FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A New Partition-based Heuristic for the Steiner Tree Problem in Large Graphs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Martin Luipersbeck
Matrikelnummer 0725756

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Lektorin Mag.rer.nat. Dr.techn. Ivana Ljubić, Privatdoz.
Mitwirkung: Dipl.-Ing. Dr.techn. Markus Leitner

Wien, 04.12.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A New Partition-based Heuristic for the Steiner Tree Problem in Large Graphs

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Martin Luipersbeck
Registration Number 0725756

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Lektorin Mag.rer.nat. Dr.techn. Ivana Ljubić, Privatdoz.
Assistance: Dipl.-Ing. Dr.techn. Markus Leitner

Vienna, 04.12.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Martin Luipersbeck
Batthyany-Allee 29, 7431 Bad Tatzmannsdorf

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

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

In erster Linie möchte ich mich bei Dr. Ivana Ljubić und bei Dr. Markus Leitner für die kompetente Betreuung beim Verfassen dieser Diplomarbeit bedanken.

Ein herzliches Dankeschön gilt auch allen meinen Freunden, die mich in den letzten Jahren unterstützt haben. Im Besonderen bedanke ich mich bei Max Resch für die gute Zusammenarbeit.

Schließlich möchte ich auch meinen Eltern danken, die mir mein Studium ermöglicht haben.

Abstract

The Steiner tree problem in graphs (STP) is a fundamental $\mathcal{NP}$-hard combinatorial optimization problem of theoretical and practical interest. Common applications range from VLSI design to problems in computational biology. The STP can be informally described as the problem of connecting a subset of special vertices called terminals in a weighted graph at minimum cost. Due to the problem's complexity the computation of optimal solutions may not always be feasible. This holds true especially for large-scale instances which are quite common in real-world scenarios. In such cases, heuristic methods specialized on finding near-optimal solutions in reasonable amounts of time, are generally the only choice.

In this master's thesis we propose a new partition-based heuristic for the efficient construction of approximate solutions to the STP in very large graphs. Our algorithm is based on a partitioning approach in which instances are divided into several subinstances which are small enough to be solved optimally. A heuristic solution of the complete instance can then be constructed through the combination of the subinstances' solutions.

To this end we combine state-of-the-art exact and heuristic methods for the STP and general graph partitioning. For the exact solution of subinstances we apply a branch-and-cut procedure. The underlying integer linear programming (ILP) model augments a formulation based on the well-known directed-cut-constraints with node variables. The associated separation procedure includes several improvements from literature. For partitioning we use the METIS graph partitioning framework as well as a greedy partitioning algorithm based on the contraction of Voronoi regions.

The implemented algorithms are also embedded into a memetic algorithm, which includes the partition-based construction heuristic, reduction tests, an algorithm for solution recombination and a variable neighborhood descent. We use common neighborhood structures from the STP literature: Steiner node insertion, Steiner node elimination, key-node elimination and key-path exchange.

All algorithms are evaluated through practical experiments on the SteinLib, a state-of-the-art benchmark set for the STP, and a set of new real-world instances from network design. The results show that our approach yields good quality solutions with reasonable runtime, even for large graphs.

Kurzfassung

Das Steinerbaumproblem in Graphen (STP) ist ein $\mathcal{NP}$-schweres kombinatorisches Optimierungsproblem, welches sowohl aus theoretischer als auch aus praktischer Sicht relevant ist. Die Anwendungsfälle reichen vom VLSI-Design bis hin zum Lösen von wissenschaftlichen Problemen in der Bioinformatik. Beim STP sollen eine Menge an Basisknoten in einem gewichteten Graphen kostenminimal verbunden werden. Da dieses Problem sehr schwierig ist, ist es nicht immer möglich eine optimale Lösung zu finden. Problematisch sind vor allem große Instanzen, die in praktischen Anwendungen relativ häufig auftreten. In solchen Fällen bleibt oft nur die Verwendung von heuristischen Methoden. Diese sind auf die Berechnung von guten, jedoch suboptimalen Lösungen in relativ kurzer Zeit spezialisiert.

In dieser Diplomarbeit wird eine neue Konstruktionsheuristik vorgestellt, die Partitionierungsmethoden nutzt, um speziell mit großen Probleminstanzen umgehen zu können. Hierzu wird eine Instanz systematisch in kleinere Instanzen zerlegt, die einfach genug sind, um sie mit einem exakten Algorithmus optimal zu lösen. Danach wird eine heuristische Lösung der ursprünglichen Instanz durch Zusammensetzen der Teillösungen erzeugt.

Zur Realisierung dieses Verfahrens werden sowohl exakte und heuristische Lösungsmethoden für das STP als auch Algorithmen zur Partitionierung von Graphen kombiniert. Für die Berechnung von exakten Lösungen wird ein Branch-and-Cut Verfahren verwendet. Das zugrundeliegende ILP-Model basiert auf den bekannten directed-cut-constraints, führt jedoch zusätzlich noch die Verwendung von Knotenvariablen ein. Der zugehörigen Separierungsmethode liegen verschiedene Verbesserungen aus der Literatur zugrunde. Zur Partitionierung wird das METIS Graph Partitioning Framework verwendet. Außerdem wird ein einfacher Greedy-Algorithmus vorgestellt, welcher eine Instanz durch die Kombination mehrerer Regionen in einem Voronoi-Diagram erstellt.

Die implementierten Algorithmen werden zusätzlich in einen memetischen Algorithmus integriert, darunter die vorgestellte Konstruktionsheuristik, Reduktionstests, ein Algorithmus zur Rekombination von Lösungen und Variable Neighorhood Descent. Die verwendeten Nachbarschaftsstrukturen basieren auf Steiner node insertion, Steiner node elimination, key-node elimination und key-path exchange.

Alle Algorithmen werden experimentell evaluiert. Die Testinstanzen dafür stammen aus der SteinLib, welche eine Sammlung von Benchmark-Instanzen für das STP darstellt, und aus einer Gruppe aus neuen Instanzen, die Netzwerkdesignprobleme aus der Praxis beschreiben. Die Ergebnisse zeigen, dass Lösungsqualität und Laufzeit des vorgestellten Verfahrens auch für große Instanzen akzeptabel sind.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Background & Motivation

Optimization problems are ubiquitous in virtually any area of human endeavour. Since resources are generally scarce, finding optimal decisions on how to use them is clearly a relevant problem. The list of practical applications is quite large. Some important examples are network design and routing. Informally, optimization can be described as finding the best solution among alternatives. Mathematically, an optimization problem corresponds to maximizing or minimizing an objective function over some domain and some set of feasible solutions which is usually restricted by a set of constraints. A combinatorial optimization problem (COP) is a specific type of optimization problem, in which the domain of choosable elements is finite or countable infinite. Such problems are of special interest, since a large number of real-world problems can be modeled in these terms. It is noteworthy that the restriction from an infinite to a finite set does not necessarily make problems any easier, since the set of possible combinations between elements is still exponentially large. A fundamental example is the $\mathcal{NP}$-hard [35] Steiner tree problem in graphs.

Definition 1.1 *Steiner tree problem in graphs (STP) [77]*
Consider an undirected graph $G = (V, E)$ where V is the set of vertices and E the set of edges, for which a cost function $c : E \to \mathbb{Q}_+$ assigns a value to each edge. Given a set $T \subseteq V$ of special vertices called terminals, the objective of the STP is to find a subgraph $S = (V_S, E_S)$ of G spanning all terminals that minimizes the cost function $c(E_S) := \sum_{e \in E_S} c(e)$.

Figure 1.1 shows a simple STP instance and a weight-minimal subgraph which connects all terminals. Here, the subgraph contains both terminals and non-terminals. A non-terminal is generally referred to as Steiner node. We observe that if the edge costs are positive and the objective is to minimize cost, the subgraph will always correspond to a tree, which is called a

Steiner minimal tree in literature [77]. However, in the remainder of this work we will refer to a Steiner minimal tree simply as an *optimal solution*, and to any subgraph connecting all terminals which is not weight-minimal as a *feasible solution*.

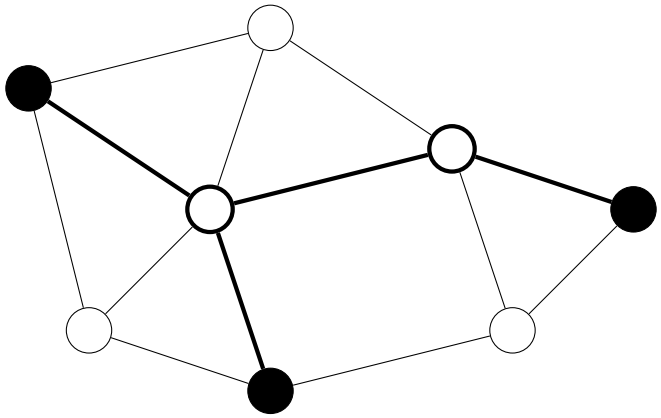


Figure 1.1: A simple STP instance and an optimal solution when assuming that all edge costs are equal. The terminals are marked black and the non-terminals are marked white. Some of the Steiner nodes are chosen to be part of the connected subgraph. The edges belonging to the solution are marked bold.

The STP appears in many practical network design problems. Some of these problems may be modeled by the STP directly. The construction of phylogeny trees in computational biology is a relevant example. A phylogeny tree describes inferred evolutionary relationships between known species. Other problems may be derived from the STP, e.g., through the augmentation of additional constraints. An example is the hop constrained Steiner tree problem, which restricts the maximum number of edges by which a terminal may be connected from a specific root terminal. Such a restriction is especially relevant for routing in computer networks [74]. In some cases the STP appears as a subproblem in algorithms for the efficient solution of other problems. For decomposition techniques like column generation and Lagrangian schemes the resulting subproblems may take the form of the STP [13].

Due to its wide applicability, much scientific effort has been invested into the design of efficient algorithms for solving the STP. A general classification of solution approaches is to distinguish between exact and heuristic methods. While exact methods guarantee that an optimal solution is found given enough time, heuristic methods give no such guarantee, but usually compensate by taking less computation time. During the last decades the exact solution approaches to COPs have made much progress through the advance of efficient multipurpose integer linear program (ILP) solvers like CPLEX. However, when applying exact methods to the class of $\mathcal{NP}$-hard problems, to which many practically relevant problems belong, their runtime is still exponential in the worst case, assuming that the conjecture $\mathcal{P} \neq \mathcal{NP}$ holds. Thus their application is often impractical for large-scale problem instances with several thousand elements to choose from, which are quite common in real-world applications. If insufficient computational

resources are available, exact methods may have to be cancelled prematurely, which effectively turns them into heuristic methods. In such cases methods that specialize primarily in the construction of heuristic solutions may create better solutions in a shorter time frame, while the exact approaches may calculate useful bounds on the value of the optimal solution. The development of efficient heuristic methods is a relevant research topic of important practical concern.

This master's thesis proposes a new partition-based construction heuristic to compute near-optimal Steiner trees in graphs, especially for large-scale problem instances. The algorithm applies heuristic partitioning techniques to divide graphs into subcomponents which are small enough to be solved by an exact algorithm. The advantages of this approach are twofold: Firstly, it provides good capabilities for parallelization and distribution. This is clearly the case, since the subproblems can be solved independently. Secondly, fine-grained control over the trade-off between solution quality and speed is achieved by adjusting the size and number of partitions. In addition, the dual ascent algorithm is used to estimate the value of the optimal solution from below.

The proposed construction heuristic is also tested in the context of a partition-based memetic algorithm. In this procedure the partitioning is guided by the best available solution and is used iteratively in combination with local search. As a local search strategy we apply Variable Neighborhood Descent (VND).

The evaluation of the presented algorithms is conducted on the SteinLib [42], a state-of-the-art benchmark set for the STP, and on a set of new real-world instances (arising from telecommunication applications) with graphs containing up to 89 000 nodes. In addition to the algorithms' evaluation of effectiveness, we hope to gain insights on the effects of heuristic partitioning on the solution quality in the context of $\mathcal{NP}$-hard problems.

The results of this work are further enhanced through the master's thesis of Max Resch [62], which focuses on the analysis and evaluation of the proposed algorithms' parallel aspects and tests the developed approaches on applications in bioinformatics.

1.2 Outline of the Thesis

The rest of this thesis is organized as follows: In Chapter 2 relevant preliminary concepts and definitions are introduced. Chapter 3 lists related works and gives an overview of the state-of-the-art concerning the STP. Chapters 4 and 5 describe the implementation of the partition-based STP heuristic and a partition-based memetic algorithm, respectively. In Chapter 6 the computational results are analyzed. Finally, Chapter 7 summarizes the results and concludes the thesis.

CHAPTER 2

Preliminaries

In this chapter, we introduce definitions and concepts relevant to this work. These include data structures for efficient and simple implementations as well as useful algorithms which have been successfully applied to the STP or related problems. The presented algorithms follow various paradigms and have different objectives, but complement each other nicely.

2.1 MST-based Construction Heuristics

Many applications require the fast construction of feasible Steiner trees from scratch. Such procedures are not only useful by themselves, but are frequently needed to build more complex exact and heuristic algorithms in which they may appear as a subcomponent. Some of the simplest algorithms in this category are based on concepts for solving the minimum spanning tree (MST) problem to optimality. The MST problem is the special case of the STP for which all vertices are terminals. This special case can be solved in polynomial time by a greedy algorithm. An algorithm is considered as greedy if for each step the currently best option is chosen, without the consideration of past decisions [40].

In the following, we will present two well-known MST-based heuristics for the STP: the distance network heuristic (DNH) [49] and the shortest path heuristic (SPH) [70]. The approximation ratio of both algorithms is roughly equivalent, and can be bounded by two, i.e., the produced solutions have cost $\leq 2 \cdot \text{OPT}$, where OPT is the objective function value of an optimal solution.

The empirical results of the algorithms can often be improved by applying a simple postprocessing phase denoted as MST-Prune [3].

MST-Prune(S):

1. Given a solution $S = (V_S, E_S)$, compute the MST S' of the subgraph induced by $V_S \in S$.
2. Recursively remove (prune) all Steiner nodes of degree one in S'.

MST-Prune can be executed in $O(|E| + |V| \log |V|)$, since an MST can be computed in $O(|E| + |V| \log |V|)$ using a Fibonacci heap data structure and the number of pruned vertices is bounded from above by $|V|$.

2.1.1 Distance Network Heuristic

The distance network heuristic (DNH) [43] computes an approximate solution to the STP by applying an exact algorithm for solving the MST problem in the so-called distance network. More formally stated, given a weighted graph $G = (V, E, c)$ and terminals $T \subseteq V$, the distance network $D_G = (T, E', d)$ corresponds to a complete network on all terminal nodes. The edge weights d correspond to the shortest distance between each pair of terminals. The DNH finds the MST on D_G and re-maps all the edges to a new graph $S = (V_S, E_S)$, to which the procedure MST-Prune is finally applied.

The main drawback of this approach is the fact that computing the full distance network is rather costly and requires $|T|$ executions of the Dijkstra algorithm. Thus the worst-case runtime is bounded by $O(|T| \cdot (|E| + |V| \log |V|))$.

In 1988 Mehlhorn [49] proposed an implementation which reaches a lower worst-case runtime by exploiting the fact that it is not necessary to compute the full distance network. The idea behind the implementation can be formalized through the concept of *Voronoi diagrams in graphs*.

Definition 2.1 *Voronoi diagram in graphs [3]*
Given a graph $G = (V, E)$ and a set $T \subseteq V$, the Voronoi diagram of G with respect to T is a partitioning of V into a set of Voronoi regions. There exists a Voronoi region for each $t \in T$, denoted by vor(t). A node v belongs to the Voronoi region vor(t) iff it is closer to t than to any other $t' \neq t$, $t' \in T$. Ties are broken arbitrarily.

Thus a node $v \in V$ belongs exactly to one Voronoi region. In the remainder of this work, we will denote an edge that connects two Voronoi regions as *border edge* and an edge that connects nodes inside a Voronoi region as *inner edge*. The improved implementation comprises three sequential steps:

1. **Construct Voronoi Diagram:**
   A Voronoi diagram can be built efficiently through a single call of a modified version of the Dijkstra algorithm, in which all terminals are considered as sources. In a heap-based implementation, this is achieved by simply inserting the terminals into the heap at the start of the algorithm (which is equivalent to adding a single artificial source connected to all terminals by zero-weight edges) [73]. The following information is stored for each node $v \in V$:

   - base(v): the terminal which is closest to v.
   - dist(v): the distance of the shortest path from v to the closest terminal.
   - pred(v): the predecessor node of v on the shortest path to the nearest terminal.

2. **Construct MST:**
   It is sufficient to apply an arbitrary exact algorithm for solving the MST on the subgraph implied by the set of border edges [49]. A new weight value is computed for each border edge $e = (v, w)$ as follows:

$$c'_e = c_e + \text{dist}(v) + \text{dist}(w)$$

3. **Construct Solution:**
   A solution S can now be constructed from the set of MST edges mst as follows:

$$S = mst$$
$$\forall e = (v, w) \in mst : S = S \cup path[base(v), v] \cup path[base(w), w]$$

   Here, $path[base(v), v]$ denotes the unique path which connects v to its nearest terminal. This path can be efficiently retrieved through the stored predecessors $pred(v)$.

4. **Prune Solution:**
   MST-Prune is applied to improve the constructed solution S.

The construction of a Voronoi diagram and the MST takes $O(|E| + |V| \log |V|)$, as does MST-Prune. Extracting the paths is bounded by $O(|E|)$. Thus the overall runtime is $O(|E| + |V| \log |V|)$, which is a vast improvement compared to computing the full distance network.

2.1.2 Shortest Path Heuristic

The shortest path heuristic (SPH) [70] is based on Prim's MST algorithm. Prim's algorithm solves the MST problem by starting with a tree S consisting of a single node. In each step, the tree is extended by the cheapest adjacent edge connecting a node $v \in V \setminus S$. This is repeated until all nodes are part of the tree. This concept can be naturally extended to the STP. Since not every vertex is a terminal, shortest paths as computed by the Dijkstra algorithm can be used to iteratively connect terminals together. We observe that the problem of finding a shortest path between two nodes is a special case of the STP, if there are exactly two terminals.

In its original implementation the runtime is bounded by $O(|T|(|E| + |V| \log |V|))$, since $|T|$ shortest paths have to be computed. Contrary to the DNH, no implementation is currently known which improves the asymptotic worst-case runtime. However, an implementation has been proposed in [3] which manages to reduce the average runtime to be empirically close to the runtime required for solving the MST problem.

The improvement is achieved by exploiting the fact that only a part of the distance labels computed by Dijkstra' algorithm may change when computing a new shortest path to connect the next terminal. Thus most of the distance data can be reused and is only updated if necessary.

Practical experiments have shown that the average solutions quality of SPH is far better than the one of DNH. Therefore the SPH is generally a better choice when using the improved implementation, considering that for both algorithms runtime and approximation factor are comparable.

2.2 Metaheuristics

A metaheuristic is a problem-independent heuristic procedure, which means the algorithmic concept is not restricted to a certain class of optimization problems [58]. This form of independence is achieved through the incorporation of basic problem-specific heuristics into a higher level framework. Such heuristics act as subcomponents which provide information to guide the metaheuristic's search process. The overall objective is to effectively and efficiently traverse the space of feasible solutions.

For $\mathcal{NP}$-hard problems, the application of metaheuristics represents an effective way to handle large-scale problems for which exact methods would be too slow. However, no guarantee is given that an optimal solution is indeed found.

Ideas for metaheuristic approaches can take all sorts of forms. Many are derived from real-world phenomena, like physical processes, biological evolution and animal behavior. However, this form of description through metaphor may also lead to confusion and makes it difficult to recognize common principles [68]. It is thus easier to categorize such metaheuristics according to their common principals. Metaheuristics can be roughly divided into *trajectory-* and *population-based* algorithms [6]. While trajectory-based methods focus on intensifying search by following the best available solution, population-based methods diversify search to cover a larger search area. Although some methods are clearly classifiable into one of these groups, most sophisticated algorithms may incorporate elements from both categories.

In the following we will present a selection of metaheuristic concepts which are relevant in the context of this thesis: Local Search, Variable Neighborhood Descent, the Greedy Randomized Adaptive Search Procedure (GRASP), Path-Relinking and Memetic Algorithms.

2.2.1 Basic Local Search

Local search is a fundamental example of trajectory-based metaheuristics [6]. Given an initial solution, changes are introduced iteratively so that the solution's objective value is improved. Due to its simplicity and effectiveness it is often applied as a subcomponent in more sophisticated metaheuristics.

An abstract view of the procedure is given in Algorithm 1. In each iteration the neighborhood $N(S)$ of the current solution S is explored and an improving solution is chosen. The elements of a neighborhood are defined by the *neighborhood structure*, usually through a transformation rule that specifies how a new solution can be derived from a given solution. The search continues until no improving solution exists in the current neighborhood. The result is a solution which is locally optimal with respect to the chosen neighborhood structure.

There exist different strategies for the exploration of $N(S)$. Common approaches are *first improvement* (select the first found improving solution) and *best improvement* (select the best improving solution in the neighborhood).

Algorithm 1: Basic Local Search

Data: A feasible solution S.
Result: A locally optimal solution S' with respect to the chosen neighborhood structure.

```
1 S' ← S
2 repeat
3 |   S' ← Improve(N(S'))
4 until no improvement possible
```

The concept of local search forms the basis for numerous extended algorithms that follow the same fundamental idea, e.g., Variable Neighborhood Search (VNS), Variable Neighborhood Descent (VND), Large Neighborhood Search (LNS), Guided Local Search (GLS) or Tabu Search (TS) [6].

2.2.2 Variable Neighborhood Descent

A natural extension of the basic local search approach is to use multiple neighborhood structures [6]. The goal is to exploit different characteristics of the search space, so that for a given solution each neighborhood potentially contains improving solutions that do not exist within another neighborhood.

This concept is illustrated by Figure 2.1. For a single neighborhood structure the local search process might get stuck, because no more improving solutions can be found. The search terminates in a so-called *basin of attraction*, which is defined by the set of solutions where the local search stops at the same local optimum. Since the basins of attraction depend on the solution space defined by a neighborhood structure, switching structures might still yield an improvement. This strategy can greatly enhance the achieved improvement, since often more improvement steps are possible.

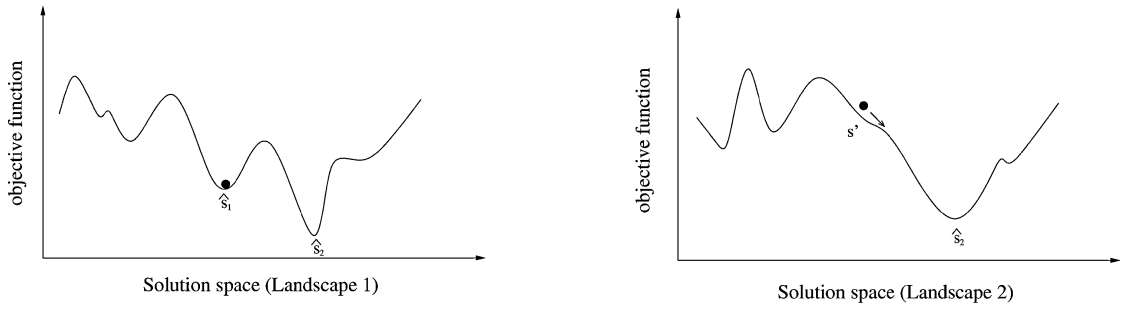


Figure 2.1: Two landscapes defined by different neighborhood structures, illustrating that switching between neighborhoods can lead to better local optima. The figure is borrowed from [6].

A high-level representation of VND is given in Algorithm 2. The procedure uses k_{max} neighborhood structures, and each N_k denotes a different neighborhood. If no improving solution is

found within the current neighborhood, the procedure switches in a deterministic manner to the next one. VND terminates if no improving solution is found in any considered neighborhood structure.

Algorithm 2: Variable Neighborhood Descent

Data: A feasible solution S.
Result: A locally-optimal solution S' with respect to the used neighborhoods.

```
1  S' ← S
2  repeat
3  |  k ← 1
4  |  while k < k_max do
5  |  |  S'' ← Improve(N_k(S'))
6  |  |  if OBJ(S'') < OBJ(S') then
7  |  |  |  S' ← S''
8  |  |  else
9  |  |  |  k ← k + 1
10 |  |  end
11 |  end
12 until no improvement possible
```

The order in which neighborhoods are explored may potentially affect solution quality as well as computation time. This issue has been addressed by Hu and Raidl [32], who propose a VND variant in which neighborhoods are ordered dynamically according to their observed success probability and required runtime. Results indicate that their method can reduce search time while preserving solution quality.

A conceptually similar technique is Variable Neighborhood Search (VNS). In contrast to VND, the basic VNS uses a randomized approach to escape basins of attraction. Each iteration consists of a perturbation and an improvement phase. In the first phase, the solution is perturbed by choosing a neighbor at random from a specified set of neighborhoods, so that the result hopefully lies within a different basin of attraction. In the second phase, the result is improved until a local optimum is reached. The benefit of this approach is that the perturbation introduces additional diversity into the search process. Both VNS and VND provide advantages, and combinations thereof are referred to as general VNS [23].

2.2.3 Greedy Randomized Adaptive Search Procedure

GRASP is a trajectory-based metaheuristic which combines a greedy randomized (semi-greedy) construction heuristic with local search in a multistart approach. The algorithm was initially proposed by Feo and Resende [20] in 1989.

Algorithm 3 depicts the procedure's structure. In each iteration a new solution is constructed and then improved by local search. After the termination conditions are met (usually a specified number of iterations) the best found solution is returned. The semi-greedy construction procedure is essential for the procedure's success. The constructed solutions should be of good quality but at the same time diverse enough to end up in different basins of attraction [51].

Algorithm 3: Greedy Randomized Adaptive Search Procedure (GRASP)

Result: A feasible solution S.

while *termination conditions not met* **do**
  $S \leftarrow$ `ConstructGreedyRandomizedSolution()`
  `ApplyLocalSearch`(S)
  if *OBJ*$(S) <$ *OBJ*(S_{best}) **then**
    $S_{best} \leftarrow S$
  end
end
$S \leftarrow S_{best}$

GRASP's construction procedure incorporates randomization into a dynamic greedy constructive heuristic. A solution is iteratively constructed by adding elements until the solution is feasible. In the case of a purely greedy algorithm, the currently best element is chosen in each iteration. The ranking of elements is defined by a greedy function, which heuristically assigns each element a value based on its perceived benefit. Such an algorithm is considered dynamic if the values of the greedy function are re-computed after each choice.

Algorithm 4: Semi-greedy solution construction

Result: A feasible solution S.

