Robot Tour Planning with High Determination Costs

Routing under Uncertainty

vorgelegt von
Dipl. Math. Dipl. Inf. Wolfgang A. Welz
geboren in Tübingen

Von der Fakultät II – Mathematik und Naturwissenschaften
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Promotionsausschuss

Vorsitzender:	Prof. Dr. Dietmar Hömberg
Berichter:	Prof. Dr. Martin Skutella
	Prof. Dr. Jörg Rambau

Tag der wissenschaftlichen Aussprache: 25. September 2014

Berlin 2014

Acknowledgements

This thesis is the outcome of some very exciting and valuable years working at the COGA group at Technische Universität Berlin. So in particular, I want to thank my supervisor Martin Skutella. He not only provided me with crucial suggestions and guidance, but also always had “open doors” for questions as well as everything else and gave strong support. Equally, I am grateful to Jörg Rambau for instantly agreeing to co-referee this thesis.

Another important requirement was the Matheon as the host of the project and its founding as a DFG research center: Many results of this thesis are fruits of the Matheon Project C30 “Automatic Reconfiguration of Robotic Welding Cells”. In this context, I would like to extend my gratitude to the colleagues involved in this project, Dietmar Hömberg, René Henrion and Chantal Landry, for many hours of collaboration and co-authorship. The company Rücker GmbH provided us with helpful insights about virtual commissioning and real-world welding cells.
I am also indebted to Thomas Erlebach and Michael Hoffmann for a short but great research visit in Leicester and for the time spent together on uncertainty problems.

Already prior to this thesis I had many fruitful discussions with Cornelius Schwarz – partially in Berlin, partially in Bayreuth – about robot optimization in the automotive industry. My fellow group members at COGA, Ashwin Arulselvan, Torsten Gellert and Julie Meißner, spent their free time proof reading different parts of this thesis and giving helpful advice – Thank you!
Furthermore, I have to mention Stefan Maak for helping with implementations as a student assistant in our project.
My office mate Ralf Hoffmann was available at all times for technical discussions and he was also always willing to solve any computer-related problem. Moreover, I wish to thank Dorothea Kiefer for her constant organizational and administrative support, which was very helpful for a smooth progress of my PhD.

And last, but surely not least, I am thankful to my friends and family for their support and their never-ending interest in my work, which has always been a great motivation.

Berlin, July 2014 Wolfgang A. Welz

Nomenclature

$\subset$	proper subset
$\subseteq$	subset or equal
(x, y)	open interval
$(x, y]$	left-open interval
$[x, y]$	closed interval
$\inf S$	infimum of a set S
$\sup S$	supremum of a set S
$\mathrm{head}(a)$	head of arc a
$\mathrm{tail}(a)$	tail of arc a
$\delta^+(v)$	set of arcs leaving node v
$\delta^-(v)$	set of arcs entering node v
$\delta^+(U)$	set of arcs with a head in U and a tail not in U
TSP	traveling salesman problem
$\leqq$	vector inequality, i.e. $\mathbf{x} \leqq \mathbf{y}$ if $x_i \leq y_i \ \forall i \in \{1, \ldots, n\}$
mod	modulo operator, i.e. $a \bmod n$ is the remainder of the division of a by n
AND	bitwise AND of a binary number
$\ll$	logical left shift of a binary number
2^S	powerset of set S
$\overline{S}$	complement of set S
OPT	cardinality of the optimal update strategy (see Def. 5.1.5)
$\lceil \cdot \rceil$	ceil function, $\lceil y \rceil$ is the smallest integer not less than x
MST	minimum spanning tree
$\mathbb{P}(A)$	probability of event A
$\mathbb{E}[X]$	expected value of random variable X
$\mathbb{Q}_{\geq 0}$	nonnegative rational numbers
$\mathbb{N}_{\geq 0}$	natural numbers (including 0)

Table of Contents

List of Algorithms

Chapter 1

Introduction

Industrial production of the future will be characterized by the strong individualization of products under the conditions of highly flexible production leading to an increasing importance of computerization and robotic automation [32]. This development is also referred to as the fourth industrial revolution (the German Federal Ministry of Education and Research called this "Industry 4.0"). Characteristics are the intelligent factory and fully computerized robotic assembly lines. This process is important for all areas of mass production and in particular the automotive industry, as especially in this sector cost reduction and productivity improvement will play a vital role to stay ahead of the newly industrialized countries.

