in 2004 [24].

Musliu proposed a tabu search algorithm for generalized hypertree decompositions in [40]. Two types of neighbourhoods were implemented. One swaps the vertex causing the largest clique during the elimination process (ties broken randomly) with a random other vertex in the elimination ordering while the other evaluates all solutions obtained by swapping the vertex causing the largest clique with its neighbours. When the algorithm moves to a different solution the swapped vertices are made tabu for a certain number of iterations.

3.4.4 *Simulated Annealing*

Another local search method that been applied to the problem of generating tree decompositions of small width is Simulated Annealing which was introduced by Kirkpatrick et al. [31] in 1983. Kjærulff et al. [32] applied this method to the decomposition of probabilistic networks.

3.4.5 *Iterated Local Search*

Any local search method can be iterated and therefore extended to an iterated local search. The motivation behind this is the inherent high probability that a local search gets stuck in a local optimum meaning that due to its neighbourhood it is impossible for the algorithm to find any better solutions even though they exist. To bypass this problem an iterated local search algorithm perturbates (modifies) the solution

returned by the local search[8] in some way and restarts the local search supplying it with this perturbated solution. This measure is based on the hope that the perturbated solution has a neighbourhood that allows the search to break out of the local optimum.

Musliu presented an iterated local search called IHA (Iterative Heuristic Algorithm) for the generation of tree decompositions in [41]. Algorithm 4 describes the application flow in pseudo-code notation where the ConstructionPhase function corresponds the local search that is being iterated. Several local search methods, pertubation mechanisms and acceptance criteria have been evaluated. For a detailed explanation of all of them please take a look at [41].

Algorithm 4: Iterative heuristic algorithm - IHA [41]

```
Generate initial solution S1
BestSolution = S1

while Termination Criterion is not fulfilled do
    S2 = ConstructionPhase(S1)

    if Solution S2 fulfills the acceptance criterion then
        S1 = S2
    else
        S1 = BestSolution

    Apply perturbation in solution S1

    if S2 has better (or equal) width than BestSolution then
        BestSolution = S2
return BestSolution
```

It was reported that IHA improved the best known upper bounds of 14 graphs in the DIMACS library. Further, it was reported that the time-performance of IHA depends highly on the local search method being applied. Using a local search method called LS1 it was possible to decrease the time needed to generate tree decompositions significantly

8 To be precise, the perturbation is not necessarily always applied to the solution returned by the local search. An acceptance criterion can be defined. If this is not fulfilled the best solution found so far is perturbated and passed on the local search in the next iteration. One acceptance criterion could be that the solution returned by the local search is at least as good as the best solution found so far.

for some instances. This was ascribed to the fact that LS1 considers only one neighbourhood solution per iteration.

4

ANT COLONY OPTIMIZATION

Ant Colony Optimization (*ACO*) is a population-based metaheuristic introduced by Marco Dorigo [14] in 1992. As the name suggests the technique was inspired by the behaviour of "real" ants. Ant colonies are able to find the shortest path between their nest and a food source just by depositing and reacting to pheromones while they are exploring their environment. The basic principles driving this system can also be applied to many combinatorial optimization problems. In this chapter we will try to explain what these principles are in general and how they can be applied to the travelling salesman problem — an example-application used for the evaluation of many variants of *ACO* algorithms. Finally, other problems ACO has been applied to are discussed briefly.

This chapter is mainly based on the structure and contents of the book *"Ant Colony Optimization"* [16] by Marco Dorigo and Thomas Stützle.

4.1 FROM REAL TO ARTIFICIAL ANTS

4.1.1 *The Double Bridge Experiment*

Deneubourg et al. [11] have investigated the foraging behaviour of the Argentine ant species Iridomyrmex humilis in controlled experiments. They connected the nest of an ant colony and a food source with a double bridge and ran various experiments with a varying ratio of the length of the two branches. With branches of equal length a vast majority of the ants preferred one of the branches after some time. Numerous repetitions of the exact same experiment showed that the ants chose one branch or the other in about the same number of trials. Using a different experimental setup with one branch being twice as long as the other one, all ants chose the shorter branch after some time in almost all trials.

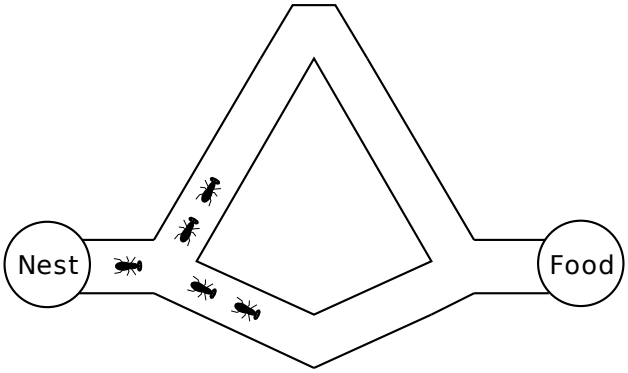


Figure 20. Double bridge experiment. Ants start exploration.

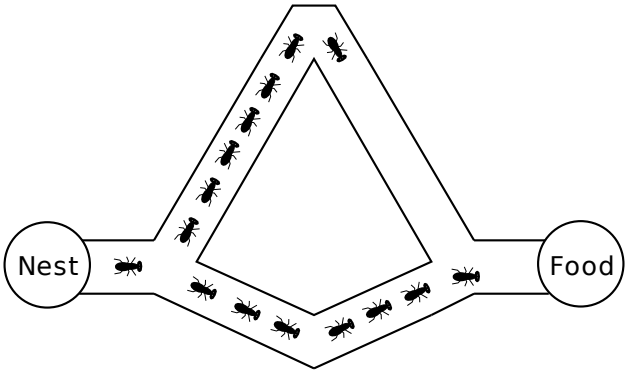


Figure 21. Ants on shorter branch reach food source.

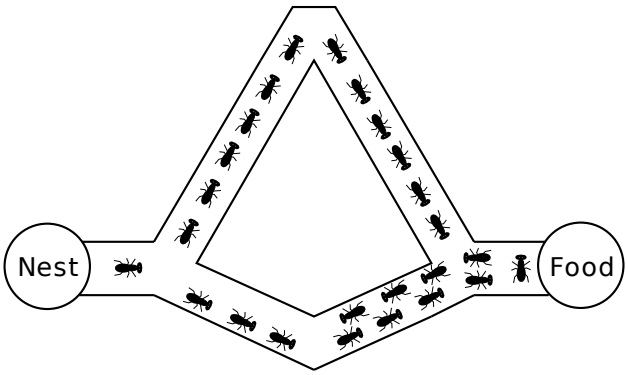


Figure 22. Ants more likely to return on shorter branch.

As can be seen in Figure 20, the ants choose one of the branches randomly in the beginning of the experiment since pheromone has been deposited on neither of them yet. Of course, the ants that have chosen the shorter branch reach the food source before the ants on the longer branch (Figure 21) and are therefore also able to deposit pheromone onto the crossing in front of the food source first. As a consequence, these ants will also be more likely to take the short branch when they return to the nest (Figure 22) and will also be the first ants to deposit pheromone onto the crossing in front of the nest enforcing the bias towards the short branch. This autocatalytic effect also influences the behaviour of the ants in the experiment consisting of two branches of equal length. The difference is that in this case the system converges to one of the branches because one of the branches is used by more ants incidentally in the beginning of the experiment.

4.1.2 *Artificial Ants*

The foraging behaviour of real ants can serve as a role model for the implementation of artificial ants that solve many different kinds of optimization problems. The exploration space used by real ants can be modelled as a graph where the artificial ants move from one vertex to another searching for a solution. Instead of constantly depositing pheromone an ant evaluates its solution after construction and deposits pheromone proportionally to its quality. During construction the ants consider not only the amount of pheromone present on the incident edges but also problem-specific heuristic information to decide where to move next.

Construction Graph

For the travelling salesman problem the construction graph consists of all given cities where each city is connected by an edge with all the others.[1] The ants start their tour at some random city since the starting point of a round-trip does not affect its overall length.

1 Remember that the travelling salesman problem (TSP) is about finding the shortest round-trip given a number of cities where each city is only visited once.

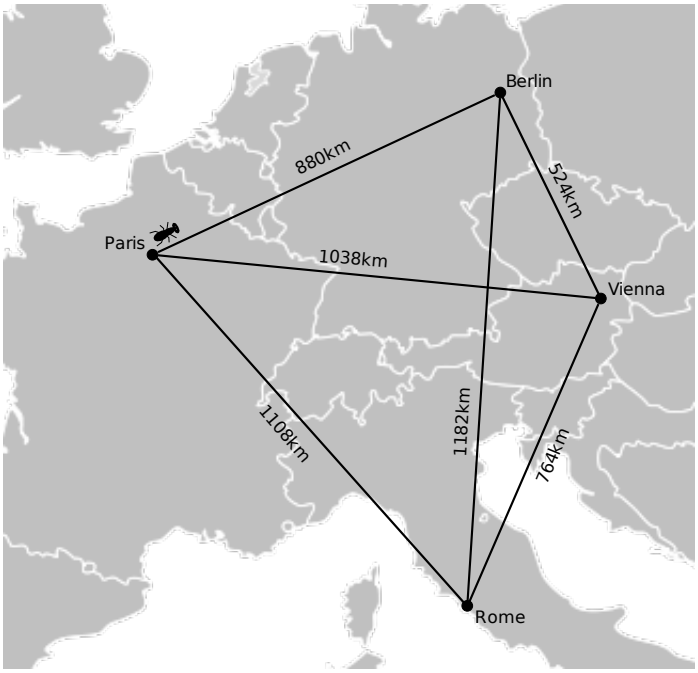


Figure 23. ACO Construction Graph for an instance of the *TSP*.

Figure 23[2] illustrates the construction graph for an instance of the TSP that deals with determining the shortest round-trip for the cities Vienna, Rome, Paris and Berlin.

Heuristic Information

At each solution construction step the ant has to decide to which neighbouring vertex to move.[3] This decision is made probabilistically based on the pheromone values and some heuristic information that especially helps finding a good solution in the beginning of the algorithm when all pheromone values are equal. The higher the heuristic value the more likely the ant will be to move to the corresponding vertex.

2 This image is a derivative of [1] licensed under the Creative Commons Attribution ShareAlike 2.5 License (http://creativecommons.org/licenses/by-sa/2.5/). Therefore this image is also covered by this license.

3 In the case of the TSP the ant may only move to a neighbouring vertex it has not visited yet.

This is why the reciprocal value of the airline distance is chosen as the heuristic for the TSP.

$$\begin{aligned}
\eta_{vr} &= \eta_{rv} = 1/764 \\
\eta_{vb} &= \eta_{bv} = 1/524 \\
\eta_{vp} &= \eta_{pv} = 1/1038 \\
\eta_{rb} &= \eta_{br} = 1/1182 \\
\eta_{rp} &= \eta_{pr} = 1/1108 \\
\eta_{pb} &= \eta_{bp} = 1/880
\end{aligned}$$

Pheromone Matrix

The pheromones associated with the edges of the construction graph are represented as a matrix.

$$\mathcal{T} = \begin{pmatrix} \tau_{vv} & \tau_{vr} & \tau_{vp} & \tau_{vb} \\ \tau_{rv} & \tau_{rr} & \tau_{rp} & \tau_{rb} \\ \tau_{pv} & \tau_{pr} & \tau_{pp} & \tau_{pb} \\ \tau_{bv} & \tau_{br} & \tau_{bp} & \tau_{bb} \end{pmatrix}$$

The variables in the matrix are initialized to some reasonable value. If this value is too low the search is biased too early towards a probably suboptimal part of the solution space. On the other hand if the initial pheromone value is too high it takes a couple of iterations until the pheromone updates have an impact on the behaviour of the ants. Dorigo and Stützle [16] reported that a good initial pheromone value is the amount of pheromone deposited by the colony per iteration on average.

Pheromone Update

After an ant has constructed its solution it deposits an adequate amount of pheromone onto the edges it traversed. For the travelling salesman problem this could be the reciprocal value of the length of the round-trip L.

$$\tau_{ij} = \tau_{ij} + \frac{1}{\text{length}(L)}, \qquad \forall (i,j) \in \mathcal{L}$$

Additionally a certain amount of pheromone evaporates after each iteration. The intended purpose of this is that the algorithm "forgets"

older solutions after some time and explores new areas of the search space. How much pheromone evaporates after each iteration can be adjusted with the evaporation rate parameter ρ.

$$\tau_{ij} = (1 - \rho)\tau_{ij} \tag{4.1}$$

4.2 ANT SYSTEM

Ant System (AS) [14] [17] was one of the first ant algorithms. It is inferior to all other variants of ACO variants discussed in this chapter. Nevertheless, it proved the concept of ant colony optimization and builds the foundation of all other ant algorithms.

4.2.1 *Pheromone Trail Initialization*

For the TSP the pheromone trails are initialized according to Equation 4.2 where m is the number of ants and C^{nn} is the length of the round-trip obtained by applying the nearest-neighbour heuristic. This heuristic randomly selects a city and then creates a round-trip by always moving to the nearest city.

$$\tau_{ij} = \tau_0 = \frac{m}{C^{nn}}, \qquad \forall (i, j) \tag{4.2}$$

Example 4.1. The ant in Figure 23 would construct the tour (Paris, Berlin, Vienna, Rome, Paris)[4] using the nearest-neighbour heuristic. This would lead to the following initial pheromone value τ_0 under the assumption that the ant colony consists of ten ants ($m = 10$).

$$C^{nn} = 880 + 524 + 764 + 1108 = 3276$$
$$\tau_0 = \frac{10}{3276}$$

4.2.2 *Solution Construction*

An ant k located at vertex i moves to a neighbouring vertex j with probability p_{ij}^k that is computed according to Equation 4.3 whereas α

4 Coincidentally this is also the least-cost tour. In general it is very unlikely that the nearest-neighbour heuristic constructs an optimal solution.

and β are parameters passed to the ant system algorithm that weight the influence of the pheromone trail τ_{ij} and the heuristic information η_{ij}.

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \qquad \text{if } j \in N_i^k \tag{4.3}$$

Example 4.2. An ant starts its tour in Paris and now has to decide where to move next. Extending Example 4.1 we assume that all pheromone trails have been initialized to $10/3276$. Further, we will determine that $\alpha = 1$ and $\beta = 1$. Using this data and the heuristic information we can compute the transition probabilities. Let's start out with the denominator d of Equation 4.3.

$$\begin{aligned} d &= 10/3276 \cdot 1/880 + 10/3276 \cdot 1/1038 + 10/3276 \cdot 1/1108 \\ &= 0.000009164 \end{aligned}$$

Next we can compute the probabilities of moving to Berlin p_{pb}, to Vienna p_{pv} and to Rome p_{pr}.

$$\begin{aligned} p_{pb} &= \frac{10/3276 \cdot 1/880}{d} \\ &= 0.378519584 \\ p_{pv} &= \frac{10/3276 \cdot 1/1038}{d} \\ &= 0.320902923 \\ p_{pr} &= \frac{10/3276 \cdot 1/1108}{d} \\ &= 0.300629273 \end{aligned}$$

Since all pheromone values are equal the probabilities are only diversified by the heuristic information in the beginning. This changes after the first iteration when the ants will have deposited pheromone for the first time.

4.2.3 *Pheromone Update*

After all ants in the colony have constructed their solutions they modify the pheromone matrix according to Equation 4.4.

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \qquad \forall (i,j) \in \mathcal{T} \tag{4.4}$$

In the case of the travelling salesman problem an ant adds the reciprocal value of the tour length C^k to the pheromone trail of all edges it traversed in its tour T^k.

$$\Delta\tau_{ij}^{k} = \begin{cases} 1/C^{k}, & \text{if arc } (i,j) \text{ belongs to } T^{k}; \\ 0, & \text{otherwise;} \end{cases}$$

Additionally pheromone evaporates on all edges of the construction graph according to Equation 4.5 where ρ is the evaporation rate.

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij}, \qquad \forall (i,j) \in \mathcal{T} \tag{4.5}$$

4.3 OTHER ANT COLONY OPTIMIZATION VARIANTS

4.3.1 *Elitist Ant System*

Elitist Ant System (EAS) was introduced by Dorigo [14] and Dorigo et al. [17]. It extends the pheromone update mechanism of Ant System with a so-called elitist ant which is the ant that has found the best solution so far. This ant is allowed to deposit additional pheromone (weighted with a parameter e) on the edges of the best-so-far solution. The idea behind this is to enforce a stronger bias towards those edges.