$S \leftarrow \emptyset$
$\alpha \leftarrow$ `DetermineCandidateListLength()`
while *solution not complete* **do**
  $RCL_\alpha \leftarrow$ `GenerateRestrictedCandidateList`(S)
  $x \leftarrow$ `SelectElementAtRandom`(RCL_α)
  $S \leftarrow S \cup \{x\}$
  `UpdateGreedyFunction`(S)
end

GRASP extends this form of solution construction by selecting new elements uniformly at random from a set of best solutions. This set is referred to as restricted candidate list RCL. Multiple approaches on how to build the RCL have been proposed. A simple strategy is to let the parameter α denote the cardinality of RCL. In this case, for $\alpha = 1$ the construction

procedure is equivalent to a deterministic greedy heuristic. In contrast, for $\alpha = n$, where n denotes the number of available elements, the construction is completely random.

The construction procedure is illustrated in Algorithm 4. In its simplest form, α is only chosen at the beginning. In each iteration the RCL is generated by selecting the α best elements. Then an element x is selected at random and added to S. Finally, the ranking of elements is adapted according to the current state of the partial solution [6].

The original GRASP does not specify any kind of long-term memory to retain information between iterations. Several improvements have been proposed that address this issue. A hash table can be applied to remember solutions which have already been constructed, so that unnecessary local search iterations are avoided. Another option is to let the set of already found solutions influence the random selection of new elements. Thus the search is optionally focused on elements which already have been part of many good solutions, or more diversity is introduced by avoiding this kind of elements. Adapting the parameter α has also been considered. In Reactive GRASP, α is changed according to the quality of previous solutions [51].

2.2.4 Path Relinking

In contrast to the approaches presented in the previous sections, Path Relinking belongs to the class of population-based metaheuristics [6, 24]. Such methods are characterized by the fact that in each iteration they operate on a set of solutions (i.e., a *population*) rather than on a single solution. In the case of Path Relinking, new solutions are generated through combination from the population. Combination corresponds to the exploration of solutions lying on the trajectory in the search space defined by two given solutions: the *initiating solution* and the *guiding solution*. The trajectory is explored through iterative transformation of the initiating solution, so that elements from the guiding solution are introduced. The intermediate results are called *trial solutions*, and may be infeasible. In such cases a repair procedure needs to be applied.

Algorithm 5: Path Relinking

Data: An initial population pop.
Result: An improved population pop.

```
1 repeat
2 |  pop' ← SubsetGeneration(pop)
3 |  trial ← SolutionCombination(pop')
4 |  trial ← ImprovementAndRepair(trial)
5 |  pop ← PopulationUpdate(pop, trial)
6 until termination criteria met
```

A high-level representation of Path Relinking is given in Algorithm 5. In each iteration, a subset pop' of the population pop is chosen and its solutions are combined among themselves. The resulting trial solutions $trial$ are repaired and then improved (usually through local search). Finally, the population is updated with the newly constructed solutions. The population size is generally limited, so only a subset of all existing solutions is kept. In addition to solution quality,

the diversity of the new population is also an important criterion for selecting solutions. Keeping the population diverse enough will make it more likely that new areas of the search space are explored during the following iterations.

Since new solutions are only generated through combination, the success of Path Relinking strongly depends on the diversity of the initial population. Exploring trajectories should discover solutions which lie in different basins of attraction. This guarantees that local search will terminate in previously unexplored areas of the search space. GRASP is usually a good choice for generating an initial population, since the constructed solutions are both locally optimal and diverse.

2.2.5 Memetic Algorithms

Memetic algorithms (MA) represent a class of evolutionary algorithms (EA) which is characterized by the intention to exploit all available knowledge about the problem at hand. In contrast to traditional evolutionary approaches, the incorporation of problem domain knowledge is a fundamental characteristic [23].

The term "memetic" is inspired by Richard Dawkins' concept of a meme which represents a unit of cultural evolution that can exhibit local refinement. In the context of MA's, memes refer to strategies used to enhance solutions [44] (e.g., local search, reduction tests, truncated exact methods, approximation and fixed-parameter tractable algorithms, heuristics, specialized recombination operators, etc. [23, 50]).

The notion of designing algorithms that are more focused on exploiting problem-specific knowledge is supported by the *No-Free-Lunch Theorems* for optimization [78]. Their general implication is that a search algorithm that specializes in solving a specific problem will outperform an algorithm for general problems.

Like EAs, MAs are population-based metaheuristics, i.e., a population of solutions is maintained over the course of the search. It is notable that the population may not only include feasible solutions, but also infeasible or partial solution, which can be extended or repaired. The basic structure of a memetic algorithm is that of a generic population-based metaheuristic, which is depicted in Algorithm 6.

Algorithm 6: A population-based search algorithm [23]

```
1 pop ← GenerateInitialPopulation()
2 repeat
3 |   pop′ ← GenerateNewPopulation(pop)
4 |   pop ← UpdatePopulation(pop, pop′)
5 |   if pop has converged then
6 |   |   pop ← RestartPopulation(pop)
7 |   end
8 until termination criteria met
```

In the previous section, Path Relinking has been explored as an example for a population-based method. The purpose of `GenerateInitialPopulation` and `UpdatePopulation` should already be apparent. On the other hand, the `GenerateNewPopulation` step is clearly the core element of a memetic algorithm. In this step, *operators* are applied to the population to generate new solutions. A given operator can be defined for any number of arguments. For example, operators that traditionally appear in EAs are recombination, in which two solutions are combined in some way to generate a set of offspring solutions, and mutation, in which a single solution is altered to introduce diversity. Memetic algorithms extend this idea, so that any problem-specific algorithm can be applied as operator. A popular example is local search, which can be interpreted as a specialized mutation operator that improves a solution.

The `RestartPopulation` step is also of particular importance in a memetic algorithm. Due to the application of problem-specific knowledge to improve solution quality, the population might converge faster to a point in which not enough diversity is available to explore more of the search space. In such a case it makes sense to restart the current population. Generally, some of the best solutions of the old population are kept, and the rest is reseeded as in `GenerateInitialPopulation`.

2.3 Exact Solution

2.3.1 Branch & Bound

Branch-and-bound (B&B) is a universal algorithmic concept which facilitates the design of exact algorithms for various optimization problems. Due to this reason it represents an important tool when attempting to solve $\mathcal{NP}$-hard problems to optimality [12]. The method itself can be considered as an intelligent enumeration scheme, in which the optimality of a solution is guaranteed by excluding the possibility that a better solution can exist anywhere in the search space of all feasible solutions. Normally this would require the enumeration of the whole search space, which is clearly not very efficient. However, in B&B this is avoided through the systematic division of the solution space and the elimination of whole subspaces.

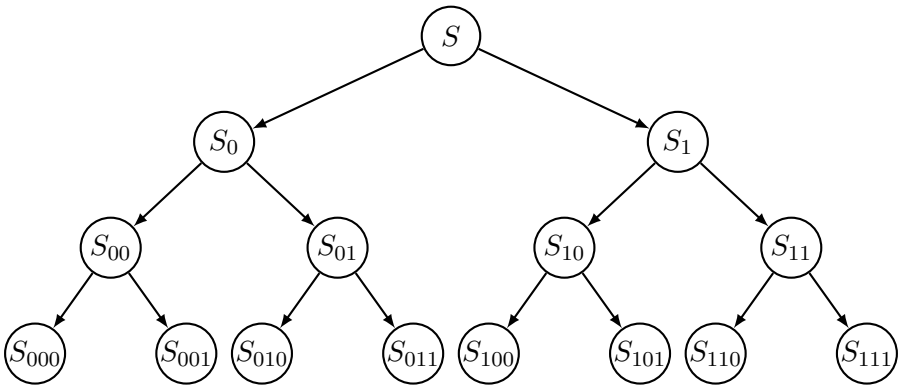


Figure 2.2: A binary B&B search tree.

The current state of the B&B procedure can be represented in the form of a search tree. An example is given in Figure 2.2. Each node corresponds to a different search space. Initially, the tree contains just a single node. Throughout the procedure the search space is split and restricted through the addition and removal of nodes. Each node corresponds to a separate solution subspace. The processing of a single node can be described in three separate operations: *bounding*, *branching* and *pruning*.

1. **Bounding:**
   Given a B&B node, a lower and upper bound to the objective value of the optimal solution is computed for the corresponding solution space. An upper bound can be computed by solving the problem heuristically, as presented in the previous sections.

   A lower bound can be obtained through the solution of a simpler form of the original problem. In this context, simpler means that it is relatively easy to solve the problem optimally. The new problem is referred to as a *relaxation* of the original problem. A problem relaxation has to fulfill the following properties:

   a) $S \subseteq S_r$

   b) $c_r(s) \leq c(s), \forall s \in S$

   The first property states that the solution space S of the original problem has to be contained in the solution space S_r of the relaxed problem. The second property states that for any solution $s \in S$, the objective function value of the relaxed problem c_r is a lower bound of the objective function value of the original problem. If the evaluation function $c_r = c$, then (a) implies (b). But in general, we may also use lower estimators of the original objective function. For a minimization problem, the value of the relaxation provides a valid lower bound of the objective function value.

2. **Branching:**
   The solution space S of a given B&B node v is partitioned into a set of disjunct subspaces $S_0 \ldots S_n$. An additional child is appended to v for each new subspace. A general idea for binary branching in COPs is to choose some element e so that:

   $$\forall s \in S_0 : e \in s$$

   $$\forall s \in S_1 : e \notin s$$

   For all solutions s in the respective solution space, e is either always or never part of s. Examples for e in the context of the STP are a single edge or Steiner node.

3. **Pruning:**
   Pruning prevents a B&B node from being branched, because exploring its solution space cannot contribute to finding a better solution. Let $lb(v)$ and $ub(v)$ denote the lower and upper bound of a B&B node v, respectively, and L the set of leaf nodes in the B&B tree. If the lower bound of the current node v is greater or equal to the best known upper bound,

then clearly no better solution can exist in this space. Hence, a node v can be pruned if the following condition holds:

$$\forall w \in L : lb(v) \geq ub(w)$$

In each iteration of the B&B procedure, a leaf node v is selected from the B&B tree. This can occur in a random manner or optionally through a more complex scheme. Then the bounding operation is executed for v. Based on the computed bounds, v is either pruned or branched. The B&B procedure terminates if no nodes remain for further evaluation, i.e., all available leaf nodes have been pruned. If the problem is feasible, the best obtained upper bound is the optimal solution.

2.3.2 Integer Linear Programming

A large number of COPs can be formulated as an integer linear program (ILP). An ILP optimizes a set of integer variables with respect to a linear objective function and a set of linear constraints. The standard form of an ILP is defined as follows:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \\ & x \in \mathbb{Z} \end{aligned} \tag{2.1}$$

Here x represents a vector of variables which have to be determined to minimize the objective function. The variables x are subjected to a number of linear constraints which are defined by the given constraint matrix A and the right hand side vector b. Vector c represents coefficients related to the cost of the variables. ILPs which are not in standard form can be transformed into this form, e.g., inequalities can be represented as equalities by introducing slack variables.

Solving ILPs is $\mathcal{NP}$-hard in general. The most efficient way to solve an ILP is through B&B. Originally, the technique was introduced in this context [45]. A natural relaxation of an ILP can be achieved by allowing the variables to take non-integer values. Such a relaxation is called a linear program (LP). Contrary to ILPs, an LP can be solved in a relatively efficient way using the *Simplex* method. Although its worst case runtime is exponential, the algorithm is generally much faster in practice. The solution of an LP is a lower bound to the corresponding ILP.

Since a COP has multiple representations as ILP, it is often favorable to select formulations which yield good LP relaxations, i.e., which are close to the optimal value. There exists a wide array of ILP formulations for the STP. For a catalog of formulations the reader is referred to [25, 53]. In this work two well-known formulations are presented: the *multicommodity-flow* formulation and the *directed-cut* formulation. Both formulations can be stated for directed or undirected instances. We will focus on the directed versions, since it can be shown that these may result in stronger relaxations.

A directed formulation can also be applied to the undirected STP. To this end the graph of an undirected STP instance $G = (V, E)$ has to be transformed into an equivalent directed instance $G_D = (V, A)$, where A is the set of arcs created by adding two anti-parallel arcs for each edge $e \in E$. Clearly, in an optimal solution of the ILP, only one arc is selected for each edge, so a directed solution can be transformed back into an undirected solution.

The presented formulations differ mainly in the idea of how the constraints ensure that the created solution is connected, i.e., the solution should take the form of a single tree. In both cases, the binary variables x_{ij} are used to represent that one arc is part of the solution. An arbitrary terminal is selected as the root r of the created arborescence. The principles used in these formulations are not restricted to the STP, but are useful for all kinds of ILPs that model connectivity constraints. The choice of the root node has no effect on the strength of the LP relaxation. For brevity, we use the following notations: Given a set $W \subset V$, we define $\delta^+(W) := \{(i, j) \in A \mid i \in W \wedge j \in V \setminus W\}$ as the set of all arcs with tail inside W and head in its complement. Conversely, $\delta^-(W)$ denotes the set of arcs pointing into W from its complement set. If W contains only a single element v, we write $\delta^+(\{v\})$ as $\delta^+(v)$ and $\delta^-(\{v\})$ as $\delta^-(v)$, respectively. Given a variable vector a and a set of arcs A, $a(A)$ denotes $\sum_{a \in A} a$, i.e., the sum of all variables associated with the arcs in A.

$$\text{(MCF)} \quad \min \quad c^T x \tag{2.2}$$

$$\text{s.t.} \quad f^k(\delta^+(v_i)) - f^k(\delta^-(v_i)) = \begin{cases} 1, & v_i = r \\ -1, & v_i = t \\ 0, & \text{else} \end{cases} \qquad \forall k \in T \setminus \{r\}, v_i \in V \tag{2.3}$$

$$f^k_{ij} \leq x_{ij} \qquad \forall (i, j) \in A, k \in T \setminus \{r\} \tag{2.4}$$

$$f^k_{ij} \geq 0 \qquad \forall (i, j) \in A, k \in T \setminus \{r\} \tag{2.5}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in A \tag{2.6}$$

In the multicommodity flow formulation, a connected graph is enforced through the addition of flow variables $f^k_{ij}, \forall k \in T \setminus \{r\}, \forall (i, j) \in A$. These simulate the flow on which one unit of commodity is sent from the root node to each terminal. The flow itself is represented by the flow conservation constraints (2.3). The constraints (2.4) ensure that for each arc on which flow exists, the corresponding arc variable x_{ij} is also set.

The multicommodity flow formulation represents the flows as independent from each other, thus for each terminal k there exist flow variables for the whole graph. This may result in a high number of variables if the graph contains many terminals, which is a disadvantage of this formulation. Alternatively, the flow for each arc could be aggregated into a single variable. The corresponding formulation is called single-commodity flow. However, it has been shown that this form of representation yields a weaker LP relaxation. Another formulation that has the same strength as (MCF) is the well-known directed cut formulation.

$$
\begin{aligned}
\text{(DCF)}\ \min \quad & c^T x && && (2.7)\\
\text{s.t.} \quad & x(\delta^+(W)) \geq 1 && \forall\, W \subset V, && (2.8)\\
& && r \in W, \\
& && (V \setminus W) \cap T \neq \emptyset \\
& x_{ij} \in \{0,1\} && \forall (i,j) \in A && (2.9)
\end{aligned}
$$

The directed cut formulation ensures a connected graph solely through the constraints (2.8). No additional variables are necessary. For each cut introduced through the set of nodes W, which contains the root terminal, and for which at least one terminal is on the other side of the cut, at least one edge has to be chosen. It can be shown that the LP relaxation of (DCF) is equal to the relaxation of (MCF) [79].

However, there exist exponentially many cuts in a graph, and thus unlike the (MCF) model, (DCF) contains an exponential number of constraints. Therefore the application of the complete formulation is generally infeasible. In practice the (DCF) can still be useful, because rarely all constraints are needed to compute an optimal solution.

2.3.3 Branch & Cut

An effective algorithm for solving an LP formulation with exponentially many constraints is the well-known *cutting-plane method*. For this purpose the LP is initialized without the set of exponential constraints. Then an optimal solution x of this relaxation is computed, which will usually violate some of the not considered constraints. So in each iteration of the cutting-plane method, inequalities are identified which are valid for the original formulation but violated by x. Geometrically, such an inequality defines a hyperplane in $\mathbb{R}^n$, where n is the number of variables, separating x from the polyhedron, hence the name "cutting plane".

The problem of identifying such inequalities is referred to as *separation problem* and a procedure that solves it as *separation method*. The identified inequalities are added to the LP and it is solved again. This process is repeated until no further violated inequalities can be identified by the separation method [27].

The application of the cutting-plane method in the context of a B&B procedure is called branch-and-cut (B&C). The simplified operation of the B&C procedure is described in Algorithm 7.

Algorithm 7: Branch & Cut procedure [41]

```
Initialization
repeat
    Select B&B leaf
    repeat
        Solve LP
        Call separation procedure
    until no violated inequalities
    Prune or branch B&B leaf
until B&B tree is empty
```

2.3.4 Dual Ascent

The dual ascent algorithm for the STP was first proposed by Wong [79] in 1984. In general, a dual ascent procedure aims to compute heuristic solutions to the dual of an LP. A general introduction to LP duality can be found in [15]. In the case of the STP, the dual of the directed multicommodity flow formulation is solved heuristically. In [79], the dual problem is stated as follows:

$$
\begin{aligned}
(\text{DUAL})^{(\text{MCF})} \max \quad & \sum_{k \in T \setminus \{r\}} (v_k^k - v_r^k) && && (2.10)\\
\text{s.t.} \quad & v_j^k - v_i^k - w_{ij}^k \leq 0 && \forall k \in T \setminus \{r\}, \quad \forall (i,j) \in A && (2.11)\\
& \sum_{k \in T \setminus \{r\}} w_{ij}^k \leq c_{ij} && \forall (i,j) \in A && (2.12)\\
& w_{ij}^k \geq 0 && \forall k \in T \setminus \{r\}, \quad \forall (i,j) \in A && (2.13)
\end{aligned}
$$

Variables v_i^k correspond to the flow conservation constraints (2.3) in the original problem. The variables w_{ij}^k correspond to the linking constraints (2.4). The objective function (2.10) maximizes the sum of the variables v_i^k. Since the constraint for the root node is redundant, we can set $v_r^k = 0$ for all $k \in T$, and therefore transform the objective function to $\sum_{k \in T} v_k^k, k \in T \setminus \{r\} \in T$.

In the presented dual ascent procedure, all dual variables are initially set to zero. The goal is to increase the variables $v_k^k, k \in T \setminus \{r\}$, since these are part of the objective function. We note that due to the constraints (2.3) and (2.4), the variables w_{ij}^k limit how much the variables v_i^k and v_j^k are allowed to diverge. Therefore, the feasibility of the current solution can only be guaranteed by increasing multiple variables in each iteration, so that all constraints are satisfied. By the duality of linear programming, any feasible solution to the $(\text{DUAL})^{(\text{MCF})}$ provides a valid lower bound to the optimal (MCF) value. Rather than explicitly keeping track of each dual

variable's current value, the procedure only manages the slack for the constraints (2.11). The dual ascent algorithm operates on the following data:

- Reduced costs $\tilde{c}$:
  The reduced cost vector $\tilde{c}$ represents the slack of the constraints (2.12). Each reduced cost variable $\tilde{c}_{ij} \in \tilde{c}$ is associated to a specific arc $(i, j) \in A$. The slack is initialized with the original arc costs c_{ij} and is decreased in each iteration.

- Auxiliary graph G_A:
  Initially, G_A contains only the nodes V of G. As the algorithm progresses, arcs from G are iteratively added to G_A. An arc $(i, j) \in A$ is added if the corresponding reduced cost $\tilde{c}_{ij}$ variable is decreased to zero. For this reason, the auxiliary graph is sometimes referred to as saturation graph, as it contains only arcs for which the corresponding constraints are saturated.

- Lower bound LB:
  The current value of the objective function of $(\text{DUAL})^{(\text{MCF})}$ is stored in LB. Whenever slack is decreased, LB is updated accordingly to reflect the change in the dual variable values.

Intuitively speaking, the algorithm decreases slack iteratively around the specific terminals, until some termination condition is met. The saturation graph governs in what manner the dual variables are increased, and thus also the objective function. The auxiliary graph G_A is used to ensure that the solution to $(\text{DUAL})^{(\text{MCF})}$ is feasible in each iteration. In the presented dual ascent algorithm the following definitions are essential to describe the operations in G_A:

Definition 2.2 *Strongly Connected Component*
In a directed graph, a set of nodes represents a strongly connected component, if from each node all other nodes of the set can be reached by a directed path.

Definition 2.3 *Dangling node*
A node i is said to dangle from node j if j is reachable from i, but not vice versa.

Definition 2.4 *Root Component*
A root component is a strongly connected component which contains at least one terminal $k \in T \setminus \{r\}$, but no terminal outside of this component dangles from any member.

The concept of root components is used to decide for which terminal $k \in T \setminus \{r\}$ the associated dual variables should be implicitly increased. We note that this is not essential for the feasibility of the dual solution, but influences the quality of the achieved objective value [79].

At each iteration, G_A contains a set of root components. As slack is decreased and G_A grows, root components are eliminated. If no root component exists anymore, this implies that each terminal $k \in T \setminus \{r\}$ can be reached from r, and the objective value stored in LB cannot be increased anymore. Thus the dual ascent algorithm terminates. In Algorithm 8 an abstract presentation of the dual ascent procedure is shown.

Algorithm 8: Dual Ascent

Data: A digraph $G = (V, A, c)$ with $T \subseteq V$.
Result: A lower bound LB.

$LB \leftarrow 0$
$\tilde{c}_{ij} \leftarrow c_{ij}, \forall (i,j) \in A$
$G_A \leftarrow (V, \emptyset)$
while *root component exists in* G_A **do**
  $R \leftarrow$ `ChooseRootComponent()`
  $W \leftarrow W(R)$
  $\Delta \leftarrow \min_{(i,j) \in \delta^-(W)} c_{ij}$
  $\tilde{c}_{ij} \leftarrow \tilde{c}_{ij} - \Delta, \forall (i,j) \in \delta^-(W)$
  $LB \leftarrow LB + \Delta$
end

The algorithm's behavior can be divided into three steps:

1. **Initialization:**
   At the beginning LB is set to zero and the slack is at its maximum, because the algorithm starts at the feasible dual solution in which all dual variables are set to zero. The auxiliary graph initially contains only the nodes V in G, but no edges.

2. **Choice of the root component:**
   The choice of a root component can be important to reach better lower bounds. Generally, choosing a root component can be done arbitrarily. However, there exist more elaborate schemes which may perform better, e.g., a heuristically constructed Steiner tree can be used to guide the choice of the next root component [56].

3. **Slack reduction:**
   The algorithm takes the set of nodes W which are reachable from t in G_A. Then the set of arcs which enters W in G is computed. The slack belonging to those arcs is reduced by the minimal slack left in this set, and the objective value is increased.
   This corresponds to an increase of the dual variables for an arbitrary terminal $t \in R$:

$$\forall i \in W : v_i^t = v_i^t + \Delta$$
$$\forall (i,j) \in \delta^-(W) : w_{ij}^t = w_{ij}^t + \Delta$$

It can be shown that the algorithm actually produces a feasible solution for the dual problem in each step. Here, we will only give an intuitive proof sketch. For an inductive proof, the reader is referred to [79].

Two things have to be made sure: For constraints (2.11), all the variables v are increased when extending a cut. In addition, the variables w of the cut have to be increased by the same

amount, to keep the first constraint satisfied. Since the slack for each arc implies that each variable w can only be decreased for this amount for all terminals, the second constraint also holds throughout the run.

In each iteration, the variables v_i, $i \in R$, of a root component are all increased by the same amount. Thus all constraints (2.11) which belong to nodes $i, j \in R$ hold. For these constraints (2.11), for which only one node is in R, the variables w_{ij}^k are increased by the same amount. Thus these constraints also hold. Due to the constraints (2.12), the reduced costs are initialized by c_{ij}, thus this is the maximum value which the sum in the constraint can effectively reach.

The dual ascent algorithm is not only useful for computing lower bounds, but can also support the construction of higher-quality primal solutions. The basic idea is to apply a heuristic method to a subgraph G_S of G, which is constructed based on the reduced costs. There exist multiple approaches of how to construct this subgraph.

In [79], G_S corresponds to the subgraph induced by the set of nodes V_A in G, where V_A denotes the set of nodes reachable from r in G_A after the termination of dual ascent. For each edge $e = \{v, w\} \in G_S$, $c(e)$ corresponds to the cost of the undirected edge in G that connects v and w. We note that the dual ascent algorithm only terminates after all terminals are reachable from r in G_A, so a feasible solution to the original instance exists in this subgraph.

In a more recent approach presented in [53], G_S is constructed in a way so that it only contains the undirected edges that correspond to edges in G_A with zero reduced costs at the termination of dual ascent. The edge costs are computed as in the previous approach.

We note that in both methods, G_S does not necessarily contain a solution that is also optimal for G. However, G_S can still encourage the construction of better primal solutions than in G, since more information is available due to the use of reduced costs.

2.4 Reduction Techniques

The purpose of reduction techniques is to simplify a given problem, i.e., to transform it into a smaller, yet equivalent problem [72]. Especially for the STP the effectiveness of reduction tests has been widely-acknowledged. Naturally, their application is strongly recommended prior to time-consuming approaches like B&B.

In the context of the STP, reductions may simply take the form of deleting edges and Steiner nodes. More complex tests require the application of transformations to the original graph, e.g., the contraction of nodes or placement of additional edges. If transformations are used, a way for efficiently mapping a solution in the reduced graph to one in the original graph is necessary.

Major classes of reduction techniques are *alternative-based* and *bound-based* tests. The first class infers information about an element through the existence of alternative solutions. If it can be shown that for any solution that an element is part of, there exists a better solution without that element, then the element can clearly be deleted. Conversely, if it can be shown that for any solution the element is part of, there exists no better solution without that element, then the element can be chosen. The second class makes use of lower and upper bounds to evaluate elements.

There exist reduction tests of varying complexity and effectiveness. Some of them are specialized to certain types of graphs, e.g., rectilinear graphs with holes, which appear in VLSI design [72]. In any case, the repeated application of different reduction tests is often beneficial, since the graphs resulting from the application of one test may be susceptible to other tests. Therefore it is generally useful to have a wide selection of tests available [53].

2.4.1 Bound-based Reductions

A simple bound-based reduction test can be constructed by using the information gained from running the previously presented dual ascent algorithm. The test requires a lower bound lb, an upper bound ub and the reduced costs $\tilde{c}_{ij}$. The argumentation presented in this work originates from [53], but the test itself is much older and was already introduced in [16].