In this thesis we consider one particular problem as it occurs in car manufacturing: The optimization of welding cells. Welding cells are essential elements in many production systems of the automotive industry. In these cells, several robots perform spot welding tasks on the same component. The tours of the robots need to be planned in such a way that within the given cycle time of the production line the robots can process all the weld points and return to their starting position. During all these operations, the robot arms must not collide with each other or the component. Given the data of the workpiece, the task is to find a feasible sequence of weld points as well as the trajectory planning for all the robots. Such a problem is called the *Welding Cell Problem (WCP).*

As for many real-world problems, it is hard to represent this problem in its entirety by one "scientific" field of research alone. The WCP, on the one hand, has similarities to the famous *Traveling Salesman Problem*, as it is necessary to determine the best order in which the welding tasks should be processed. On the other hand, it also contains traditional elements of *Robot Motion Planning*, which have been analyzed in robotics for many years.

The key, however, to efficiently handle this problem is by combining both aspects – the continuous motion planning as well as the combinatorial tour and sequence planning –

as this leads to far superior and more concise results than a separation of the WCP into its two scientific parts.

This research has also been the key part of the DFG Research Center MATHEON project called “Automatic Reconfiguration of Robotic Welding Cells”. We describe some of the results that have been found in this context especially for the integration of those two parts.

Combinations of continuous and combinatorial optimization have rarely been analyzed so far. One of the very few results is by Andrews and Sethian [2], in which the authors consider the so-called *Continuous Traveling Salesmen Problem*. Here, the actual shortest path between two cities is unknown, but it can be derived from an underlying known metric, which determines the cost of moving through each point of the given domain. The authors propose an approach where the optimal paths between pairs of cities are created “on the fly” as needed. However, the proposed solution method differs greatly from ours, as their method focuses on reusing information from previous distance computations to speed up future calculations.

Also, the isolated planning of given sequences has been well studied: For example in the recent paper by Spensieri et al. [81] where welding cells are optimized under the assumption that the robot trajectories are predefined and can only be rescheduled to avoid collisions between different robots. The problem then corresponds to a variation of *Job Shop Scheduling*; this aspect is also considered in Rambau and Schwarz [70].

How to read this thesis

The thesis consists of two main parts, both covering different aspects – theoretical and practical – of the WCP and of related problems. The Parts I and II are largely self-contained and can be read separately, although this is not recommended. The description of practical aspects as well as solution approaches for the real-world welding cell problem in the first part are complemented with theoretical results for the underlying uncertainty structure and related routing problems in the second part.

This chapter serves as an introduction and gives a short overview about the preliminary definitions and notions used in this thesis.

In Chapter 2, **Integrated Approach for the Welding Cell Problem**, we introduce our model for the welding cell problem as a combination of discrete and continuous optimization. It consists of two mayor subproblems. The first one, the *discrete WCP*, was first introduced in Welz [85]. Its solution approach as it is described in this chapter has also been published in Skutella and Welz [78].

The other key component is the generation of optimal trajectories between two points. We shortly introduce how this optimal control problem can be solved and explain in more detail how initial trajectories can be produced using methods from combinatorial optimization. The concepts of the efficient trajectory computation and initialization as well as the actual integrated algorithm (also described in Chapter 4) will also be published in the prospective journal paper by Landry, Welz, and Gerdts [60].

In Chapter 3, **Pricing – Shortest Path Problem with Forced Time Windows**, we take a separate look at the shortest path problem as it arises as a special subproblem for the discrete WCP. At first glance, this problem might seem as a typical shortest path problem with resource constraints, as they occur in many routing problems. However, there are fundamental differences as conflict times can only be avoided by reducing the motion velocity along certain trajectories, which might then lead to new infeasibilities along these trajectories.

We propose a graph transformation that implicitly handles these situations by introducing additional arcs. The resulting problem can then be solved within the existing resource constraint shortest path framework at the cost of a potentially much larger graph.

On the practical side, we develop a fast algorithm that can also cope with these difficulties. We evaluate its performance using several highly efficient data structures for storing and detecting the next conflict time.

Chapter 4, **Combining Discrete Optimization with Nonlinear Optimization**, is used to describe our main result of the first part, the design and implementation of an algorithm that efficiently relies on the interplay of both parts as described in Chapter 2. This combined algorithm is then tested on two-dimensional WCP instances, as they are easier to visualize. These instances are also used to evaluate the number of updates this approach requires. The combination algorithm is also part of Skutella and Welz [79].

