MAGISTERARBEIT

An Ant Colony Optimisation Algorithm for the Bounded Diameter Minimum Spanning Tree Problem

ausgeführt am

Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter der Anleitung von

Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl

Univ.Ass. Dipl.-Ing. Martin Gruber

durch

Boris Kopinitsch, Bakk.techn.

Römerweg 31

A-2801 Katzelsdorf

Wien, Jänner 2006

ZUSAMMENFASSUNG

Im Rahmen dieser Magisterarbeit wurde ein Ant Colony Optimisation Algorithmus für das durchmesserbeschränkte minimale Spannbaum Problem erarbeitet. Bei diesem Problem handelt es sich um ein $\mathcal{NP}$-schweres kombinatorisches Optimierungsproblem mit zahlreichen praktischen Anwendungsgebieten, zum Beispiel im Netzwerkdesign. Der Algorithmus wurde mit einer lokalen Optimierungsheuristik, nämlich einem Variable Neighbourhood Descent, erweitert. Diese lokale Optimierungsheuristik arbeitet auf vier verschiedenen Nachbarschaftsstrukturen, bei deren Entwicklung besonders auf eine effiziente Evaluierung der Nachbarschaft einer Lösung Wert gelegt wurde. Vergleiche mit verschiedenen evolutionären Algorithmen und einer variablen Nachbarschaftssuche auf euklidischen Instanzen bis zu 1000 Knoten hinsichtlich Lösungsqualität als auch Berechnungszeit zeigen, dass der Ant Colony Optimisation Algorithmus bei ausreichend Zeit die bisher besten bekannten Ergebnisse zum Teil deutlich übertreffen kann, hingegen bei Testläufen mit starker Zeitbeschränkung nicht die Lösungsqualität der variablen Nachbarschaftssuche erreichen kann.

ABSTRACT

This master thesis presents an ant colony optimisation algorithm for the bounded diameter minimum spanning tree problem, a $\mathcal{NP}$-hard combinatorial optimisation problem with various application fields, e.g. when considering certain aspects of quality in communication network design. The algorithm is extended with local optimisation in terms of a variable neighbourhood descent algorithm based on four different neighbourhood structures. These neighbourhood structures have been designed in a way to enable a fast identification of the best neighbouring solution. The proposed algorithm is empirically compared to various evolutionary algorithms and a variable neighbourhood search implementation on Euclidean instances based on complete graphs with up to 1000 nodes considering either solution quality as well as computation time. It turns out that the ant colony optimisation algorithm performs best among these heuristics with respect to quality of solution, but cannot reach the results of the variable neighbourhood search implementation concerning computation time.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The bounded diameter minimum spanning tree (BDMST) problem is a combinatorial optimisation problem. Combinatorial optimisation problems belong to the group of optimisation problems, that in turn are divided into two groups. One, encoding solutions with real-valued variables and one, encoding solutions with discrete variables. Combinatorial optimisation problems belong to the latter one. The definition of a combinatorial optimisation problem given here, follows that by Blum and Roli in [6].

Definition 1: A combinatorial optimisation (CO) problem $P = (f, S)$ has a set of variables $X = \{x_1, \ldots, x_n\}$, variable domains $D_1, \ldots, D_n$, constraints among variables, and an objective function to be minimised or maximised, where $f : D_1 \times \ldots \times D_n \rightarrow \mathbb{R}^+$.
The set of all feasible assignments is $S = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in D_i,\ s \text{ satisfies all constraints}\}$. S is also called search or solution space.

To solve a combinatorial optimisation problem as defined above, a solution $T' \in S$ with either minimum objective value function, $f(T') \leq f(T)\ \forall\ T \in S$, or maximum objective value function, $f(T') \geq f(T)\ \forall\ T \in S$, has to be found. T' is called a global optimal solution. Representative combinatorial optimisation problems are the travelling salesman problem, the quadratic assignment problem, timetabling and scheduling problems.
After having introduced the definition of the combinatorial optimisation problem that of the bounded diameter minimum spanning tree problem can be given.

Definition 2: Given an undirected, connected graph $G = (V, E)$ of $n = |V|$ nodes and $m = |E|$ edges, where each edge e has associated costs $c_e \geq 0$, the bounded diameter minimum spanning tree problem is defined as the spanning tree $T = (V, E_T)$, with edge set $E_T \subseteq E$, of minimum weight $\sum_{e \in E_T} c_e$, where the diameter D is bounded above by a constant ≥ 2.

The *eccentricity* of a node v, with $v \in V$, is defined as the maximum number of edges on a path from v to any other node in the minimum spanning tree T. The diameter bound D is the maximum eccentricity a node is allowed to have.

From this definition follows that the centre of T is either, in case of an even diameter bound, the single node or, in case of an odd diameter bound, the pair of adjacent nodes of minimum eccentricity. Thus the bounded diameter minimum spanning tree problem can also be interpreted as the search for a minimum spanning tree rooted at an unknown centre (having depth 0) and whose maximal depth is restricted to $\lfloor D/2 \rfloor$.

The BDMST problem is known to be $\mathcal{NP}$-hard for $4 \leq D \leq n-1$ [15]. Within the context of this master thesis, simply BDMST problem instances based on complete graphs are considered, since incomplete graphs can be anytime transformed into complete graphs, by setting the edge costs for not really available edges extremely high, so that these edges do not surface in solutions.

The BDMST problem is not just of scientific interest, there are various real world applications, for example in communication network design. When considering certain aspects of quality of service, e.g. a maximum communication delay or minimum noise-ratio, BDMSTs become of great importance.

A further practical application can be found in [5]: When planning a Linear Light-wave Network (LLN) an undirected multi-graph $G = (V, E)$ is used, representing the network. This multi-graph G has to be decomposed into edge disjoint trees forming at least one spanning tree. Nevertheless, the aim of this decomposition process is to gain many spanning trees with a small diameter.

Another application field of the BDMST problem is data compression. Bookstein et al. have introduced a way of transferring the problem of compressing correlated bit-vectors into the problem of finding a minimum spanning tree [7]. The decompression time of a given bitmap vector is proportional to its depth within the tree. Thus the whole running time depends on the height of the built tree. So a MST with a bounded diameter is preferable.

The BDMST problem is also met in the field of mutual exclusion algorithms. When considering costs of the distributed mutual exclusion algorithm proposed in [32], the number of messages required to enter a critical section has a major effect on these costs. As the upper bound for these messages is $2D$, where D is the diameter of the underlying tree, it is quite obvious, that the network topology the algorithm is based on is of major importance.

Furthermore, the BDMST problem can also be found as a subproblem in other combinatorial optimisation problems such as vehicle routing [2]. In the vehicle routing problem a set of vehicles has to serve a set of costumers minimising a cost function like the path

covered by the vehicles or the number of vehicles to be used. In general some additional constraints have to be met, for example each costumer has to be visited exactly once by exactly one vehicle. Another one could be, that the demand each vehicle has to satisfy is not allowed to exceed the vehicle's capacity D. Or that each vehicle has to start and end its tour in the depot. A further constraint could be that the total tour length of each vehicle is not allowed to exceed a bound L. From these example constraints follows that the vehicle routing problem (VRP) is closely related to the travelling salesman problem (TSP), as a solution for the VRP consists of several TSP solutions with common start and end cities. Famous heuristics for solving TSP instances, e.g. the Christofides heuristic, use minimum spanning trees. Thus, in case an additional constraint requires that the lengths of the vehicle tours are limited to size L, for applying these heuristics to discover a tour for a vehicle they have to be extended in terms of using BDMSTs instead of MSTs.

This master thesis will present an ant colony optimisation algorithm for the bounded diameter minimum spanning tree problem. The ant colony optimisation is extended with a local search strategy, namely a variable neighbourhood descent, to improve overall solution quality. The following chapter will give an overview of already evolved exact as well as heuristic methods for solving a BDMST instance. In chapter 3 the basic concepts of two metaheuristics, namely ant colony optimisation and variable neighbourhood search, are introduced. Chapter 4 presents four different neighbourhood structures the variable neighbourhood descent algorithm is based on. In chapter 5 the ant colony optimisation algorithm for the BDMST problem, developed within the context of this master thesis, is described in full details. Chapter 6 gives an overview of the implementation of the ant colony optimisation algorithm. And finally in Chapter 7 the performance of the ant colony optimisation algorithm compared to that of the variable neighbourhood search implementation for the BDMST, proposed by Gruber and Raidl [21], and some other state-of-the-art metaheuristics is discussed.

2. PREVIOUS WORK

A couple of exact algorithms for the bounded diameter minimum spanning tree problem has been designed. The majority of them relies on network flow-based multi-commodity mixed integer linear programming (MIP) or Miller-Tucker-Zemlin (MTZ) formulations. By introducing multi-source multi-destination network flow models [27] Magnanti and Wong performed fundamental work of using MIPs in the field of network design. A multi-commodity formulation for the BDMST problem has been proposed by Achuthan and Caccetta [3]. Achuthan et al. suggested an improvement of this formulation in [4].

A couple of different advanced multi-commodity flow (MCF) formulations for the BDMST problem, counting and limiting the hops on paths from a virtual root node to any other node, has been introduced by Gouveia and Magnanti [18]. For the sake of achieving tight LP-relaxation bounds, they had to accept a quite large number of variables in their models. In [19] Gouveia et al. proposed an extended flow formulation for the BDMST problem in case of an odd diameter D.

Santos et al. [14] used modified and lifted Miller-Tucker-Zemlin subtour elimination inequalities, ensuring that the diameter bound is not violated. This MTZ-based model is claimed to work well on BDMST problem instances having a diameter bound D close to the diameter of the unconstrained minimum spanning tree.

A compact 0-1 integer linear programming (ILP) model embedded into a branch and cut environment has been published by Gruber and Raidl [22]. Computational results turned out that BDMST instances, having an underlying dense graph, with small to moderate diameter bounds, are solved significantly faster than by the MTZ-based model from Santos et al. [14]. However the looser the diameter bound the faster the MTZ-based model becomes. Therefore it cannot be said, that one approach dominates the other. When comparing the ILP with various MCF formulations from Gouveia and Magnanti [18], similar results were yielded.

Experiments turned out that all these exact approaches for solving a BDMST problem instance can only be applied to relatively small problem instances, not more than 100

nodes when considering fully connected graphs. As these exact algorithms are not able to yield the optimal solution for large problem instances in practicable time, several heuristics have been evolved to obtain solutions for large instances, too.

One of these heuristics is the so called one time tree construction (OTTC) algorithm by Abdalla et al. [1]. This greedy construction heuristic, based on the minimum spanning tree algorithm of Prim, starts by selecting a single node at random and then repeatedly extends the tree by adding the cheapest available edge, that connects a new node to the so far built tree. To guarantee that the diameter bound is not violated it must keep track of the eccentricities of all already appended nodes and update this information every time a new node is connected to the tree, a time consuming procedure. It turned out that the choice of the first node has crucial effects on the quality of the solution.

Julstrom [26] modified the OTTC approach by starting from a predetermined centre node. This idea simplifies the OTTC algorithm significantly, as the diameter constraint can be displaced by a height restriction of $\lfloor D/2 \rfloor$ for the spawned MST tree. This centre-based tree construction (CBTC) provides relatively good solutions on random instances. Nevertheless, on Euclidean instances the randomised centre-based tree construction (RTC) [31], that uses a permutation of all nodes and connects them with the cheapest possible edge to the tree in the order given by this permutation, whereas the centre is build by the first (two) node(s), yields much better results.

Clementi et al. [8] describe other construction heuristics, e.g. a modification of Kruskal's MST algorithm, for the related height-constrained MST problem.

As the solutions obtained by these construction heuristics were still not satisfactory, different evolutionary algorithms (EAs) [31, 24, 25] have been developed, to further improve the quality of these solutions. The initial population for these evolutionary algorithms is provided by one of the greedy construction heuristics. On instances up to 1000 nodes the evolutionary algorithms are in a position to significantly improve these initial solutions.

The best metaheuristic published so far for Euclidean instances, outperforming the EAs mentioned above in solution quality as well as running time, is a variable neighbourhood search by Gruber and Raidl [21]. Unfortunately it was not tested on random instances yet.

3. METAHEURISTICS

Since combinatorial optimisation (CO) problems are of major interest for the scientific as well as for the industrial world, several algorithms have been evolved to meet them. These algorithms can be classified into exact algorithms and heuristics. Exact algorithms guarantee to find an optimal solution for finite size CO problems in bounded time [30]. As long as not proofed that $\mathcal{P} = \mathcal{NP}$, no polynomial time algorithms exist for $\mathcal{NP}$-hard CO problems [15]. Due to their heigh computational time complexity exact algorithms are in general only applicable to instances of relatively small or moderate size. To attack even bigger instances heuristics have been evolved. Heuristics abandon the guarantee of finding an optimal solution for the sake of returning a valid solution in useable time. For some heuristics, referenced as approximate algorithms, it has been proven, that the yielded solution is not more (when the objective function has to be minimised) or less (when the objective function has to be maximised) than a factor ϕ times the optimal solution.

Among heuristics two basic types, constructive and local search methods, are differentiated. Constructive methods, as the name already presumes, create a solution for a CO problem from scratch. They append step by step to an initially empty solution components until a valid solution is reached. Local search methods start with an initial solution. They try to replace this initial solution T by a better solution from a predefined neighbourhood of T. Both definitions given here, either that of a neighbourhood of a solution T as well as that of a local optimal (minimum) solution with respect to a predefined neighbourhood structure $\mathcal{N}$, follow those by Blum and Roli in [6].

Definition 3: The neighbourhood structure is a function $\mathcal{N} : S \rightarrow 2^S$ [1] that assigns each $T \in S$ a set of neighbours $\mathcal{N}(T) \subseteq S$, where S is the search or solution space. $\mathcal{N}(T)$ is called neighbourhood of T.

Definition 4: A solution T' is called a local optimal (minimum) solution with respect to a neighbourhood structure $\mathcal{N}$, if $\forall\ T \in \mathcal{N}(T')$: $f(T') \leq f(T)$, where f is the objective

[1] 2^S is the power set of S

function to be minimised. A solution T' is called a strict local optimal (minimum) solution, if $f(T') < f(T) \; \forall \; T \in \mathcal{N}(T')$.