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + e\Delta\tau_{ij}^{bs}, \qquad \forall (i,j) \in \mathcal{T} \tag{4.6}$$

$$\Delta\tau_{ij}^{bs} = \begin{cases} 1/C^{bs}, & \text{if arc } (i,j) \text{ belongs to } T^{bs}; \\ 0, & \text{otherwise;} \end{cases}$$

Dorigo [14] and Dorigo et al. [17] reported that an appropriate value for e allows EAS to find better solutions in fewer iterations than AS.

4.3.2 *Rank-Based Ant System*

Rank-Based Ant System is another extension of Ant System and was introduced by Bullnheimer et al. [6]. As in Elitist Ant System the ant with the best-so-far solution may update the pheromone trails. This additional pheromone is multiplied by a parameter w. Additionally, the iteration-best $w-1$ ants deposit pheromone in every iteration. Their update is multiplied by $w-r$ where r is their rank among all ants (e.g. the update of the ant having the second best solution is multiplied by $w-2$). Equation 4.7 sums this up.

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1} (w-r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs} \qquad \forall (i,j) \in \mathcal{T} \tag{4.7}$$

The computational results presented by Bullnheimer et al. [6] suggest that Rank-Based Ant System performs slightly better than EAS and significantly better than Ant System.

4.3.3 *Max-Min Ant System*

Stützle and Hoos [52] [53] proposed a variant of Ant System called Max-Min Ant System that introduces not only one but a couple of new ideas. Unlike the other variants discussed so far Max-Min Ant System allows only the iteration-best or the best-so-far ant to update the pheromone trails. For example, an implementation could allow the iteration-best ant to deposit pheromone in the even iterations while the best-so-far ant deposits pheromone in the odd iterations. The ratio of these updates determines whether the algorithm tends to be explorative or more focused on the search space around the best solution found so far.

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{best} \qquad \forall (i,j) \in \mathcal{T} \tag{4.8}$$

Furthermore the Max-Min variant of AS introduces limits $[\tau_{min}, \tau_{max}]$ on the values of the pheromone trails. This shall avoid algorithm stagnation which otherwise could appear due to pheromone being accumulated on certain trails. The pheromone trails are initialized to the maximum limit τ_{max}. This causes the algorithm to be more explorative

in the beginning (what is usually desirable) until enough pheromone evaporates and the algorithm concentrates on a certain search area.

Another measure against stagnation proposed by Max-Min Ant System is pheromone trail reinitialization. If the algorithm was not able to improve the best known solution for a given number of iterations the pheromone trails are reinitialized in order to enforce exploration of the search space.

4.3.4 *Ant Colony System*

Ant Colony System (ACS), introduced by Dorigo and Gambardella [15], stands out from the other variants based on Ant System. It differs in the way solutions are constructed and pheromone trails are updated. ACS even introduces a new kind of pheromone update that removes pheromone during the construction phase of the algorithm.

An Ant in ACS decides where to move next using a so-called *pseudo-random proportional rule*.

$$j = \begin{cases} \arg\max_{l \in N_i^k} \{\tau_{il}[\eta_{il}]^\beta\}, & \text{if } q \leqslant q_0; \\ \text{Equation 4.3}, & \text{otherwise;} \end{cases} \tag{4.9}$$

If a randomly generated number in the interval $[0, 1]$ q is less or equal than q_0, a parameter passed to ACS, the ant moves to the vertex which is the "best" one according to the pheromone trail and the heuristic information. Otherwise a probabilistic decision is made as in Ant System.

Another important aspect of ACS is that only the best-so-far ant is allowed to deposit pheromone after each iteration. Additionally only the edges that are part of the best-so-far tour T^{bs} are evaporated. Thus, both pheromone deposition and pheromone evaporation can be summarized by Equation 4.10.

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \rho\Delta\tau_{ij}^{bs}, \qquad \forall (i, j) \in T^{bs} \tag{4.10}$$

These measures reduce the complexity of the pheromone update from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ [15] compared to Ant System.

In addition to this global pheromone update ACS introduces a *local pheromone update*. Immediately after an ant has moved from one vertex

to another it reduces the pheromone located on the edge connecting these two vertices in order to make this edge less desirable for the following ants. This measure favors exploration of other areas of the search space and in practice avoids stagnation.

$$\tau_{ij} \leftarrow (1 - \xi)\tau_{ij} + \xi\tau_0 \tag{4.11}$$

Equation 4.11 shows how ACS implements the local pheromone update where ξ is a parameter in the interval $(0, 1)$. Dorigo and Gambardella [15] reported that experiments led to the conclusion that 0.1 is a good value for ξ when ACS is applied to the travelling salesman problem.

4.4 PROBLEMS ACO HAS BEEN APPLIED TO

4.4.1 *Travelling Salesman Problem*

The travelling salesman problem has served as a benchmark application for nearly all ant colony optimization algorithms. As already mentioned beforehand the travelling salesman problem is the problem of finding the least-cost round-trip given a number of cities. Formally and more generally speaking, it is the problem of finding the least-cost hamiltonian tour in a given graph.

Dorigo and Stützle [16] compared the performance of all ACO variants discussed in this thesis on instances of the travelling salesman problem. They reported that all extensions of Ant System were able to find much better solutions than Ant System itself. Further, it was reported that Ant Colony System (ACS) and Max-Min Ant System were the best performing ACO variants, whereas ACS, being the more aggressive variant, found the best solutions for very short computation times.

4.4.2 *Car Sequencing Problem*

Car manufacturing usually takes place in three successive shops. In the first two shops the body is constructed and painted while the third shop is responsible for assembling all the different components. Each car has

its own configuration that is specified by its options (e.g. air condition, sound system). Some of these options are more labour-intensive than others and therefore it is desirable to find a permutation of the cars to be manufactured that smooths the workload at the workstations. Additionally upper and lower bounds on the number of same-coloured cars that are allowed to be arranged consecutively might be imposed. If this number was too low the painting nozzles in the paint shop would have to be purged more often leading to a wastage of solvent and working time. The motivation behind the upper bound is the assumption that the staff responsible for purging the painting nozzles would get imprecise due to the lack of variation.

For example, constraints for the car sequencing problem can be formulated as follows...

- Out of q successive cars at most p of them may require option o.
- At least l successive cars must require the same colour.
- At most m successive cars may require the same colour.

The goal of the car sequencing problem is now to find a permutation of the cars to be manufactured that minimizes the number of constraint violations.

Solnon presents a constraint solver called Ant-P-Solver in [51] designed to solve a general class of combinatorial problems (permutation constraint satisfaction problems). This constraint solver was successfully applied to the car sequencing problem.

Gottlieb et al. introduce an improved version of Ant-P-Solver in [26] that is more dedicated to the car sequencing problem. Their ACO variant incorporates new greedy heuristics and ideas from Max-Min Ant System (e.g. upper and lower bounds on pheromone trails) discussed in Section 4.3.3.

While [51] and [26] only consider capacity constraints related to the options in the assembly shop, Gagné et al. [21] present an ACO metaheuristic for solving a multi-objective formulation of the problem, i.e. considering other constraints such as those imposed by the paint shop.

4.4.3 *Network Routing Problem*

When a data packet is sent on a computer network such as the internet from one node to another the nodes in between have to forward the packet to a neighbouring node of their choice. They base their decision on local information that is maintained by a routing protocol. A routing protocol defines different methods that can be used to exchange such information about the structure of the network between its nodes. Due to the fact that the topology of a computer network is usually highly dynamic[5] ACO seems to be a very reasonable choice for this kind of application.

Di Caro and Dorigo present a multi-agent system for network routing based on ACO called AntNet in [7]. In this system so-called *Forward Ants* are sent regularly to arbitrary destination nodes. On their way they maintain a stack containing the IDs of the network nodes they traversed and keep track of the time they needed to reach each of them. When the ant arrives at the destination node it generates a *Backward Ant* that takes over the stack and returns to the source node updating the routing information in each node on its way. Di Caro and Dorigo compared two different variants of AntNet to other well known routing algorithms using a realistic simulator of best-effort datagram networks and reported that "both instances of AntNet show superior performance" [7].

4.4.4 *Other Problems*

- The *Total Weighted Tardiness Problem* [10] is the problem of efficiently scheduling n jobs that run (without interruption) on a single machine with respect to various time constraints.
- The *Vehicle Routing Problem* [45] deals with the coordination of a fleet of vehicles that have to deliver goods to a certain number of customers from a central depot such that the total travel time is minimized.

5 Meaning that nodes are frequently added to the network as well as removed from it.

- The goal of the *University Course Timetabling Problem* [50] is to assign a set of lectures to given time slots and rooms such that a set of hard and soft constraints[6] are satisfied.

ACO has been applied to a wide variety of other combinatorial problems. For further information and references on the problems above and numerous other problems take a look at the fifth chapter of [16].

6 A feasible solution must satisfy all hard constraints. Soft constraints determine the quality of a feasible solution.

5

ACO APPROACH FOR TREE AND HYPERTREE DECOMPOSITIONS

In this chapter we will describe our approach of applying the Ant Colony Optimization metaheuristic to the problem of finding tree and generalized hypertree decompositions of small width. We have implemented all of the ACO variants described in Section 4.2 and Section 4.3 with minor modifications that we will point out in this chapter. Furthermore, we combined these variants with different guiding heuristics, local search methods and pheromone update strategies that we will discuss after giving an explanation of the basic structure of the algorithm.

5.1 A HIGH-LEVEL OVERVIEW ON THE ANT COLONY

The ant colony consists of a fixed number of ants. In every iteration each of these ants constructs an elimination ordering by climbing the construction tree. An ant decides which branch to choose next probabilistically based on the pheromone trails and a guiding heuristic. We have implemented the heuristics min-degree and min-fill that can be used as a guiding heuristic alternatively.

After all ants have constructed an elimination ordering an optional local search either tries to improve only the iteration-best or all of the constructed solutions. Two different local search methods (described in Section 5.4) were implemented that also can be used alternatively.

Finally, the ants deposit pheromone proportional to the quality of their solutions onto the edges of the construction graph they traversed. Which ants are allowed to deposit pheromone depends on the particular ACO variant. Moreover, we implemented two different pheromone update strategies that are described in detail in Section 5.3: one that assigns an equal amount of pheromone to all edges belonging to the same solution and another that tries to evaluate each edge individually.

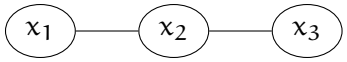


Figure 24. Constraint graph $\mathcal{G}$.

These steps are repeated until one of the following termination criteria is satisfied:

- A given number of iterations have been performed.
- A given number of iterations have passed without improvement of the best solution found so far.
- A given time limit is exceeded.
- The chosen stagnation measure (see Section 5.5) falls below a given value.

Algorithm 5 summarizes this high-level view of the algorithm:

Algorithm 5: High-Level ACO Approach

while *termination criterion not satisfied* **do**
    Solution Construction
    Local Search (optional)
    Pheromone Update

Next we are going to discuss each of these steps in detail. We will start out with the solution construction phase followed by descriptions of the algorithms for the pheromone update and the local search.

5.2 SOLUTION CONSTRUCTION

Figure 24 shows a very simple constraint graph that will serve as an example throughout this chapter.

The goal of our algorithm is now to determine the treewidth of this graph or at least to find a good upper bound for it. Therefore, the ants construct elimination orderings by climbing the so-called construction tree.

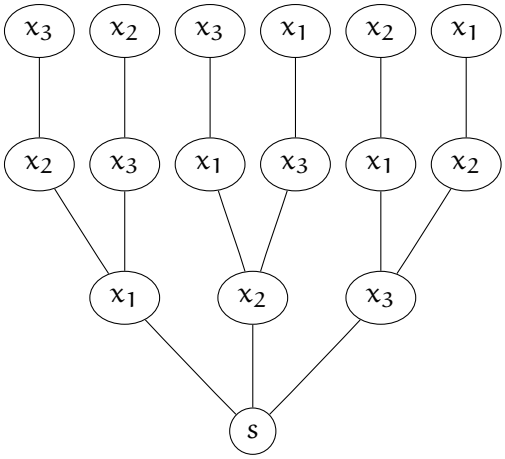


Figure 25. ACO Construction Tree

5.2.1 *Construction Tree*

The construction tree can be obtained from the constraint graph as follows:

1. Create a root node s that will be the starting point of every ant in the colony.
2. For every vertex of the constraint graph append a child node to the root node s.
3. To every leaf node append a child node for every vertex of the constraint graph that is neither represented by the leaf node itself nor by an ancestor of this node.
4. Repeat step 3 until there are no nodes left to append.

Figure 25 illustrates the construction tree obtained by applying this algorithm to the constraint graph in Figure 24.

All possible elimination orderings for the constraint graph can now be represented as a path from the root node s to one of the leaf nodes in the construction tree. Therefore each of the ants finds such a path and at each node on its way the ant decides where to move next probabilistically based on the pheromone trails and a heuristic value both associated with the outgoing edges.

5.2.2 *Pheromone Trails*

A pheromone trail constitutes the desirability to eliminate a certain vertex x after another vertex y. The more pheromone is located on a trail the more likely the corresponding vertex will be chosen by the ant. An obvious way to clearly represent the pheromone trails of our construction tree is the matrix as shown below:

$$\mathcal{T} = \begin{pmatrix} \tau_{x_1x_1} & \tau_{x_1x_2} & \tau_{x_1x_3} \\ \tau_{x_2x_1} & \tau_{x_2x_2} & \tau_{x_2x_3} \\ \tau_{x_3x_1} & \tau_{x_3x_2} & \tau_{x_3x_3} \\ \tau_{sx_1} & \tau_{sx_2} & \tau_{sx_3} \end{pmatrix}$$

Each row contains the amounts of pheromone located on the trails connecting a certain node with all the other nodes. For example, the first row contains the pheromone levels related to the node x_1 describing the desirability of eliminating x_2 ($\tau_{x_1x_2}$) respectively x_3 ($\tau_{x_1x_3}$) immediately after x_1.[1] The last row is dedicated to the root node s that is the starting point for every ant. This is also the reason why there is no fourth column since an ant must not move backwards and thus it is impossible for an ant to return to the root node.

All pheromone trails are initialized to the same value in the beginning of the algorithm that is computed according to the following equation:

$$\tau_{ij} = \frac{m}{W_\eta} \qquad \forall \tau_{ij} \in \mathcal{T}$$

W_η is the width of the decomposition obtained using the guiding heuristic (min-degree or min-fill) while m is the size of the ant colony.

Example 5.1. Let us assume we are using the min-degree heuristic and the ant colony consists of ten ants. Applying the min-degree heuristic would result in a tree decomposition having a width of 1.[2] Thus the pheromone trails are initialized to:

$$\tau_{ij} = \frac{10}{1} = 10 \qquad \forall \tau_{ij} \in \mathcal{T}$$

1 The pheromone trails $\tau_{x_1x_1}$, $\tau_{x_2x_2}$ and $\tau_{x_3x_3}$ are just given for the sake of completeness and are never considered in the solution construction process since it is impossible to eliminate the same vertex more than once.

2 In practice we would now abstain from executing the rest of the algorithm since we have proven that the constraint graph is acyclic.

Figure 26. Elimination graph $E(\mathcal{G}, \langle x_2 \rangle)$.

5.2.3 *Heuristic Information*

As already mentioned, the ants make their decision which vertex to eliminate next not solely based on the pheromone matrix but also consider a guiding heuristic. We have implemented two different heuristics. In order to compute both of these heuristic values we need to maintain a separate graph in addition to the construction tree. We will call this graph the *elimination graph* because this graph is obtained from the original constraint graph by successively eliminating the vertices traversed by the ant in the construction tree. Further, we will denote this graph as $E(\mathcal{G}, \sigma)$ where $\mathcal{G}$ is the original constraint graph and σ is a partial elimination ordering.

Example 5.2. Let us assume an ant has made its first move from the root node s to the node x_2. Consequently the elimination graph, which is equal to the original constraint graph in the beginning, is updated by eliminating the vertex x_2 resulting in the graph $E(\mathcal{G}, \langle x_2 \rangle)$ illustrated in Figure 26.

Min-Degree

The value for the min-degree heuristic is computed according to this equation:

$$\eta_{ij} = \frac{1}{d(j, E(\mathcal{G}, \sigma)) + 1}$$

The expression $d(j, E(\mathcal{G}, \sigma)$ represents the degree of vertex j in the elimination graph $E(\mathcal{G}, \sigma)$.