Proposition 2.1 *Given a directed network $G = (V, A, c)$ with $\tilde{c}_{ij} \leq c_{ij}$. Let $G' = (V, A, c')$ be a directed network with $c' = c - \tilde{c}$ and let lb' be a lower bound of the costs of any feasible solution in G'. Then for each feasible solution S in G the following inequality holds:*
$lb' + \tilde{c}(S) \leq c(S)$ [53].

The proposition holds, since $c(S) = c'(S) + \tilde{c}(S)$ and $lb' \leq c'(S)$. The lower bound lb' corresponds to the lower bound lb returned by the dual ascent algorithm, and is also a lower bound for the graph G, thus $lb + \tilde{c}(S) \leq c(S)$. Through this proposition, a simple reduction test for Steiner nodes and edges can be devised:

1. **Elimination test for Steiner nodes:**
   Assume a Steiner node v is part of the optimal solution S. We observe that in this case k will be connected by at least two paths in S. The costs of the optimal solution are not known. However, we can bound $\tilde{c}(S)$ from below through the reduced costs of two arc-disjoint paths which connect v: one connecting it to the root r, and the other one connecting it to an arbitrary terminal node. Let $\tilde{p}[a, b]$ describe the costs of a directed path $p[a, b]$ using the cost vector $\tilde{c}$. The costs of the optimal solution can be approximated by an upper bound ub. The Steiner node v and its adjacent edges can be eliminated if the following inequality holds:

$$lb + \tilde{p}[v, r] + \tilde{p}[v, t] > ub$$

2. **Elimination test for edges:**
   An edge $e = \{i, j\}$ can be deleted if the cost of both arcs (i, j) and (j, i) in the directed graph is too high for the bounds. The test can be represented by the following inequality:

$$lb + \min(\tilde{c}_{ij}, \tilde{c}_{ji}) > ub$$

The second test can clearly be executed in $O(|E|)$. The first test can run in $O(|E| + |V| \log |V|)$. For the Steiner node elimination, two executions of Dijkstra are sufficient to compute all necessary data: The first run computes the distances from the root node to all other nodes. The second run starts at the terminal nodes, but traverses the directed graph in reverse. This is essential, since the arcs in a solution of a directed STP instance have to form an arborescence rooted at r. However, the tests require that the dual ascent algorithm has been applied before to generate the essential information. It is also noteworthy that running dual ascent from various roots may result in different reduced costs, and thus eliminate different parts of the graph. Running the tests from multiple roots in a row can therefore increase the test's efficiency [53].

CHAPTER 3

Previous & Related Works

The STP in graphs was originally formulated independently by Hakimi [29] and Levin [46] in 1971. During the last decades the scientific community has dedicated much effort to the design of efficient algorithms. The presently available techniques span several conceptually different areas, including exact methods, heuristic methods, approximation algorithms and reduction tests.

As an exact method, B&B has been applied successfully to solve general problem instances to optimality. Several approaches have been proposed to compute lower bounds. In this context linear programming formulations have been analyzed extensively. Surveys of known formulations can be found in [25, 53]. Koch et al. have shown that the directed-cut formulation can yield successful results when applied in a B&C algorithm [41]. They have also proposed several methods that can speed up the solution process by enhancing the separation procedure.

Alternative methods for bound generation include Lagrangian relaxation and dual ascent. In [5], Lagrangian relaxation is applied to a MST-based problem formulation. In [79], a dual ascent algorithm is proposed which solves the dual problem of the well-known multicommodity-flow formulation heuristically. The algorithm's performance has been improved in [2] through the addition of several dual heuristic techniques.

Some special cases of the STP can be efficiently solved to optimality, e.g., on strongly chordal graphs. A graph is chordal if every cycle of length at least four contains a chord, i.e., an edge that is not part of the cycle but which connects two nodes within the cycle. A graph is strongly chordal if it is chordal and every cycle of even length at least six has an odd chord, i.e., a chord for which the distance between its incident nodes in the cycle is odd. A parallel algorithm is presented in [14] which solves the STP on this type of graphs in $O(\log^2 n)$. Other restrictions to the instance structure may facilitate the application of efficient algorithms by exploiting the problem's fixed parameter tractability. Such an algorithm has been proposed for planar graphs [52]. Another method has been proposed by Polzin and Daneshmand [53], which solves instances efficiently in linear time if the corresponding graph is bounded by a small width parameter. The applied width concept is similar to path-width.

Polzin and Daneshmand [53] propose a more complex framework which combines a wide array of algorithmic techniques, integrated into a B&B procedure. B&B ensures that optimal

solutions are found eventually, however the essential work is done by the algorithms executed within each B&B node. These include an extensive set of reduction tests, a dual ascent heuristic, a repetitive SPH enhanced by heuristic preprocessing and an algorithm for the exact solution of instances restricted by a width-parameter closely resembling path-width. Through the iterative tightening of bounds and strong reduction methods, problems can often be heavily reduced before branching occurs. For a large part of the examined test instances branching is not even necessary at all. To the date of this master's thesis Polzin and Daneshmand have achieved the best available practical results for the STP.

The importance of preprocessing in the form of reduction tests also becomes apparent in other works, since most competitive algorithms are applied only after an initial preprocessing step. Several reduction tests for general instances can be found in [16]. Most instances occurring in real-world applications can be greatly reduced in size. For this reason more challenging benchmark instances have been proposed, which have been artificially constructed to be resistant to known reduction tests [67]. The experiments performed in [41] indicate that these general reduction tests are not very successful at reducing grid graphs that appear in STP instances for VLSI design. This issue has been addressed in [64] and a set of specialized reduction tests for this type of instances has been proposed. An additional set of extended reduction tests has been proposed by Polzin and Daneshmand [54], which examine more complex graph structures for reduction. They are able to remove larger parts of the graph at the cost of a higher time complexity.

Aside from exact methods numerous classic and novel metaheuristic approaches have been evaluated with the goal to compute good feasible solutions within a short time.

Genetic algorithms (GA) have been proposed by Kapsalis et al. [34] and Esbensen [19]. In both approaches solutions are encoded by bitstrings, where each bit denotes whether a corresponding Steiner node is part of the solution or not. The typical GA operators crossover and mutation are used in both algorithms to evolve a population. In the approach by Kapsalis et al. the evolution of infeasible solutions is allowed, but is penalized to favor feasible solutions. In contrast, in Esbensen's approach solution feasibility is preserved in each step. A solution is decoded from the bitstring through application of the DNH. Furthermore, an additional GA operator referred to as "inversion" is added. The algorithm has been designed with the aim to provide robustness with respect to the chosen GA parameters, so that they do not have to be adapted for each problem instance. Esbensen's approach outperforms the GA by Kapsalis et al. with respect to runtime and solution quality.

A parallel GRASP has been applied by [64]. The approach uses a randomized version of Kruskal's MST-algorithm for solution construction. Ribeiro et al. [64] propose an enhanced GRASP approach, referred to as hybrid GRASP with perturbations and path relinking (HGPPR). HGPPR does not use the classic GRASP construction method, since not all MST-based construction heuristics are well suited for randomized construction. Instead, perturbation is introduced by adjusting the edge weights after each iteration. A so-called strategic oscillation approach is applied, which regularly introduces intensification and diversification based on the number of constructed solutions an edge has already been part of. Path relinking is applied to the population that results from the GRASP procedure. Two different relinking strategies are proposed. The first one iteratively evaluates all trial solutions between two given solutions. The second one

only adjusts the edge weights of the original graph, such that edges that are part of two given solutions are preferred in a subsequent heuristic solution.

Tabu search, an extension to the well-known local search, has been the topic of several works [4, 22, 63]. A memory structure called tabu list is used to prevent the search process from revisiting solutions, thus hopefully finding improved solutions. Ribeiro and Souza [63] extend the work of Gendreau et al. [22] through several improvements aimed at speeding up local search: move estimations, elimination tests and neighborhood reduction techniques. Their approach is based on the neighborhood structure defined through Steiner node insertion/elimination.

An algorithm based on the well-known metaheuristic concept of ant colony optimization (ACO) has been applied to the STP in [47]. Multiple generations are performed in which a whole population is constructed through SPH. The greedy choices of the construction heuristic are guided by a long term memory, which provides information on the estimated importance of each Steiner node. Fundamentally, a Steiner node is ranked higher than others if it is part of many good quality solutions. The long term memory is updated using different schemes after each solution construction and generation. According to the performed experiments, the achieved solution quality is comparable to the tabu search proposed in [22]. However, the consumed computation time has been higher. The authors note that none of the well-known improvements for ACO have been considered, e.g., lookahead, local search or backtracking.

A novel metaheuristic called *pilot method* has been proposed by Duin and Voß [17]. The basic algorithm is greedy, but with the extension that a so-called pilot heuristic is called repeatedly to estimate the influence of each possible greedy insertion. In the specific case of the STP, the SPH is used as pilot heuristic. Since a straight-forward application of this algorithm leads to a high-order time complexity, several improvements have been demonstrated that decrease runtime without sacrificing solution quality. Furthermore, the calls to the pilot heuristic within a single iteration are relatively independent and can potentially be executed in parallel.

It is notable that the STP is not fully approximable in polynomial time. It has been proven that computing an approximation closer than $\frac{96}{95}$ times the optimal cost is already $\mathcal{NP}$-complete [11]. Through MST-based construction algorithms one can reach an approximation-ratio of two [3]. An improved rate has been achieved by Zelikovsky et al. [36, 80]. In their contraction based-algorithm different contraction schemes have been devised, which have eventually led to an approximation ratio of 1.55 [65, 66]. The algorithm has also been applied in a practical framework [10]. Until now the best achieved approximation-ratio has been reached by Byrka et al. [8], who propose an LP-based approximation algorithm which achieves an approximation ratio of 1.39.

Components which are present in almost all frameworks are the fast generation of feasible solutions, local search procedures and reduction tests. Accordingly, great effort has been expended into creating efficient implementations for these techniques. For the construction of feasible solutions, MST-based heuristics have been proposed in [3]. Improvements for local search neighborhoods have been introduced in [73]. In [55], several fast implementations for reduction tests have been proposed that perform in $O(|E| \cdot |V| \log |V|)$. Furthermore, in [53] a version of the dual ascent algorithm has been implemented which also achieves $O(|E| \cdot |V| \log |V|)$, but produces slightly worse bounds. For large-scale instances, such implementations are often indispensable.

The concept of partitioning is generally very effective to design algorithms for solving special cases of $\mathcal{NP}$-hard problems or for efficient approximation schemes. A parallel algorithm for solving the STP in chordal graphs [14] has been designed which makes use of the optimal substructure of the problem. Polzin et al. [53] employ partitioning in a reduction technique focused on discovering independent subproblems, i.e., subgraphs which are only connected to the rest of the graph through terminals. Clearly, the solution of such a subgraph does not depend on the rest of the graph.

A heuristic construction algorithm which is quite similar to the one presented in this work was proposed for the Euclidean Steiner tree problem (ESTP) by Kalpakis and Sherman [33] in 1994. The algorithm has also been empirically successful as experiments show [60]. In the ESTP, a given set of terminal points in the plane has to be connected at minimum length using straight lines. Steiner points can be introduced anywhere as additional intersection points for the lines. The proposed algorithm partitions terminals recursively by splitting the current set at the terminal with the median coordinates, until the sets are small enough. The split terminal is added to both adjacent partitions, to ensure that the resulting Steiner tree is connected. The sets of terminals are then connected independently using an exact algorithm. The resulting Steiner tree is improved by computing an MST over the full graph of all terminal and Steiner nodes. After that, further ESTP-specific post-processing heuristics are applied.

During the past years attempts have been made to develop efficient parallel and distributed solvers for $\mathcal{NP}$-hard COPs which can leverage the power of cluster computing. Budiu et al. propose DryadOpt [7], a library for distributed execution of branch-and-bound. For evaluation the system has been applied to the STP. DryadOpt has been implemented on top of DryadLINQ, a technology similar to Hadoop which follows the *Map-Reduce* paradigm. The idea is to divide a computation-intensive task into smaller, completely independent tasks that can be processed in parallel. To maximize utilization of the cluster's commodity hardware using branch-and-bound, problems concerning load balancing, task scheduling and fault-tolerance had to be solved. In their experimental results the obtained speed-up through parallelization has been linear. Furthermore, the known bounds for several previously unsolved instances from the SteinLib could be improved.

We note that practically successful algorithms are rarely based on a single algorithmic concept, but are rather hybrids of multiple algorithmic concepts. This type of combined effort often proves to be very effective at tackling complex problem instances that cannot be handled by applying a single technique alone. However, classifying and relating available approaches can prove rather complex when a high number of different techniques is integrated. Puchinger and Raidl [57] propose a classification for combinations between metaheuristics and exact algorithms, based on how those algorithms interact. They distinguish between collaborative and integrative combinations. A collaborative combination means that algorithms exchange information, but are not part of each other. An integrative combination means that one algorithm acts as a master, which embeds at least one algorithm as a subcomponent. This form of classification is not only useful to classify hybrids between exact and heuristic methods, but also provides a more general perspective for understanding frameworks which combine different algorithmic concepts in some manner.

CHAPTER 4

A Partition-based Construction Heuristic

This chapter presents the implementation and reasoning behind the partition-based construction heuristic (PCH), which is one of the main contributions of this thesis. The proposed algorithm is a hybrid that integrates well-known methods for an exact and heuristic solution of the STP through the application of graph partitioning techniques.

A feasible solution is constructed in four simple steps: *partition*, *decompose*, *solve* and *repair*. In the first step, a heuristic partition of the instance graph is constructed. In the second step, the partition is used to decompose the original instance into several subinstances. In the third step, these subinstances are solved by an exact algorithm. Finally, in the fourth step, the solutions to the subinstances are combined to form a single solution of the original problem, which is repaired heuristically in case of infeasibility. Figure 4.1 depicts each stage of the process when applied to a simple problem instance.

In the context of the proposed algorithm graph partitioning is used as a form of heuristic problem decomposition, a concept that has been applied in the past to compute good feasible solutions to large-scale instances of $\mathcal{NP}$-hard problems which require too much computational effort for current exact methods [28]. This approach has both advantages and disadvantages that affect runtime and solution quality, respectively.

The main advantage and primary goal of decomposition is to decrease the computational effort necessary to find a solution to a given problem instance. The resulting subinstances are smaller than the original instance and thus generally far easier to be solved by exact methods than the original instance. This is clearly the case since known exact methods for $\mathcal{NP}$-hard problems require exponential computation time in the worst case.

We note that size is not the only indicator for instance complexity. Some instance types can be solved to optimality solely through the application of simple reduction tests. In most cases the instance size is considerably decreased. Therefore we assume that preprocessing in the form of reduction tests is applied prior to the PCH. Another considerable benefit of reduction tests is that they tend to make instances more sparse. This is especially important for finding

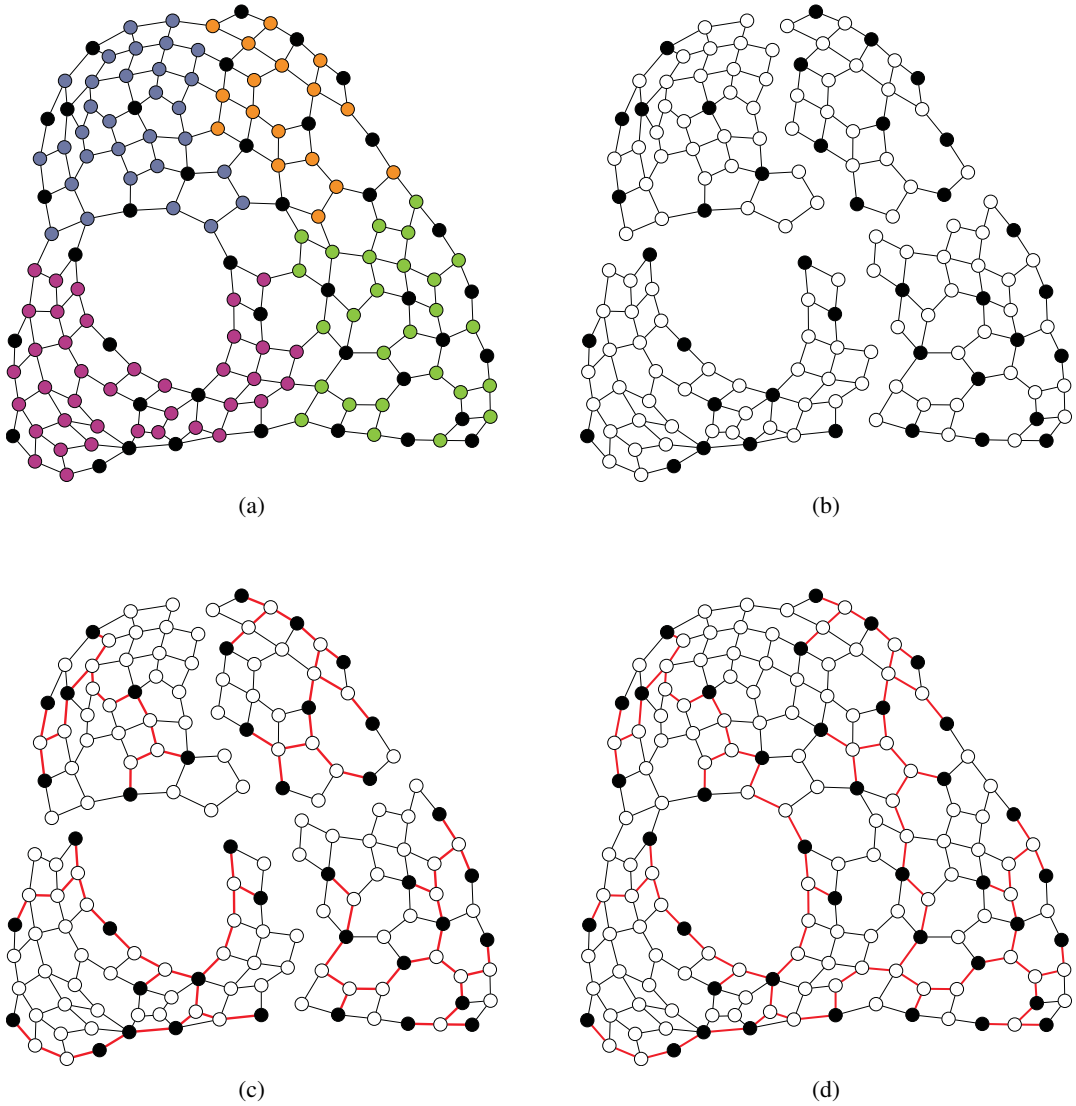


Figure 4.1: PCH solution construction. Figure (a) shows the graph after a partition has been computed. Each partition is marked in a different color. The terminal nodes are colored black. Figure (b) shows the set of subinstances, which is result of the decomposition step. Figure (c) shows the solutions that have been constructed for each subinstances. The red edges indicate computed solutions. Figure (d) shows the solution to the original instance after repair.

a good heuristic partition, which minimizes dependency between subinstances. Minimizing dependency is clearly essential, since otherwise solutions that are optimal within a subinstance may be quite suboptimal when considering the original instance. For a highly connected graph, this task is generally difficult to achieve, which adversely affects solution quality.

In contrast to large instances that can be solved easily through reduction tests, there also exist relatively small instances that are not susceptible to such tests, and in addition are difficult for exact methods. Such instances exhibit a regular, symmetric cost structure, which implies the existence of a large number of solutions with the same objective value. B&B procedures are not well-suited for these methods, and known relaxations yield a large optimality gap. For this type of instances we do not insist on optimal solutions, but rather limit the time available to an exact method to compute a feasible solution.

The potential for designing parallel and distributed algorithms is another area that clearly benefits from decomposition. We note that the exact solutions of the created subinstances are completely independent from each other, so no communication or synchronization is required during the solve-step of the proposed procedure. Therefore the algorithm is also well-suited for execution on a distributed cluster which implements the Map-Reduce paradigm.

Another beneficial aspect of using a heuristic procedure for problem decomposition is that it can be adapted to control the necessary computational effort to construct a feasible solution. In the proposed algorithm this is achieved through limiting the size of the resulting subinstances. Solving a larger number of small subinstances through exact methods will generally take less time than solving fewer larger subinstances. We note that this choice will not only affect computation time, but also solution quality, since the problem decomposition is only of a heuristic nature. Dividing the original instance into more separate problems is likely to decrease solution quality – the inherent price of heuristically decomposing a problem which does not contain truly independent subproblems.

In this context it is important to note that there also exist exact decomposition approaches for the STP, e.g., applying dynamic programming to tree decompositions of graphs. Dynamic programming on tree decompositions yields only efficient algorithms for certain instances that adhere to a low width-parameter with respect to the instance graph. In contrast, the algorithm proposed in this work decomposes a problem heuristically, and is aimed at providing a fast, although heuristic, procedure for general instances. This approach sacrifices the possibility to achieve provable optimality, because there exists no guarantee that the created decomposition yields a solution that is optimal. Finding a partitioning that would yield an optimal solution is $\mathcal{NP}$-hard in general. Nevertheless, it has been shown that heuristic partitioning is an effective approach to design heuristic procedures for other $\mathcal{NP}$-hard problems, e.g., for the graph coloring problem [21].

PCH is defined through the interaction between multiple algorithmic components. Thus we will begin by introducing a high-level framework which defines the algorithm's structure, and shows how the various components are allowed to interact with each other. After this we will examine the components themselves and their respective parameters.

Figure 4.2 gives a graphical representation of the framework. The four steps of the procedure are executed in a strictly sequential manner. The input is an instance of the STP, i.e., an undirected weighted graph and a subset of terminals. The output is a feasible solution, which is

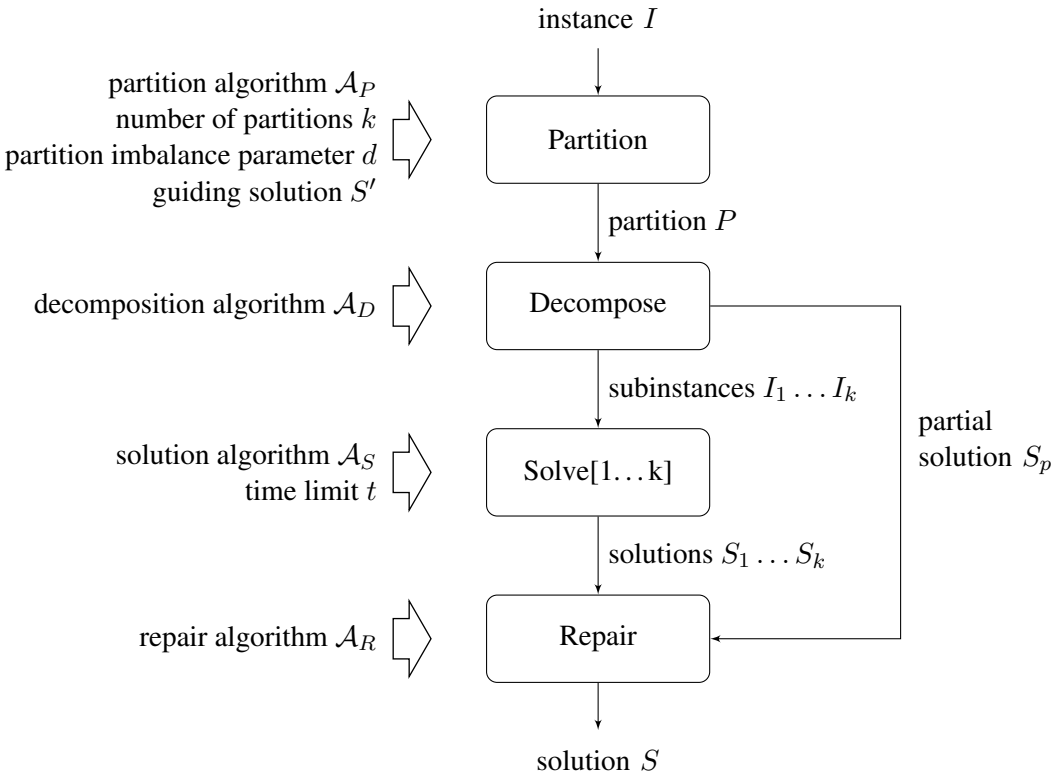


Figure 4.2: A partition-based procedure for heuristic solution construction.

stored as the set of associated edges. Each step is represented by a separate component, which applies a specified algorithm and creates a result that is passed on to the next component. The white arrows on the left enumerate the parameters which are used to influence the behavior of the respective component. The arrows between them represent the information passed on. The purpose of each step and thus each component can be succinctly described as follows:

- **Step 1: Partition**
  Given the preferred number of partitions k and the partition imbalance parameter d, apply the specified partitioning algorithm $\mathcal{A}_P$ to divide the given graph G into a set of k subsets, each containing no more than $d \cdot \frac{|V|}{k}$ nodes. Thus d should reside in the range of $1 \dots k$. Optionally, a guiding solution S_G can be specified, so that partitioning process also attempts to minimize the number of edges from the solution which cross subsets.

- **Step 2: Decompose**
  The original instance I is decomposed into the subinstances $I_1, \dots, I_k$ according to the partition P and the specified decomposition algorithm $\mathcal{A}_D$. Certain decomposition algorithms will produce an additional partial solution S_p that is used to connect subinstances.

- **Step 3: Solve**
  Construct a solution S_i for each subinstance $I_1, \ldots, I_k$. Exact or heuristic methods can be selected through the solution algorithm parameter $\mathcal{A}_S$.

- **Step 4: Repair**
  Use the subinstance solutions $S_1, \ldots, S_k$ and the optional partial solution S_p to build a feasible solution for the original instance I. Based on the previously chosen decomposition algorithm, the solution might still be infeasible, and may need to be repaired according to the chosen repair algorithm $\mathcal{A}_R$.

4.1 Partitioning Algorithms

The partitioning step is clearly a critical part of the proposed procedure, since the chosen partition P will substantially affect solution time of subinstances. However, this is not the only important aspect. Since the subinstance solutions S_i will later be combined into a solution S to the original instance, the question arises whether P can be chosen in a way so that the quality of S is maximized.

The used partitioning algorithms in this work are all based on the same heuristic objective: terminals which are likely to be directly connected in the subgraph that corresponds to optimal solution should be grouped into the same partition. By directly connected we mean that two terminals are connected through a path that does not contain any other terminals in between. Of course, this assumption can only capture a restricted view on how terminals depend on each other.

Two partitioning algorithms which focus on different aspects of the input graph's topology are applied. In the first algorithm, a partition is chosen which minimizes the number of edges between subsets. The second algorithm attempts to minimize the distance between the terminals within a partition. Both algorithms solve their respective problem heuristically.