Local search algorithms are also called iterative improvement algorithms, as moves are only performed if they result in an improvement of the existing solution. To identify an improvement move usually one of the two main strategies, namely first improvement and best improvement, is used. First improvement means, that the local search algorithm stops searching the neighbourhood $\mathcal{N}(T)$ of an existing solution T as soon as an improvement move has been identified. By contrast the best improvement strategy enumerates the complete neighbourhood $\mathcal{N}(T)$ to identify the best improvement move. Local search algorithms terminate as soon as a local optimal (minimum) solution has been reached.

Beside these basic heuristics so-called metaheuristics have been developed. The aim of this concept is to build high level frameworks based on several heuristics, allowing an effective and efficient exploration of the whole solution space. Until now no universal valid definition of the term metaheuristic has prevailed, therefore a multitude of definitions from various scientists and organisations exists. In my opinion, the most appropriate one is the following from Voß et al.:

"A metaheuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method."[35]

The term metaheuristics, that was first introduced in 1986 by Glover [16], also reflects the aim of this concept, since it is composed of two Greek words, *heuriskein*, meaning "to find" and *meta*, meaning "beyond, in an upper level". Nowadays, several metaheuristics have been proposed. Ant colony optimisation, variable neighbourhood search, evolutionary algorithms, simulated annealing, iterated local search and tabu search, to name just a selection.

The following two sections present the basic ideas of the variable neighbourhood search and ant colony optimisation metaheuristic.

3.1 Variable Neighbourhood Search

The variable neighbourhood search (VNS) metaheuristic was first proposed by Hansen and Mladenović [28, 29]. The basic principle of VNS is to improve a given solution T by a systematic change of different neighbourhood structures. VNS is based on following three simple facts.

Fact 1: A local minimum T with respect to a single neighbourhood structure does not imply that T is a local minimum to any other neighbourhood structure.

Fact 2: The optimal solution has to be a local minimum with respect to all neighbourhood structures.

Fact 3: For many problems local minima with respect to one or more neighbourhood structures are relatively close to each other.

It can be gathered from these three facts that in general a local optimal solution with respect to at least one neighbourhood structure has something in common with the global optimal solution. It is only for sure that the local minimum provides some information, that can also be found in the global optimal solution. Unfortunately, this information cannot be extracted from the local minimum. So an organised search in the predefined neighbourhood structures of this local minimum is performed in order to improve it. There are three possible ways to guide this neighbourhood search: deterministic, stochastic or both combined.

The deterministic approach, called variable neighbourhood descent (VND), manages in a deterministic way the change of neighbourhoods. The VND concept exploits fact one, namely that a local optimal solution with respect to one neighbourhood structure is not necessarily one to another neighbourhood structure. Algorithm 1 gives an overview of the VND concept. Before executing Algorithm 1 the different neighbourhood structures $\mathcal{N}_1, \ldots, \mathcal{N}_k$ have to be defined. The idea of the VND concept is to compute iteratively a local optimum with respect to all different neighbourhood structures. Every time a new best solution is discovered, assume by exploiting neighbourhood structure $\mathcal{N}_i$ with $i \leq k$, the algorithm jumps back to the first neighbourhood structure as it cannot be guaranteed that this new best solution is also a local minimum with respect to all preceding neighbourhood structures $\mathcal{N}_j$ with $1 \leq j < i$. So each time the algorithm starts searching for better solution

Algorithm 1: Variable Neighbourhood Descent

```
1 create initial solution T;
2 i = 1;
3 while i ≤ k do
4     find best neighbouring solution T' ∈ N_i(T);
5     if T' better than best solution found so far then
6         save T' as new best solution T;
7         i = 1;
8     else i = i + 1;
```

by exploiting neighbourhood structure $\mathcal{N}_i$, it is ensured that the actual best solution is a local optimal solution with respect to all preceding neighbourhood structures $\mathcal{N}_j$ with $j < i$. At the end the algorithm terminates in a solution T that is a local minimum with respect to all neighbourhood structures defined for the VND.

The stochastic approach is referenced as reduced variable neighbourhood search (RVNS). Algorithm 2 presents the basic idea of the RVNS concept. After defining the different neighbourhood structures $\mathcal{N}_1, \ldots, \mathcal{N}_k$ the algorithm starts by computing an initial solution T. In the RVNS concept each neighbourhood structure is no longer explored completely, identifying the best move. Instead, in each iteration (lines 4 to 9), including all neighbourhood structures $\mathcal{N}_i$ with $1 \leq i \leq k$, a solution $T' \in \mathcal{N}_i(T)$ is arbitrarily chosen and compared if it is better than the so far best solution T. This process is called *shaking*. Again, as already shown in Algorithm 1, an improvement of the best solution T terminates the current iteration and starts a new one, beginning with neighbourhood $\mathcal{N}_1$.

Algorithm 2: Reduced Variable Neighbourhood Search

```
1 create initial solution T;
2 while termination condition not met do
3     i = 1;
4     while i ≤ k do
5         select completely random a solution T' ∈ N_i(T);
6         if T' better than best solution found so far then
7             save T' as new best solution T;
8             i = 1;
9         else i = i + 1;
```

However, this shaking makes it impossible to guarantee that at any time the best solution found so far is a local minimum with respect to all neighbourhood structures. Therefore a termination condition has to be introduced, e.g. a maximum number of iterations without

further improvement or a maximum CPU time. The RVNS method seems to be practicable if the enumeration of a whole neighbourhood is too cost-intensive, e.g. in case of a large neighbourhood with an exponential number of neighbouring solutions.

And finally, the approach including a deterministic as well as a stochastic component is known as basic variable neighbourhood search. In principle the basic VNS is an extension of the RVNS concept. Algorithm 3 presents the idea of the basic VNS. Assuming the different neighbourhood structures $\mathcal{N}_1, \ldots, \mathcal{N}_k$ have been defined, the algorithm starts by creating an initial solution, that is simultaneously the best solution T found so far. After shaking T, that is always performed within the context of the actual neighbourhood under consideration $\mathcal{N}_i(T)$ with $i \leq k$, the resulting solution $T' \in \mathcal{N}_i(T)$ is tried to be further improved by local search methods. This complete random selection of a solution $T' \in \mathcal{N}_i(T)$, where T represents the best solution found so far, avoids any possible cycling. A drawback of this behaviour is, as already mentioned for the RVNS concept, that it cannot be assured at any time that a solution T is locally optimal with respect to all neighbourhood structures. Therefore the use of a termination condition becomes essential. Again if the local search methods yield an improvement of the overall best solution T found so far, the current iteration is aborted and a new one is started, beginning by shaking T within $\mathcal{N}_1(T)$. This shaking yields a solution $T' \in \mathcal{N}_1(T)$ as input for the local search methods. As proposed in [23], these local search methods can be replaced by a complete VND. However, as a consequence of using a VNS/VND combination the shaking process has to be extended, since VND always yields a local optimal solution T with respect to all neighbourhood structures. Therefore, within a shaking process $m \geq 2$ moves based on a single neighbourhood have to be performed to facilitate this VNS/VND combination an escape from local optimal solutions with respect to all neighbourhood structures.

Algorithm 3: Basic Variable Neighbourhood Search

```
1  create initial solution T;
2  while termination condition not met do
3      i = 1;
4      while i ≤ k do
5          select completely random a solution T' ∈ N_i(T);
6          try to improve T' by using local search methods;
7          if T' better than best solution found so far then
8              save T' as new best solution T;
9              i = 1;
10         else i = i + 1;
```

3.2 Ant Colony Optimisation

The ant colony optimisation (ACO) metaheuristic belongs to the class of ant algorithms. Ant algorithms were first proposed by Dorigo et el. [10, 13] as a multi agent approach for various difficult combinatorial optimisation problems, e.g. the travelling salesman problem (TSP) or the quadratic assignment problem (QAP). Ant algorithms are based on the behaviour of real ants. Of main interest is their foraging behaviour and, in particular, their ability to find the shortest paths between their nest and food sources.

The key component of this ability is a chemical substance, called *pheromone*. While walking ants deposit pheromone on the ground, building a pheromone trail. Pheromone can be sensed by ants. Therefore these pheromone trails enable ants to find their way back as well as other ants to find food sources, discovered by nest-mates. Furthermore, when ants have to choose between several trails, they tend to pick, in probability, those with high pheromone concentration. It has been experimentally shown that in case of different trails to the food source, this pheromone trail following behaviour is responsible for finding the shortest path. One of these conducted experiments to study this behaviour in controlled conditions is the binary bride experiment by Deneubourg et al. [9]. In this experiment the nest was separated by two branches of same length from the food source. Initially the two branches were pheromone free. Due to the fact of random fluctuations a few more ants randomly selected one branch, in the experiment the upper one. As ants deposit pheromone, the greater number of ants on the upper branch laid more pheromone on it, that in turn stimulated more ants to choose the upper branch, and so on. Finally nearly all ants chose the upper branch. A modification of this experiment, where the branches are of different length, can be found in [17].

Remarkable is the fact that a single ant has only the capability of finding a path to the food source. Only the coaction of the whole ant colony enables finding the shortest path between the nest and the food source. Therefore the foraging behaviour of ant colonies can bee seen as a distributed optimisation process, using only indirect communication, accomplished through pheromone trails. This indirect communication between nest-mates is known as *stigmergy* [20].

The key design component of the ACO metaheuristic is to portion computational resources to a set of agents (artificial ants), that in turn provide through cooperative interaction good solutions. As artificial ants are a reproduction of real ants, we can distinguish between qualities adopted from real ants and those added to make them more efficient and effective:

- **Ant colony:** Ant algorithms consist of a finite size colony of artificial ants. These artificial ants, forming the colony, act as real ants independently and concurrently. Despite this two properties artificial ants show – as their counterparts in the real world – a cooperative behaviour.

- **Stigmercy:** As real ants, artificial ants deposit pheromone on the paths used for constructing a solution. This pheromone laying behaviour modifies the problem representation, that in turn is the basis for the indirect communication among artificial ants. This way of indirect communication is, as already mentioned, called stigmercy.

- **Local moves:** Real ants are not able to jump to any position they want to. So do artificial ants, since they only move through adjacent problem states. Therefore artificial ants accomplish as their natural counterparts only local moves.

- **State transition policy:** As real ants, artificial ants apply a stochastic local decision policy while moving through adjacent problem states. This decision policy is exclusively based on local information, consisting of the pheromone trails and sometimes some a priori problem-specific information. Furthermore, local means that this information can only be accessed from the state in which it was released. Sometimes this condition is relaxed so that local information can also be accessed from neighbouring states. So as real ants, artificial ants do not make use of lookahead concepts to predict future states.

- **Memory:** Memory is a characteristic of artificial ants, that is not found in their natural counterparts. Each artificial ant has some memory capacity to store past activities, that in turn can be used to prevent ants from entering invalid problem states. Another practical application of using memory capacity is to compute the quality of the generated solution.

- **Amount of pheromone:** In contrast to real ants, artificial ants are able to evaluate the fitness of the solution found. Depending on this value they are able to bias the amount of pheromone laid on the paths forming the solution. In general the amount of pheromone deposited by an ant is proportional to the fitness of the solution constructed by this ant.

- **Time of pheromone laying:** Artificial ants are not only capable of regulating the amount of pheromone they deposit, they are also able to influence the time of pheromone laying. Real ants deposit pheromone while walking. By contrast ant

algorithms adapt the time when pheromone is laid to the problem. For many problems the deposition of pheromone after generating the solution turned out to be more practical.

- **Additional capabilities:** The basic capabilities can be extended to improve overall performance. Lookahead, backtracking or local optimisation are examples of these extra capabilities, to name just a few.

On principle the activity of the ant colony optimisation metaheuristic can be described as follows. In each iteration of an ACO algorithm an ant colony, consisting of a finite size of artificial ants, equipped with the above listed characteristics, tries to find good solutions to a difficult combinatorial optimisation problem. A solution is a minimum cost path through the different problem states. It should be clear that this path has to satisfy all problem constraints. The basic ACO concept envisions two tasks for each artificial ant. The first one is constructing either whole solutions or parts of a solution. Important is that while accomplishing this task ants only move through adjacent problem states. Each time an artificial ant moves from one state to the next, it uses a stochastic local decision policy, based solely on local information, that is information that can be only accessed from the actual state. To avoid ants to enter invalid problem states this local decision policy can be extended with private information of the artificial ant. The second task each artificial ant has to achieve is depositing pheromone. This can be arranged in several ways. The one extreme is that while moving through adjacent problem states ants deposit pheromone on the paths used. The other is that after finishing task one, thus the whole solution or a part of it has been constructed, ants evaluate the fitness of the solution and deposit pheromone depending on its quality. Combinations and variations of these two extremes are also possible pheromone depositing strategies. This pheromone laying behaviour is the key component of the ACO metaheuristic, since it changes the representation of the problem, more precisely the way how the problem is experienced by other artificial ants, and therefore influences the further search process. After accomplishing these two tasks, creating or improving a solution and depositing pheromone, the artificial ant "dies", respectively is deleted from the colony.

Beside these two basic tasks of artificial ants, two further procedures can be used to enhance the performance of the ACO metaheuristic, *pheromone trail evaporation* and *daemon activities*. The process of decreasing intensity of pheromone trails, assuming no further pheromone is deposited on them, is called pheromone trail evaporation. Pheromone trail

evaporation becomes essential when prohibiting a too fast convergence. The strength of pheromone trail evaporation and pheromone depositing on the one hand, as well as the stochasticity of the local decision policy on the other hand are responsible that the ant colony optimisation metaheuristic is not trapped soon in a local optimum, but exploits already accumulated knowledge. To find the right balance between the pheromone update process and the stochasticity of the decision policy is essential, since only then an exploration of many different search areas of the solution space is possible. Daemon activities describe actions that cannot be accomplished by a single ant. Therefore they are executed by a global guard, called a daemon. A typical activity is biasing the search process by depositing some additional pheromone on paths forming an especially good solution.