In Chapter 5, **Aspects of Uncertainty in Optimization Problems**, we analyze the WCP in the context of edge uncertainty problems as proposed by Erlebach et al. [29]. Since this concept of uncertainty has as yet only been thoroughly analyzed for the minimum spanning tree problem, we expand this concept for a shortest path problem and the traveling salesman problem, as they are both key components of the WCP. We also extend the model to incorporate a certain probabilistic uncertainty. Our main result in this chapter is the existence of an algorithm in this setting that calculates a $(2+\delta)$-approximation for the TSP with only $\mathcal{O}(n)$ edge updates in expectation. This result relies on an algorithm for the MST that is similar to the one proposed in [29]. The solution process, however, does not require any restarts and can also be easily applied for an approximate version of the MST.

Some of the results in this chapter are based on joint work together with Thomas Erlebach and Michael Hoffmann (both University of Leicester).

In Chapter 6, **Subway Challenge**, we study a problem which is known from the Guinness World Records as the “Fastest time to travel to all New York City Subway stations”[1]. Although this problem is related to other well-known routing problems, such as the rural postman problem or the generalized traveling salesman problem, this variation has rarely been considered in the context of combinatorial optimization. By applying concepts that were originally developed as subproblems for the discrete WCP, we present a branch-and-cut algorithm that finds the optimal solution for this *Subway Challenge* of Berlin in less than a second. This approach relies on the special structure of the underlying transportation network and therefore differs from branch-and-cut approaches used for the other mentioned routing problems.

1.1 Preliminaries

In this section we introduce some of the basic terms and concepts that are used as foundation in all parts of the thesis. Nevertheless, we still assume a basic knowledge of combinatorial optimization and algorithms for reading this thesis.
A broader and more detailed introduction to all the topics discussed in the following can for example be found in Schrijver [75] or Korte and Vygen [58]. We also recommend Cormen et al. [17] for implementation details on basic algorithms and data structures.

A short tabular overview of the used notation can also be found in the **Nomenclature** in the very beginning of this thesis on page V.

1.1.1 Graphs

Graphs represent one of the most important structures in combinatorial optimization. Although they are fairly basic, they can still be used to model countless real-world phenomenons. For the most part, the problems discussed in this thesis are based on graphs. Although graphs play a vital role in almost all of the introductory literature on combinatorial optimization, their actual definitions sometimes vary slightly. We give a short overview of the notion of graphs and paths that is used throughout this thesis.

Undirected Graphs: An *undirected graph* is a pair $G = (V, E)$, where V and E are both finite sets. The elements of V are called *nodes* and the elements of E are the *edges.* Each edge $e \in E$ is an unordered set $\{u, v\}$ of two nodes $u, v \in V$. An edge $e = \{u, v\}$ is a *loop*, if $u = v$ holds. For an edge $e = \{u, v\} \in E$, the nodes u and v are called *end points*

---

[1] `www.guinnessworldrecords.com`

of e. We define $\delta_G(v)$ as the set of edges of the undirected graph $G = (V, E)$ that have the node v as an endpoint.

Directed Graphs: Very similar to the previously defined undirected graph, a *directed graph* $G = (V, A)$ consists of a finite set of vertices V and a finite set of arcs A, where each arc $a \in A$ is an ordered pair (u, v) of vertices with $u, v \in V$. The elements of V are also sometimes called *nodes*. For an arc $a = (u, v) \in A$, the vertices u and v are called the *ends* of a, and u is called the *tail* of a, while v is the *head*. An arc $a = (u, v) \in A$ is a *loop*, if its head and tail are identical, i.e. $u = v$. A directed graph $G = (V, A)$ where A corresponds to a multiset of arcs, in which arcs are allowed to appear more than once once, is called a *directed multigraph*. In this case, two arcs $a, b \in A$ are called *parallel*, if $\text{tail}(a) = \text{tail}(b)$ and $\text{head}(a) = \text{head}(b)$.
An arc $a = (u, v) \in A$ is said to *connect* the nodes u and v. We further say that $a = (u, v)$ leaves u and enters v. A directed graph is called *complete*, if each pair of graph vertices is connected by an arc.
The set denoted by $\delta^+_G(v)$ contains all the arcs of G leaving one of its nodes v. Analogously, the set $\delta^-_G(v)$ contains all the arcs entering node v in G. The set $\delta^+_G(v)$ is also sometimes referred to as the *outgoing* arcs of v, while $\delta^-_G(v)$ represents the *incoming* arcs.