4.1.1 Edge-based Partitioning

Partitioning the nodes in a graph with respect to the edge-cut between subsets is a well-known COP with many important applications. The edge-cut of a partition refers to the set of edges with endpoints in different subsets. Problems in this category are generally $\mathcal{NP}$-hard and heuristic methods are essential when dealing with large-scale graphs. Luckily, there already exist many publicly available implementations for computing a graph partition efficiently. Exploring their suitability with respect to the task of identifying relatively independent regions within an STP instance is thus a natural first choice. Good performance can be expected for instances in which graphs contain areas which are more sparse than others. An example is the area of VLSI design, in which obstacles on the chip area are represented as holes in the graph. Reduction tests can often help to further intensify such structures.

The problem of partitioning a graph into k roughly equal sized parts based on the edge-cut can be formulated as follows:

Definition 4.1 *k-way graph partitioning problem (kGPP) [37]*
Let $G = (V, E)$ be a graph with node weights w_i, $i \in V$, and edge weights c_{ij}, $e = \{i, j\} \in E$. Given an integer $k > 1$, the goal is to find a partition of V into k disjoint, balanced subsets $V_1, \ldots, V_k$, so that $\bigcup_i V_i = V$ and the weight of edges between different subsets is minimized. By balanced we mean that for each subset $\sum_{i \in V_i} w_i = \sum_{i \in V} w_i / k$.

The kGPP is $\mathcal{NP}$-hard for general graphs. Furthermore, a partition that is exactly balanced might not even exist for the given graph. However, we note that for our purposes an approximate balance is sufficient, since only the occurrence of large subinstances has to be prevented. A relaxed version of the kGPP can be formulated by allowing a certain factor of deviation from the exact balance:

$$\sum_{i \in V_i} w_i \leq \sum_{i \in V} w_i \cdot \frac{d}{k}$$

The partition imbalance parameter d is used to limit the maximum size of each subset in a partition. For $d = 1$, the balance of the resulting partition is close to the exact balance.

To compute a feasible k-way graph partition, the publicly available framework METIS [37] is used. It contains several graph partitioning algorithms, including a multilevel procedure that computes a heuristic k-way partition. This algorithm has been chosen for the partitioning of STP instances.

The operation of the multilevel k-way partitioning procedure can be divided into three phases: *coarsening*, *partitioning* and *uncoarsening*. Figure 4.3 gives a graphical representation of the multilevel partitioning scheme for $k = 2$ as implemented in METIS. The procedure manages to partition a graph in linear time with respect to the number of nodes, since partitioning operations with a higher time-complexity are only applied to a coarsened version of the original graph.

In the coarsening phase, the given graph is transformed into a smaller graph in several passes. In each pass, a subset of adjacent nodes is contracted with each other. Since the goal is to find a partition that minimizes the edge-cut, nodes that are connected by heavy edges should be contracted first. To this end, edges are chosen for contraction in each pass according to a maximal matching. A matching is a set of edges in a graph in which no two of them are incident to the same node. A matching is maximal if each edge not in the matching is incident to a node that is incident to an edge in the matching. The process is repeated until the original graph reaches a certain size.

After that, a k-way partitioning is computed for the coarsened graph using a recursive bisection algorithm. Due to the coarsening, the required runtime is greatly decreased. Since the resulting partition has to fulfill the balance property, the nodes in the coarsened graph reflect the weight of the previously contracted nodes.

In the uncoarsening phase, the coarsened graph is iteratively projected back to the original graph. In each iteration, the contractions introduced in the respective coarsening phase are reverted. The resulting nodes are assigned to the same subset as the node they result from. This approach may reduce the quality of the partition, so a refinement algorithm is used to improve the partition in each iteration. An extended version of the Kernighan-Lin algorithm [39] is used

as a greedy refinement procedure. The algorithm performs several iterations, in which nodes that lie on the boundary between subsets are checked if moving them to another subset can improve the edge-cut's weight.

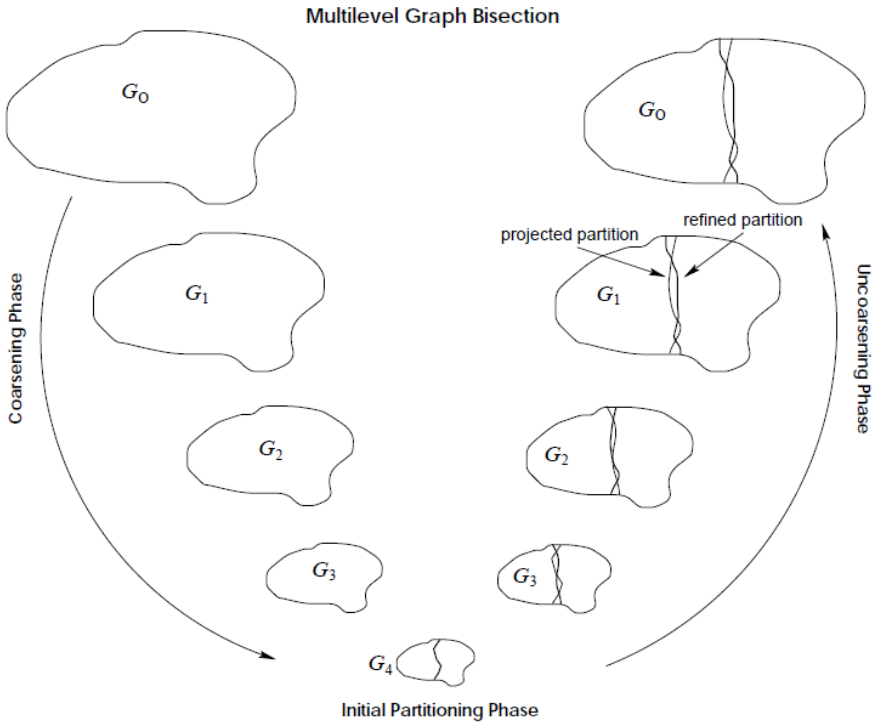


Figure 4.3: The multilevel partitioning scheme. The figure is borrowed from [37].

To successfully achieve the goal of computing a partition that corresponds to roughly independent regions in an STP instance, the algorithm has to be configured specifically for this objective. To this end, edge and node weights are assigned which reflect the specific STP information.

1. **Node Weight**
   Each node $v \in V$ is assigned weight $w_v = 1$ if it is a terminal, and $w_v = 0$ else. Node weights are used solely to balance the complexity of generated subinstances. We note that this weighting scheme only considers terminals for balancing purposes, and ignores Steiner nodes. However, the number of Steiner nodes certainly has an impact an instance's complexity. Preliminary experiments have shown that assigning weight to Steiner nodes is generally not beneficial, since it could potentially result in components that do not contain any terminal nodes.

2. **Edge Weight**
   Two weighting schemes have been considered for edges. Let c' denote the new edge weight used in the partitioning algorithm, and c the weight in the original STP instance (c_{max} corresponds to the weight of the heaviest edge in E).

   a) Uniform: $\forall e \in E : c'(e) = 1$

   b) Original Weight: $\forall e \in E : c'(e) = c_{max} - c_e$

   The uniform weighting scheme minimizes the size of the edge-cut. The second scheme uses the original weight of the problem instance, but inverts it. This is due to the fact that the partitioning algorithm minimizes the weight of the edge-cut. However, for identifying independent components, this weight needs to be maximized.

   Both weighting schemes can incorporate additional heuristic information provided by a *guiding solution*. The weights c' of edges which are part of the guiding solution are scaled by a certain priority factor p to become heavier than regular edges. The goal is to make these edges less likely to be included into an edge-cut, since partition subsets should be computed so that they decompose the guiding solution into subtrees. For our experiments we have chosen $p = 2$.

The number of partitions k and the partition imbalance parameter d can be directly applied to the k-way partitioning algorithm. We note that trying to force balance too aggressively is generally detrimental to solution quality for most practical problem instances. If their structure is not completely regular, the edge-cut will be bigger, because it is sacrificed for more balance. An example showing the different values of d is given in the Figures 4.4 and 4.5.

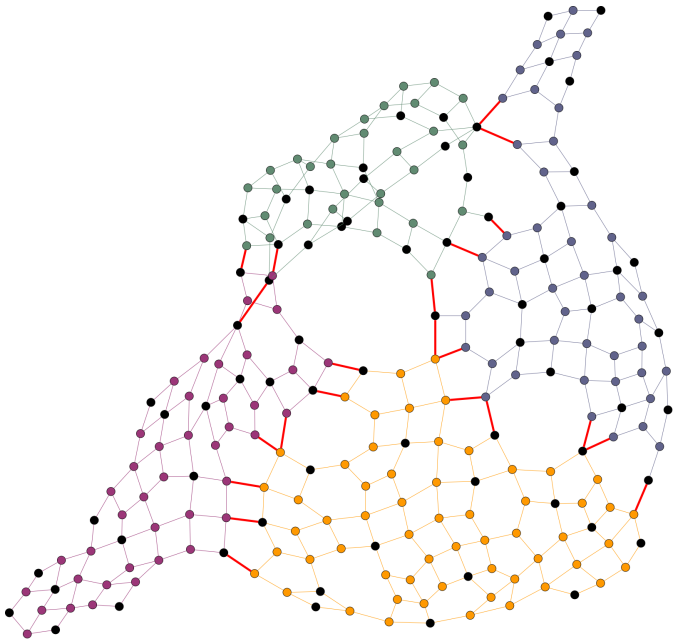


Figure 4.4: The STP instance eil101fst partitioned based on an edge-cut for $k = 4$ and $d = 1.1$. The cut edges are marked in red. A subset's Steiner nodes and edges are presented in color.

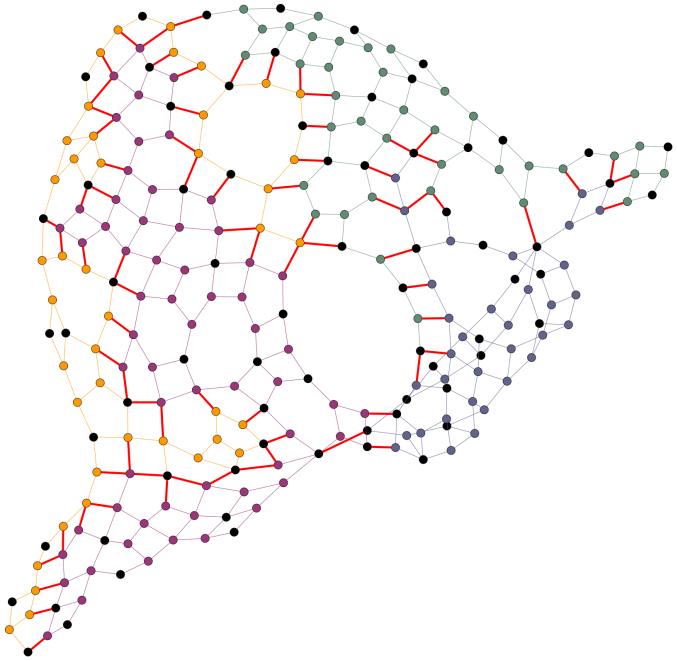


Figure 4.5: The STP instance eil101fst partitioned based on an edge-cut for $k = 4$ and $d = 1.1$. This parameter choice for d clearly shows that without too much freedom in partition imbalance, the algorithm cannot find good partitions.

4.1.2 Voronoi-based Partitioning

Partitioning an STP instance based on the edge-cut as shown in the previous section may not always be a favorable choice for the heuristic identification of independent regions. This holds true especially for instances which do not contain regions that are more sparse than others. Here, the worst-case example is clearly a complete graph. In this case an edge-cut-based partition simply groups together terminals that are adjacent in the corresponding distance network, but no additional information about their independence would be incorporated.

In this section, we consider an alternative partitioning scheme based on the instance graph's Voronoi diagram, induced by the set of terminals (see Section 2.1.1 for a formal definition). Clearly, the Voronoi diagram encodes some kind of dependency information for Steiner nodes, as each Steiner node is assigned to the Voronoi region associated to its closest terminal. Furthermore, the Voronoi diagram already defines an initial partition of the graph, in which each subset contains exactly one terminal node. It is thus balanced with respect to the number of terminal nodes in each subset.

The proposed partitioning algorithm aims to exploit these aspects for the identification of independent regions. To this end a Voronoi region is considered as an atomic unit. Instead of finding a partition on the original instance graph, a coarsened graph is considered, in which each Voronoi region corresponds to a node in the graph. The objective is to find a partition on this coarsened graph which minimizes the distances among nodes within the same subset.

The initial partition defined by the Voronoi diagram serves as a starting point which is iteratively transformed into a partition that complies to the specified partitioning parameters k and d. To this end, the algorithm employs a simple strategy in which regions are merged greedily. In the remainder of this work, we will refer to this algorithm as *Voronoi-based partitioning*.

The procedure is depicted in Algorithm 9. Initially, a Voronoi diagram $\mathcal{V}or$ is built based on the given instance graph G and the set of terminals T. Subsequently, the partition defined by $\mathcal{V}or$ is encoded into an auxiliary graph $G_A = (V_A, E_A, c')$, where each node $v \in V_A$ corresponds to a subset and each edge $e \in E_A$ corresponds to an edge crossing between subsets. For each edge $e = \{v, w\} \in E_A$, the edge weight $c'(e)$ is computed based on the Voronoi diagram as in Mehlhorn's implementation of the DNH (see Section 2.1.1):

$$c'(e) = c_e + \text{dist}(v) + \text{dist}(w)$$

Furthermore, to encode the current state of the constructed partition, for each node $v \in V_A$ the set $p(v)$ contains all nodes that currently belong to the associated subset. The sets $p(v)$ are initialized based on $\mathcal{V}or$ and are thus initially equivalent to the set of Voronoi regions (i.e., sets of nodes which share the same base).

In the main part of the algorithm, the nodes in G_A are contracted greedily to form larger subsets. In each iteration, the node v with the smallest associated subset (i.e., $\min(|p(v)|)$) is chosen for contraction. A priority queue implemented by heap H is used to realize this behavior efficiently. Initially, H contains a node with key $|p(v)|$ for each $v \in V_A$. The node with the minimal key in H is extracted in each iteration.

Let $E_A(v)$ denote the set of edges that are incident to the node $v \in V_A$. After the selection of v, an adjacent node w is chosen so that the weight $c'(e)$ of the connecting edge $e \in E_A(v)$

is minimal. Intuitively, this means that the terminals residing in the associated subsets can be connected by a path of length $c'(e)$.

The selected nodes are only contracted if the size of the resulting subset does not violate the partition balance constraint as specified through d. Two nodes v and w are contracted in G_A by deleting v and associating all edges in $E_A(v)$ to w. Loops are deleted, and for multi-edges only the cheapest edge is kept. The node set $p(w)$ of the remaining node w now also contains $p(v)$. Its priority is updated by reinserting it into H.

The procedure continues until the number of nodes in G_A equals the specified number of partitions k, or if no further nodes exist that can be contracted without violating the balance specified by d. Before termination, a partition vector P is constructed which encodes the final state of the partition by assigning each $v \in V$ an integer value that denotes its subset.

Algorithm 9: Voronoi-based partitioning scheme

Data: An instance $G = (V, E)$, $T \subseteq V$, partition number k, imbalance parameter d.
Result: A partition vector P of G.

```
// Initialization
Vor ← buildVoronoiDiagram(G, T)
G_A = (V_A, E_A, c') ← buildAuxiliaryGraph(Vor)
foreach v ∈ V_A do
    p(v) ← {w ∈ V : Vor.base(w) = v}
    H.insert(|p(v)|, v)
end
// Contraction
while |V_A| > k ∧ ¬H.empty() do
    v ← H.extractMin()
    e ← min_{c'}(E(v))
    w ← G_A.opposite(v)
    if |p(v)| + |p(w)| < d · |V|/k then
        G_A.contract(v, e)
        p(w) ← p(w) ∪ p(v)
        H.insert(|p(w)|, w)
    end
end
P ← buildPartitionVector(p)
```

We note that due to the employed greedy strategy, there exists no guarantee that the resulting partition actually has only k subsets. This becomes especially apparent when a very strict balance is enforced (e.g., $d = 1$). Since Voronoi regions are regarded as atomic, a greedy strategy may end up at a point where all available contraction operations are not allowed due to the balance constraint. In this case the algorithm will terminate with $|V_A| > k$.

The employed greedy approach does not only affect k, but also the partition balance. We note that in contrast to the employed edge-based partitioning scheme, the Voronoi-based partitioning scheme considers Steiner nodes in its balancing constraint. Clearly, this is an advantage, since Steiner nodes contribute to the solution time needed by exact methods.

However, it should be apparent that the Voronoi-based partitioning does not focus on distributing nodes uniformly into subsets. The balancing objective is a subordinate to the goal of minimizing distance between terminals. Only the maximum subset size as defined by d is enforced. We consider this as acceptable, since it is in line with the overarching goal of preventing the generation of subinstances that are too time-consuming for an exact method.

The only aspect that is beneficial to achieving a rough balance between subsets is the priority scheme that is employed to choose the next candidate node for contraction. Always choosing the node v with smallest $|p(v)|$ increases the likelihood that v can be contracted without violating the balance constraint. Thus the number of subsets with low cardinality that remain at termination will most likely be small.

In contrast, a Voronoi region with a very large neighborhood of Steiner nodes is likely to remain uncontracted until the end. This behavior is generally beneficial, since contraction of such nodes will include lots of Steiner nodes and increase subinstance complexity. So we do not want to solve them together, but rather hope that another, high-level method is used to connect them.

We note that the employed greedy strategy might not always yield good results, since the only criterion for the contraction of two subsets is that there exists a cheap path between them. Clearly, this is only a very simplified perspective on how edges in the optimal solution are chosen. If terminals are closer together, they are more likely to be connected, but the distance to other terminals in the graph still plays a role. This kind of global information is not considered in the presented greedy strategy. As in the edge-based partitioning scheme, we propose the use of a heuristic *guiding solution* to improve the results of the Voronoi-based partitioning.

If a guiding solution S is available, the partitioning scheme is adapted as follows: Whenever a node $v \in G_A$ is chosen for contraction, an adjacent edge $e \in E_A(v)$ with minimal costs is selected if the corresponding edge in the original graph is part of S. Thereby, two subsets are only contracted if a path connects them directly in S, i.e., without passing through another subset first.

Proposition 4.1 *A partition can be computed by the Voronoi-based partitioning heuristic in* $O(|T| \cdot |E|)$ *time.*

A Voronoi diagram $\mathcal{V}or$ can be built in $O(|E|+|V| \log |V|)$ [49]. An auxiliary graph G_A can be built in $|E|$ by examining $base(v)$ and $base(w)$ of each edge $e \in E$. We note that initially $|V_A| = |T|$. The sets $p(v)$ for each node $v \in V_A$ can be built in $|V|$ by simply examining $\mathcal{V}or.\texttt{base}(v)$ for each node $v \in V$.

The priority queue is implemented using a binary heap data structure. The number of elements that reside in the heap at any time is bounded by $|T|$. Thus the operation `insert` takes $O(\log |T|)$ time. The operations `extractMin` and `opposite` take always $O(1)$ time.

Each subset is inserted into the heap at most two times – once during initialization and once when another node is contracted to it. The number of contractions is bounded by $|T| - 1$, since

in each iteration the number of elements in the heap is decreased by one. The runtime for finding the minimum cost incident edge of a node and `contract` require both $O(|E|)$ time.

Thus the overall runtime of the loop is $O(|T| \cdot (|E| + \log |T|))$. Keeping track of each subset p can be done efficiently using a *union-find* data structure. A partition vector can be built in $O(|V|)$ time.

4.2 Instance Decomposition

During the decomposition step a set of subinstances $I_0, \ldots, I_k$ is constructed from the original instance $I = (G, T)$, where $G = (V, E, c)$ is the instance graph and $T \subseteq V$ is the set of terminals. The decomposition is performed based on the partition P of G. Without further additions, this procedure is a simple process. The edge-cut defined by P already divides V into a set of disjunct components, so that $V = V_0 \cup \cdots \cup V_k$. Let E_P be the edge-cut defined by partition P. Removing E_P from E separates G into a set of subgraphs $G_0, \ldots, G_k$. Each subgraph $G_i = (V_i, E_i, c)$ and its set of contained terminals $T_i = V_i \cap T$ defines a subinstance $I_i = (G_i, T_i)$.

After the decomposition step, the set of subinstances is solved in the solution step, resulting in a corresponding set of solutions $S_0, \ldots, S_k$. The combination of these solutions forms an infeasible solution S in I, since $I_0, \ldots, I_k$ are disjunct. To make S feasible, additional edges have to be added during the repair step. We note that the described decomposition process can also result in a subinstance which only contains a sole terminal. In such cases the solution to the corresponding subinstance contains no edges, and the terminal is only connected after repair.

A potential improvement to this approach is to augment $I_0, \ldots, I_k$ with additional, heuristic information. The idea is based on the observation that after the previously described simple decomposition procedure, each I_i is solved separately from each other, without any consideration of the neighboring subinstances' structure. Such a situation suggests that the resulting solution S, which results from the connection of $S_0, \ldots, S_k$, will not necessarily be close to the optimal solution of I.

Therefore it can be assumed that additional improvement is possible by adjusting each subinstance, so that their optimal solution takes into account neighboring subinstances that are likely to be connected later on by the repair step. A straight-forward approach to influencing the optimal solution of an instance is to *fix Steiner nodes*. For a given subinstance I_i and a Steiner node $v \in V_i$, fixing v corresponds to adding it to the set of terminal nodes, so that $I_i = (G_i, T_i \cup \{v\})$.

In the following we present a decomposition procedure in which Steiner nodes are fixed based on a heuristically computed partial solution S_p to I. The intuition behind this approach is that the solutions to $I_0, \ldots, I_k$ are solved while already taking S_p into account. To this end, each Steiner node incident to the edges in S_p is fixed in its corresponding subinstance.

A side effect of this approach is that the combination of S_p and $S_0, \ldots, S_k$ forms a connected subgraph which connects all terminals in the original instance, thus resulting in a feasible solution. Therefore, no sophisticated repair procedure is necessary.

In the rest of this work, we will refer to the decomposition strategy without fixation of Steiner nodes as *simple decomposition* and to the other one as *augmented decomposition*.

The implementation of the first approach is straight-forward. The second approach is more sophisticated, since the main goal is to identify a good heuristic connection between subsets. We note that at this point, no information on the solutions of subinstances is available. Algorithm 10 shows the procedure that is used to identify the edges for S_p. A simple greedy heuristic has been implemented which connects partitions. The heuristic is similar to the DNH, but here we generalize a neighborhood into having multiple terminals.

Algorithm 10: Augmented instance decomposition

Data: Original instance $I = (G, T)$ and subinstances $I_0, \ldots, I_k$
Result: A partial solution S_p, augmented subinstances $I_0, \ldots, I_k$

$U \leftarrow$ `initUnionFind`(V)
foreach $I_i = (G_i = (V_i, E_i), T_i) \in I_0, \ldots, I_k$ **do**
  U.`unionSubgraph`(V_i)
  $\mathcal{V}or_i \leftarrow$ `buildVoronoiDiagram`(G_i)
end
$\mathcal{V}or \leftarrow \bigcup_0^k \mathcal{V}or_i$
$E_P \leftarrow E \setminus \{E_1, \ldots, E_k\}$
foreach $e = \{v, w\} : E_P$ **do**
  $c'(e) \leftarrow \mathcal{V}or.dist(v) + c(e) + \mathcal{V}or.dist(w)$
end
$S_p \leftarrow$ `RestrictedMST`(E_C, U, c')
foreach $e = \{v, w\} \in S_p$ **do**
  `fixSteinerNode`(v)
  `fixSteinerNode`(w)
end

For the algorithm, we assume that the simple decomposition has already been executed. Thus the initial parameters are the original graph and the subgraphs.

A union-find data structure (denoted by U) is used to keep track of the node set of each subgraph. We use `unionSubgraph(V)` to denote that the sets corresponding to the nodes in V is merged in the union-find U. The union-find is thus initialized so that the nodes in each subgraph correspond to a different set in U.

First, a Voronoi diagram $\mathcal{V}or$ is constructed on each subgraph G_i. The purpose is to compute the distance for each Steiner node to the nearest terminal in the subgraph. All Voronoi diagrams are then combined to form a single Voronoi diagram for the whole graph.

Subsequently, this information is used to compute alternative weights $c'(e)$ for each edge $e \in E_P$. These weights represent the cost to connect two terminals in neighboring subgraphs. The operation `RestrictedMST(E,U,c)` computes an MST on the given edge set E, such that the sets defined by U are connected, and the cost of the MST is minimized with respect to the provided cost vector c.

Finally, for each edge e, both incident nodes are fixed. The operation $fixSteinerNode(v)$ adds v to the terminals of the corresponding subinstance if necessary (i.e., if it is a Steiner node).

4.3 Solution Repair

The objective of the repair-step is the construction of a feasible solution for the original instance. Based on the chosen decomposition algorithm, the combination of all partial solutions to the subinstances might already form a feasible solution. Otherwise, the resulting solution might be infeasible. For example, a black-box algorithm solves each subinstance I_i, let S_i denote the obtained solution, and let S_p be the set of edges connecting the subinstances with each other. The union of all S_i plus S_p may be feasible, but suboptimal. Even worse, in most cases the union is infeasible, and therefore it is necessary to repair the obtained solution. We have therefore provided two repair algorithms that turn a set of disjoint partial solutions into a feasible solution:

1. **Heuristic Repair**
   The remaining edges are chosen through a simple greedy construction heuristic. The SPH is applied with a randomly selected terminal as root node.

2. **Exact Repair**
   An exact algorithm is applied to identify the optimal connection between partial solutions. The same B&C algorithm is applied which is also used for the solution of subinstances.

Algorithm 11 describes a generic procedure which is applied to repair an infeasible solution. First, all edges corresponding to a solution in the graph are contracted into a single node. Loops are deleted and for multi-edges only the cheapest edge is kept. Each resulting node is marked as a terminal. The method `solve` denotes the application of a heuristic or exact algorithm to the new graph. The union of the resulting solution S_p and the remaining partial solutions form a solution for the original instance S. Subsequently, the improvement procedure MST-Prune is applied to S, since no guarantee can be given that S is optimal.

Algorithm 11: Repair partial solution

Data: An instance $G = (V, E)$, $T \subseteq V$, a disjoint set of partial solutions $S_1 \dots S_k$.
Result: A feasible solution S.