Over the years many different ACO metaheuristics have evolved. More or less they all have their roots in the first ACO algorithm developed, the so called ant system (AS) proposed in 1991 by Dorigo [10, 13]. Three ant system algorithms have been introduced, that differ only in the pheromone update process. In *ant-density* (constant amount of pheromone) and *ant-quantity* (amount of pheromone inversely proportional to the costs of the chosen trail) algorithms ants deposit pheromone while constructing the solution. On the contrary to this two approaches, in *ant-cycle* algorithms ants lay pheromone after having finished building a solution. Computations on a set of benchmark problems [10, 13] showed, that the performance of ant-cycle algorithms was superior to those of ant-density and ant-quantity algorithms. So research concentrated on the characteristics of ant-cycle, which is nowadays known as ant system, whereas the two other algorithms were abandoned. Other remarkable characteristics of ant system algorithms are that every ant deposits pheromone and that the amount of pheromone is proportional to the fitness of the solution found. AS algorithms do not make use of daemon activities, since the pheromone trail evaporation process is delegated to every ant. So each artificial ant is not only responsible for depositing pheromone on the paths used, but also for decreasing pheromone on all paths, either those used as well as those not. The memory capability is used to prohibit ants from entering invalid problem states. In general, before starting an AS algorithm, each path, connecting two adjacent problem states, is initialised with a small positive amount of pheromone. Experimental results on small TSP instances [13] showed that the AS approach reached and sometimes even exceeded the performance of some general purpose heuristics compared to. Unfortunately, on larger problem instances it could not keep up with other heuristics. So several other ACO metaheuristics have evolved to improve the performance of the ant system.

One of this refined approaches was proposed by Stützle and Hoos [34, 33] in 1997. They presented a $\mathcal{MAX} - \mathcal{MIN}$ ant system (MMAS) for the travelling salesman problem, that is in the main an ant system with three slightly differences. First, the pheromone trails are updated by a daemon that deposits extra pheromone on the best discovered tour. Second, the amount of pheromone on each path connecting two adjacent problem states has to be within a given interval $[\tau_{Min}, \tau_{Max}]$. The pheromone amount τ is not allowed to fall under τ_{Min}, nor to exceed τ_{Max}. And finally, as a third difference, the pheromone amount on each path is initialised to the maximum τ_{Max}.

Another ACO metaheuristic developed to improve the poor performance of the basic AS is the ant colony system (ACS) introduced by Dorigo and Gambardell in 1996 [12, 11]. The main difference to the ant system is the daemon activity guarding the pheromone update. It picks within one iteration only the best solution constructed by an ant and deposits pheromone only on the paths forming this solution. So only a single artificial ant of the whole colony deposits its pheromone. Furthermore, the daemon is responsible for the pheromone trail evaporation process. Another difference can be found in the details of the local decision policy.

To conclude, the basic concepts of the ant colony optimisation heuristic are quite simple, inspired by nature. Nevertheless, for a concrete ACO implementation various problem specific decisions have to be made, for example the number of artificial ants the colony should consist of, the kind of pheromone update to be used, including depositing as well as evaporation, the usage of a local decision policy and a-priori problem-specific local information. Another problem dependent key component is the initialisation of the pheromone amount on the paths, connecting adjacent problem states. Experience shows that the ACO metaheuristic can only yield good results when applied to problems where each state in the search space does not have a large number of neighbouring solutions. Otherwise the probability that artificial ants will visit the same state is marginal due to the huge number of possible moves. As a consequence the key component of an ACO, making local decisions based on pheromone information, does not work anymore because of the small pheromone differences on the various paths.

4. NEIGHBOURHOOD STRUCTURES

This chapter will present the four different neighbourhood structures, namely Edge Exchange, Node Swap, Centre Exchange and Level Change. These neighbourhood structures have been developed together with Gruber and Raidl, who used already improved versions of some of them in their VNS for the BDMST [21]. Beside the definitions and data structures for these neighbourhood structures also pseudo-code to completely enumerate the different neighbourhoods is introduced, since the local search strategy used by the ant colony optimisation approach, presented in this master thesis, follows a best improvement strategy. As already explained in the previous chapter using a best improvement strategy requires to explore the whole neighbourhood of a given solution T to identify the most profitable move. An incremental enumeration of a neighbourhood, after having it completely explored once, depends on the neighbourhood structure. Precondition for an incremental enumeration is the possibility to store information during the exploration of a neighbourhood that allows a faster identification of the successive best move, as well as a locally restricted influence on this information of an executed move. Such an incremental enumeration of the neighbourhood is introduced for Node Swap and Level Change.

A solution T is interpreted as a directed outgoing arborescence, i.e a directed tree rooted at a centre to be determined. The following data structures are used:

- For all nodes $v \in V$ a list $succ(v)$, storing all immediate successors of node v.
- An array *pred*, containing the direct predecessor for each node. For the centre node(s) the value will be NULL.
- An array *lev*, storing for each node $v \in V$ the level v is assigned to. Valid levels, nodes can be assigned to, are within the range $[0, ..., \lfloor D/2 \rfloor]$, where a level of 0 denotes a centre node. Note, that the level a node v is assigned to is not always equal to the depth of v in the tree with respect to the centre.
- For each level $l = 0, ..., \lfloor D/2 \rfloor$ a set $V_l \subset V$, storing all nodes at level l.

For representing a valid solution, for example the successor lists or the predecessor array would be sufficient. The appreciation for using additional data structures is to speed up computation in various situations.

The four different neighbourhood structures can be separated into two groups according to the data structures the neighbourhood enumeration is based on. On the one hand there are Edge Exchange and Node Swap concentrating on the predecessor and successor relationships of the nodes. As these relationships are usually associated with trees, we denote this group as tree based. Whereas on the other hand for Centre Exchange and Level Change the level a node is assigned to is of major interest. We will refer to this group as level based.

Another important difference between the four neighbourhood structures is their ability to change the centre of a solution T. This is quite an important fact, since we never know (assuming we are dealing with instances that cannot be solved by exact algorithms in pleasing time) if the current centre is the best choice for a certain instance. As the name already presumes Centre Exchange concentrates its neighbourhood enumeration on finding a new centre, but also Node Swap includes the possibility of a centre change.

4.1 Tree Based Neighbourhoods

Edge Exchange and Node Swap operate on a tree structure, since they use basically the successor lists and the predecessor array. The *lev* array is utilised to store the depth of a node v with respect to the centre. From the level sets only V_0 is of interest, since it facilitates immediate access to the centre node(s).

4.1.1 Edge Exchange Neighbourhood

The Edge Exchange neighbourhood of a feasible solution T is defined as follows: Neighbours are all feasible solutions T' that differ only in a single directed edge. In other words, Edge Exchange simply cuts off a subtree S and reconnects this subtree S at any other feasible place of T, see Figure 4.1.

To fully and efficiently explore the neighbourhood of the current solution T we introduce some additional data structures:

- A static $n \times (n-1)$ array, named sorted neighbour list, storing for each node $v \in V$ all adjacent nodes in the order of ascending connection costs.

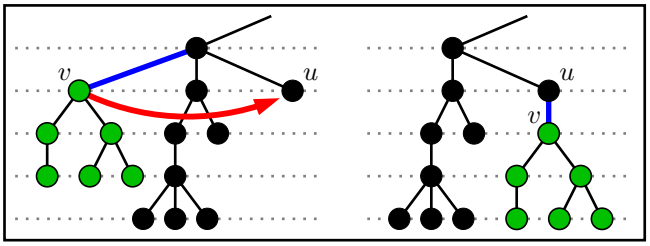

Fig. 4.1: Edge Exchange neighbourhood

- An array h storing for each node $v \in V$ the height of the subtree rooted at v. For leaf nodes this value will be 0.

The idea behind this sorted neighbour list is to consider only nodes as new predecessor for a disconnected subtree, that are cheaper than the current one. This static array can be computed in advance in $O(n^2)$. The height array is calculated in $O(n)$.

Since a best improvement strategy is used, the complete enumeration of the neighbourhood of a given solution T can be described as in Algorithm 4. For each node v, except those forming the centre, a cheaper predecessor is tried to be found using the sorted neighbour list of v. The actual move is saved as new best move if three conditions (line 7) are met. First, only the most profitable move is saved. Second, it has to be guaranteed that the diameter bound is not violated when moving the whole subtree S rooted at v. Third, it has to be assured that after reconnecting subtree S no circle has been introduced. So the third condition ensures that the subtree S is not appended to any node u within S.

Algorithm 4: Edge Exchange Neighbourhood Search

```
1 Δc* = 0;
2 for each node v ∈ V \ V_0 do
3     i = 0;
4     while c(N_i(v), v) < c(pred(v), v) do
5         u = N_i(v);
6         Δc = c(pred(v), v) − c(u, v);
7         if
          (Δc > Δc*) ∧ (lev(u)+1+h(v) ≤ ⌊D/2⌋) ∧ (lev(u) ≤ lev(v) ∨ v ∉ path(V_0, u))
          then
8             store this move as new best move m* = (u, v) and update Δc* = Δc;
9         i = i + 1;
```

The total size of the Edge Exchange neighbourhood is $O(n^2)$. For computing the new objective value when evaluating a move only cost differences have to be considered, this

can be done in constant time. Thus, the total time to explore the whole neighbourhood and to identify the best move is also $O(n^2)$.

An incremental enumeration of the neighbourhood after executing the most profitable move seemed not to be practicable, as the information collected while computing the best improvement move cannot be exploited efficiently. So for reaching a local optimal solution Algorithm 4 has to be called as long as it yields a profitable move. After the best improvement move has been identified it is performed and the data structures are updated accordingly. Therefore node v is deleted from the successor list of its old predecessor and appended to the successor list of its new one. Furthermore, the predecessor of node v is actualised. Moreover, the height value of the new and old predecessor has to be updated and – if necessary – the depth for all nodes, that are part of the moved subtree S rooted at v.

4.1.2 Node Swap Neighbourhood

The Node Swap neighbourhood of a feasible solution T is defined as follows: Neighbours are all feasible solutions T' where a node v and one of its immediate successors $u \in succ(v)$ have changed their position within the tree. In other words, node v that was predecessor of node u before the move is now an immediate successor of node u. Furthermore, node u inherits all successors of node v and the successor list of node v, $succ(v)$, is cleared. Reconnecting all immediate successors of v to u ensures that the diameter bound is not violated. Figure 4.2 illustrates this issue.

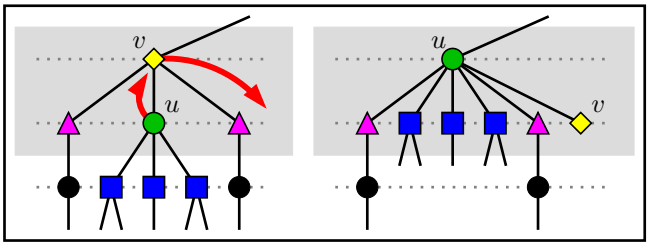

Fig. 4.2: Node Swap neighbourhood

For this neighbourhood it is possible to implement an incremental exploration after having identified the most profitable move. Therefore we introduce additional data structures:

- For every profitable move the following information is saved: The old root node v, the new root node u, the predecessor $pred(v)$ and successor list $succ(v)$ of the old root node, the cost difference Δc that will be saved if node u and v are swapped,

and finally a list l of all nodes that have to be checked once again when the move is performed, since only their situation alters if node u and v change their position. Members of this list l are pred(v), u, and all nodes $n \in \text{succ}(v) \backslash \{u\}$.

- A priority queue Q containing all profitable moves, including the information saved for each move. The priority queue Q has always the most lucrative move on top. After having explored the whole neighbourhood and identified the most profitable move, this priority queue plays the key role for computing the successive best move within the neighbourhood.

Using again a best improvement strategy the full exploration of the Node Swap neighbourhood of a solution T can be described as in Algorithm 5. Already the first line points out a major difference to the Edge Exchange neighbourhood. The Node Swap neighbourhood does not exclude the centre nodes, enabling neighbouring solutions T' having a different centre. After having selected a node v the algorithm computes in lines 3 to 8 the current costs c_v of the connections from v to its immediate successors, $succ(v)$, and, if $v \notin V_0$, also the current costs from v to its predecessor. Then each node $u \in succ(v)$ is tried to be swapped with node u. Lines 10 to 15 calculate the costs c_u of this potential new arrangement. This time the connections from u to each node in $succ(v) \setminus u$ are essential plus the connection from u to v. And again, if $v \notin V_0$, also the costs from u to v's current predecessor. In case the diameter is odd the costs from u to the second centre node have to be taken into account. In line 16 the costs saved by a node swap of v and u are computed. If this leads to an improvement greater than the best found so far then node u is marked as best node for a swap with node v. After having tested every node $u \in succ(v)$ line 20 checks if a profitable move has been identified. If so the move and the corresponding data is put into the priority queue Q.

The total size of the Node Swap neighbourhood is $O(n)$. The evaluation of a single neighbouring solution depends on the degree of the node under consideration. Therefore the time required to completely enumerate the whole neighbourhood is $O(n \cdot d_{max})$, where d_{max} denotes the maximum degree of any node in the current solution.

For reaching a local optimal solution with respect to the Node Swap neighbourhood an incremental enumeration can be applied after having explored the whole neighbourhood once. Since the worst case scenario has to be taken into account the incremental enumeration still requires time in $O(n \cdot d_{max})$, nevertheless, it speeds up computation in practice significantly.

Algorithm 5: Node Swap

```
1  for each node v ∈ V do
2      Δc* = 0; bestNode = NULL;
3      if v ∈ V_0 then
4          c_v = Σ_{m∈succ(v)} c(v, m);
5          if diameter D is odd then
6              c_v += c(v, other_centre_node);
7      else
8          c_v = c(pred(v), v) + Σ_{m∈succ(v)} c(v, m);
9      for each u ∈ succ(v) do
10         if v ∈ V_0 then
11             c_u = Σ_{m∈(succ(v)∪{v})\{u}} c(u, m);
12         else
13             c_u = c(pred(v), u) + Σ_{m∈(succ(v)∪{v})\{u}} c(u, m);
14         if diameter D is odd ∧ v ∈ V_0 then
15             c_u += c(u, other_centre_node);
16         Δc = c_v − c_u;
17         if (Δc > Δc*) then
18             Δc* = Δc;
19             bestNode = u;
20     if Δc* > 0 then
21         put move into priority queue Q;
```

Incremental enumeration of the neighbourhood of solution T

After executing the most lucrative move, only certain nodes have to be checked in order to have once again all possible improvement moves stored in the priority queue Q. The nodes affected when executing an improvement move are part of the data stored for it in Q, they can be found in list l. So for an incremental enumeration of the Node Swap neighbourhood only a slight modification of Algorithm 5 is required: Instead of executing it for all nodes, as stated in the first line, it is only executed for those affected when performing the best move received from Q.