Paths and Tours: Let $G = (V, A)$ be a directed graph. A sequence $P = (a_1, \dots, a_k)$ with $a_i = (v_i, v_{i+1}) \in A$ is a *path* in G. A path is called *s-t path*, if $v_1 = s$ and $v_{k+1} = t$. We then also say that the path starts in s and ends in t. The path P is called *tour*, if $v_1 = v_{k+1}$. A tour is a *cycle*, if the arcs $(a_1, \dots, a_k)$ are pairwise distinct. A path or tour is called *elementary*, if all the nodes $v_1, \dots, v_{k+1}$ (and thus also the arcs $a_1, \dots, a_k$) are pairwise distinct.
Every path induces a *subgraph* $G' = (V', A')$ of G with $V' = \{v_1, \dots, v_{k+1}\}$ and $A' = \{a_1, \dots, a_k\}$. A path P is also sometimes identified by its corresponding subgraph, which, for example allows us to write $v \in P$, if v is a node in the induced subgraph.
For an undirected graph $G = (V, E)$, we define an *s-t path* P as a sequence of edges $(e_1, \dots, e_k)$ with $e_i = \{v_i, v_{i+1}\} \in E$. This allows us to analogously define (elementary) tours and cycles for undirected graphs as well.

Cuts: Let $G = (V, A)$ be a directed graph and let $U \subseteq V$ be a subset of nodes. We define:

$$\begin{aligned} \delta^+_G(U) &:= \{a = (u, v) \in A : u \in U \text{ and } v \in V \setminus U\} \\ \delta^-_G(U) &:= \{a = (u, v) \in A : u \in V \setminus U \text{ and } v \in U\}. \end{aligned}$$

The cut induced by U is a subset B of A, with $B = \delta^+_G(U) \cup \delta^-_G(U)$.

Trees: An undirected graph $G = (V, T)$ is called *forest*, if it does not contain any cycles. Is the graph also *connected*, i.e. there is an u-v path for all $u, v \in V$, it is called a *tree*. A tree $G = (V, T)$ can also be identified by its set of edges T alone, as the nodes in V can be easily induced from T.
We call a tree $G = (V, T)$ *spanning tree* of the undirected graph $G' = (V', E')$, if $V = V'$.

1.1.2 Shortest Paths

After we defined the basic concepts of graphs on the previous pages, we now briefly introduce shortest paths and how they can be calculated. Shortest path problems are one of the main themes of this thesis, as they arise as an important subproblem of the WCP. Chapter 3 is even entirely dedicated to a special variation, the *Shortest Path Problem with Resource Constraints.*

Shortest Path Problem: Let $G = (V, A)$ be a directed graph with cost $c_a \geq 0$ for each arc $a \in A$, let $s, t \in V$. The task is to find an s-t path P in G, such that $\sum_{a \in P} c_a \leq \sum_{a \in P'} c_a$ for all possible s-t paths P'.

Algorithm 1: Dijkstra's algorithm

```
Input: a directed graph $G = (V, A)$ with arc costs $c_a \geq 0$ and a source node $s$
Output: shortest distances from $s$ to all nodes $v \in V$

foreach vertex $v \in V$ do
    $d_v \leftarrow \infty$;
$d_s \leftarrow 0$;
$Q \leftarrow V$;
while $Q \neq \emptyset$ do
    choose node $u \in Q$ with lowest distance $d_u$;
    $Q \leftarrow Q \setminus \{u\}$;
    foreach arc $a = (u, v) \in \delta^+(u)$ do
        if $d_v > d_u + c_a$ then
            $d_v \leftarrow d_u + c_a$;
return $d$;
```

Dijkstra's Algorithm: The shortest path problem where each arc has a nonnegative cost can be solved by using the famous *Dijkstra's Algorithm* as it is described in Algorithm 1. It represents one of the most simple variations of a label-based algorithm for shortest paths as it is formally introduced in Chapter 3. The algorithm manages tentative distances for every node, starting with zero for node s itself. In every iteration a node, for which the optimal distance is already known, is selected and the tentative distances to all its neighbors are updated. After all nodes of V have been processed, the shortest distance from s to all nodes has been calculated.