Example 5.3. Using the min-degree heuristic the following values would be computed when the ant is located at the root node s of the construction tree:

$$\eta_{sx_1} = \frac{1}{d(x_1, E(\mathcal{G}, \langle\rangle)) + 1} = \frac{1}{1+1} = \frac{1}{2}$$
$$\eta_{sx_2} = \frac{1}{d(x_2, E(\mathcal{G}, \langle\rangle)) + 1} = \frac{1}{2+1} = \frac{1}{3}$$
$$\eta_{sx_3} = \frac{1}{d(x_3, E(\mathcal{G}, \langle\rangle)) + 1} = \frac{1}{1+1} = \frac{1}{2}$$

Min-Fill

The value for the min-fill heuristic is computed according to this equation:

$$\eta_{ij} = \frac{1}{f(j, E(\mathcal{G}, \sigma)) + 1}$$

The expression $f(j, E(\mathcal{G}, \sigma)$ represents the number of edges that would be added to the elimination graph due to the elimination of vertex j.

Example 5.4. Using the min-fill heuristic the following values would be computed when the ant is located at the root node s of the construction tree:

$$\eta_{sx_1} = \frac{1}{f(x_1, E(\mathcal{G}, \langle\rangle)) + 1} = \frac{1}{0+1} = 1$$
$$\eta_{sx_2} = \frac{1}{f(x_2, E(\mathcal{G}, \langle\rangle)) + 1} = \frac{1}{1+1} = \frac{1}{2}$$
$$\eta_{sx_3} = \frac{1}{f(x_3, E(\mathcal{G}, \langle\rangle)) + 1} = \frac{1}{0+1} = 1$$

5.2.4 *Probabilistic Vertex Elimination*

We will now take a more detailed look on how exactly the ants move from node to node on the construction tree. All of the ACO variants with the exception of Ant Colony System use Equation 5.1 alone to compute the probability p_{ij} of moving from a node i to another node j

where α and β are parameters that can be passed to the algorithm in order to weight the pheromone trails and the heuristic values.

$$p_{ij} = \frac{[\tau_{ij}]^{\alpha} [\eta_{ij}]^{\beta}}{\sum_{l \in E(\mathcal{G},\sigma)} [\tau_{il}]^{\alpha} [\eta_{il}]^{\beta}}, \qquad \text{if } j \in E(\mathcal{G},\sigma) \tag{5.1}$$

This probability is computed for each vertex left in the elimination graph. According to these probabilities the ant decides which vertex to eliminate next.

Example 5.5. Under the assumptions that min-fill is used as the guiding heuristic, $\alpha = 1$, $\beta = 1$, $m = 10$ and that the ant is located at the root node s of our example construction tree, we compute the probabilities that the ant moves to the nodes x_1, x_2 respectively x_3.

First of all we compute the denominator of Equation 5.1:

$$\sum_{l \in E(\mathcal{G},\langle\rangle)} [\tau_{sl}]^1 [\eta_{sl}]^1 = (10 \cdot 1) + (10 \cdot 1/2) + (10 \cdot 1) = 25$$

After that we just need to insert appropriate term for each candidate node into the fraction:

$$p_{sx_1} = \frac{10}{25} = \frac{2}{5}$$
$$p_{sx_2} = \frac{5}{25} = \frac{1}{5}$$
$$p_{sx_3} = \frac{10}{25} = \frac{2}{5}$$

Hence, the probability that the ant moves from the root node s to x_2 is one fifth while the probability that it moves to x_1 or x_2 is for each two fifth.

As described in Section 4.3.4, Ant Colony System introduces an additional parameter q_0 that constitutes the probability that the ant moves to the "best" node instead of making a probabilistic decision:

$$j = \begin{cases} \arg\max_{l \in E(\mathcal{G},\sigma)} \{[\tau_{il}]^{\alpha} [\eta_{il}]^{\beta}\}, & \text{if } q \leqslant q_0; \\ \text{Equation 5.1}, & \text{otherwise;} \end{cases} \tag{5.2}$$

If a randomly generated number q in the interval of $[0, 1]$ is less or equal q_0 then the ant moves to the node that otherwise would have the highest probability to be chosen. Ties are broken randomly.

Example 5.6. Given the same conditions as in Example 5.5, we apply Ant Colony System and set $q_0 = 0.3$. We generate a random number in the interval of $[0, 1]$ which happens to be 0.24. Since 0.24 is less than 0.3 we make a random choice between x_1 and x_2 because both share the highest probability of being chosen according to Equation 5.1.

Ant Colony System also introduces a so-called local pheromone update. After an ant has constructed its solution it removes pheromone from the trails belonging to its solution according to the following equation whereas ξ is a variant-specific parameter and τ_0 is initial amount of pheromone:

$$\tau_{ij} \leftarrow (1 - \xi)\tau_{ij} + \xi\tau_0$$

The motivation for this is to diversify the search so that subsequent ants will more likely choose other branches of the construction tree.

5.3 PHEROMONE UPDATE

After each of the ants has constructed an elimination ordering (that optionally has been improved by a local search thereafter) the values in the pheromone matrix are updated reflecting the quality of the constructed solutions which will enable the subsequent ants in the following iteration to make decisions in a more informed manner. Moreover, pheromone is removed from the pheromone trails so poor solutions can be forgotten that might have been the best known solutions in earlier iterations of the algorithm.

5.3.1 *Pheromone Deposition*

Given an elimination ordering σ_k that was constructed by an ant k we need to determine for each subsequent elimination (i, j) in σ_k the amount of pheromone that will be deposited on the corresponding pheromone trail τ_{ij}. We implemented an edge-independent and an edge-specific pheromone update strategy. The first adds the same amount of pheromone to all trails belonging to σ_k while the latter adds more or less pheromone to individual trails depending on the quality of a certain elimination.

Edge-Independent Pheromone Deposition

The edge-independent pheromone update strategy adds the reciprocal value of the tree decomposition's width to all pheromone trails that are part of σ_k:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{W(\sigma_k)}, & \text{if } (i,j) \text{ belongs to } \sigma_k; \\ 0, & \text{otherwise;} \end{cases}$$

Example 5.7. Let us assume that an ant k has constructed the elimination ordering $\sigma_k = \langle x_2, x_1, x_3 \rangle$ for our example constraint graph $\mathcal{G}$ in Figure 24. The resulting tree decomposition has width two. Hence, the following pheromone trails will receive this pheromone addition:

$$\Delta\tau_{sx_2}^{k} = \Delta\tau_{x_2x_1}^{k} = \Delta\tau_{x_1x_3}^{k} = \frac{1}{2}$$

Edge-Specific Pheromone Deposition

In contrast to the edge-independent update strategy the edge-specific update strategy deposits different amounts of pheromone onto the trails belonging to the same elimination ordering:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{d(j,E(\mathcal{G},\sigma_{kj}))/|E(\mathcal{G},\sigma_{kj})|} \cdot \frac{1}{W(\sigma_k)}, & \text{if } (i,j) \text{ belongs to } \sigma_k; \\ 0, & \text{otherwise;} \end{cases}$$

This amount depends on the ratio between the degree of the vertex j when it was eliminated $d(j, E(\mathcal{G}, \sigma_{kj}))$[3] and the number of vertices left in the elimination graph $|E(\mathcal{G}, \sigma_{kj})|$ at that time. The lower the degree and the greater the number of vertices not eliminated yet the more pheromone will be deposited. The idea behind this is to favor eliminations that produce tree decomposition nodes of small width early in the elimination process because those eliminations usually influence the quality of the overall solution most.

Example 5.8. As in Example 5.7, the elimination ordering $\sigma_k = \langle x_2, x_1, x_3 \rangle$ is given which produces a tree decomposition of width two. For the computation of the values for $\Delta\tau_{ij}^{k}$ we need to consider the state of

3 σ_{kj} is the partial elimination ordering that is obtained from σ_k by omitting j and all vertices that are eliminated after j.

the elimination graph when j was eliminated. At first x_2 is eliminated which has a degree of two in the constraint graph consisting of three vertices. If we now insert the values for $d(x_2, E(\mathcal{G}, \langle\rangle))$, $|E(\mathcal{G}, \langle\rangle)|$ and $W(\sigma_k)$ into our equation we obtain the following result:

$$\Delta\tau_{sx_2} = \frac{1}{2/3} \cdot \frac{1}{2} = \frac{3}{2} \cdot \frac{1}{2} = \frac{3}{4}$$

Due to the elimination of x_2, an edge is introduced between the vertices x_1 and x_3 in the elimination graph. Next, x_1 is eliminated which has a degree of one in $E(\mathcal{G}, \langle x_2 \rangle)$ that now only consists of two vertices. Using these values we can compute the amount of pheromone to be added to $\tau_{x_2x_1}$ and after that $\Delta\tau_{x_1x_3}$ can be determined similarly:

$$\Delta\tau_{x_2x_1} = \frac{1}{1/2} \cdot \frac{1}{2} = \frac{2}{1} \cdot \frac{1}{2} = 1$$
$$\Delta\tau_{x_1x_3} = \frac{1}{1/1} \cdot \frac{1}{2} = \frac{1}{1} \cdot \frac{1}{2} = \frac{1}{2}$$

Differences between ACO variants

As already discussed in Section 4.2.3 and Section 4.3, which ants are allowed to deposit pheromone and how this pheromone is weighted varies between the different ACO variants. For the sake of completeness we also summarize these differences in the following.

In Simple Ant System all ants are allowed to deposit pheromone. None of these depositions are weighted.

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

All ants deposit pheromone in Elitist Ant System as well. Additionally the ant bs that has found the best known solution out of all iterations is also allowed to deposit pheromone. Moreover, this amount of pheromone is weighted with the variant-specific parameter e.

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + e\Delta\tau_{ij}^{bs}$$

In Rank-Based Ant System only w ants deposit pheromone whereas w is a variant-specific parameter. Among those ants are the $w-1$ best ants of the iteration and the ant that has found the best known solution.

All of these depositions are weighted according to their rank among all of these ants whereas the pheromone of the ant having the best known solution is weighted with the factor w.

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1} (w - r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs}$$

Max-Min Ant System allows either the iteration-best ant or the ant that has found the best known solution to deposit pheromone. Which of these two ants deposits pheromone in a certain iteration is determined based on a frequency f that is given as a variant-specific parameter. For instance, if $f = 3$ the ant with the best known solution is allowed to deposit pheromone in every third iteration. In all other iterations the iteration-best ant updates the pheromone trails. Furthermore, Max-Min Ant System imposes upper on lower bounds on the values of the pheromone trails τ_{ij}. The upper bound is set to the initial pheromone value that is computed as described in Section 5.2.2 while the lower bound is set to a fraction of the upper bound whereas the denominator is given as the variant-specific parameter a.

In Ant Colony System only the ant having the best known solution is allowed to deposit pheromone:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{bs}$$

5.3.2 *Pheromone Evaporation*

After the pheromone has been added to the trails a certain amount of pheromone is removed. This amount is determined based on the pheromone evaporation rate ρ:

$$\tau_{ij} = (1 - \rho)\tau_{ij} \qquad \forall\tau_{ij} \in \mathcal{T}$$

While all the other ACO variants remove pheromone from every pheromone trail Ant Colony System only removes pheromone from the trails belonging to the best known elimination ordering σ_{bs}:

$$\tau_{ij} = (1 - \rho)\tau_{ij} \qquad \forall(i, j) \in \sigma_{bs}$$

Note that our implementation of Ant Colony System differs a little bit from the one that is discussed in Section 4.3.4 which combines pheromone decomposition and evaporation into one equation.

5.4 LOCAL SEARCH

All ACO variants can optionally be extended with one of two local search methods. Both of these algorithms try to improve the quality of the solutions that were constructed by the ant colony by changing the positions of certain vertices in the elimination orderings.

The local search methods are alternatively applied to the elimination ordering of the iteration-best ant or to all solutions constructed in the corresponding iteration depending on the configuration of the algorithm.

5.4.1 *Hill Climbing*

A local search that is based on the concept of *hill climbing* defines a function that returns a set of solutions when given another solution. This set of solutions is called the solution's neighbourhood. If this neighbourhood contains a solution that is better than the initial solution the neighbourhood of this better solution is evaluated. This is repeated until a neighbourhood is generated that does not contain any better solution. This would mean that a local optimum has been found.

Our hill climbing local search (documented as Algorithm 6) determines the vertices that had the greatest degrees in the elimination graph when being eliminated according to the given elimination ordering. If this is more than one vertex, one of these is chosen randomly. The neighbourhood is now defined as the set of elimination orderings that can be generated by swapping this vertex with any other vertex in the elimination ordering. This neighbourhood is searched and when a better elimination ordering is found the neighbourhood of this ordering is evaluated.

5.4.2 *Iterated Local Search*

The other local search method is an iterated local search similar to the algorithm proposed by Musliu [41] which is also briefly described in Section 3.4.5 on Page 36 of this thesis.

Algorithm 6: Hill Climbing for Tree Decompositions.

Input: an elimination ordering $\sigma = (v_1, \ldots, v_n)$
$\sigma_{best} = \sigma$
$\sigma_{current} = \sigma$
repeat
  Let v_j be one of the vertices causing the largest clique in $\sigma_{current}$
  $i = 1$
  better-solution = false
  while $i \leqslant n \wedge \neg$better-solution **do**
    $\sigma_{neighbour} = \sigma_{current}$
    swap v_i and v_j in $\sigma_{neighbour}$
    if $W(\sigma_{neighbour}) < W(\sigma_{current})$ **then**
      better-solution = true
      $\sigma_{current} = \sigma_{neighbour}$
      $\sigma_{best} = \sigma_{current}$
    $i = i + 1$;
until $\neg$better-solution
return σ_{best}

Local Search

The local search being iterated corresponds to the function ConstructionPhase that is called in Algorithm 4 on Page 37. It takes as input an elimination ordering and randomly selects a vertex out of the vertices causing the largest cliques when eliminated. This vertex is then swapped with another randomly chosen vertex in the elimination ordering. This is repeated until the best known elimination ordering has not been improved this way for a given number of subsequent iterations.

Acceptance Criterion

If the elimination ordering returned by the local search yields a decomposition having smaller width than the width of the best known decomposition plus three, then this elimination ordering is used in the perturbation mechanism that is discussed next. Otherwise the best known elimination ordering is going to be perturbated.

Perturbation

We have implemented two perturbation mechanisms called MaxCliquePer and RandPert that gave the best results in [41] when both were combined.

- MaxCliquePer determines the vertices in the elimination ordering causing the largest cliques when being eliminated and inserts each one of them into a randomly chosen new position.
- RandPert selects two vertices at random and moves each one of them into a new random position in the ordering. This differs from the perturbation proposed by Musliu which adjusts the number of vertices being moved depending on the feedback of the search process.

The algorithm described in [41] switches between these two mechanisms when the iterated local search runs for 100 iterations without improvement. In contrast, our implementation chooses one of these perturbation methods randomly in each iteration.

Termination Criterion

If the iterated local search is unable to find a new best solution for a given number of subsequent iterations, the algorithm stops and returns the best known elimination ordering.

5.5 STAGNATION MEASURES

If the distribution of the pheromone on the trails becomes too unbalanced due to the pheromone depositions, the ants will generate very similar solutions causing the search to stagnate. In order to enable the algorithm to detect such situations we have implemented two stagnation measures proposed by Dorigo and Stützle [16] that indicate how explorative the search behaviour of the ants is. The ACO algorithms can be configured to terminate if the chosen stagnation measure falls below a certain value.

5.5.1 *Variation Coefficient*

The variation coefficient is defined as the quotient between the standard deviation of the widths of the constructed decompositions and their average width. The drawback of this measure is that it ignores the structure of the elimination orderings. Theoretically, it is possible that the ants construct very different elimination orderings that happen to yield decompositions of equal width. The advantage of this measure is that is can be implemented very efficiently.

5.5.2 $\overline{\lambda}$ *Branching Factor*

The $\overline{\lambda}$ branching factor [16] is more meaningful than the variation coefficient because it evaluates the pheromone matrix that is more significant concerning search stagnation. It takes a look at every vertex i and determines its λ branching factor that is defined as the number of vertices j that fulfill the following equation for i:

$$\tau_{ij} \geqslant \tau^i_{min} + \lambda(\tau^i_{max} - \tau^i_{min})$$

We decided to set λ to 0.05 for our implementation as was done by Dorigo and Stützle [16] for the evaluation of the stagnation behaviour of their ACO algorithms for the TSP.[4]

The $\overline{\lambda}$ branching factor is defined as the average of all branching factors. Consequently, it represents the average number of vertices that have a relatively high probability of being chosen by an ant during the construction of an elimination ordering.