The complete algorithm to reach a local optimal solution is now as follows: First the whole neighbourhood is explored as described in Algorithm 5 to identify the first best move. Afterwards, the priority queue Q contains all profitable moves, having the best one on top. Then, as long as the priority queue Q is not empty, the first element, containing the best improvement move, is received from Q. In the following it has to be checked if this move is still valid. Therefore the current local situation is compared to the situation when the

move was evaluated. This means that it is verified, if node v, that will be replaced by one of its successors u, still has the same predecessor and the same successors. Only in this case the move is considered to be still valid and will be executed. Otherwise it is ignored and the next move is fetched from the priority queue Q. Therefore the set of immediate successors of node v, succ(v), and its predecessor, pred(v), are saved along with the move.

Assuming a move turns out to be valid it is executed immediately: The successor list of pred(v) has to be updated, by deleting node v and adding node u. Furthermore, each node that was a direct successor of node v before performing the move has to be deleted from the successor list of node v and added, excluding node u, to the successor list of node u. In addition the predecessors of all rearranged nodes have to be set accordingly: Node v gets node u as its new predecessor, u gets the old predecessor of node v, pred(v), and all old successors of node v, succ(v), get node u as their new predecessor. The last step before fetching the next move from the priority queue Q is the update of all affected nodes as described above. A local optimal solution is reached when the priority queue Q is empty.

At this place it has to be mentioned that Gruber and Raidl demonstrated in [21] that the exploration of this Node Swap neighbourhood can be implemented more efficiently by directly manipulating the moves stored in the priority queue when updates are required. As a consequence Q always contains only valid moves and so there is no need for the corresponding validity test and list l.

4.2 Level Based Neighbourhoods

The Centre Exchange and Level Change neighbourhood structures are not based on the predecessor and successor relationships of the nodes but on the levels ($0 \leq lev(v) \leq \lfloor D/2 \rfloor, v \in V$) the nodes are assigned to. There are always exactly $1 + D \bmod 2$ nodes with a level of 0, building the centre. From the level information a local optimal tree can easily be derived: Each node $v \in V \setminus V_0$ is connected to the least cost predecessor p with $lev(p) < lev(v)$. If there are multiple potential predecessors for a node v with the same minimum edge costs, v always connects to one with minimum level.

4.2.1 Centre Exchange Neighbourhood

The Centre Exchange neighbourhood of a feasible solution T is defined as follows: Neighbours are all feasible solutions T' where exactly one centre node $c \in V_0$ is replaced by any other node $u \in V \setminus V_0$ and set to level $\lfloor D/2 \rfloor$, see Figure 4.3. This maximises the number

of potential predecessors for c. As a consequence, all immediate successors of c have to find a new predecessor.

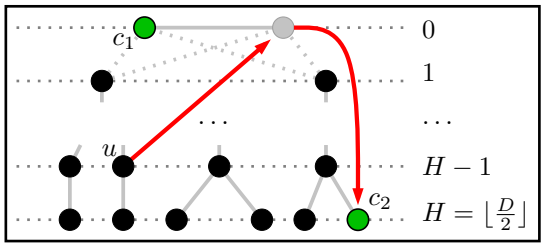

Fig. 4.3: Centre Exchange neighbourhood

Again, for a full and especially efficient exploration of the whole neighbourhood some additional data structures are advantageous:

- For each move the old and the new centre nodes are stored. Furthermore, a list l of new successor $\leftrightarrow$ predecessor relations ((u, v): node u will get node v as new predecessor).

Algorithm 6 describes the complete enumeration of the Centre Exchange neighbourhood. As the name suggests this neighbourhood focuses on finding a new centre. This is quite an important fact, since beside Centre Exchange there is only the Node Swap neighbourhood that is able to change the centre, but not on this scale. In line 3 the centre node c is assigned to level $\lfloor D/2 \rfloor$. Afterwards, a new predecessor has to be found for c. Candidates are all nodes v with $0 \leq lev(v) < \lfloor D/2 \rfloor$. Note that a potential predecessor for the moved centre node c at level 0 exists only in case of an odd diameter. In addition, for all successors $u \in succ(c)$ with $lev(u) > 1$ new predecessors have to be found (lines 5, 6). Everything happened so far is only computed once. In lines 8 to 19 each node, excluding those forming the (old) centre, is tried as new centre node. For each node $u \in V_1$ it has to be checked in case of an odd diameter if it is cheaper to connect u to the new centre node v or the remaining second centre node. The even diameter case is much simpler since all nodes $u \in V_1$ have to be appended to v, because v is their only available predecessor. This happens in lines 11 to 14. For all nodes at the levels 2 to the old level of v it has to be checked if now v is a cheaper predecessor than their current one. In line 16 the attempt installing node v as new centre node is evaluated. If it turns out that moving node v to the centre is the most profitable move found so far it is saved as new best move (lines 17-19).

The total size of this neighbourhood is $O(n)$. The worst-case time complexity to evaluate a single neighbouring solution is also $O(n)$ when only considering cost differences. Therefore

Algorithm 6: Centre Exchange Neighbourhood Search

```
1  for node c ∈ V_0 do
2      Δc* = 0;
3      assign c to level ⌊D/2⌋;
4      find least cost predecessor p for c with 0 ≤ lev(p) < ⌊D/2⌋; add (c, p) to l;
5      for each node v ∈ succ(c) with lev(u) > 1 do
6          find a new predecessor p for node v with lev(p) < lev(v); add (v, p) to l;
7      helperList = l;
8      for each node v ∈ V \ V_0 do
9          l = helperList;
10         for each node u ∈ V_1 do
11             if diameter D is odd ∧ c(u, second_centre_node) < c(u, v) then
12                 add (u, second_centre_node) to l;
13             else
14                 add (u, v) to l;
15         check for all nodes at the levels 2 to l_v if u would be a cheaper predecessor
           than their current one; if yes add the corresponding pair to l;
16         evaluate Δc of whole move;
17         if Δc > Δc* then
18             save move as new best move using c, v, l;
19             Δc* = Δc;
```

a complete exploration of the neighbourhood of a given solution T, including identifying the most lucrative move, can only be done in $O(n^2)$.

An incremental enumeration of the neighbourhood is not presented here, since an implementation is not straightforward and we assume that the solution this neighbourhood structure is applied to is already of some good quality. Therefore it is unlikely to have a longer chain of consecutive improvement moves. A local optimal solution can be reached by calling Algorithm 6 as long as it yields an improvement move. After having identified the most profitable move of the whole neighbourhood it is executed by deleting the old centre node c from the set V_0 and the new centre node v from the set $V_{lev(v)}$. Then they are appended to their new level sets, c to $V_{\lfloor D/2 \rfloor}$ and v to V_0. Furthermore, the level array lev is updated by setting $lev(c) = \lfloor D/2 \rfloor$ and $lev(v) = 0$. To complete the move the predecessor array and successor lists are updated based on the information stored in the list l. In case of an odd diameter the algorithm is executed consecutively on both centre nodes.

4.2.2 Level Change Neighbourhood

The Level Change neighbourhood of a feasible solution T is defined as follows: Neighbours are all feasible solutions T' where the level of exactly one node $v \in V \setminus V_0$ is either decremented or incremented. $1 \leq lev(v) \leq \lfloor D/2 \rfloor$ must hold after moving node v. Figure 4.4 visualises the Level Change neighbourhood.

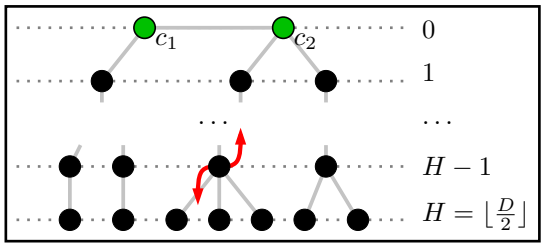

Fig. 4.4: Level Change neighbourhood

The Level Change neighbourhood is the second one for which an incremental enumeration, after having explored the whole neighbourhood once, is presented. As for the incremental enumeration of the Node Swap neighbourhood again additional data structures are required.

- For every profitable move the following information has to be stored: First the direction of the move, indicating if it is an increment or decrement one. Furthermore, the costs Δc saved by it, and finally a list l representing new successor $\leftrightarrow$ predecessor relations.
- A priority queue Q always having the most profitable move on top.
- An array *dec*, saving for each node a pointer to its decrement move within Q, and one array *inc*, storing a pointer to its increment move. These two arrays are necessary to have immediate access to all moves saved in the priority queue Q.

Algorithm 7 describes the exploration of the whole Level Change neighbourhood. As stated in line one this algorithm is executed for all nodes, excluding those forming the centre. The algorithm is split into two parts. In the first one (lines 2 to 11) the algorithm decrements the level of node v and saves this move if profitable. A decrement move is only possible if $lev(v) > 1$, since this neighbourhood does not effect the centre, but decrementing a node u with $l(u) = 1$ would put this node u into the centre. After decrementing the level of v for each node u at the old level of v it has to be checked if it is now cheaper to reconnect u to become successor of v (lines 4 to 6). If the level of v's predecessor is less than the new level

Algorithm 7: Level Change Neighbourhood Search

```
1  for node v ∈ V \ V_0 do
2      if lev(v) > 1 then
3          lev_new(v) = lev(v) − 1;
4          for each node u ∈ V_lev(v) do
5              if c(u, v) < c(u, pred(u)) then
6                  append (u, v) to list l;
7          if lev(pred(v)) == lev_new(v) then
8              find new predecessor p for node v with lev(p) < lev_new(v); add (v, p) to l;
9          evaluate Δc of whole move;
10         if Δc ≥ 0 then
11             put move into priority queue Q;
12     if lev(v) < ⌊D/2⌋ then
13         lev_new(v) = lev(v) + 1;
14         helper = pred(v);
15         for each node u ∈ V_lev(v) do
16             if c(u, v) < c(helper, v) then helper = u;
17         if helper ≠ pred(v) then
18             add (v, helper) to l;
19             for each node u ∈ succ(v) do
20                 find new predecessor p for node u with lev(p) < lev(u); add (u, p) to l;
21         evaluate Δc of whole move;
22         if Δc > 0 then
23             put move into priority queue Q;
```

of v, v's predecessor is kept. Otherwise, a new predecessor has to be found (lines 7, 8) in the appropriate levels. In line 9 the costs of this move are evaluated. If it turns out that decrementing v will improve the solution this move is put into the priority queue Q. Even if the move does not change the objective value instantly ($\Delta c = 0$) it is put into the priority queue since a node at a lower level can act as potential predecessor for a larger number of other nodes, this could be valuable for subsequent iterations. Such a move, where the level of a node is decremented but the tree derived from the level information does not change, is called a *null move* in the following.

In the second part (lines 12 to 23) the algorithm increments the level and saves this move if profitable. An increment of the level of node v is only possible if $lev(v) < \lfloor D/2 \rfloor$, since after incrementing a node u with $lev(u) = \lfloor D/2 \rfloor$ the diameter bound would be violated in general. So when incrementing node v a node at the old level of v has to be found where

v can be connected as successor in a cheaper way (lines 15, 16). If no such predecessor can be found an improvement is impossible. Assuming a cheaper predecessor was found, in lines 18 to 20 the algorithm assigns to all old successors u of v ($u \in succ(v)$, with $lev(u) = lev_{new}(v)$) a new predecessor with an appropriate level, since u and v are now on the same level and so cannot be connected any longer. In line 21 the increment move of v is evaluated and if it gains an improvement it is put into the priority queue Q.

As the Level Change neighbourhood is of size $O(n)$ and in the worst-case scenario for each decrement or increment move a node has to be compared with nearly every other node, a complete enumeration of it requires time in $O(n^2)$.

A local optimal solution has been reached when the priority queue Q is empty. In the following section an incremental enumeration of the Level Change neighbourhood is presented that can be applied after having explored a whole neighbourhood once. In this section it will also be described how to efficiently update information stored in Q when executing a move which has local impact on other nodes and their ability to improve the solution with a level increment or decrement. The incremental enumeration cannot reduce the worst-time complexity of $O(n^2)$, but still speeds up the computation time essential in practice.

Incremental enumeration of the neighbourhood of solution T

Assuming the whole neighbourhood has been explored applying Algorithm 7 and all improvement moves are saved in the priority queue Q, now the best move is fetched and performed. Therefore the node to be moved is deleted from its old and added to its new level: According to the direction of the move it is assigned either to the higher level, in case of an increment move, or to the lower level in case of a decrement move. Furthermore, the predecessor array and successor lists are updated based on the list l included in the data stored for a move in Q.

In order to have again all possible improvement moves in a valid state in the priority queue Q, the following rechecks and updates have to be done after executing a move. We have to distinguish between a decrement and increment move.

Case 1: The performed move decremented the level of node v from $l + 1$ to l.

- If there is an increment move for node v in the priority queue, delete it (it would be no longer valid).
- Reevaluate the following moves:

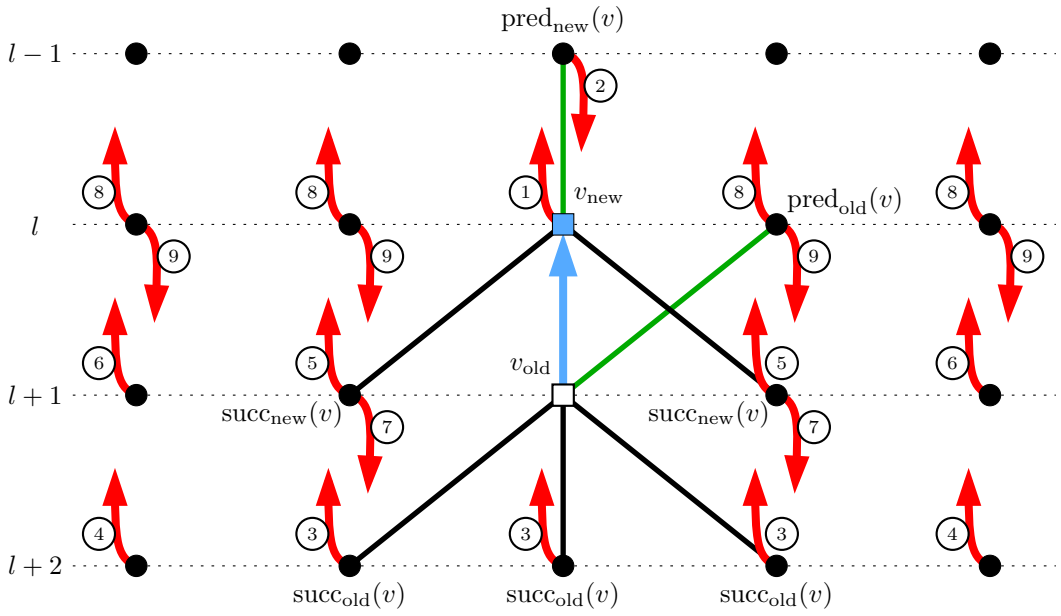

Fig. 4.5: Decrement move of node v (case 1)

1. Decrement the level of node v.

   Node v continues moving one level down, as this move was not possible before.