$G_A \leftarrow (V, E, c)$
$T_A \leftarrow \emptyset$
foreach $S_i \in S_1, \dots S_k$ **do**
    $v \leftarrow$ `contractSubgraph`(G_A, S_i)
    $T_A \leftarrow T_A \cup \{v\}$
end
$S_p \leftarrow$ `solve`(G_A, T_A)
$S \leftarrow S_p \cup S_1 \cup \dots \cup S_k$
`MST-Prune`(S)

4.4 Solving Subproblems to Optimality

The algorithm for the exact solution of the STP is based on a branch-and-cut (B&C) approach similar to the one proposed by Koch et al. [41]. It has been chosen in favor of a more sophisticated B&B procedure as proposed by Polzin [53], even though the achieved results by the latter one are significantly better. This decision is based mainly on the fact that Polzin's procedure is rather complex and derives its effectiveness from a large pool of different techniques, which all contribute to the end result in equal parts. In contrast, the implementation of a state-of-the-art B&C procedure for the STP that includes available improvements recommended in literature is rather straight-forward when using the ILOG CPLEX framework. In addition, an even faster exact algorithm would not really have contributed much to our objective, which is the evaluation of the effects of partitioning on solution quality and runtime.

4.4.1 ILP Model

The used ILP model is an extension of the well known directed cut formulation by Wong [79]. In [41], this formulation has been extended through the addition of *flow balance inequalities*, which have originally been proposed by [16]. These constraints can strengthen the LP relaxation and decrease overall solution time of the ILP. In our formulation we add node variables for Steiner nodes, through which a consistent runtime improvement has been achieved in practical experiments.

For each arc $(i, j) \in A$, an arc variable x_{ij} denotes membership of the corresponding arc to the Steiner tree ($x_{ij} = 1$) or not ($x_{ij} = 0$). Similarly, additional node variables y_i for $i \in (V \setminus T)$ denote if i is spanned by the Steiner tree ($y_i = 1$) or not ($y_i = 0$). An arbitrary terminal is chosen as root node r.

$$
\begin{aligned}
\text{(EDCF)} \quad \min \quad & \sum_{(i,j)\in A} c_{ij} \cdot x_{ij} && && (4.1)\\
\text{s.t.} \quad & x(\delta^-(i)) = 1 && \forall i \in T \setminus \{r\} && (4.2)\\
& x(\delta^+(i)) \geq y_i && \forall i \in V \setminus T && (4.3)\\
& x(\delta^-(i)) = y_i && \forall i \in V \setminus T && (4.4)\\
& x(\delta^+(r)) \geq 1 && && (4.5)\\
& x(\delta^-(r)) = 0 && && (4.6)\\
& x(\delta^+(W)) \geq 1 && \forall W \subseteq V,\ r \in W,\ (V \setminus W) \cap T \neq \emptyset && (4.7)\\
& x_{ij} + x_{ji} \leq y_i && \forall (i,j) \in A && (4.8)\\
& x_{ij} \in \{0, 1\} && \forall (i,j) \in A && (4.9)\\
& y_i \in \{0, 1\} && \forall i \in V \setminus T && (4.10)
\end{aligned}
$$

The objective function (4.1) minimizes the weight of the selected arcs. Constraints (4.2) ensure that each terminal except the root has exactly one incoming arc. Constraints (4.3) and (4.4) ensure that Steiner nodes that are part of the solution have only one incoming arc and at least one outgoing arc. Constraints (4.5) and (4.6) ensure that the root terminal r has no

incoming arc and at least one outgoing arc. Constraints (4.7) are the directed cut constraints. Finally, constraints (4.8) ensure that only one arc of each antiparallel pair is selected. We note that introducing node-variables and extending constraints (4.3), (4.4) and (4.8) does not improve the lower bound of the cut model, but provides a computational speed-up if a lot of branching is necessary during solution.

In the following we describe important components of the used B&C procedure:

1. **Initialization:**
   The undirected instance G is transformed into a directed instance. As already stated, the LP is initialized without any cut constraints (4.7) at all. However, the time needed to separate all violated constraints may be decreased by initializing the LP with a set of heuristically separated cuts. We have generated an initial set of cuts through Wong's dual ascent algorithm [79]. The expected runtime saving is as high as the time required by B&C to reach the same lower bound as dual ascent [2, 53].

2. **Primal heuristic:**
   Good upper bounds are clearly important for an exact algorithm like B&B. They can be used to prune nodes from the search tree early and thus decrease runtime and memory requirements. Improved results can be achieved through the application of problem-specific heuristics for the STP.

   We apply SPH combined with MST-Prune to the directed instance in which adapted weights are computed using the assignment of the arc variables in the LP solution:

   $$c'_{ij} = c_{ij} \cdot (1 - \max(x_{ij}, x_{ji})) \qquad \forall \{i, j\} \in E$$

   The heuristic is called after each cutting-plane iteration.

3. **Separation procedure:**
   The separation procedure takes the current LP solution as input and identifies a set of violated inequalities which need to be added to the model. The procedure's concrete operation is described in the next section.

4. **Branching and node selection strategy:**
   The branching strategy should create a roughly balanced B&B tree. When branching on arc variables, this implies that an arc is part of the solution in one branch, and excluded in the other one. Since in most cases the probability that an arc will be part of the solution is lower than not being part of the solution, this results in an unbalanced tree. Furthermore, the LP relaxation may not be influenced much through the exclusion/inclusion of a single edge [13].

   A more effective choice is to branch on Steiner nodes. The inclusion/exclusion of a Steiner node restricts the set of available solutions more strongly, because multiple arc variables may be affected. In addition, the number of possible B&B nodes is smaller for dense graphs. Due to the inclusion of node variables in the model, node branching using CPLEX can be achieved by simply setting a higher branching priority for node variables.

As a B&B node selection strategy we have chosen strong branching [1]. Strong branching evaluates the LP-value resulting from selecting all possible variables and then branching on the one which leads to the best LP relaxation.

4.4.2 Separation

Given a current LP solution $\tilde{x}$ an efficient way to solve the separation problem is to view the directed graph $G_D = (V, A, c)$ of the instance as a flow network. In this network, the arc capacities correspond to the assignment of the arc variables x in the current LP solution. A violated cut inequality can then be identified through the application of a maximum flow algorithm. For this we note that the maximum flow corresponds to the minimum cut in the network (max-flow min-cut theorem). If a violated inequality exists in the flow network, the computed minimum cut will clearly be a violated inequality. In our implementation the *push-relabel maximum flow* algorithm [9] is applied, which runs in $O\left(|V|^2 \cdot \sqrt{|E|}\right)$. Through the application of a dynamic tree data structure the worst-case runtime of the algorithm could be theoretically improved to $O\left(|V| \cdot |E| \cdot \log(|V|^2/|E|)\right)$ [26]. This improvement has not been considered.

The separation procedure is described in Algorithm 12. The procedure `MaxFlow` computes a maximum flow / minimum cut from the root node r to each other terminal $t \in T \setminus \{r\}$. The current LP solution $\tilde{x}$ is used as arc the capacities for G_D. The maximum flow is stored in f. Since the used implementation computes the flow in both directions (from r to t and t to r), two different cut sets are returned, S_r and S_t.

Algorithm 12: Separation procedure

Data: The directed graph $G_D = (V, A)$, $T \subseteq V$, LP variable values $\tilde{x}$.
Result: A set of violated inequalities inserted into the LP.

```
for (i, j) ∈ A do
    x̃_ij = x̃_ij + ε                                  // minimum cardinality cuts
end
repeat
    for t ∈ T \ {r} do
        f = MaxFlow(G_D, x̃, r, t, S_r, S_t)
        if f < 1 − ε then
            Insert the violated cut Σ_{(ij)∈δ+(S_r)} x_ij ≥ 1 into the LP
            x̃_ij = 1   ∀(i, j) ∈ δ+(S_r)
            Insert the violated cut Σ_{(ij)∈δ−(S_t)} x_ij ≥ 1 into the LP     // back cuts
            x̃_ij = 1   ∀(i, j) ∈ δ−(S_t)
        end
    end
until no cuts added                                   // loop for nested cuts
```

A violated cut is identified if the computed maximum flow f is lower than one minus a small value ϵ (in our implementation $\epsilon = 10^{-6}$). The subtraction of ϵ is essential, since due to the numerical inaccuracy of the LP solver, inequalities may be added even though the LP solution is valid. Thus the termination of the row generation process would not be guaranteed.

Three improvements from literature as presented by Koch and Martin [41] have been incorporated into the separation procedure, namely *back cuts*, *nested cuts* and *minimum cardinality cuts*. The objective is to improve the strength and number of the separated cuts.

A higher number of separated inequalities per call of the separation method usually decreases the number of necessary row generation iterations, and thus may decrease overall runtime. Another factor which can influence runtime is the strength of the separated inequalities. If the inequalities are strong, less row generation iterations may be necessary to reach a valid LP solution. The LP model stays more compact and resolving takes less time.

1. **Back Cuts:**
   If the maximum flow is computed in both directions for each terminal, from source to sink and from sink to source, two cuts can be added in each call by the separation method instead of one. The extra cuts can be identified efficiently through an extended implementation of the maximum flow algorithm. As already stated, the used implementation returns the cut set from the root to a terminal in S_r and the cut set from the terminal to the root in S_t.

2. **Nested Cuts:**
   The number of separated inequalities can be further increased through the application of nested cuts. The objective is to add violated inequalities until the Steiner tree corresponding to the LP solution is connected. This can be achieved by setting the LP variable values $x_{ij} = 1$ which correspond to the arcs of the new violated inequality. The maximum flow algorithm is called iteratively until no further violated cuts are found. This approach generally adds many violated inequalities in one step, so fewer row generation iterations are necessary.

3. **Minimum Cardinality Cuts:**
   A way to strengthen the cut inequalities is to ensure that the maximum flow algorithm returns cuts which are not only minimal with respect to the arc capacity, but also to the number of arcs. Clearly, such inequalities restrict the solution space more, since less variables are part of each constraint.

   In an LP solution, many arc variables may be set to zero, so these arcs are normally not considered when computing a minimum cut. To include all arcs, a small value ϵ is added to all arcs. Although this makes computing a minimum cut take longer since the flow network is denser, for the STP it has been shown in practical experiments that the stronger inequalities lead to an overall runtime decrease for most instances.

4.4.3 Application of Bound-based Reductions

An additional improvement to the row generation approach using directed cut constraints is proposed in [13]. The application of the bound-based reduction tests (also described in section 2.4.1) may be executed efficiently in between iterations to fix variables in the LP.

However, we have been unable to include this improvement in our implementation due to the way the ILOG Concert framework handles reduced costs. In the Concert framework, the reduced costs are not accessible during the B&C procedure. They are only accessible after termination. Alternatively, the CPLEX C API provides methods for accessing the reduced costs even during B&C. Nevertheless we have refrained from adapting this approach in our framework. The model which CPLEX uses internally to compute an LP relaxation does not necessarily correspond to the originally provided model. A set of reductions is applied to the model, which makes the available reduced costs incompatible with the original model. Thus the reduced costs cannot be used to execute any reductions.

CHAPTER 5

A Partition-based Memetic Algorithm

In the previous chapter, the partition-based construction heuristic (PCH) has been described as a stand-alone procedure. In this chapter, we present a memetic algorithm in which PCH is applied in combination with several other problem-specific algorithms for the STP. The objective is to evaluate PCH with respect to emergent synergy effects arising from the interaction between multiple algorithmic components. This approach seems especially promising, since many state-of-the-art optimization algorithms achieve impressive results through hybridization. In the remainder of this work we will refer to the new memetic algorithm as MPCH.

The role of PCH in this new procedure slightly deviates from how it has been presented in the previous chapter. Rather than constructing feasible solutions from scratch, PCH is always supplied with already available heuristic information in the form of a guiding solution. Given a population of solutions, PCH can be interpreted as a specialized mutation operator, which introduces new information and potentially enhances a given solution. Its application is also similar to the iterative improvement provided by a local search procedure. In the proposed algorithm PCH is not only applied to a solution once, but several times. The solution produced in the previous iteration is subsequently used as a guiding solution in the next step. This procedure is allowed to continue until no improving solution has been found after a specified number of iterations. In the end, the best found solution replaces the original guiding solution in the population.

PCH is not the only employed problem-specific algorithm. The following methods have been incorporated into the memetic algorithm:

1. **SPH:** construction of a diverse set of starting solutions by the shortest path heuristic followed by MST-Prune (cf. Section 2.1)
2. **DA:** computation of lower bounds using the dual ascent approach (cf. Section 2.3.4)
3. **Bound-based reduction tests:** reduction of the instance size based on reduced costs (cf. Section 2.4.1)

4. **Solution recombination:** construction of new solutions with a specialized recombination algorithm
5. **Solution archive:** already calculated solutions are stored to avoid multiple evaluations of one and the same solution
6. **VND:** improvement of solutions through a Variable Neighborhood Descent algorithm

Algorithm 13 shows the structure of MPCH. Basic population-based parameters are the population size $popmax$ and the maximum number of generations g. The parameter n restricts the number of PCH applications without improvement. The whole procedure returns the best found solution S^* and an associated lower bound lb as an estimate of the solution's quality. In the following, let $\texttt{best}(pop)$ return the best solution of the population pop with respect to the objective value, and let $\texttt{obj}(S)$ denote the objective value of a solution S.

As a population-based approach, the algorithm can be divided into two phases: generation of the *initial population* and the *generation step* process.

The initial population is generated as follows: In each iteration, the dual ascent algorithm (DA) is executed with an arbitrary terminal as root node. The result is a reduced graph G_R (cf. Section 2.3.4), a lower bound lb' and the reduced costs $\tilde{c}$. Subsequently, the shortest path heuristic (SPH) is run in G_R to construct a feasible solution. The result is improved through the application of Variable Neighborhood Descent (VND). This procedure is described in detail in Section 5.3. The resulting solution S is inserted into the population pop. Finally, the currently best lower bound lb and upper bound (i.e., the objective value of the best solution in pop), as well as the reduced costs $\tilde{c}$ from the current iteration are used to apply bound-based reduction tests to the instance I.

After the completion of the population initialization, the following goals will hopefully have been achieved: A population pop of diverse, feasible solutions has been generated, a lower bound lb is available, and the instance graph G has been reduced.

In the main phase, the population is evolved for a fixed number of iterations, also called "generations". Each generation consists of two steps: individual improvement and solution recombination.

In the first step, PCH followed by VND is applied in a multi-start fashion to each solution $S \in pop$. For the first iteration of the multi-start procedure, S is used as a guiding solution for PCH. The guiding solution for each subsequent iteration is the solution from the previous iteration. The multi-start procedure continues until no better solution has been found within the last n iterations. Let S' denote the best obtained solution within this multi-start. After the termination of the multi-start, S' replaces S in the population.

In the second step, the current population is recombined to potentially construct new high quality solutions. An elaborate description of this process is given in Section 5.1.

MPCH employs a solution archive with the purpose to speed up PCH and solution recombination. The objective is to prevent the calculation of the exact solutions of subinstances that have already been solved. Again, an extensive description is given in Section 5.2.

Algorithm 13: Partition-based Memetic Algorithm

Data: Instance $I = (G, T, c)$, population size $popmax$,
maximum number of generations g, number of iterations without improvement n
Result: The best found solution S^* and an associated lower bound lb.

$pop \leftarrow \emptyset$
$lb \leftarrow 0$
for *popmax individuals* **do** `// Initialize population`
  $(lb', G_R, \tilde{c}) \leftarrow \texttt{DA}(I)$
  $I' \leftarrow (G_R, T, c)$
  $S \leftarrow \texttt{SPH}(I')$
  $S \leftarrow \texttt{VND}(I, S)$
  `insertInPopulation`(pop, S)
  $lb \leftarrow \max(lb, lb')$
  `reduce`$(I, \tilde{c}, lb, \text{obj}(\text{best}(pop)))$
end
for *g generations* **do** `// Generation step`
  foreach $S \in pop$ **do**
    $S' \leftarrow S$
    repeat
      $S \leftarrow \texttt{PCH}(I, S)$
      $S \leftarrow \texttt{VND}(I, S)$
      $S' \leftarrow min_{obj}(S', S)$
    until *n iterations without improvement*
    `replaceInPopulation`(pop, S, S')
  end
  $pop \leftarrow$ `recombination`(pop)
end
$S^* \leftarrow \texttt{best}(pop)$

5.1 Solution Recombination

The purpose of the solution recombination step is the generation of improved solutions from the current population. A straight-forward way to achieve this objective is to choose good solution parts from a subset of the population and recombine them into a new solution. If the population is diverse enough and the chosen solutions correspond to different local optima of the search space, the result of the recombination is likely to be a solution of higher quality.

In the context of the proposed algorithm only two solutions are combined at each time. A new solution is constructed through the following procedure:

1. Given two feasible solutions S_1 and S_2, construct a graph G_U that corresponds to the union of the subgraphs defined by the solutions. Clearly, all terminals are reachable in G_U, since S_1 and S_2 are feasible.

2. Solve the STP for G_U through the application of an exact or heuristic algorithm.

In each recombination step, each solution in pop is combined with a randomly chosen other solution. The algorithm ensures that each recombination pair is distinct from each other, i.e., the same pair of solutions is only combined once during a recombination step. The new population consists of the $popmax$ best solutions with respect to the objective value. All additional solutions are discarded.

5.2 Solution Archive

The concept of storing already visited solutions is an approach worth considering whenever the calculation of the objective function / optimal subsolution is an expensive procedure. Especially for population-based methods like EA, the use of a solution archive has been proposed to prevent wasted computation time [31], which is especially important if solution decoding is costly, and to prevent the search from getting stuck if not enough diversity is available.

In MPCH, the solution archive is solely used as a means to save computation time. This is especially important in the case of the PCH, which uses exact solutions of subinstances to construct a feasible solution. Due to the fact that it is applied iteratively, this can be a rather computation-intensive task. Thus the use of a solution archive seems straight-forward. This proposition is further encouraged by the observation that iterative repartitioning is likely to change only some regions of the graph. Some subsets may stay the same. In such cases, we expect the computational load to drastically decrease.

The solution archive is implemented as follows: A hash is computed based on the subinstance graph. The associated optimal solution of the subinstance is stored for further reference. No optimizations have been made according to efficient storage. However, we note that a trie data structure may be used [31, 59].

In addition to improving the runtime of PCH, the solution recombination procedure can also profit. Unions between solutions that already yield similar graphs do not have to be solved exactly a second time.

5.3 Solution Improvement

In this section several neighborhood structures are presented which are subsequently combined in a Variable Neighborhood Descent (VND) procedure. Improved algorithms for fast neighborhood evaluation have been proposed recently by Uchoa and Werneck [73]. The ideas behind the fast neighborhood evaluation are explained and re-implementations are presented.

There exist several neighborhood structures for the STP. We consider the following four well-known neighborhood structures: *Steiner node insertion*, *Steiner node elimination*, *key-path exchange* and *key-node removal*.

The following notations are used to describe the neighborhood evaluation algorithms for a given solution S:

E_S	...	edges part of S
V_S	...	nodes part of S
E_C	...	candidate edges for the construction of a neighboring solution S'
$E(S, v)$	...	the set of edges $\{(v, w) \mid w \in V_S\}$ that connects v to V_S.
K_S	...	the set of key nodes in S (a key node is a Steiner node with a degree of at least three in the solution S)
C_S	...	the set of crucial nodes in S, i.e., $C_S = K_S \cup T$

The given neighborhood structures mainly differ according to how S is represented. In the Steiner node insertion / elimination neighborhoods, S is defined by its nodes V_S. For any given solution S, a solution of equivalent cost can be constructed by computing the MST of the subgraph induced by V_S.

The other two neighborhood structures use *key paths* and *key nodes* to represent S. A key node is a Steiner node in S with degree at least three. For a given solution S, a solution of equivalent cost can be constructed by applying the DNH to the graph, where all *crucial nodes* C_S are interpreted as terminals. A key path is a path which connects two crucial nodes in S, with no other crucial node in between.

We note that the neighborhood exploration methods considered in this work do not correspond to the classic algorithms, but to the improved methods proposed in [73]. These algorithms aim to efficiently explore the whole neighborhood of a solution while introducing multiple improvements. Each algorithm evaluates all solutions in its respective neighborhood in $O(|E| \log |V|)$. We have re-implemented procedures described in [73] and in the following we give a description of our re-implementation.

This runtime bound is achieved through the use of several types of non-trivial data structures (dynamic tree, heap, union-find) and by exploiting the fact that the difference between a solution and one of its neighbors is generally quite small. Hence, to evaluate the cost of a neighbor, it is sufficient to compute the cost of this difference. Some of the presented algorithms depend on evaluating their respective neighborhood in a specific order. This way, some of the information that is necessary to evaluate a given neighbor can be reused from previous evaluations, instead of computing it from scratch.

5.3.1 Steiner Node Insertion

The Steiner node insertion neighborhood structure contains all feasible solutions which can be constructed from an initial solution S through the insertion of a single Steiner node $v \notin V_S$ and the application of the procedure MST-Prune to the subgraph induced by V_S. The re-implemented fast neighborhood evaluation algorithm has been presented originally in [73].

To explain how the improved neighborhood evaluation works, we first consider the naive approach, in which a neighbor is evaluated by computing the MST on the subgraph induced by $V_S \cup \{v\}$, where v is the inserted Steiner node. It has been shown that it is not necessary to

compute the MST on the full induced subgraph, but that it is enough to restrict the computation to $G = (V_S \cup \{v\}, E_S \cup E(S, v))$, see Spira and Pan [69]. The intuition behind this improvement is that the insertion of v does not change the structure of the MST, but instead some of the edges $E(S, v)$ are exchanged with the edges in E_S. A more precise description is given by Tarjan's *red rule* [71], which states that the heaviest edge on any cycle in a graph G cannot belong to the corresponding MST. This rule can be applied to compute the required MST more efficiently, by adding the edges $E(S, v)$ to E_S and removing the heaviest edges on each cycle.

Algorithm 14: Steiner node insertion: neighborhood evaluation

Data: A feasible solution S.
Result: A potentially improved version of S.

```
1  D ← InitializeDynamicTree(S)
2  for v ∉ V_S : |E(S, v)| > 1 do
3      i = 0
4      for e = {v, w} ∈ E(S, v) do
5          if i = 0 then
6              D.link(e)
7          else
8              f ← D.findHeaviestEdge(v, w)
9              if c_f > c_e then
10                 D.cut(f)
11                 D.link(e)
12             end
13         end
14         i ← i + 1;
15     end
16     D.pruneSteinerNodes(E_S)
17     if obj(D) > obj(S) then
18         D.revertChanges(v)
19     end
20 end
21 S ← ConvertToSolution(D)
```

Algorithm 14 presents an abstract representation of the procedure described in [73]. The algorithm makes use of a dynamic tree data structure (denoted by D) to represent the solution S. Using this data structure, the heaviest edge in a cycle can be detected efficiently. At each point in time, the data structure represents all nodes in the graph as a forest of trees. Nodes can be connected or disconnected using the `link` and `cut` operation. Other operations include finding the root of a given node in a tree and finding the heaviest edge on the path from a node to its root node. All operations of D can be performed in $O(\log |V|)$.

At the beginning of the algorithm, the solution S is represented in D by linking all nodes V_S. In the main loop, each node $v \notin V_S$ that is adjacent to V_S is checked for insertion.

As already stated, we attempt to find the heaviest edge in each cycle introduced by adding v to the solution. For a node $v \notin V_S$, an arbitrary edge (v, u) for which $u \in V_S$ is added by linking v and u in D. Afterwards, the addition of a further edge (w, u) would form a cycle. We note however that in a dynamic tree data structure, cycles cannot be present. Therefore, edges are linked iteratively and for each cycle that would be introduced, the heaviest edge is removed before linking the next edge. In Algorithm 14, finding the heaviest edge on a potential cycle is abstracted as `findHeaviestEdge`(v, w) for two given nodes v and w that would be connected through adding the next edge. If the returned edge f is heavier than the edge e that we currently try to add, these edges are exchanged in D. Otherwise, we do not add e and continue.

The operation `findHeaviestEdge`(v, w) actually describes several operations, including the identification of the nearest common ancestor (NCA) for v and w, as well as the identification of the heaviest edge on both paths to their NCA node. The NCA is formally defined as follows:

Definition 5.1 *Nearest Common Ancestor (NCA) [30]*
Given an arborescence S with the root r, the nearest common ancestor of a pair of nodes $v, w \in V$, $v, w \neq r$ denoted by nca(u, v) is the first node that the paths from v to r and w to r have in common.

After all edges from $E(S, v)$ have been checked for insertion, S is pruned in D to remove unnecessary Steiner nodes of degree one. Finally, we check if the new solution is cheaper than the original solution. For brevity, let $obj(D)$ denote the objective value of the solution represented by D. If no improvement has been achieved, the changes in D have to be reverted. To this end, we store all executed link/cut operations, so that the original state of D can be restored efficiently.

After all nodes have been checked, the solution stored in D is converted back into a solution S, represented by an edge set.

In contrast to the other neighborhood evaluation algorithms presented in this work, the Steiner node insertion does not require a specific order of insertion. The algorithms that are presented in the following sections all evaluate their respective neighborhood in post-order for efficiency reasons.

5.3.2 Steiner Node Elimination

The Steiner node elimination neighborhood structure contains all feasible solutions that can be constructed by removing a single Steiner node $v \in V_S$ from the solution and applying MST-Prune to the subgraph induced by V_S. The described fast neighborhood evaluation algorithm has been proposed in [73].

The given solution S is seen as a tree rooted at an arbitrary terminal r. Without loss of generality, we assume that S is a tree which has only terminals at its leaves. All nodes $v \in V_S$ are checked for removal in post-order. The elimination of v splits the tree into the subtrees $S_0, \dots S_k$, where $r \in S_0$ and $S_1, \dots S_k$, are rooted at the children $v_1, \dots, v_k$ of v. To construct

an optimal solution for the given node set $V_S \setminus \{v\}$ it suffices to find an MST that connects the components corresponding to these subtrees. The other edges in E_S remain unchanged.