A more detailed description, especially on favorable data structures, and the proof of correctness can for example be found in Cormen et al. [17].

Part I

Practice

Chapter 2

Integrated Approach for the Welding Cell Problem

It is the goal of the WCP to find optimal tours for the robots such that all the given weld points on the component can be processed in the provided cycle time. The robot arms are equipped with different welding tongs and can only process specific weld points. During all these operations the robot arms must not collide with each other and safety clearances have to be kept.

The WCP consists of two major components:
- Finding the optimal assignment of weld points to the robots and their sequence.
- Finding the optimal trajectory between two weld points that does not collide with the component.

However, especially the second part corresponds to hard, computational very expensive problems and both parts heavily depend on each other. For example, the sequencing and assignment both depend on the distances between all reachable weld points.

In this chapter we will discuss the components of an approach that efficiently combines these two parts. This concept does not rely on any precomputed trajectories, as it iteratively identifies a subset of "promising" trajectories that only need to be considered. Further, during each iteration as much information as possible is also reused from the computations in the last iteration, which considerably reduces the computational effort for each new iteration.

In Section 2.1 we first classify the integrated WCP and its components, before a detailed explanation of a solution approach for its first component (assignment and sequencing) is given in Sections 2.2 and 2.3. In the next section the trajectory calculation is explained. The actual calculation as an optimal control problem is only described by means of a brief example. The trajectory calculations, however, rely on the combinatorial subproblem of finding "good" initial solutions. Algorithms for this class of problems are explained in Section 2.5.

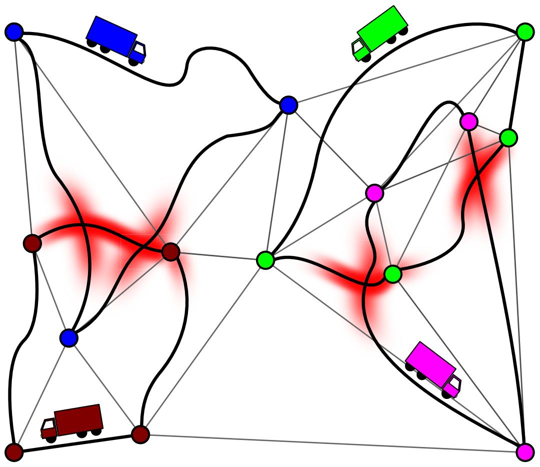


Figure 2.1: Sketch of the WCP where robots are represented as vehicles

2.1 Problem Classification

For the Welding Cell Problem we are given a representation $\mathcal{W}$ of the workpiece containing certain weld points J. Further, a set of robots R is given. For the robots to start their actual welding process, a certain joint configuration needs to be reached. Furthermore, it is, due to technical limitations, not possible for each robot to reach all the weld points. Therefore, some points are exclusive for a certain subset of robots.
The task is to plan trajectories for each of the robots so that in the end all weld points are processed while the makespan of all trajectories must be below the given cycle time of the production process.

The Welding Cell Problem consists of the following four key components:

Assignment: It needs to be decided which weld point is processed by which robot. There are some constraints that limit the possible assignments, but as in general the jobs are not fixed to one robot, the assignment step is not trivial.

Sequencing: When the jobs are assigned to the robots, it is necessary to plan the order in which they are processed before the robot returns to its starting position.

Trajectory calculation: For each sequence of weld points, trajectories for that robot need to be found. Hereby, the robot should reach all the weld points as fast as possible without colliding with the workpiece. The trajectories then correspond to the actual paths the robot takes to get from one of its jobs to the next.

Detecting and avoiding robot collisions: During all this operations the robots must not collide with each other and safety clearances have to be kept.

This list already indicates that none of its points can be considered separately. Each component depends on information from other parts. It is for example not possible to find a feasible sequence or assignment without the knowledge of distances between the weld points.
Even further, the different components also require completely different solution approaches: The sequencing and assigning relates to well-known combinatorial optimization problems such as vehicle routing and scheduling problems. The collision-free trajectory planning corresponds to a continuous optimal control problem.
This idea as well as the connections between the different components are visualized in Figure 2.2.