4 The value for λ could range over the interval $[0, 1]$ whereas higher values cause the measure to be more sensitive concerning the detection of stagnation.

6

IMPLEMENTATION

In this chapter we will describe the most important aspects of our implementation of the algorithms that were presented in Chapter 5. Further, we present a C++ library for the Ant Colony Optimization metaheuristic called *libaco* that was implemented in the course of this thesis.

All of the following implementation artefacts resulted from this thesis and their current source code is available under the GNU Lesser General Public License[1] at `http://code.google.com/p/libaco/`:

- The *acotreewidth* program is the command line application implementing the problem specific algorithms for tree and hypertree decompositions. This is the application that was used to obtain the computational results reported in Chapter 7. A description of the command line interface of the *acotreewidth* program is given as Appendix A.1.
- The entire logic specific to the ACO metaheuristic was encapsulated in a library we decided to name *libaco*. This library is also used by the acotreewidth application.
- Another library called *liblocalsearch* contains the basic logic of the local search methods described in Section 5.4.
- In order to demonstrate the flexibility of the libaco library, we have implemented the *acotsp* application that applies the functionality of the library to the travelling salesman problem.
- The project *acotemplate* serves as a template for developers who wish to implement a command line client similar to acotreewidth and acotsp for another combinatorial optimization problem.

1 `http://www.gnu.org/licenses/lgpl.html`

The exact versions of these components that were used for the computational experiments described in Chapter 7 can be downloaded from http://libaco.googlecode.com/files/libaco-thesis.tar.gz.

6.1 IMPLEMENTATION DETAILS

We used the C++ programming language and its standard library for all of our practical work. The command line applications additionally make use of the *Templatized C++ Command Line Parser Library*[2] for the implementation of their command line interface.

The source code was compiled with the C++ compiler that is part of the *GNU Compiler Collection* (version 4.1.3). The software was compiled and tested on the 64 bit server edition of Ubuntu[3] 7.10 and the 32 bit desktop edition of Ubuntu 8.10.

6.1.1 *Vertex Elimination Algorithm*

The implementation of the vertex elimination algorithm is very performance-critical. In the following we will describe the data structures and the operations dealing with them during the elimination process.

Data Structures

The data structures given below are used to represent and keep track of the structure of the elimination graph:

- An $n \times n$ (n representing the number of vertices in the constraint graph) adjacency matrix T consisting of boolean values. If a vertex x_i is connected to another vertex x_j, then $T[i][j] = T[j][i] = \mathit{true}$.
- Another $n \times n$ matrix A consisting of integer values whereas the i^{th} row corresponds to the adjacency list of the vertex x_i. An adjacency list contains the neighbours of a certain vertex x_i. Additionally to this matrix we need to maintain the following arrays:

2 http://tclap.sourceforge.net/
3 http://www.ubuntu.com

- An array of integers L that holds the initial length of the adjacency list for each vertex of the elimination graph.
- An array of integers D that maintains the length of the adjacency list for each vertex of the elimination graph during the elimination process.
- An array of boolean values E that keeps track of which vertices have been eliminated. For instance, if E[i] is true, then the vertex x_i has already been eliminated.

Algorithm

The vertex elimination algorithm is executed every time an ant constructs a solution and if $\beta > 0$. We do not need to maintain an elimination graph if beta equals 0 since the heuristic information does not influence the movement of the ants. In this case we return the same heuristic value for each feasible vertex. However, if the maintenance of an elimination graph is necessary and the current ant moves to another vertex x_i in the construction tree the following changes have to be applied to the graph:

1. An edge has to be inserted for every pair of neighbours of x_i that are not already connected. The neighbours of x_i are obtained by looking at every vertex x_j from A[i][0] to A[i][D[i]] and by collecting those vertices that have not been eliminated yet (i.e. E[j] is false). Then an edge is introduced for every pair (x_k, x_l) of non-adjacent (i.e. T[k][l] is false) neighbours of x_i. This is achieved by performing the following operations:
   - Setting T[k][l] and T[l][k] to true.
   - Appending x_k to A[l] and x_l to A[k].
   - Increasing D[k] and D[l] by one.
2. Afterwards E[i] is set to true since x_i has just been eliminated.

The vertex elimination algorithm is also executed every time a complete elimination ordering is evaluated. In this case the algorithm above is optimized as proposed by Golumbic [25]. Instead of adding edges for all neighbours of x_i that are not connected yet, we only introduce edges for the neighbour of x_i that is going to be eliminated next. This

optimization is also applied during the evaluation of the solutions generated by the local search procedure. The reason why this optimization is not applicable while the ants are constructing solutions is that the neighbour that is eliminated next cannot be determined since the ants are moving probabilistically.

Another minor optimization we have implemented implicates that the vertex elimination algorithm terminates when the number of vertices left in the elimination graph is less than the width of the tree decomposition that was generated by the eliminations so far.

After the vertex elimination algorithm terminates the original constraint graph is restored from the existing data structures as follows:

1. For each vertex x_i set $E[i]$ to false.
2. For each vertex x_i set $D[i]$ to $L[i]$.
3. Set all values in the adjacency matrix T to false.
4. For each vertex x_i set $T[i][j]$ to true if x_j is an element of the adjacency list $A[i][0]$ to $A[i][L[i]]$.

6.2 THE LIBACO LIBRARY

The *libaco* library implements the following variants of Ant Colony Optimization algorithms that are described in Chapter 4:

- Simple Ant System (class *SimpleAntColony*)
- Elitist Ant System (class *ElitistAntColony*)
- Rank-Based Ant System (class *RankBasedAntColony*)
- Max-Min Ant System (class *MaxMinAntColony*)
- Ant Colony System (class *ACSAntColony*)

The library takes care of all problem-independent details of these algorithms. That is to say, the client application does not need to worry about the probabilistic construction of solutions, manipulating the pheromone matrix or remembering the best solution among many other things. What the client application is required to define are such

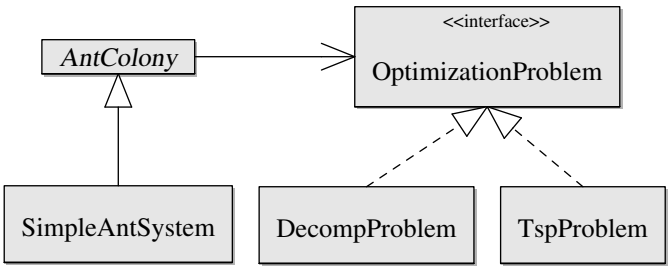


Figure 27. Simplified libaco class diagram.

things as the structure of the construction graph and a function that measures the quality of a constructed solution.

6.2.1 *Interface*

Figure 27 shows a class diagram that illustrates the basic structure of the interface that connects the client code and the library. The client code needs to implement the *OptimizationProblem* interface that defines a number of callbacks that are called by the library.[4] An instance of the resulting class (e.g. *TspProblem*) is passed to an instance of a class that extends the abstract class called *AntColony*.[5] Listing 1 demonstrates this using an instance of the class *TspProblem* that is passed to the constructor of *SimpleAntColony*.[6] The second argument to this constructor is an instance of the *AntColonyConfiguration* class. This class contains public member variables representing the ACO parameters that all variants have in common. The constructor of *AntColonyConfiguration* sets these parameters to default values. In Listing 1 we override this default value for α on line three.

Once the ant colony has been instantiated we can run iterations of the algorithm by calling the *run*-method as on line six and seven in Listing 1. On line eight and line nine we retrieve the best solution that

4 In the following we will often refer to classes implementing this interface as *client classes*.

5 In order to save some space only the class *SimpleAntSystem* is given in the class diagram. Actually all other implementations of ACO variants inherit from *AntColony* too.

6 In this example, we assume that *create_tsp_problem* takes care of creating an instance of *TspProblem* (reading the file containing the coordinates of the cities, etc.).

```
1  void main() {
2    AntColonyConfiguration config;
3    config.alpha = 3.0;
4    TspProblem problem = create_tsp_problem();
5    SimpleAntColony colony(tsp, config);
6    colony.run(); // run one iteration
7    colony.run(); // run a second iteration
8    std::vector<unsigned int> tour = colony.get_best_tour();
9    double length = colony.get_best_tour_length();
10 }
```

Listing 1. Running Simple Ant System.

was found among all iterations and its "length".[7] As can be noticed in this example, the vertices of the construction graph are represented as unsigned integers. Consequently, the representation of an ant's tour is a vector of unsigned integers.

6.2.2 *Configuration*

An ant colony is configured with an instance of the corresponding configuration class (i.e. an *ElitistAntColony* expects an *ElitistAntColonyConfiguration*). Table 2 lists the available parameters of every configuration class and their default values. Note that each variant-specific configuration class inherits all parameters from *AntColonyConfiguration* and defines its additional variant-specific parameters.

6.2.3 *Libaco Implementation for the TSP*

We will now take a closer look at the libaco implementation for the travelling salesman problem. Listing 2 contains the *OptimizationProblem* interface that the library client class *TspProblem* needs to implement.

The *TspProblem* class additionally contains three different member variables:

7 The smaller the length the better the solution.

8 The type of the local_search parameter is *AntColonyConfiguration::LocalSearchType*, an enumeration containing values for LS_NONE, LS_ITERATION_BEST and LS_ALL.

CLASS	PARAMETER NAME	DEFAULT
AntColonyConfiguration	alpha	2.0
	beta	5.0
	number_of_ants	20
	evaporation_rate	0.1
	initial_pheromone	1.0
	local_search[8]	LS_ITERATION_BEST
ElitistAntColonyConfiguration	elitist_weight	2.0
RankBasedAntColonyConfiguration	elitist_ants	1
MaxMinAntColonyConfiguration	best_so_far_frequency	3
	a	5
ACSAntColonyConfiguration	xi	0.1
	q0	0.5

Table 2. Libaco configuration parameters.

```
1  class OptimizationProblem {
2    public:
3      virtual ~OptimizationProblem() {}
4      virtual unsigned int get_max_tour_size() = 0;
5      virtual unsigned int number_of_vertices() = 0;
6      virtual std::map<unsigned int,double>
           get_feasible_start_vertices() = 0;
7      virtual std::map<unsigned int,double>
           get_feasible_neighbours(unsigned int vertex) = 0;
8      virtual double eval_tour(const std::vector<unsigned int> &
           tour) = 0;
9      virtual double pheromone_update(unsigned int vertex, double
            tour_length) = 0;
10     virtual void added_vertex_to_tour(unsigned int vertex) = 0;
11     virtual bool is_tour_complete(const std::vector<unsigned
           int> &tour) = 0;
12     virtual std::vector<unsigned int> apply_local_search(const
           std::vector<unsigned int> &tour) { return tour; }
13     virtual void cleanup() = 0;
14 };
```

Listing 2. The *OptimizationProblem* interface.

```
1 std::map<unsigned int,double> get_feasible_start_vertices() {
2   std::map<unsigned int,double> vertices;
3   start_vertex_ = Util::random_number(distances_->rows());
4   vertices[start_vertex_] = 1.0;
5   return vertices;
6 }
```

Listing 3. Declaring the ant's starting point.

- A quadratic matrix of unsigned integers *distances_* containing the distances between the cities. For instance, *(*distances_)[2][5]* gives the distance between city 2 and 5. Thus, the number of rows respectively the number of columns of the matrix corresponds to the number of cities the problem instance consists of.
- A map *visited_vertices_* that contains a boolean value for each city indicating whether this city has been visited yet.
- The *start_vertex_* representing the starting point of the tour. This is also where the tour has to end.

Before an ant starts its tour on the construction graph a starting point has to be determined. Therefore the library will call the method of our *TspProblem* class given in Listing 3. This method is supposed to return a map. Each key of this map corresponds to a feasible starting point for the ant. The value assigned to each of these keys is the heuristic value η associated with this vertex. The greater this value the more promising is the choice of this vertex for the ant.

For the travelling salesman problem the first city of the tour is irrelevant regarding the quality of the overall solution. This is why we just choose a random starting point and set its heuristic value arbitrarily to 1.0[9]. Further, we remember the starting point of our tour in the member variable *start_vertex_* because our tour has to end at the same city.

After the ant has moved to a new vertex in the construction graph the library notifies the client code of its decision by calling the method given in Listing 4.

9 We could set this value to any other positive value since we return the randomly chosen city as the only feasible starting point.

```
1 void added_vertex_to_tour(unsigned int vertex) {
2   visited_vertices_[vertex] = true;
3 }
```

Listing 4. Ant has moved on the construction graph.

```
1  std::map<unsigned int,double> get_feasible_neighbours(unsigned
       int vertex) {
2    std::map<unsigned int,double> vertices;
3    for(unsigned int i=0;i<distances_->cols();i++) {
4      if (!visited_vertices_[i]) {
5        unsigned int distance = (*distances_)[vertex][i];
6        vertices[i] = 1.0 / (distance);
7      }
8    }
9
10   if(vertices.size() == 0) {
11     vertices[start_vertex_] = 1.0;
12   }
13   return vertices;
14 }
```

Listing 5. Declaring feasible neighbours.

Due to the fact that a valid tour for the travelling salesman problem must not visit the same city twice (with the exception of the starting city) we need to remember the cities the ant has already visited. Therefore we update the map that maintains a boolean value for each vertex indicating whether this vertex has been already visited.

When an ant wants to move from one vertex to another the library calls the method illustrated in Listing 5 in order to obtain a map containing the feasible neighbours of the ant's current vertex and their corresponding heuristic values.

For the travelling salesman problem all vertices are feasible that have not been visited yet. The heuristic value that is assigned to each feasible vertex is the reciprocal value of the distance between this vertex and the ant's current vertex (line 5 and line 6). If there are no vertices left that have not been visited yet the ant has to return to its starting point (lines 10 to 12).

```
1 bool is_tour_complete(const std::vector<unsigned int> &tour) {
2   return tour.size() == (distances_->rows() + 1);
3 }
```

Listing 6. Deciding whether the given solution is complete.

```
1 double eval_tour(const std::vector<unsigned int> &tour) {
2   double length = 0.0;
3   for(unsigned int i=1;i<tour.size();i++) {
4     length += (*distances_)[tour[i-1]][tour[i]];
5   }
6   return length;
7 }
```

Listing 7. Evaluating a solution.

The client code also needs to specify whether a given solution is complete. Therefore the library calls the method in Listing 6 every time an ant has added a vertex to its tour. If the method returns the value *true* the ant is finished with the construction of its solution.

A solution for the travelling salesman problem is complete if all cities have been visited once and the ant has returned to its starting point. Therefore a given tour is complete if its size equals the number of cities plus one.[10]

When the ant has constructed a complete solution the library will call the method given as Listing 7 for the evaluation of the solution. Libaco acts on the assumption that it is dealing with a minimization problem. Hence, libaco expects that the client code returns smaller values for better solutions.

The quality of a given solution for the travelling salesman problem is defined as the sum of the distances that are covered by the tour.

The library also demands to know how much pheromone to deposit on the individual pheromone trails belonging to a certain solution.

10 The starting point is the first and last city of the tour while all other cities have been visited once.

```
1 double pheromone_update(unsigned int v, double tour_length) {
2   return 1.0 / tour_length;
3 }
```

Listing 8. Deciding how much pheromone to deposit.

```
1 void cleanup() {
2   visited_vertices_.clear();
3 }
```

Listing 9. Clean-up actions after a solution has been constructed.

Therefore it calls the method in Listing 8 supplying it with a vertex v that is part of the overall solution[11] and the length of this solution.

We add the same amount of pheromone to all trails belonging to the same solution. This amount of pheromone is defined as the reciprocal value of the tour's length so the ants will deposit more pheromone for shorter tours.

Once the ant is finished with the construction of its tour the library calls the *cleanup*-methods of our *OptimizationProblem*. This is where the state of the library client can be reset so that the next ant is able to construct its solution.

The *cleanup* method of the *TspProblem* class needs to clear the map of visited vertices (Listing 9) since the subsequent ant has not visited any vertices yet.

6.2.4 *Template Project*

The *acotemplate* project is a good starting point to experiment with libaco. It already contains the code for a command line application that is similar to the *acotsp* and *acotreewidth* programs. In order to apply this code to another combinatorial optimization problem the following two things have to be done:

11 This vertex is passed to the method so the client code can deposit different amounts of pheromone for trails belonging to the same solution depending on the quality of certain decisions made by the ants.