2. Increment the level of the new predecessor of v, $pred_{new}(v)$, if its level is $l-1$ and if this move is already in the priority queue Q. To verify if this move is already in the priority queue Q the *inc* array is used.

   The case that $pred_{new}(v)$ loses v as successor is new. So a new predecessor for v has to be found, this information must be added to the successor $\leftrightarrow$ predecessor list and the costs saved by this improvement Δc have to be updated accordingly.

   If the improvement Δc becomes negative or equal to zero after the update, the corresponding increment move can be removed from the priority queue Q (this holds true for every following increment move, too).

3. Decrement the level of each old successor u of v with $lev(u) = l+2$. Note that these nodes will remain successors of v.

   This move might not change the represented tree and therefore the objective value, but as already mentioned it should be considered as an improvement move since it may allow further nodes to decrement their level leading to other improvement moves.

   If such a move is already in the priority queue Q only the predecessor of u and the costs saved Δc have to be updated.

Otherwise this move has to be evaluated from scratch with the exception that there is no need to search for a predecessor for u, since it can keep v for sure.

4. Decrement the level of each node u at level $l+2$ having its predecessor $pred(u) \neq v$ at level $l+1$.

   Node u will lose its current predecessor and now v might become its new one.

   If such a move can be found in the priority queue Q only the predecessor of u stored in this move has to be checked against v. In case v would be the better choice, the predecessor of u and Δc have to be updated accordingly.

   Otherwise, the decrement move of u has to be evaluated completely new including the search for a new predecessor.

5. Decrement the level of each node u being a new successor of v (note that all these new successors are on level $l+1$) if such a decrement move is already in the priority queue Q.

   Before moving v to level l node u had another predecessor, assume w. So the calculated Δc of this move was based on the connection between u and w. Now u has v as its predecessor ($c(u,v)$ has to be less than $c(u,w)$). In addition, the move can contain the information that u would have become the new predecessor of v at its old level $l+1$, but after moving v to level l this is no longer possible. Furthermore, the move can include new successors of u now connected to v; they have to be checked if still u would be the better predecessor for them.

   So there are three possible impacts to Δc and the successor $\leftrightarrow$ predecessor list of the move that have to be updated accordingly.

6. Decrement the level of each node u at level $l+1$ not being a new successor of v if there is already a corresponding move in the priority queue Q.

   The situation is more or less the same as above with the restriction that the predecessor of u does not change when moving v to level l.

   So there are only two possible impacts to Δc and the successor $\leftrightarrow$ predecessor list. In this case the objective value gain can only decrease, making this move worse.

7. Increment the level of each node u being a new successor of v if such a move is already in the priority queue Q.

   The only possibility for an increment move to be profitable is that afterwards the node incremented can be connected much cheaper to a node at its old level.

There are two cases:

(a) In the move stored in Q, v would have become the new predecessor of u. After moving v to level l this is already the case so the move can be deleted without any further investigations.

(b) Otherwise u is now connected cheaper than it was before when the move was evaluated so Δc of the corresponding move has to be updated accordingly.

8. Decrement the level of each node u at level l, excluding v, if u would be a cheaper predecessor for v than its current predecessor, i.e. $c(v, u) < c(v, pred_{new}(v))$.

   If a decrement move for u is already found in Q only the new connection between u and v has to be added. Furthermore, Δc of the corresponding move has to be updated.

   Otherwise, the move has to be calculated from scratch.

9. Increment the level of each node u at level l excluding v.

   Now node v could be a new predecessor for u and for all old successors of u at level $l + 1$, as they lose their predecessor.

   If there is already an increment move for u in the priority queue two things have to be checked:

   (a) The move contains another new predecessor, assume w, for u. If v would be a cheaper predecessor than w, update the corresponding information and the improvement Δc.

   (b) Old successors of u at level $l + 1$ will lose their predecessor. For all of these nodes there will be a new predecessor stored in the move.

   - In the meantime (after the decrement move of v) it is possible that an old successor, assume w, of u is now successor of v. In this case w will not lose its predecessor, the corresponding information stored in the move record of u must be removed and Δc has to be updated.
   - For all nodes still successors of u (after performing the move of v) the new predecessor stored in the move has to be compared to v if v would be a better choice and if so, the corresponding information and Δc must be updated.
   - In case u is the old predecessor of v there will be a new predecessor for v stored in the move. This information can be removed and Δc has to be updated accordingly.

If no increment move for u can be found in the priority queue Q, the move has to be calculated completely new.

Case 2: The performed move incremented the level of node v from $l-1$ to l.

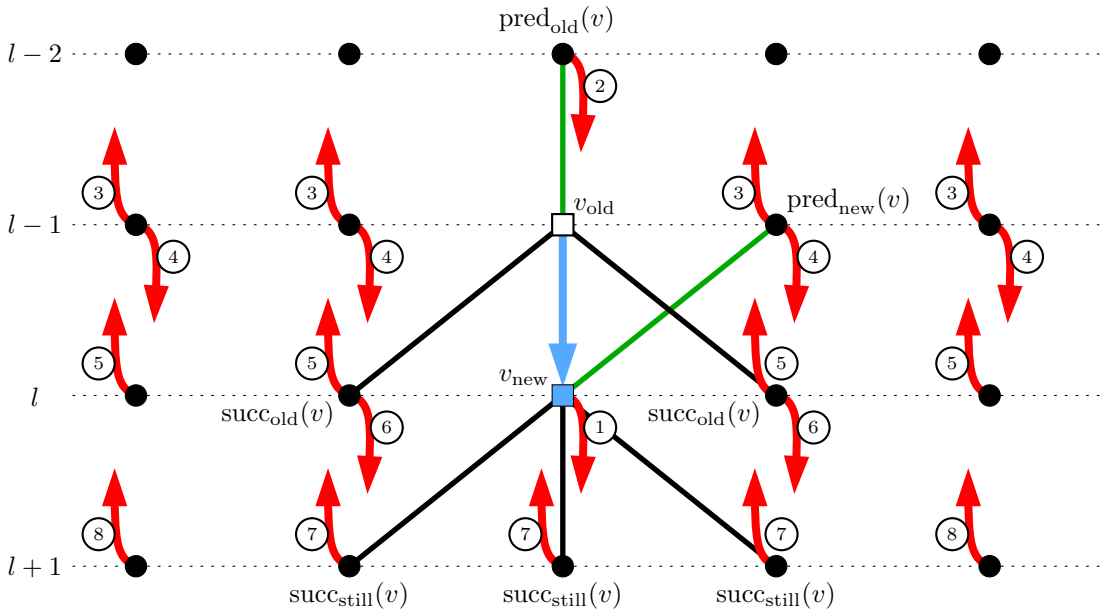

Fig. 4.6: Increment move of node v (case 2)

- If there is a decrement move of node v in the priority queue, delete it (it would be no longer valid).
- Reevaluate the following moves:

  1. Increment the level of v.

     Node v continues moving a level up, as this move was not possible before.

  2. Increment the level of the old predecessor of v, $pred_{old}(v)$, if its level is equal to $l-2$.

     Note that in this case v will not lose its predecessor. Therefore, if the move is already in the priority queue Q, only the objective value gain Δc will change and the new connection of v to another predecessor can be deleted.

     Otherwise, the move has to be evaluated from scratch.

  3. Decrement the level of each node u at level $l-1$ if such a move is already in the priority queue Q and u would be a cheaper predecessor for v than the old predecessor of v, i.e. $c(v,u) < c(v, pred_{old}(v))$.

If this move is not in the priority queue Q it still has not to be considered, as nothing changed to make this move profitable. Nevertheless, if the move is already in Q the costs saved by this move Δc include the benefit of connecting v to u instead of connecting it to $pred_{old}(v)$. Therefore, Δc and the fact that v is already connected to u have to be updated.

4. Increment the level of each node u at level $l-1$ if such a move is already in the priority queue Q.

   The move of v to level l has a lot of side effects on already existing increment moves of other nodes from level $l-1$ to l. For example v is no longer available as predecessor for u and all old successors of u at level l. In addition, v and some old successors of v at level l could now have u as their new predecessor and so would require to find a new node they can connect to. Due to these various special cases an increment move of u already found in Q is evaluated from scratch.

5. Decrement the level of each node u at level l excluding v.

   In case such a move is found in the priority queue Q two things have to be checked.

   (a) Node v could become a new successor of u if u is a cheaper predecessor for v than its new one, i.e. $c(u,v) < c(pred_{new}(v), v)$.

   (b) Old successors of v at level l have lost their predecessor. Node u could now be attractive as new predecessor, if this connection is not already saved in the decrement move of u.

   In both cases the objective value gain Δc and the successor $\leftrightarrow$ predecessor list of the move object have to be updated accordingly.

   If such a move is not found in the priority queue Q it has to be evaluated from scratch. For efficiency the computation can be split into two steps. The only things changed are the two points described above, u is a possible new predecessor for v and for old successors of v. If the evaluation of these changes does not lead to a positive Δc the move cannot be valuable. Therefore, the computation can already be aborted at this point.

6. Increment the level of each old successor u of v at level l.

   After executing the move of node v from level $l-1$ to l node u has got a new predecessor and is now connected with higher costs. With an increment move u

can get back its old and cheaper predecessor v.

If such a move is already in the priority queue Q then only Δc has to be updated accordingly, since before v's increment move u was successor of v and u has now a different predecessor. The increment move of u stored in the priority queue Q means that there is another, cheaper predecessor for u at level l so it is not necessary to check the new predecessor of u stored in the move record against v.

If no such move can be found in the priority queue Q the increment move of u has to be evaluated completely new.

7. Decrement the level of each successor u of v at level $l + 1$ if such a move is already in the priority queue.

   If no such move is found in Q it can be ignored since nothing happened that would now make a decrement of node u profitable.

   In case there is such a move, u will lose v as predecessor so a new predecessor has to be found leading to a different objective value gain Δc. The rest of the move can be taken without any change.

8. Decrement the level of each node u at level $l + 1$ with predecessors at level l, excluding successors of v, if such a move is already stored in the priority queue Q and u would have become successor of v.

   Node u will lose its best predecessor v after performing its decrement move. Again, the move itself can keep unchanged, only a new predecessor has to be found and the improvement Δc must be updated accordingly.

5. ANT COLONY OPTIMISATION FOR THE BDMST PROBLEM

In this chapter the concrete design of the ant colony optimisation algorithm for the bounded diameter minimum spanning tree problem, developed within the context of this master thesis, is presented. As already mentioned, ACO algorithms can be extended with extra capabilities to improve overall system efficiency, e.g. lookahead, local optimisation or backtracking strategies, to name just a few. The ACO algorithm introduced here is equipped with a local optimisation heuristic, namely a variable neighbourhood descent. So before going into the details of the ACO algorithm for the BDMST problem itself, first a VND for the BDMST is introduced, as this VND plays an important role within the ant colony optimisation algorithm.

5.1 Variable Neighbourhood Descent

In Chapter 3.1 various strategies to guide a search within a certain neighbourhood have been described. For the ACO the deterministic variable neighbourhood descent (VND) framework is used. The different neighbourhood structures for this VND are those four presented in the preceding chapter, namely Edge Exchange, Node Swap, Centre Exchange and Level Change.

The VND algorithm introduced here varies a little from the general VND framework as proposed by Hansen and Mladenović [23]. When following this general VND framework, the algorithm stops exploring a certain neighbourhood of a solution T, $\mathcal{N}_i(T)$ with $i > 1$, after identifying the best neighbouring solution of T with respect to this neighbourhood structure, $T' \in \mathcal{N}_i(T)$. It sets T to T' and switches back immediately to the first neighbourhood structure $\mathcal{N}_1$. However, as a best improvement strategy is used to identify an improvement move, the whole neighbourhood has to be enumerated. Depending on the definition of the neighbourhood structure it is sometimes possible to store information while exploring the whole neighbourhood (as shown in the previous chapter). This information allows a faster computation of the successive best move when a move has only local impact on it. Such an incremental enumeration has been introduced for the Node Swap and Level

Algorithm 8: VND(T)

```
1 k = 1;
2 while k ≤ 4 do
3     improve T until local optimum reached with respect to neighbourhood ...
4     switch k do
5         case 1 : Edge Exchange;
6         case 2 : Node Swap;
7         case 3 : Centre Exchange;
8         case 4 : Level Change;
9     if T improved and k ≠ 1 then k = 1; else k = k + 1;
```

Change neighbourhood structures. To benefit from this incremental exploration the VND algorithm presented here does not switch back to the first neighbourhood after having identified the best improvement move. Instead it computes a local optimal solution with respect to the current neighbourhood structure under consideration and only jumps back to the first neighbourhood $\mathcal{N}_1$, in case the solution could be improved.

After defining the different neighbourhood structures the VND is based on, the order these neighbourhoods should by applied to a solution has to be determined. This decision has crucial impact either on the computation time as well as on the quality of solutions obtained. The neighbourhood order for this VND algorithm has been taken from the VNS for the BDMST proposed by Gruber and Raidl [21], as the neighbourhood structures used in their VNS algorithm are basically the same as those utilised within this VND framework.