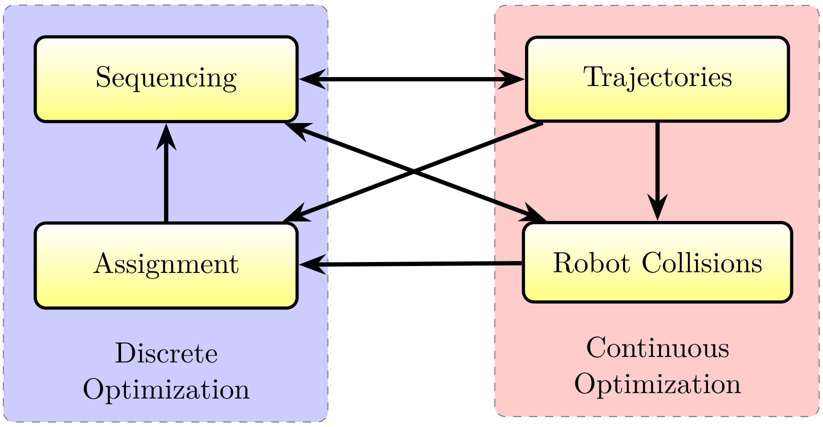


Figure 2.2: Sketch of WCP components and dependencies

So, how can the WCP still be solved in one integrated solution approach?
One approach is to first compute the trajectories for all possible job pairs. As the weld points can only be accessed in one particular way, it is then possible to just string these short trajectories together to get a valid long trajectory for a sequence of weld points. Unfortunately, this approach has two major drawbacks:

1. The collisions between two robots still depend on the actual times these trajectories are used, i.e. the sequencing.
2. The trajectory calculations are computationally expensive. It is therefore just not possible to calculate all $|J|^2$ trajectories within reasonable time.

The first drawback can be resolved by using a slightly more conservative definition of collisions for the discrete approach: Two trajectories do not collide in the discrete setting, if and only if the trajectories do not collide for any combination of start times in the continuous setting. Using this approach we might exclude some feasible solutions, but at least we can assure that all solutions that respect these collision constraints are still feasible for the continuous and exact collision calculations.

The second point, however, cannot be avoided, as distance information between all points is essential for any sequencing algorithm. The only way to effectively speed up the calculations of all trajectories is to parallelize their computations so that as many trajectories as possible are calculated at the same time on different nodes of a computer cluster. But this approach then requires a vast amount of computational power that is usually not available.

Our proposed approach addresses both points more efficiently and more elegantly: It can be best explained using two subproblems, one for the discrete part and the other one for the continuous optimization.

1. In the beginning only the coordinates of the weld points as well as the starting positions are known. These coordinates can be used to derive some very easy initial estimates, such as the Euclidean distances, on the length of the trajectories.
2. This information is then given to the *discrete WCP* that now calculates a feasible assignment and scheduling based on the information known so far.
3. The resulting tour is then fed into the *continuous WCP* returning corresponding trajectories and checking whether this solution is also feasible in the exact setting.
4. If this is not the case, we go back to point 2, but this time also the exact trajectories (their lengths and collisions) found so far are taken into account.

These points, as presented here, are not meant as complete and precise solution approach (such an algorithm is given in Section 4.1), but we can use them to formulate the requirements of the two key parts:

The **Discrete WCP** takes the distances and already determined conflicts as arguments and then returns a feasible tour for each robot. Since it has no further knowledge of the actual circumstances of the collisions, it has to find solutions where conflicting arc pairs are never in use at the same time.
A solution approach for the discrete WCP under these prerequisites is presented in Section 2.2.

The **Continuous WCP** takes a sequence of weld points for each robot and then calculates the optimal trajectory to reach all points in that order as fast as possible. The computation of such a trajectory that avoids obstacles and observes the dynamic laws as well as the bounds on the acceleration is also called *kinodynamic motion planning* [22]. It represents a continuous optimization problem that can be solved using optimal control of ordinary differential equations. As this research problem exceeds the scope of this thesis, we only give a short introductory example of this approach in Section 2.4. However, in kinodynamic planning the significance of a good initial trajectory is crucial to the overall performance [54]. Such an initial solution can be computed by solving a drastically simplified optimal control problem under the assumption that a set of good via points is available. Such points can be derived by discretizing the workspace and then performing combinatorial shortest path computations; this is discussed in Section 2.5.2.