- The *Problem* class has to be implemented. Its definition and the stubs of its methods reside in the files *include/acotemplate/template.h* and *src/template.cpp*.
- An instance of this class has to be created and initialized in the *main* method that is defined in the file *src/acotemplate.cpp*. Where exactly this has to happen is marked by a multi-line comment.

When this has been accomplished the program can be compiled by issuing the *make* command in the root directory of the project:

```
~/libaco/acotemplate/trunk$ make
```

This will also cause the compilation of the *libaco* and *liblocalsearch* projects that have to reside on the same directory level as the *acotemplate* project. In this example the makefile will assume that these projects are located at *~/libaco/libaco* respectively *~/libaco/liblocalsearch*.

If the compilation is successful a binary called *acotemplate* will reside in the *bin/* directory of the project. In order to get an overview of the different command line options execute the binary with the help flag:

```
~/libaco/acotemplate/trunk$ ./bin/acotemplate --help
```

7

COMPUTATIONAL RESULTS

Within this chapter we document our experiments with different ACO algorithm configurations in Section 7.1. Based on the results of these experiments we determine the best ACO algorithms and apply these to instances of the Second DIMACS Graph Coloring Challenge [2] and instances of the CSP Hypergraph Library [22]. These final results are presented in Section 7.2.

7.1 EXPERIMENTS

In order to evaluate and compare the performance of the different ACO algorithms discussed in Chapter 5, a series of experiments was planned whereas each of these experiments was designed with a certain research goal in mind:

1. Identification of good values for variant-independent parameters.
2. Comparison of the guiding heuristics min-fill and min-degree.
3. Evaluation and optimization of each ACO variant.
4. Comparison of all ACO variants. Identification of best performing variant.
5. Comparison of edge-specific and edge-independent pheromone update strategies using the best performing ACO variant.
6. Comparison of local search methods using the best performing ACO variant.

All of these experiments were performed for the ten instances of the DIMACS Graph Coloring Challenge listed in Table 3. Column $|V|$ contains the number of vertices graph wheres column $|E|$ presents the number of edges belonging to the constraint graph. In the following, we will refer to this set of instances as the *experimental set*.

Instance	|V|	|E|
DSJC125.1	125	736
games120	120	638
le450_5a	450	5714
le450_5b	450	5734
miles500	128	1170
myciel6	95	755
myciel7	191	2360
queen12_12	144	2596
queen8_8	64	728
school1	385	19095

Table 3. Experimental Set

Since good elimination orderings for tree decompositions usually also give good results for generalized hypertree decompositions no special experiments were performed for any hypergraph instances. Instead of that we simply extended the best ACO algorithm for tree decomposition with the greedy set cover algorithm described in Section 2.6 in order to obtain the results for generalized hypertree decomposition presented in Section 7.2.2.

7.1.1 *Tuning the Variant-Independent Parameters*

All ACO variants with the exception of Simple Ant System have their own variant-specific parameters. Therefore we decided to experiment with the variant-independent parameters before optimizing the additional parameters of each ACO variant.

No experiments were executed to investigate different values for the following two parameters:

τ_0 As described in Section 5.2.2 we initialize the pheromone trails to m/W_η because according to [16] it is a good heuristic to initialize the pheromone trails to a value that is slightly higher than the expected amount of pheromone deposited by the ants in one iteration. If τ_0 was too low the first (probably suboptimal) solutions generated by the ants would bias the search of the ants in the

second iteration too much. Otherwise if τ_0 was too high it would take numerous iterations until enough pheromone evaporates and the pheromone trails would make a difference in the search behaviour of the ants.

ρ The pheromone evaporation rate ρ is the fraction of pheromone that is removed after each iteration from the pheromone trails. If this rate is too high the algorithm will not be able to focus on a certain search space. On the contrary, if this rate is too low the algorithm might not be able to "forget" suboptimal solutions causing stagnation of the search. We decided to set ρ to 0.1 for our experiments because that also worked well for the travelling salesman problem according to [16].

For all other variant-independent parameters we compared the performance of a set of different values in a series of experiments.

Tuning α and β

We found empirically that the results of all variants seem to be highly dependent on the ratio between the parameters α and β where much greater values for β seem to favor the generation of tree decompositions of small width. Therefore we decided to experiment with different combinations of values for α and β:

$$[(\alpha, \beta), \dots] = [(1, 10), (2, 20), (3, 30), (5, 40), (2, 50)]$$

Each variant used the min-degree guiding heuristic and performed five runs per combination on each of the ten instances part of the experimental set with a time limit of 200 seconds. The time limit is checked after each iteration of the algorithm. If it is not exceeded another iteration is completely executed regardless of whether the time limit is exceeded during execution.

All other parameters that we were going to investigate later have been set to values that were obtained through trial and error:

- Variant-independent parameters: $m = 10$
- Elitist Ant System: $e = 3$

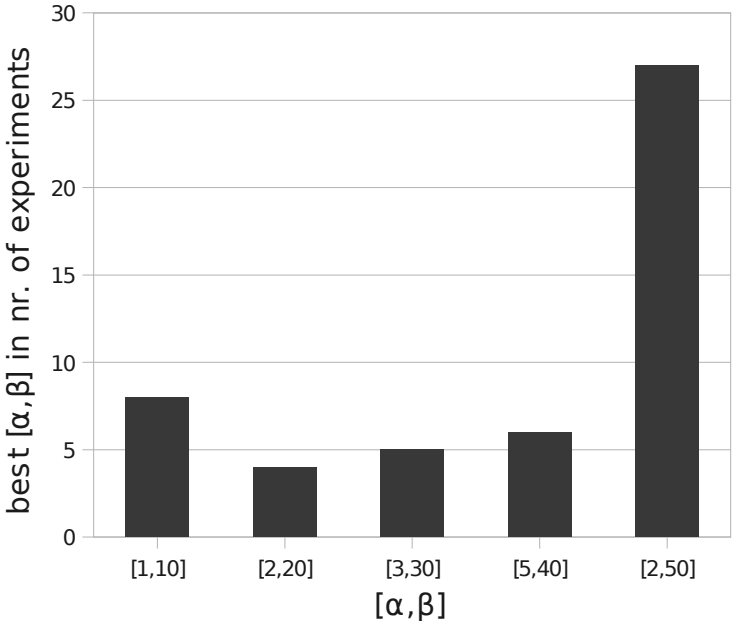


Figure 28. Comparison of combinations for α and β.

- Rank-Based Ant System: $w = 3$
- Max-Min Ant System: $a = 5$, $f = 3$
- Ant Colony System: $q_0 = 0.3$, $\xi = 0.1$

The bar chart in Figure 28 illustrates for how many experiments a certain (α,β)-combination yielded the best average width over five runs. If multiple combinations yielded equal average widths, the combination that found its best solution on average in the least amount of time is considered to be the best one. The combination of $\alpha = 2$ and $\beta = 50$ outperformed the other combinations in 27 of 50 experiments.

Table 4 contains the same data as Figure 28 but it is broken down to the individual ACO variants. We find that the (α, β)-combination (2,50) also yielded the best average width for most problem instances for every ACO variant.

Due to the fact that our results show that for the majority of problem instances the combination of $\alpha = 2$ and $\beta = 50$ gave the best

Variant/(α,β)	(1,10)	(2,20)	(3,30)	(5,40)	(2,50)
Simple Ant System	4	0	0	0	**6**
Elitist Ant System	3	0	1	0	**6**
Rank-Based Ant System	1	1	1	1	**6**
Max-Min Ant System	0	0	3	3	**4**
Ant Colony System	0	3	0	2	**5**

Table 4. Number of best solutions per variant

Instance	Minimum Width		Average Width	
	min-degree	min-fill	min-degree	min-fill
DSJC125.1	64	**63**	64	**63.2**
games120	37	**35**	37.2	**36.6**
le450_5a	312	**296**	313.4	**301**
le450_5b	312	**305**	315.1	**306.4**
miles500	25	**23**	25	**23**
myciel6	35	35	35	35
myciel7	68	**66**	68.8	**66**
queen12_12	111	**109**	112.6	**109.8**
queen8_8	47	47	47	47
school1	231	**225**	234	**225**

Table 5. Comparison of guiding heuristics.

results, we decided to keep these parameter settings for the subsequent experiments.

Comparison of Guiding Heuristics

We compared the min-degree and min-fill heuristics by applying each ACO variant to every problem instance in the experimental set with either of them. Due to the results of our prior experiments α was set to 2 and β to 50 while all other parameters have been retained unchanged. Again, five runs were performed per algorithm configuration with a time limit of 200 seconds.

Table 5 contains the best minimum widths and the best average widths of the tree decompositions generated by the ACO algorithms using the min-degree and the min-fill heuristic.

Variant/m	5	10	20	50	100
Simple Ant System	0	1	1	3	**5**
Elitist Ant System	0	1	0	3	**6**
Rank-Based Ant System	1	0	2	**4**	3
Max-Min Ant System	1	1	**4**	3	1
Ant Colony System	**3**	2	2	1	2

Table 6. Comparison of ant colony sizes.

The results clearly indicate that the min-fill heuristic gives better results. Nonetheless, the min-degree heuristic is much more time-efficient. For instance, the ACO algorithms were only able to complete one iteration within 200 seconds for the problem instance le450_5a using the min-fill heuristic while 144 iterations could be performed using the min-degree heuristic. This is why we decided to use the min-degree heuristic for all remaining experiments since we would otherwise be unable to investigate the impact of the pheromone trails on the search behaviour of the ants due to the small number of iterations.

Tuning the Number of Ants

The number of ants influences the quality of the pheromone update. The more ants construct solutions per iteration the more useful the pheromone matrix will be for the ants in the following iterations. For example, if only one ant constructs a solution per iteration this solution probably will be suboptimal. Nevertheless, only the edges associated with this solution will receive additional pheromone. On the contrary, if a hundred ants construct solutions per iteration this solution will receive less pheromone than other better solutions.

We experimented with ant colonies consisting of 5, 10, 20, 50 and 100 ants. Each ACO variant performed five runs for each instance of the experimental set using ant colonies of these different sizes.

Table 6 shows for how many instances each variant yielded the best average width over five runs using a certain number of ants.

The results suggest that especially Simple Ant System and Elitist Ant System seem to prefer bigger ant colonies. This is quite coherent since those are the only ACO variants that allow all of their ants to deposit

pheromone and therefore might require more ants in order to obtain a useful pheromone matrix.

For our subsequent experiments we decided based on these results to use ant colonies consisting of 100 ants for Simple and Elitist Ant System, 50 ants for Rank-Based Ant System, 20 ants for Max-Min Ant System and five ants for Ant Colony System.

7.1.2 *Tuning the Variant-Specific Parameters*

Elitist Ant System

Elitist Ant System adds pheromone to the trails belonging to the best known solution in every iteration. This pheromone update is multiplied by an elitist weight e. We have applied Elitist Ant System to every instance of the experimental set with different elitist weights of 2, 4, 6 and 10. Five runs were performed for each value. For four instances Elitist Ant System gave the best average width over these five runs using an elitist weight of 10. Elitist Ant System with $e = 6$ respectively $e = 4$ delivered the best average width for three instances in each case while the algorithm with e set to 2 never returned the best average width.

Rank-Based Ant System

Rank-Based Ant System allows a certain number of ants w to deposit pheromone in every iteration. We have compared the average widths over five runs achieved by the algorithm for each instance of the experimental set when setting w to 3, 5, 7 and 10. As can be seen in Figure 29, we have obtained the best results for the majority of problem instances when w was set to 10.

Max-Min Ant System

For Max-Min Ant System we experimented with the following different combinations of the parameters a and f:

$$[(a, f), \ldots] = [(10, 2), (3, 5), (10, 5), (3, 2)]$$

The combination $(10, 2)$ results in a more focused search while the combination $(3, 5)$ should cause the algorithm to be more explorative.

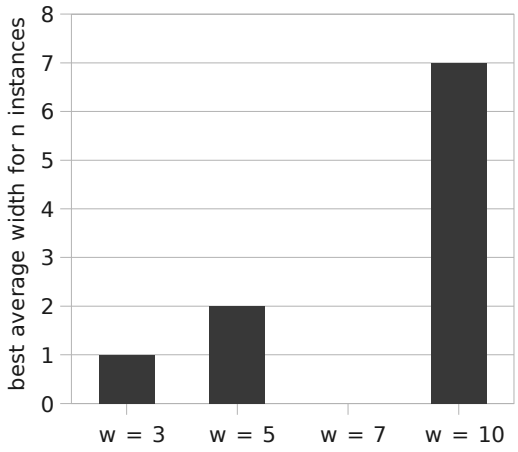


Figure 29. Comparison of different values for parameter w of Rank-Based Ant System. The best average width over 10 runs was obtained for seven problem instances with $w = 10$.

The other two combinations represent trade-offs between these two approaches.

Max-Min Ant System performed best on four instances of the experimental set using the explorative $(3, 5)$ parameter combination. The combinations $(10, 2)$ and $(10, 5)$ each caused Min-Max Ant System to yield the best average width over five runs for three problem instances.

Ant Colony System

Ant Colony System defines two additional variant-specific parameters q_0 and ξ. Again, we experimented with different combinations of parameter values:

$$[(q_0, \xi), \dots] = [(0.1, 0.3), (0.5, 0.05), (0.1, 0.05), (0.5, 0.3)]$$

The results of our experiments clearly suggest that $(0.5, 0.3)$ causes Ant Colony System to give the best results among all of these combinations. For eight out of the ten instances of the experimental set $(0.5, 0.3)$ outperformed all other combinations. The best performing combination for the other two problem instances was $(0.1, 0.05)$.

Instance	Minimum Width					Average Width				
	SAS	EAS	RAS	MAS	ACS	SAS	EAS	RAS	MAS	ACS
DSJC125.1	65	65	65	64	**63**	65.6	65.4	65.4	64	**63.8**
games120	37	38	38	37	37	38.8	38.8	38.6	37.4	**37**
le450_5a	311	312	**303**	308	309	313.6	314	**310.2**	312.8	311.4
le450_5b	313	314	311	**307**	312	313.6	315.8	314.8	**313.2**	313.4
miles500	25	25	25	25	25	25.4	25.2	25.8	**25**	25.2
myciel6	35	35	35	35	35	35	35	35	35	35
myciel7	68	68	69	68	69	68.8	68.8	69	68.8	69
queen12_12	114	113	112	112	**111**	114.6	114	114.4	113.2	**112**
queen8_8	48	48	48	47	47	48	48	48.2	47	47
school1	237	231	232	**228**	232	238	235.2	235.4	233.4	**233.2**

Table 7. Comparison of minimum and average widths achieved by all ACO variants over 10 runs. Given in bold are those values that represent single-best solutions among all variants.

7.1.3 *Comparison of ACO Variants*

After we had found good parameter settings for each ACO variant we were now ready to compare them. Therefore, five runs were performed by each variant for every instance of the experimental set with $\alpha = 2$, $\beta = 50$, $\rho = 0.1$ and a time-limit of 500 seconds. Min-degree was used as the guiding heuristic. Additionally, the following parameter settings were applied based on the results of the prior experiments:

- Simple Ant System: $m = 100$
- Elitist Ant System: $m = 100$, $e = 10$
- Rank-Based Ant System: $m = 50$, $w = 10$
- Max-Min Ant System: $m = 20$, $a = 3$, $f = 5$
- Ant Colony System: $m = 5$, $q_0 = 0.5$, $\xi = 0.3$

Table 7 lists the minimum and the average width achieved by each ACO variant for each problem instance. According to these results Max-Min Ant System and Ant Colony System performed slightly better than

the other variants. Only once, for the problem instance le450_5a, Rank-Based Ant System was able to achieve better results than Max-Min Ant System and Ant Colony System. For all other problem instances one of these two variants delivered the best minimum and average width. Since Ant Colony System more often gave the single best solution among all variants, we decided to focus our remaining investigations on this ACO variant.