Algorithm 8 gives an overview of the VND framework used within the context of this master thesis. Starting from a solution T, first the Edge Exchange neighbourhood tries to improve T by moving around whole subtrees. When a local minimum with respect to the Edge Exchange neighbourhood is reached, T is tried to be further improved by optimising the arrangement of every node and its immediate successors (Node Swap). This is again done – as already mentioned and in contrast to the general VND – until a local optimal solution T' is found. Afterwards, in case T' is better than T, T' is saved as new so far best solution T within this VND and the algorithm jumps back to the first neighbourhood structure. If T was already a local minimum due to these two neighbourhoods the Centre Exchange neighbourhood tempts to further improve it by changing the centre node(s). Again, if the Centre Exchange neighbourhood yields an improvement move, the algorithm does not switch back immediately to the Edge Exchange neighbourhood. Instead, it computes a local minimum T' due to the Centre Exchange neighbourhood structure and jumps back

afterwards. And finally, if T is a local minimum with respect to the Edge Exchange, Node Swap and Centre Exchange neighbourhood, Level Change tries to optimally arrange all non-centre nodes within the level structure. When this VND algorithm terminates it reached a local optimal solution with respect to all four different neighbourhood structures.

5.2 Ant Colony Optimisation

As already presented in Chapter 3.2, several different ant algorithms have been developed. The ACO for the BDMST problem, proposed within the context of this master thesis, is based on the ant colony system (ACS), first proposed by Dorigo et al. in 1996 [12, 11]. A property taken from the ACS for the travelling salesman problem (TSP) is the pheromone initialisation process. Another adapted characteristic is that every artificial ant is only responsible for constructing a solution, since the pheromone update process is managed by a daemon. This daemon identifies, as it is representative for ACS algorithms, the best solution found by the whole colony in one iteration and updates the pheromone information proportional to the fitness of this best solution found. Pheromone trail evaporation is also performed by the daemon. A main difference between the ACS concept and the ant colony optimisation metaheuristic presented here lies in the local decision policy. In general the decision policy of ACS algorithms includes a heuristic component when computing a state transition. This heuristic component for the local decision policy is completely omitted for this ACO algorithm.

For the implementation of the ACO algorithm some data structures are required, most of them already known from the four different neighbourhoods:

- Again the successor lists *succ*, storing for each node its immediate successors, the predecessor array *pred*, saving for each node its predecessor in the directed tree from the centre to the leaves, the level information *lev*, stating the level a node is assigned to and the level sets V_l with $l = 0, \ldots, \lfloor D/2 \rfloor$, containing all nodes at the level l, are used.

- In addition, a two dimensional $n \times (\lfloor D/2 \rfloor + 1)$ pheromone matrix is needed to keep one pheromone entry for each combination of a node and a level. The value $\tau_{i,j}$ reflects the desirability of assigning node i to level j.

The level information *lev* and level sets $V_l \subset V$ are of major interest, as the ant colony optimisation algorithm developed in the context of this master thesis uses a different approach

regarding pheromone deposition in comparison to the basic ACO concept for the TSP proposed by Dorigo [10, 13]. There they operate on a graph $G = (V, E)$ where pheromone is deposited – either by ants themselves or by a daemon – on edges $e \in E$ contained in the solution. The ACO algorithm presented here does not deposit pheromone on edges but exploits the fact that once given the level information $lev(v)$ for all nodes $v \in V$ an optimal solution for the bounded diameter spanning tree problem with respect to this level information lev can be easily derived. Therefore simply for each non centre node v the least cost predecessor p with $lev(p) < lev(v)$ has to be found. In case of several possible least cost predecessors, always one with minimum level is selected. Thus ants construct a solution not by "running" through the graph structure, but by assigning nodes to levels, building the level information lev. So the pheromone information reflects instead of the desirability of using an edge the desirability of assigning a node to a certain level. Algorithm 9 gives an overview of the ant colony optimisation algorithm developed in the context of this master thesis.

To be able to construct a solution based on the information stored in the pheromone matrix, it has to be first initialised. This initial amount of pheromone for each combination of node and level has to be in relation to those values the pheromone update process is going to write into the matrix. E.g., if the initial values are too big compared to those, the pheromone update process is going to deposit, it will last very long until the ant colony will be capable of exploiting the information stored in the pheromone matrix. To guarantee that ants are able to use the collected information very soon the pheromone initialisation process follows the scheme of ACS algorithm for the TSP [12, 11], where the initial amount of pheromone deposited is proportional to the product of an initial solution value and the total number of nodes of the instance graph. Therefore in the first line of Algorithm 9 a solution S using the randomised tree construction (RTC) heuristic [31] is computed. The solution value obtained from the RTC heuristic, together with the total number of nodes n is used to initialise the pheromone matrix with $\tau_{i,j} = 1/(S * n)$, $i = 0, \ldots, n-1$ and $j = 0, \ldots, \lfloor D/2 \rfloor$.

Until no termination condition is met (line 3), which can be either a time limit, a maximum number of iterations without further improvement or both combined, in each iteration an colony of a finite number of artificial ants tries to find good solutions based on the so far collected information on the problem instance.

Each artificial ant builds a solution based on the information stored in the pheromone matrix. As already mentioned, the solution construction process exploits the fact that a given level information lev for each node is enough to construct the local optimal tree.

Algorithm 9: ACO for the BDMST

```
1  create an initial solution S using the RTC heuristic;
2  initialise the pheromone matrix with 1/(S * n);
3  while termination condition not met do
4      for each ant of the colony do
5          create a solution T, based on the information saved in the pheromone matrix;
6          VND(T);
7      determine best solution T' found by the whole colony;
8      update pheromone matrix using T';
9      if T' better than bestSolution then
10         bestSolution = T';
```

So each ant simply sets each node to a certain level. The level a node is assigned to is selected randomly with a probability proportional to the pheromone information stored for this node, i.e. $P_{i,l}$ is the probability that node i is assigned to level l.

$$P_{i,l} = \frac{\tau_{i,l}}{\sum_{l'=1}^{\lfloor \frac{D}{2} \rfloor} \tau_{i,l'}}$$

In general – due to the stochastic level selection process – each node can be assigned to any level as long as no entry in the pheromone matrix is zero. This is a desirable behaviour except for level 0, where the number of nodes is fixed in advance based on the diameter bound. Therefore the centre node(s) is (are) selected separately based on the information stored in the first column of the pheromone matrix (the pheromone values of level 0 for each node). As said above this local decision policy has no heuristic component, every node is assigned to a certain level independently from all other nodes, only the decoding step – deriving a tree from the level information – makes use of a problem specific heuristic when using for each non centre node its cheapest available predecessor.

To improve solutions built by artificial ants the variable neighbourhood descend for the BDMST, presented in the previous section, is applied to these solutions. Thus each ant constructs a solution based on the information stored in the pheromone matrix. Afterwards the VND algorithm tries to improve this solution (lines 5 to 6).

The daemon is responsible for collecting all solutions found by the ant colony in the current iteration (line 7). From this pool of solutions it selects the best one T' and determines the amount of pheromone, that is proportional to the fitness of this best solution, going to be written into the pheromone matrix for the appropriate node/level combinations.

This behaviour, that only the best ant deposits pheromone, was first proposed for ACS algorithms. The quantity of pheromone deposited is determined as follows:

$$\Delta\tau_{i,j} = \begin{cases} 1/T' & \text{if } lev(i) == j \text{ in } T' \\ 0 & \text{otherwise} \end{cases}$$

After having calculated the amount of pheromone to be deposited, the pheromone matrix is updated (line 8). This update process combines the pheromone laying and the pheromone evaporation process and follows the scheme introduced for ACS algorithms:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \rho\Delta\tau_{i,j}$$

The parameter ρ, with $0 \leq \rho \leq 1$ has a crucial impact on the convergence of the ant colony. The greater ρ the faster the ant colony commits itself to a certain solution, it converges faster, perhaps too fast. On the contrary, the smaller ρ the later the ant colony is able to exploit already collected knowledge of the problem instance, in terms of the pheromone matrix, to construct good solutions, but therefore the solution space is explored in more detail. So using this parameter the behaviour of this ant colony optimisation algorithm can be influenced substantially and adjusted for different applications.

In line 9 and 10 the daemon verifies if the best solution found by the ant colony in the current iteration is better than the best solution found so far. If so this solution is saved as new best solution.

6. IMPLEMENTATION

The ant colony optimisation metaheuristic, introduced in this master thesis, is implemented in C++, following the C++ coding standard, e.g. representing each class through a header file, stating the attribute and method declarations, and an implementation file, stating the program code. The only non-standard class library used is LEDA (Library of Efficient Data Types and Algorithms) in terms of version 5.0.1. The aim of the LEDA project, launched in 1988, was to build a small, but extendable library of data structures and algorithms for combinatorial computing in a form making them easy to use. Since February 1st, 2001, Algorithmic Solutions Software GmbH is responsible for maintaining and developing LEDA[1].

Figure 6.1 shows the UML class diagram of the major classes, their relationships as well as the most important attributes and methods of the respective classes.

The `Instance` class encapsulates all relevant information concerning a BDMST problem instance that does not change during computations, e.g. the underlying graph or the diameter bound. Due to this fact, each run of the ant colony optimisation algorithm has exactly one `Instance` object.

The `InstanceParser` is responsible for parsing the input file and storing the relevant information in the corresponding data structures. The `InstanceParser` is able to parse various file formats, e.g. GNUplot or EA graph instances. GNUplot files were created at the Institute of Computer Graphics and Algorithms for testing and visualising purposes. EA files are instances from the Beasley's OR-Library[2] which haven been originally intended for Euclidean Steiner tree problems. As in [31, 25, 21] these instances have been used for comparing this ACO with other state-of-the-art metaheuristics.

The `Solution` class is used for representing solutions for a certain problem instance. To express a valid solution the attribute `successorVector`, storing for each node its immediate successors, would be sufficient, but to enable faster computations additional attributes

[1] `http://www.algorithmic-solutions.com/enleda.htm`

[2] `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/esteininfo.html`

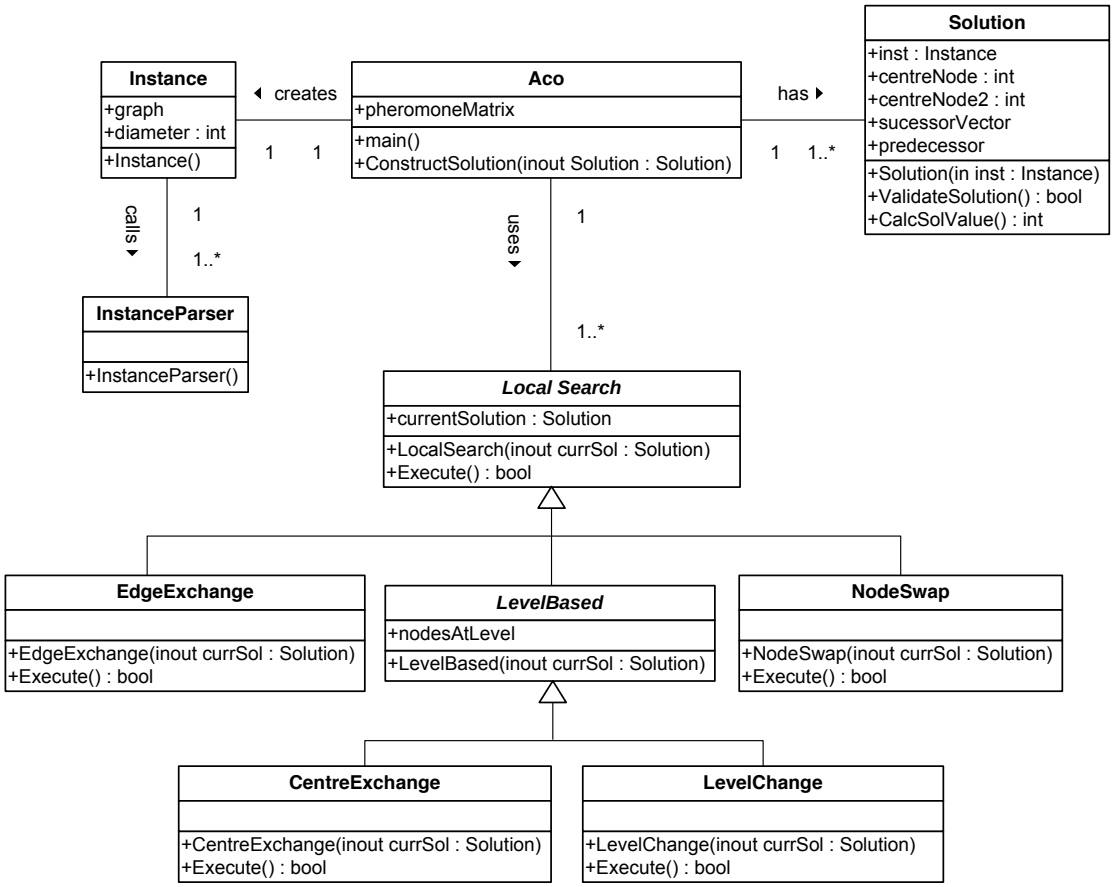

Fig. 6.1: Class diagram

are used. The `predecessor` attribute stores for each node its direct predecessor and `centreNode` keeps the centre node. In case of an odd diameter `centreNode2` saves the second one. To express the correlation between a `Solution` object and the instance it belongs to, a pointer to the `Instance` object is passed to in the constructor of the `Solution` class. This pointer enables the `Solution` object to access instance specific information, like number of nodes, diameter bound or connection costs between two nodes. Important methods of the `Solution` class are `ValidateSolution()` and `CalcSolValue()`. The `ValidateSolution()` method verifies if the solution represented by this `Solution` object contains all nodes of the problem instance, if the diameter bound is not violated and if the solution is free of circles. If it turns out that the `Solution` object represents a valid BDMST `true` is returned, otherwise `false`. The `CalcSolValue()` method calculates the total costs of the solution represented by the object and returns this objective value.