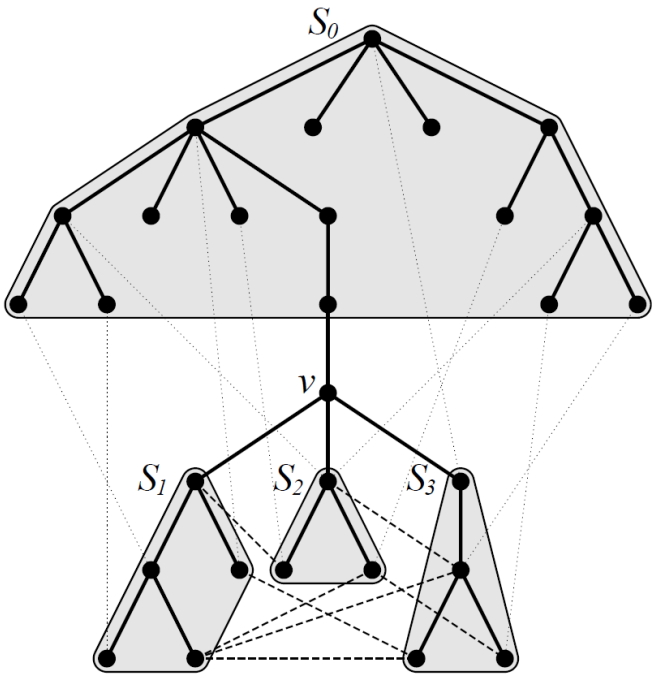


Figure 5.1: A set of subtrees that results from the elimination of node v. A solution edge is marked as bold, while the horizontal and vertical edges are marked as dashed and dotted, respectively. The figure is borrowed from [73].

At this point we note that the number of candidate edges E_C that needs to be evaluated to compute the MST from scratch in each iteration may still be very high (up to $\Theta(|E|)$). This set includes all edges that connect the subtrees $S_0, \ldots S_k$ with each other. However, due to the employed post-order traversal, parts of E_C can be eliminated or pre-built.

To this end, the candidate edges E_C are divided into two groups, relative to the currently removed node v and the resulting subtrees: *vertical edges* and *horizontal edges*. A vertical edge connects the root tree S_0 with a child tree S_i ($i > 0$), while a horizontal edge connects two child trees S_i and S_j ($i > 0$). Figure 5.1 shows an example.

Both the set of horizontal and vertical edges can be built efficiently. We begin with the set of horizontal edges, which are defined succinctly using the definition of the *nearest common ancestor* (NCA). We observe that an edge $e = \{a, b\}$ is horizontal with respect to a given node $v \in V_S$, if v is the NCA of a and b. It is therefore sufficient to compute the NCA for each pair of nodes in V_S which are connected by an edge. The NCAs for all adjacent nodes in S can be built in $O(|E| \log |V|)$ using a dynamic tree which represents the solution S as an arborescence rooted at the terminal r. We note that there also exist algorithms which perform this task in linear time. For each node v, a list $L(v)$ stores all edges $e = \{a, b\}$ for which $nca(a, b) = v$.

Concerning vertical edges, we note that only the cheapest edge for each subtree can potentially be part of an MST which connects the subtrees. These edges are identified as follows: Let $H(v)$ be a heap that stores all edges that are incident to the subtree rooted at v. Cheaper edges are given higher priority.

Algorithm 15: Steiner node elimination: neighborhood evaluation

Data: A feasible solution S.
Result: A potentially improved S.

```
1  L ← computeNCAs(V_S)
2  postT ← postOrderTraversal(S, V_S)
3  U ← initializeUnionFind(V_S)
4  H ← initializeEmptyHeaps(V_S)
5  for v ∈ postT do
6      if v ∉ T then
7          E_C ← L(v)
8          for w ∈ children(v) do
9              while {a, b} = H(v).top() : a ∉ S_0, b ∉ S_i, i > 0 do
10                 H(v).extractMin()
11             end
12             E_C ← E_C ∪ H(v).top()
13         end
14         E_MST ← MST-Components(E_C, U)
15         if c(E(S, v)) > c(E_MST) then
16             E_S ← E_S \ E(S, v) ∪ E_MST
17         end
18     end
19     for w ∈ children(v) do
20         H(v).merge(H(w))
21         U.union(v, w)
22     end
23     for {v, w} ∈ E(S, v) : w ∉ S_i, i > 0 do
24         H(v).insert({v, w})
25     end
26 end
```

All heaps can be built efficiently while traversing the tree in post-order. For this we observe that each heap $H(v)$ contains all edges that are present in the heaps of its child nodes. This is the case, since the subtrees rooted at the child nodes are part of the subtree rooted at v. The heap $H(v)$ is thus a combination of all edges within the children's heaps and $E(S, v)$. An efficient implementation can exploit this fact by using heap data structures which perform merging in

constant time (we use a Fibonacci heap). The cheapest vertical edge can be extracted from $H(v)$ by iteratively calling `extractMin`, until the top element is a vertical edge.

The evaluation of the Steiner node elimination neighborhood using these data structures is shown in Algorithm 15. At the beginning, the list $L(v)$ is built for each node v using a dynamic tree and a post-order traversal is performed for the given solution S. We use a union-find data structure (denoted as U) to efficiently represent the sets of disconnected subtrees.

In the main loop all nodes in V_S are traversed. Thereby, terminals cannot be removed, but the corresponding data structures have to be updated. If the chosen node v is not a terminal, it is considered for removal. To evaluate the resulting neighbor, we initialize the candidate list E_C for computing the MST. All horizontal edges stored in $L(v)$ are added to E_C. Subsequently, the vertical edges are added. Edges are removed from each child heap, until the top element is a vertical edge.

After building E_C, the MST is computed by calling `MST-Components`. This MST algorithm chooses the cheapest set of edges so that each subtree is connected and where v is left out. The union-find data structure is used to represent the subtrees. To check if the new solution is better than the original, we only have to compare the costs of the changed edges, i.e., the cost of the new connection (E_{MST}) and the cost of the old connection ($E(S, v)$).

In any case, the heap and union-find data structures need to be updated. Thereby, the heaps are combined, and the new edges are added. In the union-find data structure, the sets of the children are combined with the node v to form a single set.

5.3.3 Key-path Exchange

The key-path exchange neighborhood structure contains all feasible solutions which can be constructed from an initial solution S through exchanging key paths P_1 by P_2, where $P_1 \in S$ and $P_2 \notin S$. A path is called a key path if its endpoints are both crucial (i.e., terminal or Steiner node with degree at least three in the solution), and no crucial nodes lie in between. The presented algorithm is a re-implementation of the fast neighborhood evaluation which is described in [73].

We note that the removal of an arbitrary key path with endpoints v and w from the solution splits the tree into two disconnected components, denoted by S_v and S_w. The classic way to compute a new solution is to apply the Dijkstra algorithm to find a path between the components. However, this is not very efficient, as it requires $O(|T| \cdot (|E| + |V| \log |V|))$ in the worst case to explore the complete neighborhood. In each iteration, all information is computed from scratch. In contrast, the exploration procedure proposed in [73] exploits the fact that in each iteration, only a part of the distance information generated by Dijkstra's algorithm will change.

In the following, a Voronoi diagram is used to encode the distance information at any point in the algorithm. All nodes in the solution (V_S) are used as the set of bases. In contrast to the previously described algorithms, key-path exchange deals with paths instead of edges. A unique path in the Voronoi diagram is described by a border edge. The border edge can be simply extended into a path based on the predecessor list stored in the Voronoi diagram.

We begin by describing the basic idea behind the presented neighborhood evaluation algorithm: For each key path in the solution S, two Voronoi diagrams exist: one before removing the path, and one after the removal. The second one changes according to the removed path, while

the first one always stays the same. A potentially better path is identified by choosing a boundary edge in the second Voronoi diagram (which corresponds to a unique key path). However, as already stated, constructing a Voronoi diagram in each iteration is costly. A more efficient approach is to only recompute the area of the original Voronoi diagram that changes. A new key path is then selected from two sets of border edges: the *original border edges* that have remained unchanged, and the *new border edges* which appear only after the removal.

Since the set of original border edges will most likely be larger than the new set, it pays off to keep track of it efficiently. Again, heap data structures are used for this task. For each base v in $\mathcal{V}or$, the heap $H(v)$ stores the set of original boundary edges with one endpoint in the subtree rooted at v.

Algorithm 16: Key-path exchange: neighborhood evaluation

Data: A feasible solution S.
Result: A potentially improved S.

$\mathcal{V}or \leftarrow$ `initVoronoiDiagram`(V_S)
$postT \leftarrow$ `postOrderTraversal`(S, C_S)
$H \leftarrow$ `initHeaps`$(\mathcal{V}or)$
$U \leftarrow$ `initUnionFind`(V)
for $v \in postT$ **do**
  $w \leftarrow parentC(v)$
  $I_{vw} \leftarrow P_{vw} \setminus \{v, w\}$
  $\mathcal{V}or$.`removeBases`(I_{vw})
  $e_b^{new} \leftarrow \mathcal{V}or$.`repair`$()$
  while $\{a, b\} = H(v)$.`top`$() : \mathcal{V}or.base(a) \notin S_v, \mathcal{V}or(b) \notin S_w$ **do**
    $H(v)$.`extractMin`$()$
  end
  $e_b^{orig} \leftarrow H(v)$.`top`$()$
  $e_b \leftarrow \min(c(P(e_b^{new})), c(P(e_b^{orig})))$
  if $c(P_{vw}) > c(P(e_b))$ **then**
    $E_S \leftarrow E_S \setminus P_{vw} \cup P(e_b)$
  end
  $\mathcal{V}or$.`revert`$()$
  for $u \in I_{vw}$ **do**
    $H(w)$.`merge`$(H(u))$
    U.`union`(w, u)
  end
  $H(w)$.`merge`(v)
  U.`union`(w, v)
end

Algorithm 16 shows an abstract representation of the neighborhood evaluation procedure. In the beginning, the Voronoi diagram (denoted by $\mathcal{V}or$) is computed, where V_S are considered as bases. Initially, each heap $H(v)$ stores the set of original boundary edges with one endpoint in v's Voronoi region. These heaps will later be merged to represent the boundary edges of whole subtrees. A post-order traversal is built for the set of crucial nodes $C_S \in S$. Again, a union-find data structure is used to keep track of the disconnected components.

In each iteration of the main loop, the solution that results from exchanging key path P_{vw} is evaluated. The end point w is retrieved by a call to $parentC(v)$, which can be computed efficiently while building the post-order traversal. Subsequently, the set of internal nodes I_{vw} of P_{vw} are considered. These nodes have to be removed from the set of bases in $\mathcal{V}or$, so that the influence of P_{vw} is removed from the Voronoi diagram. This also requires a call to $\mathcal{V}or$.`repair`, which recomputes the stored information only for the affected nodes (i.e., the set of nodes which had a base in I_{vw}). During this repair step, new border edges are created in $\mathcal{V}or$. The best new border edge is stored in e_b^{new}.

After this, the boundary edges which do not connect the subtrees S_v and S_w are discarded by iteratively calling `extractMin`. In the end, the best original boundary edge that connects these subsets is stored in e_b^{orig}. Let $P(e)$ denote the corresponding key path of boundary edge e. The path with the lowest cost is chosen from e_b^{orig} and e_b^{new}. If the cost of the new key path is smaller than the removed key path, the current solution is adapted accordingly.

Finally, $\mathcal{V}or$ has to be reverted to its original state, and the data structures are updated. All heaps and sets in the union-find data structure are merged for all nodes on path P_{vw}.

5.3.4 Key-node Elimination

The key-node elimination neighborhood structure contains all feasible solutions which can be constructed from an initial solution S through the removal of a single key node $v \in K_S$ and its incident key paths. We note that this neighborhood also contains all solutions from the Steiner node elimination. A straight-forward way to evaluate each neighbor is to apply the DNH to the remaining set of crucial nodes C_S. The presented algorithm provides a more efficient approach by combining elements from the previously described algorithms for key-path exchange and Steiner node elimination.

First, we observe that the removal of a key node v and its incident key paths from S creates a set of disconnected subtrees $S_0, \ldots, S_k$, where S_0 is the parent component rooted at r. Let $v_1, \ldots, v_k$ be the roots of the subtrees S_i ($i > 0$). The goal is to identify a set of key paths that connect all subtrees at minimal costs.

Again, the concept of updating a Voronoi diagram to reflect the changes after the removal is used for efficiency. Like in the key-path exchange algorithm, we deal with new and original boundary edges. Similar to the Steiner node elimination algorithm, we now consider horizontal and vertical boundary edges. To this end, the already familiar data structures are used, but with some changes.

The list $L(v)$ has already been used to store horizontal edges for the Steiner node elimination, but now we store the original horizontal boundary edges. An original boundary edge is contained in $L(v)$, if its endpoints have v as nearest common ancestor, i.e., $nca(base(u), base(w)) = v$.

Since these boundary edges correspond to the state of the Voronoi diagram before any removal, it can be precomputed. For each $v \in V_S$, a heap $H(v)$ contains all boundary edges with one endpoint in the subtree rooted at v. They are used to identify the cheapest original vertical boundary edge for each subtree. The set of new boundary edges, both vertical and horizontal, are computed during the repair of the Voronoi diagram.

Algorithm 17 shows the neighborhood evaluation procedure. In the beginning, the data structures are initialized. The initialization is exactly the same as in the key-path exchange procedure, with the exception that $L(v)$ is computed for each $v \in V_S$.

In each iteration of the main loop, a neighbor solution is created by the elimination of one key node $v \in K_S$ and subsequently evaluated. Let P_i denote the path that connects v to subtree S_i and let I_i denote the set of internal nodes for each path. Furthermore, let $childrenC(v)$ and $parentC(v)$ denote the set of child nodes and the parent node, respectively, when only crucial nodes are considered in the solution tree. Similarly, $adjacentC(v)$ is the union of $childrenC(v)$ and $parentC(v)$.

At the beginning of an iteration, the internal nodes of I_w, where w is the crucial parent of v, are combined into a set in the union-find data structure. Since C_S is evaluated in post-order, the same operation has already been executed for each I_{v_i}, for all $v_i \in childrenC(v)$.

The next step is to clean data structures $L(v)$ and $H(v)$. In contrast to Steiner node elimination where the full list $L(v)$ is always considered, here the boundary paths could potentially have endpoints that are internal nodes. All such paths have to be removed from $L(v)$.

The heaps $H(v_i)$, where $v_i \in childrenC(v)$ have to be cleaned in a similar manner. Each heap contains boundary edges associated to paths with an endpoint in S_i. The top element of each heap is removed using `extractMin` until the path of the top boundary edge has one endpoint in S_0 and one endpoint in S_i ($i > 0$). This edge is thus the cheapest vertical boundary.

Until now, we have only dealt with the data structures that contain original boundary edges. The missing part are the new boundary edges that are the result of repairing the Voronoi diagram. To this end, the Voronoi diagram $\mathcal{V}or$ needs to be adapted to the state that appears after v and its adjacent key paths are removed from the solution. First, v and the set of internal nodes in I_i for each subtree are removed from the set of bases in $\mathcal{V}or$. Second, $\mathcal{V}or$ is repaired by applying the Dijkstra algorithm to the set nodes of nodes without a base. The set of new boundary edges that appears during this repair process is stored in $R(v)$.

Finally, a set of candidate boundary edges E_C is built as the union of $R(v)$, $L(v)$ and the top of each heap for each child node v_i. The neighboring solution is computed by finding the MST in this set of edges which connects each subtree, represented by the union-find data structure U. Let $P(S, v)$ denote the set of key paths that connect v in S. For brevity, for a set of boundary edges E_b, let $P(E_b)$ be the set of boundary paths. The costs of the removed and the new connecting boundary paths are compared. If the new connection is cheaper, the current solution is adapted.

Before starting the next iteration, $\mathcal{V}or$ has to be reverted to its original state, and the data structures have to be updated along the key paths.

Algorithm 17: Key-node elimination: neighborhood evaluation

Data: A feasible solution S.
Result: A potentially improved S.

```
1  L ← computeNCAs(V_S)                          // Initialize data structures
   Vor ← initVoronoiDiagram(V_S)
2  H ← initHeaps(Vor)
3  postT ← postOrderTraversal(S, C_S)
4  U ← initUnionFind(V)
5  for v ∈ postT do
6      if v ∈ K_S then                                          // Test key node
7          w ← parentC(v)
8          U.union(I_w)
9          foreach e = {a, b} ∈ L(v) : base(a) ∉ I_i, base(b) ∉ I_i, i > 0 do
10             L(v).remove(e)
11         end
12         foreach w ∈ childrenC(v) do
13             while {a, b} = H(w).top() : base(a) ∉ S_i, base(b) ∉ S_0, i > 0 do
14                 H(w).extractMin()
15             end
16         end
17         foreach w ∈ adjacentC(S, v) do Vor.removeBases(I_w)
18         R(v) ← Vor.repair()
19         E_C ← R(v) ∪ L(v)
20         foreach u ∈ childrenC(v) do E_C ← E_C ∪ H(u).top()
21         E_MST ← MST-Components(E_C, U)
22         if c(P(S, v)) > c(P(E_MST)) then
23             E_S ← E_S \ P(S, v) ∪ E_MST
24         end
25         Vor.revert()
26     end
27     for w ∈ childrenC(v) do                          // Update data structures
28         H(v).merge(H(w))
29         U.union(v, w)
30         for u ∈ I_w do
31             H(v).merge(H(u))
32             U.union(v, u)
33         end
34     end
35 end
```

5.3.5 Variable Neighborhood Descent

VND is used as the primary method for solution enhancement within the proposed algorithm. The application of VND to the STP has already yielded good results in other heuristic procedures like GRASP and Path Relinking [48, 64].

Experiments from literature suggest that using multiple neighborhood structures in combination is essential for achieving robustness with respect to solution quality, since their effectiveness is generally influenced by the instance structure. In [64], tests on SteinLib benchmark instances indicate that the Steiner node insertion / elimination neighborhoods are more effective for dense graphs that contain a large number of terminals, while key-path exchange performs better on sparse graphs with fewer terminals.

Furthermore, we note that the Steiner node elimination neighborhood is dominated by the key-node elimination neighborhood, so it is excluded from the VND. We have specified the following order for the exploration of neighborhoods:

1. Steiner node insertion
2. Key-path exchange
3. Key-node elimination

We note that in general the order in which these neighborhoods are evaluated can have an effect on the achieved solution quality. However, in [64] the experiments do not suggest any significant differences for the STP, at least for the tested neighborhoods (Steiner node insertion / elimination, key-path exchange). In our own preliminary experiments, also for the key-node elimination no significant influence has been measured. Thus, in the applied VND we only iterate through neighborhoods in a fixed, deterministic manner.

CHAPTER 6

Computational Results

In this chapter the computational results are presented and analyzed. All algorithms in this work have been implemented in C++ and compiled using GCC 4.8.1 with the full compiler optimization flag (-O4). A collection of pre-existing implementations and software libraries has been incorporated to facilitate the solution of the more complex tasks. Table 6.1 lists each element and its purpose.

Table 6.1: External implementations and libraries

Implementation / library	*Purpose*
ILOG CPLEX 12.5	solution of ILPs through B&C
METIS [38]	computation of a k-way graph partition
dtree [18]	an implementation of link-cut trees used for efficient Steiner node insertion
bossa [76]	preprocessing of STP instances
Push-relabel maximum flow [26]	implementation of the B&C separation procedure

The presented experiments are grouped into four parts. First, a set of suitable benchmark instances is identified through preprocessing. Naturally, only instances which are still large after the application of common reduction tests are of relevance for testing PCH and MPCH, which are both aimed at the solution of large-scale instances. Second, the performance of B&C and the VND procedure is evaluated by themselves. Both are algorithmic components which play a major role within the other algorithms. Third, we experiment with different parameter combinations for PCH and MPCH to identify their best configurations. Finally, we compare the proposed algorithms to state-of-the-art methods.

All tests excluding the comparison with other methods have been executed on a Phenom II X4 965 3.4 GHz with 12 GB RAM. The test runs for the final comparison have been computed

on an Intel Xeon E5540 2.53 GHz with 24 GB RAM. In each case only a single core of the processor is utilized.

Unless noted otherwise, all average values are computed as shifted geometric mean A_{sg}. For a set of n data values $x_1, \dots x_n$, and a given shift value s, this measure is computed as follows:

$$A_{sg} = \sqrt[n]{\prod_{i=1}^{n}(x_i + s)} - s$$

Using the geometric mean instead of the arithmetic mean has the advantage that a small number of extreme outliers does not affect it as much. However, if a single one of the data values is zero, the result is also zero. Therefore, we use $s = 1$, so that no data value can be zero.

In all tests the performance of algorithms is measured in runtime and solution quality. All runtime values are given in seconds. The solution quality of a given solution S is denoted as the percentage gap to the best known or optimal objective value (denoted by OPT), computed as follows:

$$gap = \frac{obj(S) - OPT}{OPT} \cdot 100\%$$

Some algorithms also give a lower bound, which can be used to estimate the quality of the constructed solutions. We measure the gap between the objective value and this lower bound lb in percent, and denote it as dual bound. It is computed as follows:

$$gap_D = \frac{obj(S) - lb}{lb} \cdot 100\%$$

To measure the quality of the lower bound independently from the best upper bound, we also measure the error in percent from the best known or optimal objective value.

$$gap_{DO} = \frac{OPT - lb}{lb} \cdot 100\%$$

6.1 Benchmark Instances

A set of benchmark instances has been chosen from the SteinLib [42]. These instances have been selected according to their sparsity and size. Since the design of PCH does not imply that good results can be achieved for small graphs with a large number of edges, this type of instances has been excluded from the tests. However, the total number of instances which are rather large with respect to the number of nodes (i.e., over 10 000) is relatively small. Since our main focus lies on the efficient construction of heuristic solutions in instances with large graphs, we have included additional real-world instances into our tests to further evaluate the capabilities of PCH and MPCH.

The benchmark instances are grouped into four sets. The first three sets (ES, TSPFST and VLSI) have been selected from the SteinLib, while the fourth (I) is the set of new large-scale real-world instances. In the following we will give a short description of each instance set.

- **Instance set 1: ES**
  The instances belong to the SteinLib sets ES1000FST and ES10000FST. The numbers within them refer to the number of contained terminals, 1 000 and 10 000, respectively. They have been generated from random instances of the Euclidean Steiner tree problem in the plane by the software GeoSteiner [75]. The ratio of terminals to nodes is generally quite high, with ~30% on average.

- **Instance set 2: TSPFST**
  These instances belong to the SteinLib set of the same name, and have also been generated through GeoSteiner. The original instance graphs are taken from the TSPLIB [61], a benchmark library for the traveling salesman problem. The ratio of terminals to nodes is the same as in ES.

- **Instance set 3: VLSI**
  This set contains instances from VLSI applications. The instances correspond to grid graphs with rectangular holes, and are taken from the SteinLib sets TAQ, ALUE and ALUT. In comparison to the other sets, only very few terminals exist in these instance graphs.

- **Instance set 4: I**
  These are the new large-scale real-world instances from infrastructure network design. Compared to the other sets the average instance size is quite high, with the largest instance consisting of over 85 000 nodes, 135 000 edges and 3 900 terminals. The terminal to node ratio is on average lower than in the sets ES and TSPFST, but still higher than in VLSI. Since the instances model the design of real-world infrastructure, the distribution of terminals within the graphs is likely to form groups which are closer to each other than to other terminals. We therefore expect to achieve good results through the application of partitioning.

6.2 Preprocessing

Prior to the application of any other algorithms, common reduction tests have been applied to all the tested instances. The preprocessing has been handled by the bossa framework [64], which implements specialized preprocessing procedures for both general and VLSI instances.

Table 6.2 and Table 6.3 list the results of preprocessing. Each entry contains the post-preprocessing and the original instance properties (numbers of nodes, edges and terminals), the preprocessing runtime in seconds and the percentage of remaining edges (denoted by $E_R\%$). The instances are grouped according to their specific instance set, and sorted in decreasing order according to the number of nodes after preprocessing. The tables are restricted to the instances that have been selected as benchmark instances for the implemented algorithms. For the sets TSPFST, VLSI and I, only the ten instances with the highest number of nodes after preprocessing have been chosen. For the set ES, all instances originally part of the SteinLib sets ES1000FST and ES10000FST have been selected.

We note that preprocessing runtimes are generally quite low, except for large-scale instances. The runtimes for I-instances are especially high, e.g., instance I024 took approximately 5.8 days to finish preprocessing. The high computational effort is most likely due to the iterative application of complex reduction tests to a large number of nodes. It is important to note that the bossa framework is probably primarily optimized for SteinLib instances, which do not contain that many nodes, and that better results could possibly be achieved through optimization of its reduction procedures.

Concerning the effectiveness of preprocessing, we note that for instances with a low number of terminals (i.e., VLSI and I), instance size is reduced heavily, so that ~60% of all edges have been typically removed. The reduction tests are less successful for the ES and TSPFST sets, e.g., the largest TSPFST instance fnl4461fst has still ~82% of its original edges after preprocessing.