Convergence Behaviour

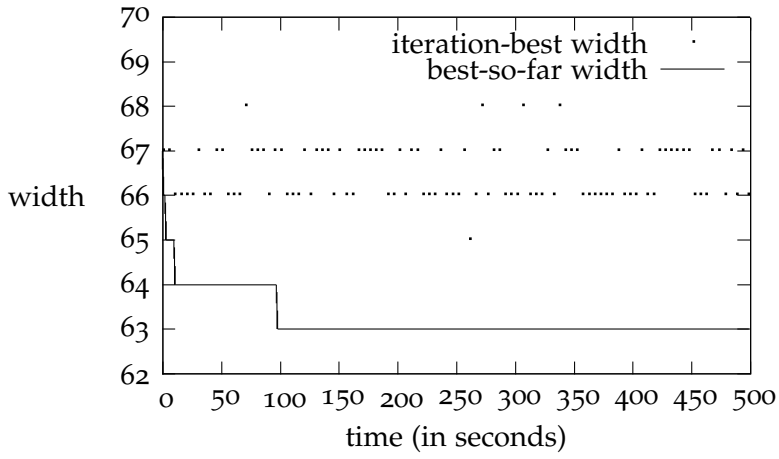
We noticed while analyzing these results that all variants found their best results or results nearly as good already in the first couple of iterations. This implies that the pheromone trails play a minor role in the search for the best result since the algorithms stagnate very early. We suppose that this is also the reason why the ACO variants MAS and ACS give slightly better results. MAS stays explorative because of the lower and upper bounds imposed on the pheromone trails while ACS performs a local pheromone update that removes pheromone from the trails belonging to solutions which were already constructed.

The charts in Figure 30 document the progress of the ACO variants ACS and RAS for a run on the problem instance DSJC125.1. RAS stagnates right after the first iteration while ACS stays more explorative and finds its best solution after approximately 100 seconds. ACS performed 18384, RAS 1748 iterations within their 500 seconds of running time. This difference is due to the different numbers of ants used by each variant ($m = 5$ for ACS, $m = 50$ for RAS). Just a subset of all iteration-best solutions is illustrated in Figure 30 for presentability reasons.

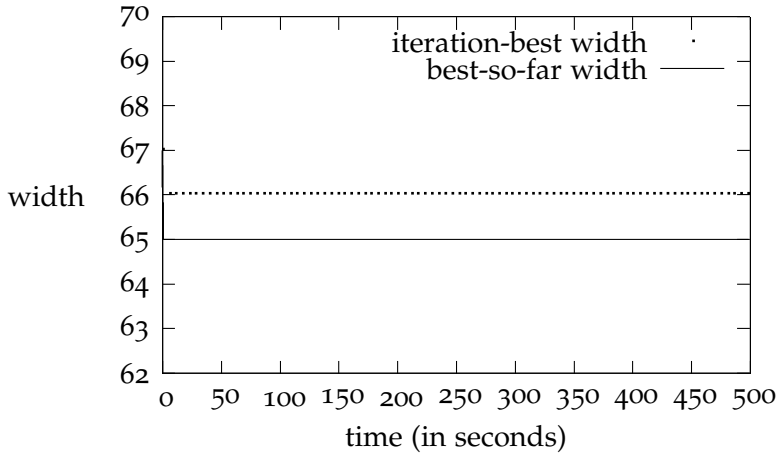
7.1.4 *Comparison of Pheromone Update Strategies*

In Section 5.3.1 we present two different pheromone update strategies. In order to compare them we have applied Ant Colony System with either of them using the same parameter settings as in the experiment described in the previous section. Again, 10 runs were performed for each problem instance with a time-limit of 500 seconds.

Table 8 gives the minimum, maximum and average widths achieved by Ant Colony System with the edge-independent (EI) and the edge-



(a) Ant Colony System (ACS)



(b) Rank-Based Ant System (RAS)

Figure 30. Convergence behaviour of the algorithms ACS and RAS for the problem instance DSJC125.1. Both find their best solution within the first 100 seconds. However, while ACS keeps exploring the search space, RAS stagnates and keeps generating tree decompositions of width 66.

Instance	|V| / |E|	EI-min	ES-min	EI-max	ES-max	EI-avg	ES-avg
DSJC125.1	125 / 736	63	63	64	64	63.8	63.8
games120	120 / 638	37	37	37	37	37	37
le450_5a	450 / 5714	311	**309**	313	**312**	312.4	**310.4**
le450_5b	450 / 5734	314	**313**	315	315	314.4	**314.2**
miles500	128 / 1170	25	25	26	**25**	25.2	**25**
myciel6	95 / 755	35	35	35	35	35	35
myciel7	191 / 2360	69	69	69	69	69	69
queen12_12	144 / 2596	111	111	113	113	**112.2**	112.6
queen8_8	64 / 728	47	**46**	47	47	47	**46.8**
school1	385 / 19095	233	**228**	235	**234**	233.6	**231.6**

Table 8. Comparison of minimum, maximum and average widths over five runs achieved by the edge-independent (EI) and edge-specific (ES) pheromone update strategies. The edge-specific pheromone update strategy gave slightly better results on some problem instances.

specific (ES) pheromone update strategy. The edge-specific pheromone update strategy seems to give slightly better results than the edge-independent strategy. Therefore, we decided to apply the edge-specific pheromone update strategy for all subsequent experiments.

7.1.5 *Combining ACO with a Local Search Method*

Our final experiments dealt with the combination of Ant Colony System with the iterated local search and the hill climbing algorithm described in Section 5.4. We have applied Ant Colony System to all instances of the experimental set combining the algorithm with each of the following local search methods:

ILS: The iterated local search whereas the local search being iterated terminates after 5 iterations without improvement of the input solution while the iterative algorithm terminates after 20 iterations without improvement of the best known solution. We obtained this configuration after investigating three different combinations of these parameters: $(20, 5)$, $(10, 10)$ and $(5, 20)$. The experiments

Instance	|V| / |E|	ACS+ILS			ACS+HC		
		min	max	avg	min	max	avg
DSJC125.1	125 / 736	63	64	63.8	63	64	63.8
games120	120 / 638	37	37	37	37	37	37
le450_5a	450 / 5714	306	308	306.8	**305**	308	**306.6**
le450_5b	450 / 5734	**306**	312	309.4	308	**310**	**308.8**
miles500	128 / 1170	25	**25**	**25**	25	26	25.2
myciel6	95 / 755	35	35	35	35	35	35
myciel7	191 / 2360	**67**	**68**	**67.2**	69	69	69
queen12_12	144 / 2596	**109**	**110**	**109.8**	111	112	111.6
queen8_8	64 / 728	**46**	**46**	**46**	47	47	47
school1	385 / 19095	**229**	**232**	**230.2**	231	233	232

Table 9. Minimum, maximum and average widths achieved over five runs by Ant Colony System in combination with the Iterated Local Search (ACS+ILS) and by Ant Colony System in combination with the Hill Climbing algorithm (ACS+HC).

showed that the combination $(5, 20)$ was the best of all three for 6 out of all 10 problem instances.

HC: The hill climbing algorithm as described in Section 5.4.

Each combination of Ant Colony System and local search performed five runs. Table 9 lists the results of each of these hybrid algorithms.

Ant Colony System in combination with the iterated local search clearly outperformed the hybrid algorithm consisting of Ant Colony System and the hill climbing local search. ACS+HC was only able to give better results than ACS+ILS for two out of the ten problem instances.

7.2 RESULTS

Based on the results of our experiments we decided to obtain the final results of this thesis with the Ant Colony System ACO variant and the following parameter settings: $\alpha = 2$, $\beta = 50$, $\rho = 0.1$, $m = 5$, $q_0 = 0.5$, $\xi = 0.3$. Further, we chose min-degree as the guiding heuristic and

used the edge-specific pheromone update strategy. We will refer to this algorithm in the following with the abbreviation ACS.

Additionally, we present in this section the results delivered by the ACS algorithm when it is combined with the iterated local search ILS. We will refer to this extended algorithm with the abbreviation ACS+ILS in the following.

For generalized hypertree decomposition, we extended the ACS algorithm with a greedy set cover algorithm and applied the resulting algorithm ACS+SC to 19 instances of the CSP hypergraph library.

All results reported in this thesis have been obtained on a machine equipped with 48GB of memory and two Intel(R) Xeon(R) CPUs (E5345) having a clock rate of 2.33GHz.

7.2.1 *ACO for Tree Decompositions*

Results of ACS and ACS+ILS

In Table 10 and Table 11 we present the minimum, maximum and average widths obtained in 10 runs with ACS and ACS+ILS for 62 DIMACS graph coloring instances. Each run was performed with a time-limit of 1000 seconds. The second column gives the number of vertices and the number of edges a certain problem instance consists of.

For 25 problem instances ACS+ILS gave a better minimum width than ACS on its own. ACS was never able to outperform ACS+ILS with the exception of the problem instances *inithx.i.2* and *inithx.i.3* for which ACS achieved a better average width than ACS+ILS.

Comparison with other decomposition methods

Table 12 on Page 98 and Table 13 on Page 99 additionally contain the best upper bounds on treewidth obtained with other decomposition methods from the literature:

- The column KBH contains the best results from a set of algorithms proposed by Koster, Bodlaender and Hoesel in [35]. The results reported were obtained with a Pentium 3 800MHz processor.

Instance	|V| / |E|	ACS			ACS+ILS		
		min	max	avg	min	max	avg
anna	138 / 986	12	12	12	12	12	12
david	87 / 812	13	13	13	13	13	13
huck	74 / 602	10	10	10	10	10	10
homer	561 / 3258	31	31	31	30	31	30.1
jean	80 / 508	9	9	9	9	9	9
games120	120 / 638	37	37	37	37	37	37
queen5_5	25 / 160	18	18	18	18	18	18
queen6_6	36 / 290	25	25	25	25	25	25
queen7_7	49 / 476	35	35	35	35	35	35
queen8_8	64 / 728	47	47	47	46	46	46
queen9_9	81 / 1056	60	60	60	59	59	59
queen10_10	100 / 1470	75	76	75.5	73	74	73.8
queen11_11	121 / 1980	92	93	92.6	89	91	90.4
queen12_12	144 / 2596	110	113	111.7	109	110	109.5
queen13_13	169 / 3328	132	134	133.1	128	130	129.4
queen14_14	196 / 4186	154	157	156	150	153	152.1
queen15_15	225 / 5180	178	182	181	174	177	175.9
queen16_16	256 / 6320	206	211	208.5	201	204	202.5
fpsol2.i.1	269 / 11654	66	66	66	66	66	66
fpsol2.i.2	363 / 8691	31	31	31	31	31	31
fpsol2.i.3	363 / 8688	31	31	31	31	31	31
inithx.i.1	519 / 18707	56	56	56	56	56	56
inithx.i.2	558 / 13979	31	32	31.6	31	32	31.8
inithx.i.3	559 / 13969	31	33	31.5	31	33	32
miles1000	128 / 3216	52	53	52.9	50	50	50
miles1500	128 / 5198	77	77	77	77	77	77
miles250	125 / 387	9	9	9	9	9	9
miles500	128 / 1170	25	26	25.1	25	25	25
miles750	128 / 2113	38	38	38	38	38	38
mulsol.i.1	138 / 3925	50	50	50	50	50	50
mulsol.i.2	173 / 3885	32	32	32	32	32	32
mulsol.i.3	174 / 3916	32	32	32	32	32	32
mulsol.i.4	175 / 3946	32	32	32	32	32	32
mulsol.i.5	176 / 3973	31	31	31	31	31	31
myciel3	11 / 20	5	5	5	5	5	5
myciel4	23 / 71	10	10	10	10	10	10
myciel5	47 / 236	19	19	19	19	19	19
myciel6	95 / 755	35	35	35	35	35	35
myciel7	191 / 2360	69	69	69	66	68	67.1

Table 10. Results of ACS and ACS+ILS for Tree Decompositions.

Instance	|V| / |E|	ACS			ACS+ILS		
		min	max	avg	min	max	avg
school1	385 / 19095	231	234	232.5	228	232	229.8
school1_nsh	352 / 14612	189	190	189.1	185	190	188.3
zeroin.i.1	126 / 4100	50	50	50	50	50	50
zeroin.i.2	157 / 3541	33	33	33	33	33	33
zeroin.i.3	157 / 3540	33	33	33	33	33	33
le450_5a	450 / 5714	309	313	311.1	304	308	305.6
le450_5b	450 / 5734	308	316	311.7	308	311	309.3
le450_5c	450 / 9803	315	318	316.9	309	318	313.9
le450_5d	450 / 9757	301	309	304	290	303	298.4
le450_15a	450 / 8168	294	297	295.3	288	293	291.2
le450_15b	450 / 8169	295	298	296.6	292	294	293.1
le450_15c	450 / 16680	372	374	373.2	368	372	370.4
le450_15d	450 / 16750	373	375	374.3	371	373	372.3
le450_25a	450 / 8260	251	254	253.4	249	252	250.6
le450_25b	450 / 8263	254	257	256	245	257	253.4
le450_25c	450 / 17343	355	358	356.4	346	352	350.2
le450_25d	450 / 17425	359	362	360.8	355	358	356.8
dsjc125.1	125 / 736	63	64	63.9	63	64	63.3
dsjc125.5	125 / 3891	108	109	108.7	108	109	108.2
dsjc125.9	125 / 6961	119	119	119	119	119	119
dsjc250.1	250 / 3218	174	176	175.6	174	176	175.3
dsjc250.5	250 / 15668	231	231	231	231	231	231
dsjc250.9	250 / 27897	243	243	243	243	243	243

Table 11. Results of ACS and ACS+ILS for Tree Decompositions.

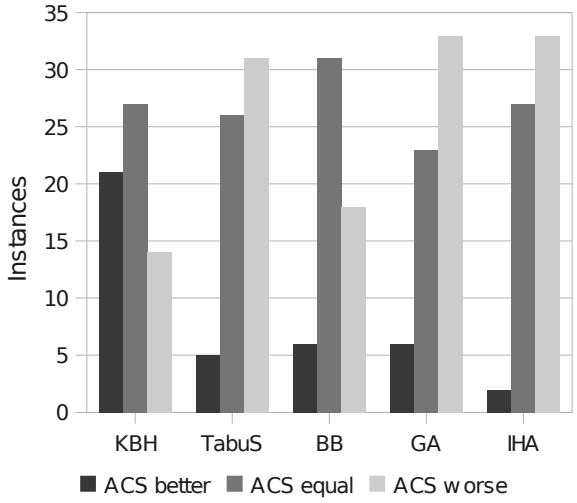
- TabuS contains the results reported for the Tabu Search algorithm proposed in [8] which were obtained with a Pentium 3 1 GHz processor.
- BB is the branch and bound algorithm presented by Gogate and Dechter in [24]. The experiments for this algorithm were performed on a Pentium 4 2.4 GHz processor using 2 GB of memory.
- The column GA contains the best results obtained with the genetic algorithm proposed by Musliu and Schafhauser in [42]. They used an Intel(R) Xeon(TM) 3.2 GHz processor and 4 GB of memory for their experiments.
- IHA stands for the iterative heuristic algorithm presented by Musliu in [41]. These results were obtained with a Pentium 4 processor 3 GHz and 1 GB of RAM.

These results indicate that the ACO algorithms were able to give results comparable to those of the other decomposition methods for many problem instances. ACS+ILS was even able to find an improved upper bound of 30 for the problem instance *homer.col*. By applying the ACS+ILS algorithm with the min-fill heuristic we could further improve the upper bound for this instance to 29. Nonetheless, especially for more complex problem instances both ACO algorithms gave results inferior to those of most other algorithms.

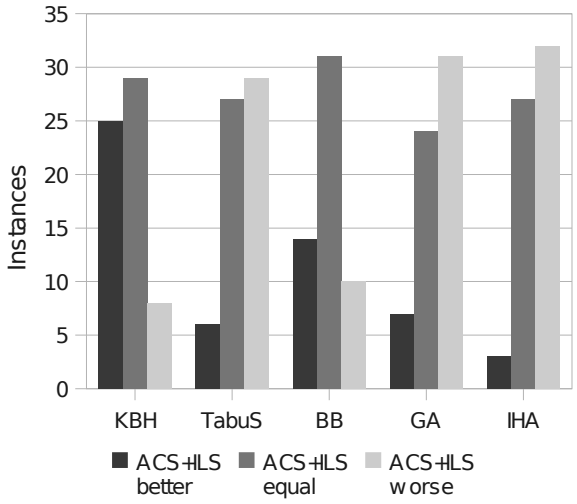
Figure 31 on Page 97 visualizes for how many problem instances ACS respectively ACS+ILS gave a better, equal or worse minimum width compared with each of the other decomposition methods. As can be seen, both algorithms outperformed KBH on more instances than vice versa but only ACS+ILS also managed to outperform BB.