The `LocalSearch` class is an abstract class and merges common attributes and methods of the implementation of the four different neighbourhood structures, namely Edge Exchange, Node Swap, Centre Exchange and Level Change. In the constructor of the

`LocalSearch` class a reference to a `Solution` object is passed to. This enables different `LocalSearch` objects, in case they are referencing the same object, to operate on the same `Solution` object in a sequential order. Additional instance specific information can be accessed via this reference to the `Solution` object and its pointer to the `Instance` object. The `Execute()` method, each subclass of `LocalSearch` has to implement, starts the improvement process, stopping in a local optimal solution with respect to the neighbourhood structure implemented by this subclass. In case the objective value of the solution passed to the constructor of the `LocalSearch` class has been improved, `Execute()` returns `true`, otherwise `false`.

`EdgeExchange` is a subclass of the abstract `LocalSearch` class and implements the Edge Exchange neighbourhood. All improvement moves found by the `Execute()` method are executed directly on the `Solution` object passed to it in the constructor. This holds true for all other subclasses of the abstract `LocalSearch` class, too.

`Node Swap`, another subclass of `LocalSearch`, implements the second neighbourhood structure based on the successor and predecessor relationships of the nodes, beside Edge Exchange, namely – as the name already presumes – Node Swap.

The `LevelBased` class is abstract and summarises attributes and methods used by its two subclasses, `CentreExchange` and `LevelChange`, implementing neighbourhoods based on the level representation of a solution. An example for such an additional attribute is `nodesAtLevel` which stores for each level all nodes assigned to it.

As already mentioned `CentreExchange` and `LevelChange` are subclasses of `LevelBased`. The `Execute()` method of `CentreExchange` computes a local optimal solution with respect to the Centre Exchange neighbourhood, whereas `LevelChange` implements the Level Change neighbourhood.

The `Aco` class is the core of this ant colony optimisation program. It is responsible for creating all the required objects, as there are for example the sole object of the `Instance` class, objects of `Solution` and subclasses of the abstract `LocalSearch` class. Furthermore, it administrates – strictly speaking initialises and updates – the pheromone matrix. Based on the information stored in this pheromone matrix each artificial ant builds a solution by calling the `CreateSolution()` method. These solutions are then tried to be locally improved by a variable neighbourhood descent algorithm using the four different neighbourhood structures implemented as subclasses of `LocalSearch`.

6.1 User Manual

The proposed ant colony optimisation metaheuristic, implemented in C++, is a simple command line program, named `Aco`. The program was developed under Linux 2.4.21. The program was compiled using `gcc` in version 3.3.1 under `glibc` in version 2.3.2. To start the program some mandatory parameters have to be specified. If they are missing the program will prompt an error message followed by a usage instruction. Furthermore, `Aco` has some optional parameters, that can be omitted when starting the ant colony optimisation program. In the following a synopsis of the program parameters is shown.

```
./Aco -t graphtype -d diameter -i instancefile [-g instancefile] [-r rho]
      [-s time_limit] [-l iteration_limit] [-a ants]
```

Here in alphabetical order the list describing all optional as well as mandatory parameters of the `Aco` program in detail:

- **a** is an optional parameter. It indicates the number of artificial ants forming the ant colony. If it is omitted a default colony size of 10 ants is used.
- **d** is a mandatory parameter, since it specifies the diameter bound. The number declared has to be a positive integer number greater or equal to 4, otherwise an error message is prompted.
- **i** has also to be specified, stating the path to the instance file.
- **g** is only a mandatory parameter when a GNUplot graph instance is used, since GNUplot instances consist of two files, a point and a line file. So in case of a GNUplot graph instance the parameter **g** takes the line file, the corresponding point file has to be specified using parameter **i**. In case of all other graph types this parameter has to be omitted, otherwise an error message is prompted.
- **l** specifies the maximum number of iterations without an improvement of the best solution found so far. As it is an optional parameter it can be omitted, in this case the program sets it per default to 1000.
- **r** is an optional parameter, with $0 \leq \mathbf{r} \leq 1$. This parameter represents the ρ value used in the pheromone update process. The greater ρ, the faster the ant colony commits itself to a certain solution. If omitted a default value of 0.5 is used.

- **s** is an optional parameter indicating the time limit in seconds, otherwise no time limit is set. Only positive integer numbers are valid values for this parameter, all other specifications will result in a corresponding error message.
- **t** is a mandatory parameter and specifies the type of the instance file. The instance parser used by this program is able to read in four different file formats:
  - 0 GNUplot graph
  - 1 CostMatrix graph (Gouveia and Magnanti [18])
  - 2 EA graph
  - 3 Santos graph (Santos et al. [14])

The output of the `Aco` program is sent to standard out (`std::cout`) per default, but it can easily be redirected to any file with the help of the appropriate Unix operator ">". The output starts with a summary of all instance relevant information, e.g. instance file, number of nodes or diameter bound. Then a summary of each iteration follows, containing various information on every ant of the colony. Each ant builds a solution exploiting the knowledge stored in the pheromone matrix. Afterwards, this solutions is tried to be improved by a VND algorithm using four different neighbourhood structures. This iteration summary states for each artificial ant the total costs of the solution found after local improvement by VND, the total costs of the best solution found so far, time needed so far and finally an indicator if the current best solution was achieved by this ant. Finally, when a termination condition is met – either time limit or x rounds without further improvement of the best found solution – a complete summary of the optimisation process is printed, containing again all instance relevant information plus total costs of the best solution found and the time in seconds until the program was terminated. Note, that it in general the time stated at the end of this summary is not the time required to achieve the best solution. In the following a sample output of an `Aco` run is presented:

```
------------------------------------------------------------------------
Bounded diameter minimum spanning tree problem: ACO  [ver. 1.0 / 10.2005]
------------------------------------------------------------------------

Reading in instance ...
Successfully read in EA instance... ../../data/ea/estein500_01.eu

    number of nodes: 500
    number of edges: 124750
    diameter:        20
```

```
Creating Solution object to save best solution ... done
Creating Solution object for computations ... done
Creating EdgeExchange object ... done
Creating NodeSwap object ... done
Creating CentreExchange object ... done
Creating LevelChange object ... done

Creating starting solution value ... done


Creating Aco object ... done
    build time:           0.35 sec.
    objective value:      2215672

-------------------------------------------------------------------------

Starting Ant Colony Optimisation:
  objective value (starting solution):              2215672
  neighbourhood order:                              escl
  number of ants (colony size):                     10
  rho:                                              0.1000
  maximum number of iterations without improvement: 1000
  time limit (in seconds, 0 for no time limit):     4000

-------------------------------------------------------------------------
* Ant 0 found     1877464 [vnd]     1877464 [ACO]     1.75 sec.
* Ant 1 found     1858671 [vnd]     1858671 [ACO]     3.65 sec.
  Ant 2 found     1885506 [vnd]     1858671 [ACO]     5.30 sec.
  Ant 4 found     1918045 [vnd]     1836331 [ACO]     8.92 sec.
  Ant 5 found     1884828 [vnd]     1836331 [ACO]     11.30 sec.
  Ant 6 found     1887286 [vnd]     1836331 [ACO]     13.10 sec.
  Ant 7 found     1945281 [vnd]     1836331 [ACO]     15.31 sec.
  Ant 8 found     1876021 [vnd]     1836331 [ACO]     18.13 sec.
  Ant 9 found     1918894 [vnd]     1836331 [ACO]     20.05 sec.
-------------------------------------------------------------------------
Updating pheromone matrix...         1836331 [ACO]     20.05 sec.

-------------------------------------------------------------------------
  Ant 0 found     1844813 [vnd]     1836331 [ACO]     21.46 sec.
                                .
                                .
                                .
* Ant 8 found     1676033 [vnd]     1676033 [ACO]     2726.53 sec.
                                .
                                .
                                .
-------------------------------------------------------------------------
```

```
Summary of optimisation:

instance:                  ../../data/ea/estein500_01.eu
number of nodes:           500
number of edges:           124750
diameter:                  20
starting solution:         2215672
best solution:             1676033
improvement:               539639
time (starting solution): 0.35 sec.
time (ACO/VND):            4000.27 sec.
time (total):              4000.62 sec.

--------------------------------------------------------------------------
```

7. COMPUTATIONAL RESULTS

This chapter takes a closer look at the performance of the ACO algorithm for the BDMST problem, developed within the context of this master thesis, and compares it to that of other heuristics for the BDMST problem. Three state-of-the-art metaheuristics have been selected for this comparison, a VNS implementation by Gruber and Raidl [21], a permutation coded EA as well as a random-key coded EA from [25]. Of major interest is the comparison of the ACO presented here and the VNS by Gruber and Raidl, since they use in principle the same neighbourhood structures for their VND as it is done in the ACO, especially because the VNS is the so far leading metaheuristic for the BDMST problem.

For comparison the same instances, diameter bounds, time limits and termination conditions as in [21] were used. The instances were taken from the Beasley's OR-Library[1] and have been originally proposed for the Euclidean Steiner tree problem. All nodes of a graph are settled within the unit square and the Euclidean distance between any pair of nodes corresponds to the edge costs between these two nodes. Furthermore, as in [21], the tests were grouped into long-term and short-term runs. The long-term runs concentrated on achieving as good results as possible. In the short-term runs the different algorithms had a relatively short time with respect to the instance size for yielding good solutions. The size of the ant colony was set to ten artificial ants for all test categories. Before presenting the computational results it should be noted, that all test runs were performed on a Pentium®4 2.8 GHz system using Linux 2.4.21 as operating system.

First the results of the long-term runs are discussed. As in [21] the first five instances of each size $n = 100$, 250, 500 available in the OR-Library were used and the diameter bound was set to 10 for instances with 100 nodes, to 15 for instances with 250 nodes, and to 20 for instances with 500 nodes. Two different termination conditions were introduced for these long-term runs for the ACO, adopted from [21]: The ACO algorithm terminated either if the last 1000 iterations yielded no improvement of the best found solution so far or if a time limit was reached. This time limit was set to 2000 seconds for instances with 100 nodes, to 3000 seconds for instances with 250 nodes, and to 4000 seconds for instances with 500

[1] `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/esteininfo.html`

Instance			permutation coded EA			random-key coded EA			VNS			ACO				
n	D	nr.	best	mean	stddev	best	mean	stddev	best	mean	stddev	ρ	best	mean	stddev	time (sec.)
100	10	1	7.818	7.919	0.07	7.831	7.919	0.05	**7.759**	7.819	0.03	0.01	**7.759**	**7.771**	0.007	196.06
		2	7.873	8.017	0.08	7.853	8.043	0.09	7.852	7.891	0.03		**7.849**	**7.869**	0.004	126.43
		3	7.990	8.139	0.08	7.982	8.137	0.09	**7.904**	7.962	0.04		7.907	**7.946**	0.014	200.40
		4	8.009	8.143	0.07	7.996	8.122	0.06	**7.979**	8.046	0.03		**7.979**	**7.995**	0.005	200.15
		5	8.193	8.335	0.08	8.198	8.313	0.08	8.165	8.203	0.03		**8.164**	**8.168**	0.002	158.37
250	15	1	12.440	12.602	0.08	12.448	12.580	0.08	12.301	12.430	0.05	0.05	**12.245**	**12.333**	0.013	1159.08
		2	12.237	12.432	0.10	12.222	12.393	0.10	12.024	12.171	0.06		**12.023**	**12.097**	0.015	1219.32
		3	12.117	12.282	0.08	12.178	12.315	0.07	12.041	12.112	0.04		**12.017**	**12.058**	0.009	1034.92
		4	12.572	12.824	0.11	12.632	12.802	0.07	12.507	12.615	0.06		**12.474**	**12.559**	0.011	1108.55
		5	12.358	12.608	0.12	12.382	12.623	0.10	12.281	12.423	0.07		**12.228**	**12.360**	0.016	1152.07
500	20	1	17.216	17.476	0.10	17.156	17.429	0.10	16.974	17.129	0.07	0.1	**16.909**	**17.028**	0.018	2346.49
		2	17.085	17.311	0.11	17.097	17.291	0.10	16.879	17.052	0.07		**16.760**	**16.875**	0.019	2726.73
		3	17.173	17.449	0.11	17.164	17.369	0.11	16.975	17.148	0.07		**16.929**	**16.990**	0.014	2433.62
		4	17.215	17.484	0.13	17.266	17.432	0.09	16.992	17.166	0.06		**16.908**	**17.071**	0.026	2384.56
		5	16.939	17.137	0.11	16.872	17.092	0.11	16.572	16.786	0.07		**16.512**	**16.640**	0.022	2890.01

Tab. 7.1: Long-term runs on Euclidean instances

nodes.

Table 7.1 shows the computational results of the long-term runs. It lists for each instance the number of nodes, the diameter bound, the instance number, and for each metaheuristic the objective values of the best and mean solutions found. Furthermore, the standard deviation of 50 (EAs) and 30 (VNS, ACO) independent runs is presented.

In addition, for the ACO the mean times for identifying the best solutions are given. As indicated in Table 7.1 ρ varies among the different instance sizes in order to obtain better solutions. The results for the EAs as well as those for the VNS are taken from [21].

Like the VNS implementation the ACO algorithm outperforms the two EAs concerning either best as well as mean solution values. Comparing the ACO with the VNS it can be seen that on instances with 100 nodes the ACO algorithm is still close to the VNS implementation concerning best solutions, but is already better regarding mean values. The greater the instances the larger the gap between the ACO algorithm and the VNS. While on instances of size $n = 250$ the ACO performs already clearly better than the VNS concerning best as well as mean solutions, on instances with 500 nodes the mean objective value of the solutions found by the ACO is close to – in one case it is even better than – the best solution identified by the VNS. In addition, the standard deviation of the ACO algorithm is much smaller than those of the three other metaheuristics. Furthermore, it is interesting to observe, that the time limit was never reached. For each instance size 1000 iterations without further improvement terminated the ACO algorithm, in contrast to, for example, the VNS, which requires the whole 4000 seconds to compute the solutions listed for the instances with 500 nodes.