Table 6.2: Preprocessing results for the instance set ES

	Instance	$\|V_{orig}\|$	$\|E_{orig}\|$	$\|T_{orig}\|$	$\|V_{prep}\|$	$\|E_{prep}\|$	$\|T_{prep}\|$	*time*[s]	$E_R\%$
ES	es10000fst01	27019	39407	10000	15464	24774	4875	10,33	62.87
	es1000fst01	2865	4267	1000	1833	2958	544	0.22	69.32
	es1000fst02	2629	3793	1000	1443	2317	465	0.2	61.09
	es1000fst03	2762	4047	1000	1609	2583	492	0.22	63.83
	es1000fst04	2778	4083	1000	1596	2569	490	0.2	62.92
	es1000fst05	2676	3894	1000	1584	2543	479	0.14	65.31
	es1000fst06	2815	4162	1000	1693	2741	486	0.19	65.86
	es1000fst07	2604	3756	1000	1453	2306	462	0.18	61.40
	es1000fst08	2834	4207	1000	1742	2824	502	0.17	67.13
	es1000fst09	2846	4187	1000	1743	2801	518	0.18	66.90
	es1000fst10	2546	3620	1000	1334	2109	437	0.14	58.26
	es1000fst11	2763	4038	1000	1638	2630	490	0.21	65.13
	es1000fst12	2984	4484	1000	1943	3126	566	0.22	69.71
	es1000fst13	2532	3615	1000	1235	1983	387	0.19	54.85
	es1000fst14	2840	4200	1000	1785	2864	525	0.2	68.19
	es1000fst15	2733	3997	1000	1597	2570	487	0.15	64.30

Table 6.3: Preprocessing results for the instance sets TSPFST, VLSI and I

	Instance	$\|V_{orig}\|$	$\|E_{orig}\|$	$\|T_{orig}\|$	$\|V_{prep}\|$	$\|E_{prep}\|$	$\|T_{prep}\|$	*time*[s]	$E_R\%$
TSPFST	fnl4461fst	17127	27352	4461	13699	22453	3290	5.87	82.09
	nrw1379fst	5096	8105	1379	3990	6528	988	0.73	80.54
	fl1400fst	2694	4546	1400	1973	3642	967	1.72	80.11
	fl3795fst	4859	6539	3795	1827	3415	895	5.39	52.23
	rat783fst	2397	3715	783	1729	2822	492	0.18	75.96
	rat575fst	1986	3176	575	1557	2566	412	0.12	80.79
	pcb3038fst	5829	7552	3038	1346	2146	532	0.59	28.42
	dsj1000fst	2562	3655	1000	1328	2130	415	0.16	58.28
	vm1748fst	2856	3641	1748	925	1474	332	0.14	40.48
	att532fst	1468	2152	532	904	1456	256	0.05	67.66
VLSI	alue7065	34046	54841	544	13073	23017	445	119.22	41.97
	alut2625	36711	68117	879	12196	22457	764	88.93	32.97
	alut2610	33901	62816	204	10760	20185	182	212.09	32.13
	alue7080	34479	55494	2344	9270	16016	1402	47.96	28.86
	taq0377	6836	11715	136	2040	3603	118	5.4	30.76
	taq0903	6163	10490	130	1851	3294	90	4.95	31.40
	alue7066	6405	10454	16	1791	3151	9	16.44	30.14
	taq0014	6466	11046	128	1766	3155	86	7.23	28.56
	alut2288	9070	16595	68	1224	2195	59	17.46	13.23
	alue5345	5179	8165	68	1191	2012	64	2.75	24.64
I	I027	85085	138888	3954	40772	60555	3490	246569.16	43.60
	I083	89596	148583	4991	34221	50301	4138	77571.68	33.85
	I024	68464	108732	3001	32357	48250	2511	503528.64	44.38
	I064	63158	107345	3458	31712	46711	3182	332814.54	43.51
	I044	68905	113889	3358	31500	46757	2954	122199.74	41.05
	I016	72038	115055	4391	27214	39824	3434	472546.85	34.61
	I002	49920	77871	1665	23800	35758	1282	165874.52	45.92
	I062	66048	110491	3343	23714	35305	2812	9369.44	31.95
	I061	39160	63659	1458	20958	31465	1337	32477.62	49.43
	I015	48833	79987	2493	20573	30541	2119	59274.99	38.18

6.3 Parameter Analysis for the Exact Approach

The B&C procedure is used for the exact solution of subinstances, for exact repair in PCH and the exact solution recombination in MPCH. Since B&C is clearly the most computation-intensive procedure, optimization is essential. In the following we compare the performance of a simple B&C (without any improvements to the separation procedure and without any heuristic separation of cuts through dual ascent) and the B&C with all the improvements presented in Section 4.4.

Table 6.4 shows the average results for each algorithm and instance set. The first column (*# inst*) gives the number of instances per set. The second and third columns (gap_D and gap) are computed as already specified. The fourth column (*time*) gives the average runtime in seconds. The fifth column (*# S*) denotes the number of solved instances and the sixth column (*# nodes*) denotes the average number of B&B nodes. For the sets ES, TSPFST and VLSI, the time avail-

able for exact solution has been limited to three hours. For the large-scale I instances, the time limit has been set to 24 hours.

Table 6.4: Performance comparison for B&C

		simple B&C					*improved B&C*				
	# inst	*gap* [%]	gap_D [%]	*time* [s]	*# S*	*# nodes*	*gap* [%]	gap_D [%]	*time* [s]	*# S*	*# nodes*
ES	16	1.19	13.30	10800	0	644	0.005	0.007	56	15	3
TSPFST	10	1.03	9.06	10800	0	346	0.037	0.065	201	7	11
VLSI	10	1.80	38.84	10800	0	253	0.186	0.187	458	8	1
I	10	0.08	0.87	86000	0	5	0.0	0.0	21907	10	2

The results show that the simple B&C has not solved any instance to optimality within the specified time limit. For all instance sets, gap_D and gap are on average relatively high, especially for the VLSI instances. Concerning the I instances, we note that in their specific case the preprocessing has already fixed many heavy edges, so the resulting gaps appear lower than for the other instance sets. On average, the percentage of fixed costs relative to the known optimum is 78% for the I instances and 10% for the VLSI instances. We note that this does not affect the usefulness of the instance set I for performance comparison purposes.

The improved B&C performs much better than the simple B&C. We observe that both gap_D and gap are quite small (significantly below 1%) after the specified time limit, and most instances have been solved to optimality. In comparison to the simple procedure, the improved B&C also requires much fewer B&B nodes, with most instances solved in the root node. We observe that the average runtime for all instance sets except I is below ten minutes. The results for the I instances are achieved on average after approximately six hours. We conclude that the B&C procedure becomes only effective due to its improvements. Without these, the exact solution takes an unacceptable amount of time.

6.4 Comparison of Local Search Heuristics

Clearly, the algorithms used for local search represent a performance-critical factor. Especially for large-scale instances, their iterative application can potentially require an unacceptable amount of time. The use of efficient neighborhood exploration algorithms is therefore essential. In this section we evaluate whether the performance of the implemented VND is also acceptable for the large-scale instances. For all tests a starting solution is constructed through a single application of SPH from an arbitrary terminal as root node, which subsequently is improved by MST-Prune.

Table 6.5 and 6.6 show the average gap to the optimal value and the average runtime, respectively. The column SPH lists the construction time and quality of the starting solution. First, each single neighorhood is separately explored, so that its potential contribution becomes apparent. The following abbreviations are used in the tables: SN_i and SN_e denote the Steiner node insertion and elimination, KP_e is key-path exchange and KN_e is key-node elimination. The last column (*VND*) lists the results for the whole VND procedure. In this procedure, we

have excluded SN_e, which is dominated by KN_e. Every local search procedure is applied until no further improvements can be found.

Table 6.5: Comparing local search procedures w.r.t. the average gap [%]

	SPH	SN_i	SN_e	KP_e	KN_e	*VND*
ES	1.770	1.366	1.482	1.058	1.310	0.793
TSPFST	1.947	1.552	1.463	1.246	1.414	0.881
VLSI	2.983	2.758	2.694	1.332	2.588	1.229
I	0.064	0.053	0.060	0.025	0.022	0.018

Table 6.6: Comparing local search procedures w.r.t. the average runtime [s]

	SPH	SN_i	SN_e	KP_e	KN_e	*VND*
ES	0.002	0.012	0.015	0.036	0.058	0.075
TSPFST	0.002	0.014	0.017	0.052	0.069	0.114
VLSI	0.003	0.016	0.015	0.059	0.057	0.114
I	0.047	0.307	0.301	2.876	5.672	6.788

First, we note that the solution quality obtained by SPH is always quite robust. For each instance set except VLSI, the average gap is less than 2%. Considering its low runtime, these results are quite acceptable. Even for the large-scale instances, the construction takes less than 50 milliseconds on average.

Subsequently, each neighborhood exploration procedure achieves a substantial improvement in solution quality. Moreover, the results clearly show that some neighborhood structures are better suited for certain types of instances. Especially noteworthy is KP_e, which manages to outperform the other neighborhoods in almost every case. This behavior is easily explained by the fact that all tested instance sets consist of large, sparse graphs, in which the insertion or elimination of single Steiner nodes does not make much difference. The performance discrepancy is especially noticeable for the VLSI-instances, which contain the smallest percentage of terminals, so key paths tend to be rather long. In contrast, the achieved improvement for SN_i and SN_e is similar to KP_e when applied to instances with many terminals, i.e., ES and TSPFST. Compared to the other neighborhoods, their exploration is also faster.

As expected, for all tested instances KN_e achieves better results than SN_e. For the instance sets ES, TSPFST and VLSI the difference in runtime is not very pronounced, with KN_e requiring only a few milliseconds more than SN_e. The runtimes of KP_e and KN_e are comparably high for the large-scale instances. In any case, KN_e achieves a higher improvement than other procedures, which implies that its use in the VND is justified.

The evaluation of single neighborhoods already proves the effectiveness of the implemented local search techniques. The results suggest that for each instance set, their combination in a VND is able to further enhance the achieved improvement. In most cases, the introduced additional runtime is negligible, as it is relatively close to the performance of the slowest neighborhood exploration technique (in this case KN_e). We therefore conclude that it is more effective and robust to apply the VND instead of only exploring a single neighborhood.

6.5 Tuning the Partition-based Construction Heuristic

In this work a number of different algorithms have been suggested that attempt to solve the subtasks of PCH. An interesting question is if there exist algorithms that clearly outperform others in their respective task, or if some of them are more suited for certain instance types. In the following, we evaluate the presented algorithms for partitioning, decomposition and repair in the context of PCH. Their performance is measured by comparing solution quality and runtime.

As there exists a large number of algorithm combinations that need to be evaluated, we set the time limit for each exact solution to $t = 100$ (seconds). This ensures that the runtime of PCH is more predictable, since in the worst case the exact solution of a subinstance may require a large amount of time. In our experiments, most subinstances can be solved before this time limit, and only certain special cases require more time.

Moreover, each algorithm combination is evaluated for certain configurations with respect to the partitioning parameters k (the number of subsets in the partition) and d (allowed imbalance in the partition P, such that $|V_i| \leq |V| \cdot d/k$ for each subset $i \in P$), which represent a trade-off between solution quality and runtime. We note that all tested instances are of different sizes, so k is always computed according to the number of terminals: $k = |T|/10$ and $k = |T|/100$ has been considered. The results are grouped accordingly. We have chosen terminals as the basis for computing k instead of the number of nodes, because the terminal to node ratio is also different for each test instance.

6.5.1 Evaluation of Partitioning Schemes

We first evaluate the proposed *partitioning algorithms*. For comparison purposes, each algorithm is applied together with the *simple decomposition* and *heuristic repair* algorithm, since these are expected to require the lowest runtime. This choice does not have any impact on the performance of partitioning, since the application of these techniques is independent from each other.

Table 6.7 and 6.8 show solution quality and runtime for the different partitioning schemes. Each column contains results for a different partitioning scheme and the given parameters k and d. The following abbreviations are used: Eb and Vb denote the edge-based and Voronoi-based partitioning schemes, c_u and c_{orig} are the uniform and original weighting strategies for Eb, while S_G denotes the use of a guiding solution. This guiding solution is constructed by applying SPH and MST-Prune in the reduced graph generated by dual ascent (cf. Section 2.3.4). In Table 6.7, the smallest average gap per instance set and partitioning scheme has been marked bold, while in Table 6.8 the configuration with the smallest runtime is marked.

First, we note that the results confirm that the parameters k and d affect PCH as intended. In most cases, a clear progression is visible concerning runtime and solution quality when the values of $|T|/k$ and d are increased. The creation of larger, imbalanced subinstances leads to an improved solution quality, but increases runtime, since the created subinstances take longer to be solved.

As was have already noted in Section 4.1.1, the solution quality for Eb is quite bad if $d = 1$. In addition, $d = 3$ also affects quality adversely, which is frequently worse than for $d = 2$. We conclude that the performance of Eb highly depends on its partitioning parameters, and requires them to be fine-tuned to achieve its full potential. For all instance sets, using c_{orig} outperforms

Table 6.7: Comparing partitioning schemes in PCH w.r.t. the average gap [%]

			Edge-based partitioning (METIS)				*New Voronoi-based partitioning*	
	$\|T\|/k$	d	c_u	c_{orig}	c_u, S_G	c_{orig}, S_G	*w/o* S_G	*w.* S_G
ES	10	1	8.88	9.13	8.57	8.13	1.54	0.98
		2	1.99	1.43	1.37	1.05	1.03	0.69
		3	1.82	1.24	1.29	0.94	0.97	0.63
	100	1	3.41	3.90	1.35	1.53	0.37	0.18
		2	0.67	0.64	0.58	**0.42**	0.22	0.14
		3	1.63	0.95	0.80	0.75	0.20	**0.13**
VLSI	10	1	55.39	44.57	45.02	42.67	3.30	1.25
		2	8.65	7.95	5.22	4.62	1.29	0.97
		3	11.82	12.92	6.61	6.74	1.00	0.81
	100	1	25.50	25.15	19.99	20.09	1.86	0.65
		2	3.30	1.61	1.34	1.14	0.63	**0.58**
		3	3.27	1.46	1.23	**1.11**	0.71	0.68
TSPFST	10	1	7.42	7.33	8.50	7.50	1.69	1.12
		2	1.82	1.34	1.37	1.14	1.07	0.82
		3	1.80	1.30	1.33	1.12	0.98	0.81
	100	1	4.55	4.24	4.03	4.02	0.43	0.28
		2	0.59	0.53	0.40	0.73	0.35	0.19
		3	0.57	**0.34**	0.41	0.35	0.23	**0.14**
I	10	1	0.903	0.863	0.986	0.826	0.094	0.039
		2	0.241	0.198	0.144	0.125	0.056	0.028
		3	0.234	0.201	0.149	0.112	0.046	0.028
	100	1	0.340	0.296	0.320	0.281	0.094	0.011
		2	0.090	0.109	0.058	0.083	0.038	0.006
		3	0.097	**0.076**	0.065	0.111	0.016	**0.006**

c_u. This implies that the distance between nodes is more important than minimizing the number of crossing edges between subsets. Furthermore, the use of a guiding solution almost always leads to better results. There exists however no strict guarantee that using a guiding solution increases solution quality. Especially when a high partition imbalance is allowed (i.e., $d = 3$), the average quality is decreased. Clearly, this is another indicator that the edge-cut of a graph is not a good measure to encourage the construction of good quality solutions.

On the other hand, we observe that the solution quality when using Vb does not depend that much on the partitioning parameters. The algorithm produces better average solution quality than all variants using Eb. In all but one cases, the increase of d leads to better results. The exception is the VLSI instance set, for which $d = 2$ achieves the best results. We conclude that in instances which contain only a very small percentage of terminals, making subsets in a partition too big may lead to less favorable results than making smaller subsets and connecting them heuristically.

Comparing the value shown in 6.8, we note that in almost every case, the average runtime of Vb is lower than the one of Eb. In Table 6.8, the lowest runtimes for each configuration are marked bold. For the instance sets ES, VLSI and TSPFST, if Eb is faster than Vb it is only by

Table 6.8: Comparing partitioning schemes in PCH w.r.t. the average runtime [s]

			Edge-based partitioning (METIS)				*New Voronoi-based partitioning*	
	$\|T\|/k$	d	c_u	c_{orig}	c_u, S_G	c_{orig}, S_G	*w/o* S_G	*w.* S_G
ES	10	1	**0.87**	0.88	0.87	0.88	0.98	1.10
		2	1.06	1.06	1.04	**0.76**	0.86	0.92
		3	1.08	1.09	1.07	1.07	**0.87**	0.93
	100	1	2.24	1.84	2.18	2.15	**1.53**	1.61
		2	4.06	3.60	4.17	4.75	**2.91**	3.21
		3	6.65	7.25	6.30	7.48	**3.74**	3.93
VLSI	10	1	1.75	2.02	1.87	**1.52**	1.56	1.73
		2	9.30	8.07	4.72	4.92	3.47	**3.01**
		3	15.07	20.93	10.54	11.40	4.01	**3.87**
	100	1	10.66	9.80	11.80	9.10	13.28	**13.05**
		2	118.53	64.72	44.38	33.05	40.07	**28.76**
		3	127.50	72.69	37.46	30.77	41.10	**28.43**
TSPFST	10	1	**1.15**	1.25	1.16	1.22	1.22	1.32
		2	1.49	1.49	1.45	1.51	**1.15**	1.22
		3	1.53	1.49	1.52	1.52	**1.44**	1.45
	100	1	7.28	7.99	7.82	**6.07**	6.97	7.69
		2	16.33	15.88	15.68	17.20	**13.14**	14.39
		3	25.21	25.90	23.80	25.29	**17.79**	18.01
I	10	1	35.33	36.51	59.48	58.68	**27.55**	53.08
		2	36.70	36.80	53.60	51.02	**24.71**	47.63
		3	36.79	37.67	52.30	56.25	**26.11**	48.14
	100	1	92.77	91.77	95.91	101.90	**43.77**	66.10
		2	**78.23**	108.83	101.92	100.12	97.46	124.90
		3	**93.60**	93.71	112.64	115.39	173.28	171.52

a small amount of time less than a second. In instance set I, Eb outperforms Vb only for larger values of d, but the longer runtimes of Vb are compensated by better solution quality.

Especially interesting is the fact that the use of a guiding solution speeds up Vb for all $d > 1$. This can be explained by the fact that due to the restricted merging process in Vb, subsets stay smaller, but at the same time the connections are more meaningful due to the guiding solution.

We conclude, that Vb performs much better both with respect to solution quality and runtime, and is also a much more robust strategy with respect to the choice of parameters. Thereby, in the remainder of this thesis, Vb with a guiding solution as specified before is used when referring to the PCH.

6.5.2 Evaluating Decomposition and Repair Methods

Next, the algorithms for *decomposition* and *repair* are evaluated. Table 6.9 lists the average gap and runtime for each instance set. Again, the results are given for different combinations of k and d. Each column shows a different combination of decomposition and repair algorithm. Here, D_s and D_a denote the simple and augmented decomposition procedures (see Section 4.2), and

Table 6.9: Comparing decomposition and reapir methods in PCH w.r.t. average gap and runtime

			D_s,R_h		D_s,R_e		D_a	
	$\|T\|/k$	d	*gap* [%]	*time* [s]	*gap* [%]	*time* [s]	*gap* [%]	*time* [s]
ES	10	1	0.98	**1.28**	**0.87**	2.04	1.18	1.35
		2	0.69	1.03	**0.62**	1.84	0.86	**1.00**
		3	0.63	1.05	**0.55**	2.01	0.89	**1.03**
	100	1	**0.18**	1.66	0.18	2.08	0.21	**1.63**
		2	**0.14**	**3.34**	0.14	3.82	0.15	3.40
		3	0.13	4.15	**0.13**	4.66	0.14	**4.10**
TSPFST	10	1	1.12	**1.50**	**0.98**	3.09	1.49	1.51
		2	0.82	1.26	0.74	**2.40**	1.03	**1.23**
		3	0.81	**1.39**	**0.73**	2.75	1.07	1.41
	100	1	**0.28**	**7.76**	0.28	8.47	0.46	8.02
		2	0.19	14.45	**0.18**	15.27	0.25	**14.09**
		3	0.14	**18.05**	**0.13**	18.91	0.21	18.99
VLSI	10	1	1.25	**1.73**	**1.12**	7.50	2.25	1.89
		2	0.97	**3.05**	**0.94**	7.58	1.81	3.37
		3	0.81	**3.88**	**0.76**	8.16	2.00	4.38
	100	1	0.66	**13.41**	**0.65**	15.28	0.65	13.78
		2	0.59	**31.04**	**0.58**	32.19	0.65	31.49
		3	0.65	**29.92**	0.65	**30.97**	0.72	30.55
I	10	1	0.039	**52.21**	**0.037**	175.58	0.062	55.50
		2	0.033	**45.44**	**0.030**	173.79	0.053	50.24
		3	0.032	**44.46**	**0.028**	166.37	0.052	50.02
	100	1	**0.010**	**60.01**	0.010	128.09	0.016	63.95
		2	0.007	**111.41**	**0.006**	200.31	0.012	118.37
		3	0.007	**169.07**	**0.007**	258.79	0.011	177.04

R_h and R_e are the heuristic and exact repair algorithms. We note that repair is only necessary for D_s, so combinations between D_a and repair algorithms do not exist. For each instance set the best average gap and the fastest runtime are marked bold.

First, we observe that for the simple decomposition, the exact repair always results in solutions that are at least as good as the heuristic repair. This is to be expected, since both algorithms are applied to the same partial solution. For $|T|/k = 100$, the difference in solution quality is only marginal. For $|T|/k = 10$, the difference is slightly higher, since the partition contains more subsets, and thus the potential for optimization is larger.

In each case, the exact repair requires more runtime than the heuristic repair. This is especially apparent for the I-instances, for which the exact solution often exceeds the specified time limit of 100 seconds. We conclude that the exact repair does not bring much benefit when the number of subsets is small, and the heuristic repair achieves the same solution quality much faster in such cases. However, if the number of subsets is large, the improvement achieved by the exact repair may be worth the increase in runtime.

Surprisingly, the augmented decomposition has not performed as well as the simple decomposition. While having a runtime that is comparable to the simple decomposition and heuristic repair, it is outperformed with respect to solution quality even without exact repair in almost

every case. This suggests that fixing Steiner nodes to optimize the solutions of subinstances decreases the overall solution quality. Thus it seems to be more effective to compute solutions independently and optimize their connection afterwards.

6.6 Tuning the Partition-based Memetic Algorithm

The combination of several algorithmic concepts results in a vast number of additional configurations that can influence the whole procedure's behavior. In this work, we concentrate on testing MPCH's aspects which we consider most essential: the influence of local search in combination with PCH and the solution recombination.

In each test, PCH is applied using the Voronoi-based partitioning scheme, simple decomposition and heuristic repair. For the partitioning parameters of PCH, we use $k = |T|/10$ and $d = 3$. The guiding solution for partitioning is always taken from the solutions within the population (cf. Section 5.1). Again, the limit for exact solution is set to $t = 100$ (seconds). This limit applies whenever the B&C algorithm is used, i.e., for solving subinstances and for the exact solution recombination. MPCH is performed for a population of ten individuals, and three generations are executed. For the iterated application of PCH, two iterations without improvement are allowed ($n = 2$).

Table 6.10 shows the performance overview for different sets of parameters. We test the algorithm both with and without VND. In addition, we apply either an exact or heuristic approach as a black-box procedure for solving the subinstances and performing recombination. As specified in Section 5.1, B&C is used to solve subinstances to optimality and SPH+MST-Prune is applied to the auxiliary graph generated by DA. For both approaches we apply VND to further enhance the solution.

For the reductions, R_n and R_e denote the average number of reduced nodes and edges, respectively. We note that only a few instances have been reduced. We conclude that the used reduction tests are not strong enough to make a significant difference compared to the elements already removed by the reduction tests during preprocessing.

In each experiment, a different configuration of local search and solution recombination algorithm is used. In the following tables, columns *VND* and *E* denote whether local search or exact solution recombination is used. If exact recombination is disabled, the solution will be recombined heuristically.

Table 6.10 is divided into three parts. The first part describes the construction of the initial population. We note that this experiment also shows the strength of a multi-start approach when using local search. The constructed solutions are likely to reside in different basins of attraction, and the results are much better when compared to a single iteration (cf. Section 6.7).

We observe that the population initialization step already yields a good result when combined with VND. Compared to the parameter tests for local search, the multistart for ten solutions can further enhance the solution quality. Compared to single start, the gap is almost halved for all instance sets (cf. Section 6.4). Since we apply SPH on the subgraph generated by DA, we also have access to a lower bound.

We note that the executed reductions only reduce the instance graph for the sets VLSI and I. These sets contain fewer terminals than the two other sets. In any case, the number of reduced

Table 6.10: Comparing different parameter configurations for MPCH

			Initialization			*Reductions*		*Result*		
	Exact	*VND*	*gap* [%]	gap_D [%]	*time* [s]	R_n	R_e	*gap* [%]	gap_D [%]	*time* [s]
ES	0	0	1.42	2.21	0.62	0	0	0.37	1.15	6.63
	0	1	0.48	1.27	1.26	0	0	0.30	1.09	11.44
	1	0	1.44	2.24	0.65	0	0	0.28	1.08	21.99
	1	1	0.48	1.27	1.19	0	0	**0.18**	0.96	25.17
TSPFST	0	0	1.60	2.43	0.56	0	0	0.44	1.30	9.70
	0	1	0.55	1.32	1.15	0	0	0.33	1.13	9.99
	1	0	1.70	2.57	0.58	0	0	0.29	1.21	66.07
	1	1	0.53	1.31	1.28	0	0	**0.19**	0.99	69.80
VLSI	0	0	1.86	3.39	2.32	1.41	2.01	0.43	1.93	16.34
	0	1	0.66	2.13	3.99	1.76	2.49	0.32	1.78	21.72
	1	0	1.86	3.39	2.18	1.41	2.01	0.29	1.77	24.09
	1	1	0.66	2.13	3.50	1.76	2.49	**0.25**	1.70	29.77
I	0	0	0.0762	0.0978	283.56	3.51	4.19	0.0078	0.0295	496.99
	0	1	0.0132	0.0349	460.72	3.55	4.24	0.0069	0.0286	963.37
	1	0	0.0762	0.0978	291.35	3.51	4.19	0.0069	0.0286	1372.59
	1	1	0.0132	0.0349	461.98	3.55	4.24	**0.0060**	0.0277	1915.04

nodes is rather small. The reason for this is that the employed reduction tests are very simple, and most of the parts which they could have reduced had already been removed by the initial preprocessing.

Finally, after the execution of three generations, the difference in the final result between starting from a locally optimal initial generation is not very large. The results are only slightly improved.

A greater change is achieved from switching between exact and heuristic combination. Using the exact combination yields much smaller gaps. This improvement comes at the price of much higher runtimes. This becomes especially apparent for the larger instance sets with many terminals. In such sets, solutions contain more edges, and thus the union subgraph is also larger. In contrast, the I and VLSI sets only double their runtimes, while the ES and TSPFST have a much higher runtime increase. The use of exact combination is thus more feasible for instances with fewer terminals.

Table 6.11 gives a more detailed view on the performance of different generations (denoted by G_1, G_2 and G_3). Here, the use of the solution archive becomes especially apparent. If diversity is too low, and solutions do not change much, the required time to apply PCH drops extremely. The drop in runtime attains a maximum if local search and exact combination are not used. Without the time required for exact combination, it is obvious that without local search, the iterative application of PCH does not introduce enough diversity to actually change the produced solutions much. Only in combination with local search, the solution is sufficiently different, so that a new partition is created. We observe that in each iteration, the gap is only improved using the exact combination.

Table 6.11: Comparing average gap and runtime per generation in MPCH

			G_1		G_2		G_3	
	Exact	*VND*	*gap* [%]	*time* [s]	*gap* [%]	*time* [s]	*gap* [%]	*time* [s]
ES	0	0	0.40	**3.48**	0.38	**1.58**	0.37	**0.96**
	0	1	0.33	5.90	0.31	2.26	0.30	2.00
	1	0	0.35	9.73	0.31	6.81	0.28	4.81
	1	1	**0.23**	10.96	**0.20**	7.09	**0.18**	5.90
TSPFST	0	0	0.47	**5.28**	0.45	**2.45**	0.44	**1.41**
	0	1	0.38	5.96	0.34	2.97	0.33	1.88
	1	0	0.38	32.37	0.32	18.42	0.29	14.70
	1	1	**0.25**	32.00	**0.20**	18.88	**0.19**	17.64
VLSI	0	0	0.50	**10.20**	0.43	**2.57**	0.43	**1.25**
	0	1	0.35	11.43	0.32	4.60	0.32	1.66
	1	0	0.39	14.18	0.31	5.43	0.29	2.30
	1	1	**0.30**	15.79	**0.26**	6.85	**0.25**	3.59
I	0	0	0.0081	**126.65**	0.0079	**51.50**	0.0078	**35.28**
	0	1	0.0072	276.89	0.0070	127.72	0.0069	97.34
	1	0	0.0076	457.86	0.0072	360.80	0.0069	262.58
	1	1	**0.0066**	679.46	**0.0061**	410.87	**0.0060**	362.14

6.7 Final Results

In this section we give a final performance overview for all implemented algorithms. In addition, the results are compared with another successful metaheuristic approach from literature, namely a hybrid GRASP with perturbations and Path Relinking (HGPPR), proposed by Ribeiro et al. [64]. Their implementation is publicly available in the program *bossa* [76].