Time-performance

Table 14 on Page 101 and Table 15 on Page 102 list the time-performance given in seconds for the algorithms TabuS, GA, IHA, ACS and ACS+ILS. The values in the column TabuS represent the overall running time of the algorithm whereas the number of iterations is limited to 20000 and the algorithm terminates after 10000 iterations without improvement. The column GA gives the running time that was necessary to obtain



(a) ACS



(b) ACS+ILS

Figure 31. Comparison of ACO algorithms for the generation of tree decompositions with other decomposition methods.

Instance	|V| / |E|	KBH	TabuS	BB	GA	IHA	ACS	ACS+ILS
anna	138 / 986	12	12	12	12	12	12	12
david	87 / 812	13	13	13	13	13	13	13
huck	74 / 602	10	10	10	10	10	10	10
homer	561 / 3258	31	31	31	31	31	31	**30**
jean	80 / 508	9	9	9	9	9	9	9
games120	120 / 638	37	33	-	32	32	37	37
queen5_5	25 / 160	18	18	18	18	18	18	18
queen6_6	36 / 290	26	25	25	26	25	25	25
queen7_7	49 / 476	35	35	35	35	35	35	35
queen8_8	64 / 728	46	46	46	45	45	47	46
queen9_9	81 / 1056	59	58	59	58	58	60	59
queen10_10	100 / 1470	73	72	72	72	72	75	73
queen11_11	121 / 1980	89	88	89	87	87	92	89
queen12_12	144 / 2596	106	104	110	104	103	110	109
queen13_13	169 / 3328	125	122	125	121	121	132	128
queen14_14	196 / 4186	145	141	143	141	140	154	150
queen15_15	225 / 5180	167	163	167	162	162	178	174
queen16_16	256 / 6320	191	186	205	186	186	206	201
fpsol2.i.1	269 / 11654	66	66	66	66	66	66	66
fpsol2.i.2	363 / 8691	31	31	31	32	31	31	31
fpsol2.i.3	363 / 8688	31	31	31	31	31	31	31
inithx.i.1	519 / 18707	56	56	56	56	56	56	56
inithx.i.2	558 / 13979	35	35	31	35	35	31	31
inithx.i.3	559 / 13969	35	35	31	35	35	31	31
miles1000	128 / 3216	49	49	49	50	49	52	50
miles1500	128 / 5198	77	77	77	77	77	77	77
miles250	125 / 387	9	9	9	10	9	9	9
miles500	128 / 1170	22	22	22	24	22	25	25
miles750	128 / 2113	37	36	37	37	36	38	38
mulsol.i.1	138 / 3925	50	50	50	50	50	50	50
mulsol.i.2	173 / 3885	32	32	32	32	32	32	32
mulsol.i.3	174 / 3916	32	32	32	32	32	32	32
mulsol.i.4	175 / 3946	32	32	32	32	32	32	32
mulsol.i.5	176 / 3973	31	31	31	31	31	31	31
myciel3	11 / 20	5	5	5	5	5	5	5
myciel4	23 / 71	11	10	10	10	10	10	10
myciel5	47 / 236	20	19	19	19	19	19	19
myciel6	95 / 755	35	35	35	35	35	35	35
myciel7	191 / 2360	74	66	54	66	66	69	66

Table 12. Comparison with other tree decomposition methods.

Instance	|V| / |E|	KBH	TabuS	BB	GA	IHA	ACS	ACS+ILS
school1	385 / 19095	244	188	-	185	178	231	228
school1_nsh	352 / 14612	192	162	-	157	152	189	185
zeroin.i.1	126 / 4100	50	50	-	50	50	50	50
zeroin.i.2	157 / 3541	33	32	-	32	32	33	33
zeroin.i.3	157 / 3540	33	32	-	32	32	33	33
le450_5a	450 / 5714	310	256	307	243	244	309	304
le450_5b	450 / 5734	313	254	309	248	246	308	308
le450_5c	450 / 9803	340	272	315	265	266	315	309
le450_5d	450 / 9757	326	278	303	265	265	301	290
le450_15a	450 / 8168	296	272	-	265	262	294	288
le450_15b	450 / 8169	296	270	289	265	258	295	292
le450_15c	450 / 16680	376	359	372	351	350	372	368
le450_15d	450 / 16750	375	360	371	353	355	373	371
le450_25a	450 / 8260	255	234	255	225	216	251	249
le450_25b	450 / 8263	251	233	251	227	219	254	245
le450_25c	450 / 17343	355	327	349	320	322	355	346
le450_25d	450 / 17425	356	336	349	327	328	359	355
dsjc125.1	125 / 736	67	65	64	61	60	63	63
dsjc125.5	125 / 3891	110	109	109	109	108	108	108
dsjc125.9	125 / 6961	119	119	119	119	119	119	119
dsjc250.1	250 / 3218	179	173	176	169	167	174	174
dsjc250.5	250 / 15668	233	232	231	230	229	231	231
dsjc250.9	250 / 27897	243	243	243	243	243	243	243

Table 13. Comparison with other tree decomposition methods.

the best result whereas the numbers listed in the column IHA represent the average running time that was necessary to obtain the best result of a certain run. In the columns t-min and t-avg we give the running time of our ACO algorithms that passed until the minimum width was found respectively the running time that passed on average until the best result of a certain run was found.

Based on the results in Table 14 and Table 15 we see that the time performance of our ACO algorithms is good.

7.2.2 *ACO for Generalized Hypertree Decompositions*

In Table 16 on Page 103 we present the minimum, maximum and average widths obtained in 10 runs with ACS+SC for 19 instances of the CSP Hypergraph Library. Each run was performed with a time-limit of 1000 seconds. The second column of the table gives the number of vertices and the number of hyperedges a certain hypergraph contains. The columns t-min and t-avg give the running time (in seconds) that was necessary to obtain the minimum width respectively the average running time that was necessary to obtain the best result of a certain run.

Table 17 gives a comparison of our results with those of other methods for hypertree decompositions. GA-ghw is a genetic algorithm proposed in [42]. The BE algorithm described in [13] creates tree decompositions according to the heuristics *maximum cardinality*, *min-degree*, and *min-fill* and afterwards applies two set cover heuristics in order to obtain hypertree decompositions. The decomposition having minimum width is then returned by the algorithm. All other methods from [13] are heuristic algorithms based on hypergraph partitioning. The results from [42] were obtained with an Intel(R) Xeon(TM) 3.20 GHz processor having 4 GB RAM. An Intel Xeon (dual) 2.20 GHz processor having 2 GB of memory was used for the experimental results reported in [13].

ACS+SC outperformed the hypergraph partitioning algorithm based on Tabu Search TS on all except of three problem instances. The algorithms BE, GA-ghw, and HM were able to clearly outperform our algorithm. The algorithm FM seems to give results comparable to those of ACS+SC for many problem instances. The chart in Figure 32 on

Instance	TabuS	GA	IHA	ACS		ACS+ILS	
				t-min	t-avg	t-min	t-avg
anna	2776.93	213	0.1	0.02	0.03	0.03	0.04
david	796.81	154	0.1	0.01	0.01	0.01	0.02
huck	488.76	120	0.1	0.01	0.01	0.01	0.01
homer	157716.56	1118	127	57.32	176.45	99.02	284.7
jean	513.76	120	0	0	0.01	0.01	0.02
games120	2372.71	462	145.8	0	0	0.67	59.09
queen5_5	100.36	33	0.1	1.45	330.36	0	0
queen6_6	225.55	51	0.1	29.47	230.72	0.01	3.49
queen7_7	322.4	92	0.1	0.62	20.72	0.08	1.1
queen8_8	617.57	167	28.8	4.55	77.95	3.01	24.23
queen9_9	1527.13	230	5.2	127.62	303.22	6.36	57.45
queen10_10	3532.78	339	28.3	100.84	264.49	449.59	288.06
queen11_11	5395.74	497	29.6	84.75	238.96	508.03	326.31
queen12_12	10345.14	633	106.7	528.98	379.04	113.59	319.71
queen13_13	16769.58	906	3266.12	93.78	272.51	431.46	240.95
queen14_14	29479.91	1181	5282.2	414.21	354.45	851.22	473.45
queen15_15	47856.25	1544	3029.51	951.32	411.4	347.41	526.45
queen16_16	73373.12	2093	7764.57	605.99	462.37	493.97	449.53
fpsol2.i.1	63050.58	1982	4.8	0.42	0.55	0.49	0.66
fpsol2.i.2	78770.05	1445	8.4	0.33	0.37	0.4	0.46
fpsol2.i.3	79132.7	1462	8.7	0.29	0.33	0.35	0.4
inithx.i.1	101007.52	3378	10.2	1.28	1.68	1.46	1.5
inithx.i.2	121353.69	2317	11.7	75.99	171.61	81.48	108.39
inithx.i.3	119080.85	2261	10.6	128.54	458.93	745.31	284.82
miles1000	5696.73	559	54.2	190.86	19.11	5.19	39.32
miles1500	6290.44	457	0.7	0.03	0.15	0.11	0.24
miles250	1898.29	242	2.9	0.02	0.02	0.02	0.03
miles500	4659.31	442	81	4.2	252.68	0.42	29.19
miles750	3585.68	536	112.2	0.06	0.44	0.17	0.57
mulsol.i.1	3226.77	671	0.1	0.06	0.08	0.08	0.1
mulsol.i.2	12310.37	584	0.8	0.05	0.06	0.08	0.08
mulsol.i.3	9201.45	579	0.5	0.05	0.06	0.07	0.08
mulsol.i.4	8040.28	578	0.9	0.05	0.06	0.08	0.08
mulsol.i.5	13014.81	584	1.1	0.05	0.06	0.08	0.09
myciel3	72.5	14	0.1	0	0	0	0
myciel4	84.31	34	0.1	0.03	0.2	0	0
myciel5	211.73	80	0.1	0	0.01	0	0.01
myciel6	1992.42	232	0.4	0.01	0.03	0.02	0.02
myciel7	19924.58	757	18.2	0.06	4.55	333.24	371.29

Table 14. Time performance of ACS and ACS+ILS for Tree Decomposition.

Instance	TabuS	GA	IHA	ACS		ACS+ILS	
				t-min	t-avg	t-min	t-avg
school1	137966.73	4684	5157.13	774.38	509.12	106.56	377.63
school1_nsh	180300.1	4239	5468.9	127.62	303.22	771.42	480.92
zeroin.i.1	2595.92	641	0.1	0.07	0.08	0.09	0.1
zeroin.i.2	4825.51	594	43	0.07	0.07	0.09	0.09
zeroin.i.3	8898.8	585	22	0.06	0.07	0.09	0.09
le450_5a	130096.77	6433	7110.3	481.17	462.39	664.03	521.8
le450_5b	187405.33	6732	5989.9	580.92	428.66	140.3	361.61
le450_5c	182102.37	5917	4934.8	410.08	479.84	634.97	400.52
le450_5d	182275.69	5402	4033.8	198.01	276.51	730.51	460.02
le450_15a	117042.59	6876	6191	159.01	237.44	940.83	232.66
le450_15b	197527.14	6423	6385.7	305.8	382.18	364.68	398.77
le450_15c	143451.73	4997	4368.9	45.4	450.63	50.68	392.54
le450_15d	117990.3	4864	3441.8	127.73	320.13	32.51	286.26
le450_25a	143963.41	6025	7377.9	982.12	671.7	824.89	571.85
le450_25b	184165.21	6045	6905.8	452.96	236.8	927.89	443.17
le450_25c	151719.58	6189	5345.9	25.25	343.46	531.27	571.19
le450_25d	189175.4	6712	4118.9	410.32	447.03	437.24	446.18
dsjc125.1	1532.93	501	334.95	521.34	83.67	137.75	293.33
dsjc125.5	2509.97	261	66.0	179.72	173.54	29.68	382.06
dsjc125.9	1623.44	110	0.1	0.03	0.13	0.05	0.09
dsjc250.1	28606.12	1878	4162.4	766.75	381.73	163.68	263.95
dsjc250.5	14743.35	648	753.5	6.36	81.3	2.73	26.58
dsjc250.9	30167.7	238	0.5	0.2	1.52	0.34	1.13

Table 15. Time performance of ACS and ACS+ILS for Tree Decomposition.

Instance	|V| / |H|	ACS+SC				
		min	max	avg	t-min	t-avg
adder_75	526 / 376	4	4	4	0.41	0.42
adder_99	694 / 496	4	4	4	0.72	0.73
b06	50 / 48	5	6	5.5	230.56	240.89
b08	179 / 170	11	11	11	0.27	11.6
b09	169 / 168	12	12	12	0.35	8.65
b10	200 / 189	17	17	17	0.72	27.05
bridge_50	452 / 452	5	5	5	0.3	0.31
c499	243 / 202	21	22	21.5	55.32	173.31
c880	443 / 383	22	23	22.8	718.52	201.65
clique_20	190 / 20	18	21	20.3	419.41	317.22
grid2d_20	200 / 200	16	17	16.7	23.6	141.37
grid3d_8	256 / 256	41	46	44	732.52	472.65
grid4d_4	128 / 128	28	30	28.8	225.95	327.31
grid5d_3	122 / 121	29	31	30.6	132.02	274.12
nasa	579 / 680	37	39	38.6	878.72	236.19
NewSystem1	142 / 84	4	4	4	0.25	6.33
NewSystem2	345 / 200	6	6	6	0.18	2.15
s444	205 / 202	6	6	6	6.77	51.98
s510	236 / 217	27	29	28.2	239.52	123.39

Table 16. ACS+SC for Generalized Hypertree Decompositions

Page 104 illustrates how often ACS+SC gave a lower, equal, and greater minimum width over 10 runs than each of the other methods for the generation of hypertree decompositions.

Instance	|V| / |H|	BE	GA-ghw	TS	FM	HM	ACS+SC
adder_75	526 / 376	2	3	5	2	2	4
adder_99	694 / 496	2	3	5	2	2	4
b06	50 / 48	5	4	5	5	5	5
b08	179 / 170	10	9	20	14	12	11
b09	169 / 168	10	7	20	13	12	12
b10	200 / 189	14	11	33	17	16	17
bridge_50	452 / 452	3	6	10	29	4	5
c499	243 / 202	13	11	27	18	17	21
c880	443 / 383	19	17	41	31	25	22
clique_20	190 / 20	10	11	11	20	10	18
grid2d_20	200 / 200	12	10	18	15	12	16
grid3d_8	256 / 256	25	21	44	25	20	41
grid4d_4	128 / 128	17	15	40	17	18	28
grid5d_3	122 / 121	18	16	49	18	19	29
nasa	579 / 680	21	19	98	56	32	37
NewSystem1	142 / 84	3	3	6	4	3	4
NewSystem2	345 / 200	4	4	6	9	4	6
s444	205 / 202	6	5	25	8	8	6
s510	236 / 217	23	17	41	23	27	27

Table 17. Comparison of ACS+SC with other decomposition methods.

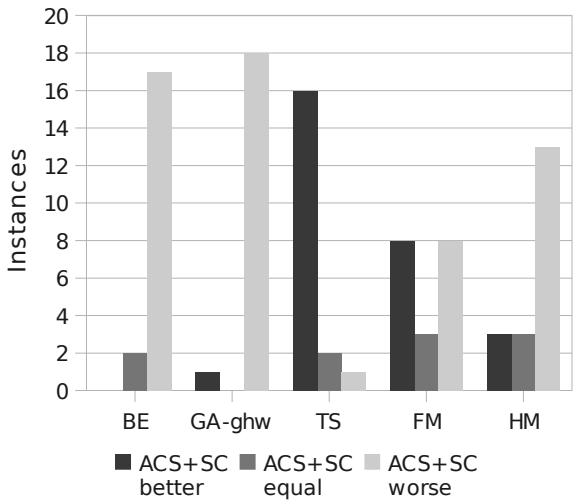


Figure 32. Comparison of ACS+SC with other decomposition methods.

8

CONCLUSIONS

In this thesis we have applied the ant colony optimization metaheuristic to the problem of finding tree and generalized hypertree decomposition of small width. For this purpose, we have adopted five different variants of ant algorithms from the literature [16] and adjusted them for the generation of tree and generalized hypertree decompositions.

Based on these algorithms we experimented with the guiding heuristics min-degree and min-fill and examined two different pheromone update strategies. Further, we proposed hybrid approaches that incorporate local search methods into the ant algorithms. One of these methods is an iterated local search similar to the algorithm proposed in [41] while the other one is a local search based on hill climbing.