As in [21] for the short-term tests the permutation coded EA was replaced by an edge-

set coded EA from [31], since it makes use of operators with linear time complexity and therefore scales much better with larger instances. The same tests should be performed but for the 1000 node instances the time limit turned out to be really though for the ACO. When using the time limit of 100 seconds no useful results could be achieved due to the following reason: In the first iteration of the colony with its ten artificial ants every solution constructed by an artificial ant is more or less random, as the pheromone matrix contains no practical information yet (it is initialised uniformly). Therefore the variable neighbourhood descent part of the ACO algorithm can substantially improve these solutions, but this improvement process requires a lot of time due to the fact that the quality of the solution constructed by an ant is very poor. As a consequence only four to six solutions could be processed within the time limit of 100 seconds. Since not even one iteration could be finished within this time limit, thus no update of the pheromone matrix was performed, no relevant results for the ACO can be presented here for the 1000 nodes instances in combination with a time limit of 100 seconds. Using a time limit of 1000 seconds enables the ACO algorithm to finish at least five to eight complete iterations.

Table 7.2 presents the results of the short-term runs. It shows the number of nodes, the diameter bound, the instance number, and the time limit in seconds. Furthermore, for each metaheuristic the objective values of the best solutions found, the mean values and the standard deviations are listed. For each instance 50 (EAs) respectively 30 (VNS, ACO) independent runs have been performed.

In addition for the ACO algorithm the parameter ρ used for the experiments is given, since varying it depending on the time limit and instance size yielded better results. Again the results for the EAs as well as those for the VNS are taken from [21].

From Table 7.2 it can be seen that the ACO algorithm consistently outperforms the two EAs. Considering the 500 node instances, the mean values of the ACO with tighter time limits, as well as those of the VNS, are even better than the objective values of the best solutions found by any of the EAs. Unfortunately, the good results of the VNS cannot be reached by the ACO. Looking at the 500 node instances it can be seen that when using a time limit of 500 seconds the ACO algorithm comes closer to the results of the VNS but cannot really exceed them. Just on two instances the ACO yields a better solution, and it beats the VNS only one time concerning the mean values. The main reason for this behaviour is – as already mentioned above – that the ACO algorithm starts with very poor, more or less random solutions, since in the first iteration the solutions constructed by the artificial ants are based on a uniformly initialised pheromone matrix. In the following

Instance			time	edge-set coded EA			random-key coded EA			VNS			ACO			
n	D	nr.	limit (sec.)	best	mean	stddev	best	mean	stddev	best	mean	stddev	ρ	best	mean	stddev
500	20	1	50	19.368	19.830	0.17	21.223	21.440	0.07	**17.753**	**18.108**	0.12	0.5	17.905	18.199	0.035
		2		19.156	19.522	0.13	20.836	21.097	0.09	**17.688**	**17.966**	0.10		17.832	18.083	0.037
		3		19.321	19.888	0.16	21.042	21.304	0.11	**17.799**	**18.114**	0.10		17.990	18.202	0.027
		4		19.464	19.866	0.19	21.129	21.432	0.09	**17.930**	**18.161**	0.11		18.181	18.308	0.023
		5		19.209	19.477	0.17	20.728	21.017	0.11	**17.464**	**17.863**	0.12		17.660	18.008	0.047
500	20	1	500	18.470	18.976	0.13	19.658	19.908	0.14	**17.290**	**17.460**	0.08	0.16	17.347	17.507	0.019
		2		18.442	18.810	0.22	19.332	19.651	0.13	17.215	17.373	0.08		**17.111**	**17.333**	0.027
		3		18.619	19.056	0.18	19.618	19.887	0.10	**17.252**	**17.464**	0.05		17.270	17.485	0.033
		4		18.745	19.116	0.17	19.654	19.905	0.11	17.318	**17.514**	0.07		**17.294**	17.571	0.029
		5		18.197	18.685	0.20	19.312	19.635	0.10	**16.932**	**17.139**	0.09		17.031	17.159	0.029
1000	25	1	100	28.721	29.265	0.16	30.996	31.288	0.11	25.850	26.188	0.13	-	-	-	-
		2		28.607	29.105	0.19	30.832	31.132	0.11	25.501	25.981	0.17		-	-	-
		3		28.410	28.905	0.17	30.515	30.856	0.12	25.340	25.705	0.09		-	-	-
		4		28.695	29.263	0.21	30.966	31.277	0.08	25.562	26.128	0.17		-	-	-
		5		28.396	28.882	0.19	30.633	31.010	0.10	25.504	25.826	0.15		-	-	-
1000	25	1	1000	26.494	26.936	0.14	30.097	30.401	0.13	**25.177**	**25.572**	0.14	0.4	25.414	25.709	0.034
		2		26.300	26.789	0.24	29.924	30.261	0.12	**25.015**	**25.342**	0.14		25.281	25.527	0.028
		3		25.762	26.556	0.21	29.586	29.981	0.12	**24.816**	**25.086**	0.11		24.895	25.443	0.051
		4		26.470	26.816	0.15	29.946	30.329	0.13	**25.289**	**25.572**	0.11		25.391	25.716	0.038
		5		26.117	26.606	0.19	29.782	30.151	0.12	25.026	**25.254**	0.12		**24.975**	25.405	0.039

Tab. 7.2: Short-term runs on Euclidean instances

these solutions can be improved significantly by the variable neighbourhood descent, but at the expense of a high running time. Another aspect that has to be taken into account in the short-term runs is the value chosen for the parameter ρ which controls how much influence the best solution identified within one iteration shall have on the pheromone and – consequently – on the probabilities calculated based on this pheromone information for the following iterations. Remember, that based on these probabilities all artificial ants construct solutions. The greater the ρ value the higher the probability to assign node v to level l also in a subsequent iteration (if this combination was part of the best solution) since the pheromone value for this combination of node and level will dominate the other ones relatively fast. As a consequence the ACO algorithm converges to a certain solution after only a small number of iterations. As the first solutions are in general of a poor quality (even after the VND local improvement step), a great ρ value will not yield good solutions at all, since a solution close to these bad solutions will be favoured very soon. On the other hand, if the value of the parameter ρ is chosen too low the pheromone updates written into the pheromone matrix are so small compared to the values already stored there that they have only marginal effects on the probabilities calculated from the pheromone information. So within the next iteration nearly the same probabilities can be met. When using a too tough time limit with respect to the ρ value it is likely to occur that the already collected information cannot be exploited at all. Table 7.2 indicates this effect. For instances with 500 nodes and a time limit of 50 seconds a relatively great ρ value has been used in order to make use of the information already written to and stored in the pheromone matrix. By

decupling the time limit an obviously smaller ρ can be applied. This smaller ρ is responsible for the behaviour that the algorithm does not favour – as fast as in the prior test category – a certain solution, but explores a greater part of the solution space before preferring a certain solution. This leads inevitably to much better results.

Generally it can be noted, that the ACO algorithm is very slow in the beginning of the computation process, as the solutions constructed in this phase are very poor and the variable neighbourhood descent part can substantially improve these solutions, but this local improvement step is very time consuming. As time proceeds the artificial ants will construct better and better solutions, due to the information stored in the pheromone matrix. As a consequence the neighbourhood descent part needs less and less time to improve these solutions. So the time required for one iteration decreases over the whole runtime of the ACO algorithm, as the time needed by an artificial ant to build a solution is constant.

8. CONCLUSIONS

This master thesis proposed an ant colony optimisation algorithm for the bounded diameter minimum spanning tree problem. Its main characteristic is that it makes use of a local optimisation heuristic, namely a variable neighbourhood descent, to improve overall solution quality. This VND approach is based on four different neighbourhood structures for the BDMST problem, namely Edge Exchange, Node Swap, Centre Exchange and Level Change. When developing these neighbourhood structures main focus was on an efficient and fast exploration of a certain neighbourhood, for example computing only cost differences when evaluating a solution. For two of them, namely Node Swap and Level Change, an incremental enumeration, applied after having identified and executed the most profitable move, was presented. This incremental exploration could not reduce the theoretical worst case time complexity, but accelerated computation essential in practise.

Another important characteristic of the ant colony optimisation algorithm, developed within the context of this master thesis, is the way artificial ants construct solutions. A tree is not built by starting from a single node and successive connecting the remaining ones, but ants distribute the nodes to various levels, since given this level information always a local optimal minimum spanning tree ensuring the diameter bound can be derived in an easy and straightforward way.

To evaluate the overall performance of the ant colony optimisation algorithm it was compared to the so-far-leading variable neighbourhood search implementation for the BDMST, that operates on the same neighbourhood structures, as well as to three evolutionary algorithms, namely a permutation, a random-key and a edge-set coded EA. Results on complete Euclidean instances turned out, that when considering solution quality the ACO algorithm is not only superior to the EAs but also to the VNS, especially on larger instances. In case the computation time is highly restricted the solutions provided by the ACO algorithm are in-between those of the EAs and those of the VNS.

To conclude, the ant colony optimisation algorithm, proposed within the context of this master thesis, has its strengths when operating on large complete Euclidean instances

without a time limit. A starting point for further research activities could be to adapt this ACO approach in a way that it also works on instances where the underlying graph is incomplete or, since the ACO algorithm was only compared to other metaheuristics on Euclidean instances, to test its effectiveness on random instances.

BIBLIOGRAPHY

[1] Ayman Abdalla, Narsingh Deo, and Pankaj Gupta. Random-tree diameter and the diameter constrained MST. *Congressus Numerantium*, 144:161–182, 2000.

[2] N. R. Achuthan and L. Caccetta. Models for vehilce routing problems. *Proceedings of the 10th National Conference of the Australian Society for Operations Research*, pages 276–294, 1990.

[3] N. R. Achuthan and L. Caccetta. Minimum weight spanning trees with bounded diameter. *Australasian Journal of Combinatorics*, 5:261–276, 1992.

[4] N. R. Achuthan, L. Caccetta, P. Caccetta, and J. F. Geelen. Computational methods for the diameter restricted minimum weight spanning tree problem. *Australasian Journal of Combinatorics*, 10:51–71, 1994.

[5] K. Bala, K. Petropoulos, and T. E. Stern. Multicasting in a linear lightwave network. In *IEEE INFOCOM'93*, pages 1350–1358, 1993.

[6] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[7] A. Bookstein and S. T. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4):387–400, 1991.

[8] A. E. F. Clementi, M. Di Ianni, A. Monti, G. Rossi, and R. Silvestri. Experimental analysis of practically efficient algorithms for bounded-hop accumulation in ad-hoc wireless networks. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), workshop 12*, volume 13, page 247.1, 2005.

[9] J.-L. Deneubourg, S. Aron, S. Goss, and J.-M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3:159–168, 1990.

[10] M. Dorigo. *Optimization, Learning and Natural Algorithms (in Italian)*. PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, IT, 1992.

[11] M. Dorigo and L. M. Gambardella. Ant colonies for the traveling salesman problem. *BioSystems*, 43:73–81, 1997.

[12] M. Dorigo and L. M. Gambardella. Ant colony system: A coopeartive learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[13] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy. Technical Report 91-016, Dipartimento die Elettronica, Politecnico di Milano, IT, 1991.

[14] A. C. dos Santos, A. Lucena, and C. C. Ribeiro. Solving diameter constrained minimum spanning tree problems in dense graphs. In *Proceedings of the International Workshop on Experimental Algorithms*, volume 3059 of *LNCS*, pages 458–467. Springer, 2004.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.

[16] F. Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13:533–549.

[17] S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, 1989.

[18] Luis Gouveia and Thomas L. Magnanti. Network flow models for designing diameter-constrained minimum spanning and Steiner trees. *Networks*, 41(3):159–173, 2003.

[19] Luis Gouveia, Thomas L. Magnanti, and Christina Requejo. A 2-path approach for odd-diameter-constrained minimum spanning and Steiner trees. *Networks*, 44(4):254–265, 2004.

[20] P. P. Grassé. La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: essai d´interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–81, 1959.

[21] M. Gruber and Günther. R. Raidl. *Variable Neighborhood Search for the Bounded Diameter Minimum Spanning Tree Problem.* PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2005.

[22] Martin Gruber and Günther R. Raidl. A new 0–1 ILP approach for the bounded diameter minimum spanning tree problem. In L. Gouveia and C. Mourão, editors, *Proceedings of the 2nd International Network Optimization Conference*, volume 1, pages 178–185, Lisbon, Portugal, 2005.

[23] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-heuristics, Advances and Trends in Local Search Paradigms for Optimization*, pages 433–458. Kluwer Academic Publishers, 1999.

[24] B. A. Julstrom and G. R. Raidl. A permutation-coded evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In A. Barry, F. Rothlauf, D. Thierens, et al., editors, *in 2003 Genetic and Evolutionary Computation Conference's Workshops Proceedings, Workshop on Analysis and Desgn of Representations*, pages 2–7, 2003.

[25] Bryant A. Julstrom. Encoding bounded-diameter minimum spanning trees with permutations and with random keys. In Kalyanmoy Deb et al., editors, *Genetic and Evolutionary Computation Conference – GECCO 2004*, volume 3102 of *LNCS*, pages 1282–1281. Springer, 2004.

[26] Bryant A. Julstrom. Greedy heuristics for the bounded-diameter minimum spanning tree problem. Technical report, St. Cloud State University, 2004. Submitted for publication in the ACM Journal of Experimental Algorithmics.

[27] T. Magnanti and R. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18(1), 1984.

[28] N. Mladenović. *A variable neighborhood algorithm - a new metaheuristic for combinatorial optimization. Abstracts of papers presented at Optimization Days.* Montreal, 1995.

[29] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers Opers. Res.*, 24:1097–1100, 1997.

[30] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization-Algorithms and Complexity.* Dover Publications, Inc., New York, 1982.

[31] Günther R. Raidl and Bryant A. Julstrom. Greedy heuristics and an evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In Gary Lamont

et al., editors, *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 747–752, New York, 2003. ACM Press.

[32] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[33] T. Stützle and H. Hoos. Introducing $\mathcal{MAX} - \mathcal{MIN}$ ant system. In *Proceedings of the International Conference on Artifical Neural Networks and Genetic Algorithms*, pages 245–249. Springer Verlag, 1997.

[34] T. Stützle and H. Hoos. $\mathcal{MAX} - \mathcal{MIN}$ ant system and local search for the traveling salesman problem. In *Proceedings of IEEE-ICEC-EPS'97, IEEE Conference on Evolutionary Computation and Evolutionary Programming Conference*, pages 309–314. IEEE Press, 1997.

[35] S. Voß, S. Martello, I. H. Osman, and C. Routacirol. *Meta-Heuristics-Advances and Trends in Local Search Paradigms for Optimization.* Kluwer Academic Publishers, Dordrecht, The Netherlands.