The tests have been executed for several configurations. First,the performance of the simple construction heuristic SPH combined with MST-Prune is tested, for which the results are enhanced by an iterative improvement through VND. The goal is to provide a basic reference for the other algorithms, since it is usually instructive to see how more complex algorithms perform in comparison to simpler ones. For both SPH and VND, root nodes are chosen at random from the set of terminals. We denote this configuration by SPH+VND.

As another reference, we apply the same procedure to the reduced graph that is constructed based on the reduced costs which results from a single execution of dual ascent (cf. Section 2.3.4). This procedure may find solutions of higher quality, but more importantly gives a lower bound for the estimation of the solution's quality. In the following, this procedure is referred to as DA+VND.

In the next configuration, PCH is used as an initial solution for VND. Here, the best parameters from the previous experiments are used: Voronoi-based partitioning with $k = |T|/100$, $d = 3$ and a guiding solution, simple decomposition and heuristic repair. The guiding solution is constructed as in DA+VND. Again, the time limit for the exact solution of subinstances is set to $t = 100$ (seconds). The whole procedure is denoted as PCH+VND.

For MPCH, the same parameters as in the previous experiments are used ($g = 3$, $popmax = 10$, $n = 2$), but only the exact solution recombination is applied. For the internal application of

PCH, the number of partitions is decreased to $k = |T|/100$, which leads to better solutions at the cost of increased runtime.

For HGPPR, the number of iterations for GRASP is fixed to 128. For the Path Relinking phase, no restriction is enforced, and the algorithm only terminates if no improved solution can be found based on the current population. Adaptive Path Relinking is used, which means that the program tests the runtime of two different Path Relinking algorithms for a few iterations, and chooses the faster one. In the following, we present both solution quality and runtime for both the GRASP phase and also the full procedure. The results after the GRASP phase are denoted by GRASP, while the result of the full procedure is denoted by HGPPR.

In addition to the heuristic approaches, we also compare the results for the B&C with limited runtime, which have been already presented in Section 6.3.

In Table 6.12 and Table 6.13, the average gaps to the known optimum and the average runtimes are shown. The results are divided into two groups: In the first group, the results for VND with different starting solutions are compared. In the second group reside the remaining, more complex algorithms. For each group the best results of each instance set are marked in bold. In figure 6.1, a graphical representation of the most important test results is given (PCH+VND, MPCH, HGPPR and B&C).

Table 6.12: Comparing different methods w.r.t. the average gap [%]

	SPH+VND	DA+VND	PCH+VND	MPCH	GRASP	HGPPR	B&C
ES	0.82	0.57	**0.12**	0.04	0.32	0.09	**0.005**
TSPFST	0.88	0.70	**0.14**	**0.03**	0.31	0.09	0.037
VLSI	1.41	0.99	**0.33**	0.17	0.30	**0.10**	0.186
I	0.0264	0.0167	**0.0019**	0.0008	0.0100	0.0044	**0.0000**

Table 6.13: Comparing different methods w.r.t. the average runtime [s]

	SPH+VND	DA+VND	PCH+VND	MPCH	GRASP	HGPPR	B&C
ES	**0.08**	0.21	4.47	69.55	148.51	365.96	**56.11**
TSPFST	**0.13**	0.26	19.16	261.02	**167.29**	380.07	201.54
VLSI	**0.11**	0.39	37.77	117.63	**47.33**	76.76	458.13
I	**4.14**	31.31	156.95	**1621.09**	6262.05	63134.15	21906.91

As already shown in the local search experiments, the combination of SPH and VND produces good quality solutions in a short amount of time, even for large-scale instances. For DA+VND, the average gap is smaller than SPH+VND in every case, but the procedure scales slightly worse with respect to the average runtime. We expect that the runtime of our dual ascent implementation can be further improved, since the algorithm has already been applied to large-scale instances successfully [53].

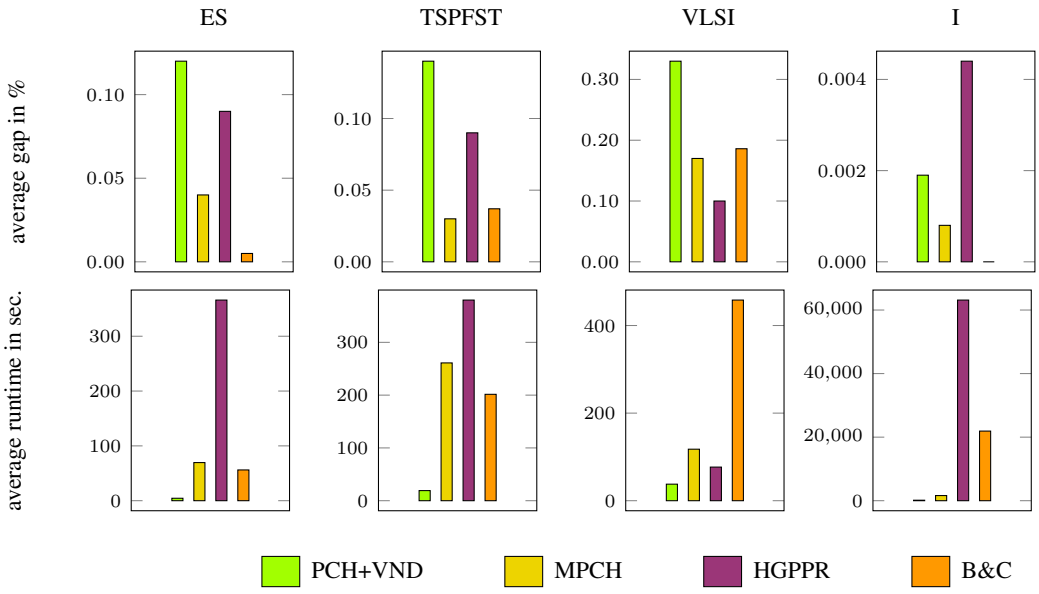
In relation to the previous procedures, PCH+VND achieves a much better average gap. However, the average runtime is also significantly higher. We note that in contrast to the other methods, the runtime of PCH could potentially be decreased through parallelization, which has not

been tested in this master's thesis. For an extensive analysis concerning PCH's parallel aspects, the reader is referred to [62].

We note that for the large-scale I instances, PCH+VND manages to outperform HGPPR both with respect to average gap and runtime. Thereby we conclude that the used partitioning scheme successfully captures the dependencies between terminals in the instance graph. The extremely high average runtime of HGPPR can be explained by the fact that the procedure iteratively applies local search procedures to suboptimal solutions created through MST-construction heuristics. In addition, HGPPR does not make use of the fast local search procedures presented in [73].

In comparison to the single-start procedure PCH+VND, MPCH always achieves a significantly better average gap. This highlights the importance of population-based procedures. However, the average runtime of MPCH is one order of magnitude higher than PCH+VND for each instance set except VLSI. We also note that for the VLSI set, MPCH yields the smallest improvement with respect to the average gap compared to PCH+VND. This may be due to the fact that MPCH does not introduce much diversity, and thus the constructed solutions are almost identical. Since a solution archive is used to speed-up the repeated solution, this results in a lower runtime.

Figure 6.1: A graphical representation of the performance comparison from Table 6.12 and Table 6.13. Only a selection of algorithms is compared: PCH+VND, MPCH, HGPPR and B&C.



We note that the use of B&C, although an exact method, is quite competitive to the other methods. The average runtimes for the sets ES, TSPFST and VLSI are not that far away from the heuristic value. The worst performance is achieved for the VLSI instances. As already stated, their regular cost structure makes it harder for exact approaches to achieve good bounds. For ES, the average runtime and gap is even better than for other approaches. The average runtime for

the I instance set is large, but not as large as HGPPR. Only MPCH seems to achieve an average gap close to B&C. We therefore conclude that B&C is a very powerful approach by itself, and that even if the time available for exact solution is limited, acceptable heuristic solutions can be achieved through the use of a primal heuristic.

In Table 6.14 we compare the dual values for each configuration that produces lower bounds (i.e., DA+VND, PCH+VND, MPCH and B&C) of the implemented procedures. Again, the best values for each instance set are marked bold. For each column, we specify the gap between the lower and the upper bound, as well as the gap between the lower bound and the optimal objective value. We note that for DA+VND and PCH+VND, the dual ascent algorithm is executed on a single terminal as root node, so the lower bounds are identical. The procedure uses ten different roots for dual ascent, however only for the VLSI instances the gap is increased more strongly. We conclude that in the VLSI instances, which contain a smaller terminal to node ratio than the other instance sets, the lower bounds created by dual ascent differ by a larger amount. For each instance set, the B&C achieves the best results, since the procedure is initialized with the cuts produced by dual ascent (cf. Section 4.4).

Table 6.14: Comparing algorithms w.r.t. the average dual gap

	DA+VND		PCH+VND		MPCH		B&C	
	$gap_D[\%]$	$gap_{DO}[\%]$	$gap_D[\%]$	$gap_{DO}[\%]$	$gap_D[\%]$	$gap_{DO}[\%]$	$gap_D[\%]$	$gap_{DO}[\%]$
ES	2.37	0.80	0.95	0.80	0.84	0.79	**0.0070**	**0.0022**
TSPFST	2.69	0.92	1.09	0.92	0.96	0.91	**0.0651**	**0.0319**
VLSI	3.92	1.51	2.37	1.51	1.64	1.46	**0.3209**	**0.1854**
I	0.0920	0.0207	0.0259	0.0207	0.0200	0.0206	**0.0000**	**0.0000**

CHAPTER 7

Conclusion

The aim of this master's thesis has been the design and evaluation of heuristic methods for the near-optimal solution of large-scale STP instances. At its center stands the concept of partitioning, which is employed as a means to find heuristic decompositions of problem instances.

A new partition-based construction heuristic (PCH) has been proposed which makes use of this concept to speed-up the construction of STP solutions, while attempting to maximize solution quality. Several partitioning schemes have been implemented and tested with respect to their performance.

Furthermore, a partition-based memetic algorithm (MPCH) has been proposed, in which PCH is used in combination with several well-known methods for the STP. The objective has been to further increase solution quality on the basis of the already high-quality solutions that are constructed by PCH.

In both algorithms exact and heuristic methods from the STP literature are used, which have been reimplemented in a common framework.

A computational evaluation has been performed, and the performance of all algorithms has been compared with respect to solution quality and runtime. In the following, the core contributions of each area are summarized:

- **Partition-based construction heuristic (PCH):**
  - Introduction of a new Voronoi-based partitioning scheme
  - Improvement of partition quality through the introduction of global information in the form of a heuristic solution (guiding solution) to the unpartitioned STP instance
  - Interchangeable use of heuristic and exact methods for different subtasks like the solution of subinstances and repair of infeasible solutions
- **Partition-based memetic algorithm (MPCH):**
  - Improvement of solution quality by multi-start application of PCH and VND

  - Estimation of solution quality through lower bounds generated by the dual ascent algorithm and instance simplification through bound-based reduction tests
  - Prevention of unnecessary computations through a solution archive
  - Construction of new high-quality solutions by a specialized recombination algorithm

- **Implementation of several state-of-the-art methods for the STP:**
  - Shortest Path Heuristic
  - Dual Ascent
  - Bound-based reductions
  - Branch & Cut
  - Fast local search procedures
    - Steiner node insertion
    - Steiner node elimination
    - Key-path exchange
    - Key-node elimination

- **Computational evaluation and comparison of algorithms:**
  - Introduction of a new set of large-scale real-world instances for benchmark purposes
  - Comparison of PCH and MPCH with other solution methods for the STP

The performed computational experiments have shown that the heuristic decomposition of an STP instance through a Voronoi-diagram-based partitioning scheme enables PCH to construct solutions that are of comparable quality to those of other methods. The procedure has been compared to a B&C procedure and a hybrid GRASP with perturbations and path-relinking. In comparison, PCH is on average orders of magnitude faster than these methods for large-scale instances.

In each case, the application of PCH in the memetic algorithm MPCH led to the construction of solutions of higher quality. Here, the exact recombination of solutions has yielded the best results, although in large-scale instances this procedure can be costly.

Bibliography

[1] ACHTERBERG, T., KOCH, T. AND MARTIN, A. Branching rules revisited. *Operations Research Letters* 33, 1 (2005), 42–54.

[2] ARAGÃO, M. P. DE, UCHOA, E. AND WERNECK, R. F. F. Dual Heuristics on the Exact Solution of Large Steiner Problems. *Electronic Notes in Discrete Mathematics* 7 (Mar. 3, 2009), 150–153.

[3] ARAGÃO, M. P. DE AND WERNECK, R. F. F. On the Implementation of MST-Based Heuristics for the Steiner Problem in Graphs. In: *ALENEX*. Ed. by Mount, D. M. and Stein, C. Vol. 2409. Lecture Notes in Computer Science. Springer, 2002, 1–15.

[4] BASTOS, M. P. AND RIBEIRO, C. C. Reactive Tabu Search with Path-Relinking for the Steiner Problem in Graphs. English. In: *Essays and Surveys in Metaheuristics*. Vol. 15. Operations Research/Computer Science Interfaces Series. Springer US, 2002, 39–58.

[5] BEASLEY, J. E. An SST-based algorithm for the Steiner problem in graphs. *Networks* 19, 1 (1989), 1–16.

[6] BLUM, C. AND ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)* 35, 3 (2003), 268–308.

[7] BUDIU, M., DELLING, D. AND WERNECK, R. F. DryadOpt: Branch-and-bound on distributed data-parallel execution engines. In: *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE. 2011, 1278–1289.

[8] BYRKA, J. et al. Steiner Tree Approximation via Iterative Randomized Rounding. *Journal of the ACM* 60, 1 (Feb. 2013), 6:1–6:33.

[9] CHERKASSKY, B. V. AND GOLDBERG, A. V. On implementing push-relabel method for the maximum flow problem. Lecture Notes in Computer Science 920 (1995). Ed. by Balas, E. and Clausen, J., 157–171.

[10] CHIMANI, M. AND WOSTE, M. Contraction-based Steiner tree approximations in practice. In: *Algorithms and Computation*. Springer, 2011, 40–49.

[11] CHLEBI K, M. AND CHLEBI KOVÁ, J. The Steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science* 406, 3 (2008), 207–214.

[12] CLAUSEN, J. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen* (1999), 1–30.

[13] CRONHOLM, W., AJILI, F. AND PANAGIOTIDI, S. On the minimal Steiner tree subproblem and its application in branch-and-price. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2005, 125–139.

[14] DAHLHAUS, E. A parallel algorithm for computing Steiner trees in strongly chordal graphs. *Discrete Applied Mathematics* 51, 1 (1994), 47–61.

[15] DANTZIG, G. B. AND THAPA, M. N. *Linear programming: 1: Introduction*. Vol. 1. Springer, 1997.

[16] DUIN, C. W. Steiner's problem in graphs. PhD thesis. PhD thesis, University of Amsterdam, 1993.

[17] DUIN, C. AND VOSS, S. The pilot method: A strategy for heuristic repetition with application to the Steiner problem in graphs. *Networks* 34, 3 (1999), 181–191.

[18] EISENSTAT, D. *dtree*. `http://www.davideisenstat.com/dtree/`. (visited on 2013-10-23). 2012.

[19] ESBENSEN, H. Computing near-optimal solutions to the Steiner problem in a graph using a genetic algorithm. *Networks* 26, 4 (1995), 173–185.

[20] FEO, T. A. AND RESENDE, M. G. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters* 8, 2 (1989), 67–71.

[21] GEBREMEDHIN, A. H. AND MANNE, F. Scalable parallel graph coloring algorithms. *Concurrency - Practice and Experience* 12, 12 (2000), 1131–1146.

[22] GENDREAU, M., LAROCHELLE, J.-F. AND SANSÒ, B. A tabu search heuristic for the Steiner Tree Problem. *Networks* 34, 2 (July 2, 2003), 162–172.

[23] GENDREAU, M. AND POTVIN, J.-Y. *Handbook of metaheuristics*. Vol. 146. Springer, 2010.

[24] GLOVER, F., LAGUNA, M. AND MARTI, R. Fundamentals of scatter search and path relinking. *Control and cybernetics* 39, 3 (2000), 653–684.

[25] GOEMANS, M. X. AND MYUNG, Y.-S. A catalog of Steiner tree formulations. *Networks* 23, 1 (1993), 19–28.

[26] GOLDBERG, A. V. AND TARJAN, R. E. A new approach to the maximum-flow problem. *Journal of the ACM* 35, 4 (1988), 921–940.

[27] GRÖTSCHEL, M., MONMA, C. L. AND STOER, M. Computational eesults with a cutting plane algorithm for designing communication networks with low-connectivity constraints. *Operations Research* 40, 2 (1992), 309–330.

[28] GUSCHINSKAYA, O. et al. A heuristic multi-start decomposition approach for optimal design of serial machining lines. *European Journal of Operational Research* 189, 3 (2008), 902–913.

[29] HAKIMI, S. L. Steiner's problem in graphs and its implications. *Networks* 1, 2 (1971), 113–133.

[30] HAREL, D. AND TARJAN, R. E. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing* 13, 2 (1984), 338–355.

[31] HU, B. AND RAIDL, G. R. An evolutionary algorithm with solution archive for the generalized minimum spanning tree problem. In: *Computer Aided Systems Theory–EUROCAST 2011*. Springer, 2012, 287–294.

[32] HU, B. AND RAIDL, G. R. Variable neighborhood descent with self-adaptive neighborhood-ordering. In: *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics*. Citeseer. 2006.

[33] KALPAKIS, K. AND SHERMAN, A. T. Probabilistic analysis of an enhanced partitioning algorithm for the Steiner tree problem in Rd. *Networks* 24, 3 (1994), 147–159.

[34] KAPSALIS, A., RAYWARD-SMITH, V. J. AND SMITH, G. D. Solving the graphical Steiner tree problem using genetic algorithms. *Journal of the Operational Research Society* (1993), 397–406.

[35] KARP, R. M. Reducibility among combinatorial Problems. In: *Complexity of Computer Computations*. Plenum Press, 1972, 85–103.

[36] KARPINSKI, M. AND ZELIKOVSKY, A. New approximation algorithms for the Steiner tree problems. *Journal of Combinatorial Optimization* 1, 1 (1997), 47–65.

[37] KARYPIS, G. AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[38] KARYPIS, G. AND KUMAR, V. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0 (1995).

[39] KERNIGHAN, B. AND LIN, S. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal* (1970).

[40] KLEINBERG, J. AND TARDOS, E. *Algorithm design*. Pearson Education India, 2006.

[41] KOCH, T. AND MARTIN, A. Solving Steiner Tree Problems in Graphs to optimality. *Networks* 32, 3 (July 2, 2003), 207–232.

[42] KOCH, T., MARTIN, A. AND VOSS, S. *SteinLib: An updated Library on Steiner Tree Problems in Graphs*. eng. Tech. rep. 00-37. Takustr.7, 14195 Berlin: ZIB, 2000.

[43] KOU, L., MARKOWSKY, G. AND BERMAN, L. A fast algorithm for Steiner trees. *Acta informatica* 15, 2 (1981), 141–145.

[44] KRASNOGOR, N. AND SMITH, J. A tutorial for competent memetic algorithms: model, taxonomy, and design issues. *Evolutionary Computation, IEEE Transactions on* 9, 5 (2005), 474–488.

[45] LAND, A. H. AND DOIG, A. G. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society* (1960), 497–520.

[46] LEVIN, A. J. Algorithm for the shortest connection of a group of graph vertices. In: *Soviet Math. Doklady*. Vol. 12. 1971, 1477–1481.

[47] LUYET, L., VARONE, S. AND ZUFFEREY, N. In: *Applications of Evolutionary Computing*. Ed. by Giacobini, M. Vol. 4448. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, 42–51.

[48] MARTINS, S. L. et al. A parallel GRASP for the Steiner Tree Problem in Graphs using a Hybrid Local Search Strategy. *Journal of Global Optimization* 17 (1999), 267–283.

[49] MEHLHORN, K. A faster Approximation Algorithm for the Steiner Problem in Graphs. *Information Processing Letters* 27, 3 (1988), 125–128.

[50] MOSCATO, P. AND COTTA, C. An introduction to memetic algorithms. *Revista Iberoamericana de Inteligencia artificial* 19, 2 (2003), 131–148.

[51] PARDALOS, P. M. AND RESENDE, M. G. *Handbook of applied optimization*. Vol. 1. Oxford University Press Oxford, 2002.

[52] PILIPCZUK, M. et al. Subexponential-Time Parameterized Algorithm for Steiner Tree on Planar Graphs. In: *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*. Ed. by Portier, N. and Wilke, T. Vol. 20. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013, 353–364.

[53] POLZIN, T. Algorithms for the Steiner Problem in Networks. PhD thesis. Saarbrücken: Saarland University, May 2003.

[54] POLZIN, T. AND DANESHMAND, S. V. Extending reduction techniques for the Steiner tree problem. In: *Algorithms—ESA 2002*. Springer, 2002, 795–807.

[55] POLZIN, T. AND DANESHMAND, S. V. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics* 112, 1 (2001), 263–300.

[56] POLZIN, T. AND VAHDATI, S. Primal-dual approaches to the Steiner problem. In: *Approximation Algorithms for Combinatorial Optimization*. Springer, 2000, 214–225.

[57] PUCHINGER, J. AND RAIDL, G. R. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In: *Artificial intelligence and knowledge engineering applications: a bioinspired approach*. Springer, 2005, 41–53.

[58] RAIDL, G. R. A unified view on hybrid metaheuristics. In: *Hybrid Metaheuristics*. Springer, 2006, 1–12.

[59] RAIDL, G. R. AND HU, B. Enhancing genetic algorithms by a trie-based complete solution archive. In: *Evolutionary Computation in Combinatorial Optimization*. Springer, 2010, 239–251.

[60] RAVADA, S. AND SHERMAN, A. T. Experimental evaluation of a partitioning algorithm for the Steiner tree problem in R2 and R3. *Networks* 24, 8 (1994), 409–415.

[61] REINELT, G. TSPLIB – A traveling salesman problem library. *ORSA journal on computing* 3, 4 (1991), 376–384.

[62] RESCH, M. Parallel Solving of the (Prize-Collecting) Steiner Tree Problem in Graphs through Partitioning. MA thesis. Vienna University of Technology: Faculty of Informatics, Dec. 2013. unpublished.

[63] RIBEIRO, C. C. AND DE SOUZA, M. C. Tabu search for the Steiner problem in graphs. *Networks* 36, 2 (2000), 138–146.

[64] RIBEIRO, C. C., UCHOA, E. AND WERNECK, R. F. F. A Hybrid GRASP with Perturbations for the Steiner Problem in Graphs. *INFORMS Journal on Computing* 14, 3 (2002), 228–246.

[65] ROBINS, G. AND ZELIKOVSKY, A. Improved Steiner tree approximation in graphs. In: *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2000, 770–779.

[66] ROBINS, G. AND ZELIKOVSKY, A. Tighter bounds for graph Steiner tree approximation. *SIAM Journal on Discrete Mathematics* 19, 1 (2005), 122–134.

[67] ROSSETI, I. et al. New benchmark instances for the Steiner Problem in Graphs. In: *Metaheuristics*. Ed. by Resende, M. G. C., Sousa, J. P. de and Viana, A. Kluwer Academic Publishers, Norwell, MA, USA, 2004, 601–614.

[68] SÖRENSEN, K. Metaheuristics – the metaphor exposed. *International Transactions in Operational Research* (2013).

[69] SPIRA, P. M. AND PAN, A. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing* 4, 3 (1975), 375–380.

[70] TAKAHASHI, H. AND MATSUYAMA, A. An approximate solution for the Steiner problem in graphs. *Math. Japonica* 24, 6 (1980), 573–577.

[71] TARJAN, R. E. *Data structures and network algorithms*. Vol. 14. SIAM, 1983.

[72] UCHOA, E., POGGI DE ARAGÃO, M. AND RIBEIRO, C. C. Preprocessing Steiner problems from VLSI layout. *Networks* 40, 1 (2002), 38–50.

[73] UCHOA, E. AND WERNECK, R. F. F. Fast Local Search for Steiner Trees in Graphs. In: *ALENEX*. Ed. by Blelloch, G. E. and Halperin, D. SIAM, 2010, 1–10.

[74] VOSS, S. The Steiner tree problem with hop constraints. *Annals of Operations Research* 86 (1999), 321–345.

[75] WARME, D., WINTER, P. AND ZACHARIASEN, M. GeoSteiner 3.1. *Department of Computer Science, University of Copenhagen (DIKU)* (2001).

[76] WERNECK, R. F. F. *Bossa*. `http://www.cs.princeton.edu/~rwerneck/bossa/`. (visited on 2013-10-06). 2003.

[77] WINTER, P. Steiner Problem in Networks: a Survey. *Networks* 17, 2 (Apr. 1987), 129–167.

[78] WOLPERT, D. H. AND MACREADY, W. G. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on* 1, 1 (1997), 67–82.

[79] WONG, R. T. A Dual Ascent approach for Steiner Tree Problems on a Directed Graph. English. *Mathematical Programming* 28, 3 (1984), 271–287.

[80] ZELIKOVSKY, A. Z. An 11/6-approximation algorithm for the network Steiner problem. *Algorithmica* 9, 5 (1993), 463–470.

Abbreviations

ACO ant colony optimization. 26

B&B branch-and-bound. 13

B&C branch-and-cut. 17

COP combinatorial optimization problem. 1

DNH distance network heuristic. 5

GA genetic algorithm. 25

GRASP Greedy Randomized Adaptive Search Procedure. 7

HGPPR hybrid GRASP with perturbations and path relinking. 25

ILP integer linear program. 2

kGPP k-way graph partitioning problem. 33

LP linear program. 15

MA memetic algorithm. 12

MPCH partition-based memetic algorithm. 48

MST minimum spanning tree. 4

PCH partition-based construction heuristic. 28

SPH shortest path heuristic. 6

STP Steiner tree problem in graphs. 1

VND Variable Neighborhood Descent. 8