Our experiments suggested that the ACO variants Max-Min Ant System and Ant Colony System give the best results for tree decompositions. We also found out that min-fill is superior to min-degree as a guiding heuristic. However, min-fill is much more computationally expensive for more complex problem instances. Moreover, we found that the iterated local search outperforms the hill climbing approach when combined with Ant Colony System. We also drew the conclusion that the usefulness of the pheromone trails is limited for our problem domain since the solutions constructed in the first couple of iterations are usually not improved significantly in later iterations. Hence, the ACO algorithms can be thought of probabilistic variations of the min-degree respectively min-fill construction heuristics.

In Chapter 6 we introduced a library for the ACO metaheuristic called *libaco*. We demonstrated how to use this library to solve arbitrary combinatorial optimization problems on the basis of an implementation for the travelling salesman problem.

In Chapter 7 we have applied Ant Colony System with and without the iterated local search to 62 benchmark graphs. The hybrid algorithm turned out to give better results than Ant Colony System on its own. It could improve the best known upper bound of the problem instance

homer.col from 31 to 29. For 28 instances the algorithm was able to return a width equal to the best known upper bound. Nevertheless, especially for more complex problem instances both algorithms gave worse results than other methods. Finally, we extended Ant Colony System with a greedy set cover heuristic and applied this algorithm to 19 benchmark hypergraphs. This algorithm outperformed one of the other methods it was compared to, but did not find any optimal solutions.

8.1 FUTURE WORK

Subject of future research is the investigation of self-adaptive parameter settings. The algorithm could make use of the stagnation measures in order to adjust parameters such as the evaporation rate ρ autonomously.

Another viable extension worth of further investigation is the application of ant colonies consisting of a number of ants proportional to the number of vertices in the constraint graph. That possibly could help to improve the quality of the pheromone updates and therefore could also improve the convergence behaviour of the algorithm.

For generalized hypertree decomposition the set cover heuristic could be applied as a guiding heuristic instead of min-fill respectively min-degree. This heuristic favors the elimination of vertices that cause cliques that can be covered by fewer hyperedges.

Certain aspects of the algorithms presented in this thesis could be analyzed in more detail via additional experiments. For instance, the ACO algorithms could be evaluated without the incorporation of a guiding heuristic. In this case only a local search would be used to guide the algorithm which probably would improve the efficiency of the algorithm since the optimization of the vertex elimination algorithm described by Golumbic [25] could also be applied during solution construction.

Another task for future research is the development and investigation of other pheromone update strategies that improve the convergence behaviour of the algorithms.

A

APPENDIX

A.1 INTERFACE OF THE ACOTREEWIDTH APPLICATION

In order to apply the application to the problem of finding tree decompositions of small width you have to specify at least the ACO variant to use and the path to the file that describes the constraint graph in DIMACS standard format:

```
$ ./acotreewidth --simple -f path/to/graph
```

If the application is intended to be used for the generation of gereralized hypertree decompositions, you have to specify the *hypergraph* flag and the input file must describe the input hypergraph in the format that is described at http://www.dbai.tuwien.ac.at/proj/hypertree/downloads.html:

```
$ ./acotreewidth --maxmin --hypergraph -f path/to/hypergraph
```

All non-mandatory parameters that are omitted will be set to a default value. You can get an overview of all command line options by executing *acotreewidth* with the *help* flag:

```
$ ./acotreewidth --help

USAGE:

   ./acotreewidth  {--simple|--elitist <double>|--rank <positive
                    integer>|--maxmin|--acs} [--iteratedls_ls_it
                    <positive integer>] [--iteratedls_it <positive
                    integer>] [--iteratedls] [--hillclimbing] [--ls]
                    [--it_best_ls] [--no_ls] [--acs_xi <double>]
                    [--acs_q0 <double>] [--maxmin_a <double>]
                    [--maxmin_frequency <double>] [-t <double>]
                    [--pheromone_update_es] [-l <double>]
                    [--stag_lambda] [--stag_variance] [-o]
                    [--hypergraph] -f <filepath> [-j <0|1>] [-g <0|1>]
                    [-p <double>] [-r <double>] [-b <double>] [-a
                    <double>] [-n <positive integer>] [-i <positive
                    integer>] [-m <integer>] [--] [--version] [-h]
```

```
Where:

   --simple
     (OR required)  use Simple Ant System
         -- OR --
   --elitist <double>
     (OR required)  use Elitist Ant System with given weight
         -- OR --
   --rank <positive integer>
     (OR required)  use Rank-Based Ant System and let the top n ants
     deposit pheromone
         -- OR --
   --maxmin
     (OR required)  use Max-Min Ant System
         -- OR --
   --acs
     (OR required)  use Ant Colony System


   --iteratedls_ls_it <positive integer>
     terminate the local search in the iterated local search after n
     iterations without improvement

   --iteratedls_it <positive integer>
     terminate iterated local search after n iterations without improvement

   --iteratedls
     use the iterated local search

   --hillclimbing
     use hill climbing

   --ls
     apply the local search to all solutions

   --it_best_ls
     apply the local search only to the iteration best solution

   --no_ls
     do not apply a local search

   --acs_xi <double>
     xi parameter for Ant Colony System

   --acs_q0 <double>
     q0 parameter for Ant Colony System

   --maxmin_a <double>
     parameter a in Max-Min Ant System
```

```
--maxmin_frequency <double>
  frequency of pheromone updates of best-so-far ant in Max-Min Ant
  System

-t <double>,  --time <double>
  terminate after n seconds (after last iteration is finished)

--pheromone_update_es
  use edge specific pheromone update

-l <double>,  --stag_limit <double>
  terminate if the stagnation measure falls below this value

--stag_lambda
  print lambda branching factor stagnation

--stag_variance
  print variation coefficient stagnation

-o,  --printord
  print best elimination ordering in iteration

--hypergraph
  input file is a hypergraph

-f <filepath>,  --file <filepath>
  (required)  path to the graph file

-j <0|1>,  --heuristic <0|1>
  0: min_degree 1: min_fill

-g <0|1>,  --graph <0|1>
  0: AdjacencyMatrix 1: AdjacencyList

-p <double>,  --pheromone <double>
  initial pheromone value

-r <double>,  --rho <double>
  pheromone trail evaporation rate

-b <double>,  --beta <double>
  beta (influence of heuristic information)

-a <double>,  --alpha <double>
  alpha (influence of pheromone trails)

-n <positive integer>,  --no_improve <positive integer>
  number of iterations without improvement as termination condition

-i <positive integer>,  --iterations <positive integer>
  number of iterations
```

```
-m <integer>,  --ants <integer>
  number of ants

--,  --ignore_rest
  Ignores the rest of the labeled arguments following this flag.

--version
  Displays version information and exits.

-h,  --help
  Displays usage information and exits.
```

BIBLIOGRAPHY

[1] Map of blank europe. URL http://commons.wikimedia.org/wiki/Image:Blank_map_of_Europe.svg. (Cited on page 42.)

[2] Graph coloring instances. URL http://mat.gsia.cmu.edu/COLOR/instances.html. (Cited on page 80.)

[3] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987. ISSN 0196-5212. (Cited on page 2.)

[4] Emgad H. Bachoore, Emgad H. Bachoore, Hans L. Bodlaender, and Hans L. Bodlaender. A branch and bound algorithm for exact, upper, and lower bounds on treewidth. In *Proceedings 2nd International Conference on Algorithmic Aspects in Information and Management, AAIM 2006, Lecture Notes in Computer Science*, pages 255–266, 2006. (Cited on page 29.)

[5] Hans L. Bodlaender, Arie M. C. A. Koster, and Thomas Wolle. Contraction and treewidth lower bounds. In *Proceedings 12th Annual European Symposium on Algorithms, ESA2004*, pages 628–639. Springer, 2004. (Cited on page 27.)

[6] Bernd Bullnheimer, Richard F. Hartl, and Christine Strauss. A new rank based version of the ant system — a computational study. Technical report, Central European Journal for Operations Research and Economics, 1997. (Cited on pages 10 and 47.)

[7] Gianni Di Caro and Marco Dorigo. Two ant colony algorithms for best-effort routing in datagram networks. In *In Proceedings of the Tenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'98*, pages 541–546. IASTED/ACTA Press, 1998. (Cited on pages 2 and 51.)

[8] F. Clautiaux, A. Moukrim, S. Negre, and J. Carlier. Heuristic and meta-heurisistic methods for computing graph treewidth. *RAIRO Operations Research*, (38):13–26, 2004. (Cited on pages 36 and 96.)

[9] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 2003. (Cited on pages 14, 22, and 24.)

[10] Matthijs den Besten, Thomas Stützle, and Marco Dorigo. Ant colony optimization for the total weighted tardiness problem. In *PPSN VI: Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pages 611–620, London, UK, 2000. Springer-Verlag. ISBN 3-540-41056-2. (Cited on page 51.)

[11] Deneubourg, S. Aron, S. Goss, and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3(2):159–168, March 1990. (Cited on page 39.)

[12] Artan Dermaku. Generalized hypertree decompositions based on hypergraph partitioning. Master's thesis, 2007. (Cited on page 35.)

[13] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *MICAI*, pages 1–11, 2008. (Cited on pages 35 and 100.)

[14] Marco Dorigo. *Optimization, Learning and Natural Algorithms [in Italian]*. PhD thesis, Dipartimento die Elettronica, Politecnico di Milano, Milan, 1992. (Cited on pages 10, 39, 44, and 46.)

[15] Marco Dorigo and Luca M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997. (Cited on pages 10, 48, and 49.)

[16] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Book, 2004. ISBN 0262042193. (Cited on pages 9, 10, 39, 43, 49, 52, 66, 67, 81, 82, and 105.)

[17] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE*

Transactions on Systems, Man, and Cybernetics-Part B, 26:29–41, 1996. (Cited on pages 10, 44, and 46.)

[18] Marco Dorigo, Mauro Birattari, Thomas Stützle, Université Libre, De Bruxelles, and Av F. D. Roosevelt. Ant colony optimization – artificial ants as a computational intelligence technique. *IEEE Comput. Intell. Mag*, 1:28–39, 2006. (Cited on page 2.)

[19] Thomas Eiter and Georg Gottlob. Hypergraph transversal computation and related problems in logic and ai. In *JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 549–564, London, UK, 2002. Springer-Verlag. ISBN 3-540-44190-5. (Cited on pages 16 and 17.)

[20] Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998. ISSN 0004-5411. (Cited on page 24.)

[21] Caroline Gagné, Marc Gravel, and Wilson L. Price. Solving real car sequencing problems with ant colony optimization. *European Journal of Operational Research*, 174(3):1427–1448, November 2006. (Cited on pages 2 and 50.)

[22] Tobias Ganzow, Georg Gottlob, Nysret Musliu, and Marko Samer. A CSP Hypergraph Library. Technical report, Database and Artificial Intelligence Group, June 2005. (Cited on pages 12, 34, and 80.)

[23] F. Glover. Tabu search — part i. *ORSA Journal on Computing*, 1(3): 190–206, 1989. (Cited on page 35.)

[24] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208, Arlington, Virginia, United States, 2004. AUAI Press. ISBN 0-9749039-0-6. (Cited on pages 18, 27, 29, 36, and 96.)

[25] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004. ISBN 0444515305. (Cited on pages 70 and 106.)

[26] Jens Gottlieb, Markus Puchta, and Christine Solnon. A study of greedy, local search and ant colony optimization approaches for car sequencing problems. In *In Applications of evolutionary computing, volume 2611 of LNCS*, pages 246–257. Springer, 2003. (Cited on page 50.)

[27] Georg Gottlob, Zoltan Miklos, and Thomas Schwentick. Generalized hypertree decompositions: np-hardness and tractable variants. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 13–22, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-685-1. (Cited on page 2.)

[28] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. In *Proceedings of the 1961 16th ACM national meeting*, pages 71.201–71.204, New York, NY, USA, 1961. ACM. (Cited on page 2.)

[29] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961. ISSN 0001-0782. (Cited on page 2.)

[30] G. Karypis and V. Kumar. hmetis: A hypergraph partitioning package version 1.5.3, 1998. (Cited on page 35.)

[31] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983. (Cited on page 36.)

[32] Uffe Kjaerulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2:7–17, 1992. (Cited on page 36.)

[33] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994. ISBN 3-540-58356-4. (Cited on pages 13, 16, and 18.)

[34] Thomas Korimort. *Heuristic Hypertree Decomposition*. PhD thesis, Vienna University of Technology, 2003. (Cited on page 35.)

[35] Arie M. C. A. Koster, Hans L. Bodlaender, and Stan P. M. Van Hoesel. Treewidth: Computational experiments. In *Electronic Notes*

in Discrete Mathematics, pages 54–57. Elsevier Science Publishers, 2001. (Cited on page 93.)

[36] Arie M. C. A. Koster, Frank Van, Den Eijkhof, Linda C. Van, Der Gaag, Hans L. Bodlaender, Hans L. Bodlaender, Arie M. C. A, Koster Frank Eijkhof, and Linda C. Van Der Gaag. Pre-processing for triangulation of probabilistic networks. In *Proceedings of the 17th Conference on Uncertainty in Arti Intelligence*, pages 32–39. Morgan Kaufmann, 2001. (Cited on page 29.)

[37] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. (Cited on page 28.)

[38] Pedro Larrañaga, Cindy M. H. Kuijpers, Mikel Poza, and Roberto H. Murga. Decomposing bayesian networks: triangulation of the moral graph with genetic algorithms. *Statistics and Computing*, 7(1):19–34, 1997. ISSN 0960-3174. (Cited on page 33.)

[39] Ben McMahan. Bucket elimination and hypertree decomposition. Technical report, Database and Artificial Intelligence Group, 2004. (Cited on page 22.)

[40] Nysret Musliu. Tabu search for generalized hypertree decompositions. In *The Seventh Metaheuristics International Conference (MIC)*, June 2007. (Cited on page 36.)

[41] Nysret Musliu. An iterative heuristic algorithm for tree decomposition. In *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, pages 133–150. 2008. (Cited on pages 11, 37, 64, 66, 96, and 105.)

[42] Nysret Musliu and Werner Schafhauser. Genetic algorithms for generalised hypertree decompositions. *European Journal of Industrial Engineering*, 1(3):317–340, January 2007. (Cited on pages 13, 33, 34, 96, and 100.)

[43] Bruno Richard Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. wiley, 1999. 660 pp. ISBN 0471-24134-2. (Cited on page 13.)

[44] Siddharthan Ramachandramurthi. The structure and number of obstructions to treewidth. *SIAM J. Discret. Math.*, 10(1):146–157, 1997. ISSN 0895-4801. (Cited on page 28.)

[45] A. Rizzoli, R. Montemanni, E. Lucibello, and L. Gambardella. Ant colony optimization for real-world vehicle routing problems. *Swarm Intelligence*, 1(2):135–151, December 2007. (Cited on page 51.)

[46] N. Robertson and P. D. Seymour. Graph minors II: algorithmic aspects of tree-width. *Journal Algorithms*, 7:309–322, 1986. (Cited on page 16.)

[47] Donald J. Rose and R. Endre Tarjan. Algorithmic aspects of vertex elimination. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 245–254, New York, NY, USA, 1975. ACM. (Cited on page 7.)

[48] Werner Schafhauser. New heuristic methods for tree decompositions and generalized hypertree decompositions. Master's thesis, 2006. (Cited on pages 7, 30, and 31.)

[49] K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 97)*, pages 185–190, 1997. (Cited on page 29.)

[50] Krzysztof Socha, Michael Sampels, and Max Manfrin. Ant algorithms for the university course timetabling problem with regard to the state-of-the-art. In *In Proc. Third European Workshop on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2003*, pages 334–345. Springer Verlag, 2003. (Cited on page 52.)

[51] Christine Solnon. Solving permutation constraint satisfaction problems with artificial ants. In *in Proceedings of ECAI'2000, IOS*, pages 118–122. Press, 2000. (Cited on page 50.)

[52] T. Stutzle and H. Hoos. Max-min ant system and local search for the traveling salesman problem. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 309–314, 1997. (Cited on pages 10 and 47.)

[53] Thomas Stützle and Holger H. Hoos. Max-min ant system. *Future Gener. Comput. Syst.*, 16(9):889–914, 2000. (Cited on pages 10 and 47.)

[54] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13 (3):566–579, 1984. ISSN 0097-5397. (Cited on page 27.)

[55] Edward Tsang. Foundations of constraint satisfaction, 1993. (Cited on page 14.)