2.2 The Discrete WCP

The *Discrete Welding Cell Problem* is a subproblem of the general WCP, where distances (not necessarily the final ones) between any pair of two weld points are given. Also, a set of *potential collisions* is given. Since no further properties of the trajectory or collisions is known, the discrete WCP has to assure that potentially colliding arcs are never used at the same time. Although this leads to a more conservative definition of collisions, feasible and collision-free tours can be guaranteed.
This collision definition is very similar to the concept of *collision zones*, which is commonly used in robotics: For every pair of trajectories it is checked whether the sweeping volumes, i.e. the volumes that a robot sweeps through when moving along the path, of the robots intersect. These collision zones are then regarded as mutually exclusive, see for example Flordal et al. [36] or Spensieri et al. [81].
With this information it is now possible to define the discrete WCP:
The discrete variation of the WCP has been introduced in Welz [85]. The following problem definition is essentially identical to the welding cell problem as described in [85], but a slightly different notation has been used to keep the notation consistent throughout this thesis.

The input data of the WCP consists of a set R of robots, a set J of jobs, each job $j \in J$ has a working set $W_j \subseteq R$ containing the robots that can process the job j, a set of collisions $\mathcal{C}$, distance information $\mathcal{D}$ and a time limit T^{limit}. Each robot $r \in R$ has a starting position s^r where its tour has to start and end. Each job $j \in J$ consists of a weld point that has to be processed by a robot in W_j. The distances are denoted by $\mathcal{D}$, i.e. the time robot r needs to get from position u to position v is given by $\mathcal{D}^r(u, v) \geq 0$. As the welding times are not explicitly given, we assume that they are included in the distance information, i.e. $\mathcal{D}^r(p, j)$ for $j \in J$ and $r \in W_j$ corresponds to the travel time from position p to j plus the processing time of job j for robot r.
For simplicity we can also interpret the input as a complete directed graph $G = (V, A)$, where $V = \{s^r : r \in R\} \cup J$. This graph is identical for all robots $r \in R$, only the arc times denoted by τ_a^r differ. If a node j cannot be visited by r, because it corresponds to the start position of a different robot or because $r \notin W_j$, the times τ_a^r of all incoming and outgoing arcs a of j are set to a value larger than the time limit T^{limit}.

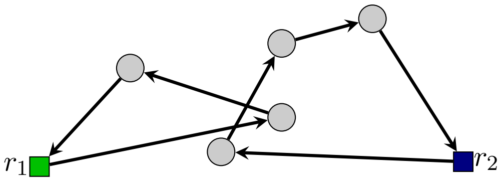
An element $(r_1, a_1, r_2, a_2) \in \mathcal{C}$ with $r_1, r_2 \in R$ and $a_1, a_2 \in A$ of the collision set $\mathcal{C}$ corresponds to a potential collision between to moving robots. We say two arcs a_1 and a_2 of an element in $\mathcal{C}$ collide, if they are in use at the same time by the corresponding robots: Let I_1 be the interval for which a_1 is used and I_2 the corresponding interval for arc a_2. The arcs a_1 and a_2 collide, if and only if I_1 and I_2 overlap.

The task is to find for each robot $r \in R$ an elementary tour $\bar{T}^r = (a_1, \ldots, a_{n^r})$, starting and ending in the depot s^r, and a schedule $\bar{I}^r$ that assigns each arc $a = (u, v) \in \bar{T}^r$ a left-closed time interval $\bar{I}^r_a$, where $\min \bar{I}^r_a$ corresponds to the departure time in u. The tuple $(\bar{T}^r, \bar{I}^r)_{r \in R}$ must have the following properties:

- Each job $j \in J$ is visited only in exactly one tour $\bar{T}^r$, i.e. j is the head of exactly one arc $a \in \bar{T}^r$.
- A job $j \in J$ can only be contained in a tour $\bar{T}^r$ for a robot with $r \in W_j$.
- For each robot $r \in R$ the schedule $\bar{I}^r$ must be feasible for the tour $\bar{T}^r$, i.e. for all $a \in \bar{T}^r$ the length of $\bar{I}^r_a$ must be at least τ^r_a and $\sup(\bar{I}^r_a)$ must be equal to the departure time of the succeeding arc.
- All robot moves are collision-free with respect to their schedule.
- The time limit is kept, i.e. for each $r \in R$ we have $\sup(\bar{I}^r_{a_{n^r}}) \leq T^{limit}$.

To further illustrate this definition, we give a simple example of scheduled tours as solutions for the discrete WCP:

Example 2.2.1 Scheduled Tours: Consider the following solution for a WCP instance with two robots r_1 and r_2. The tours $\bar{T}^{r_1}$ and $\bar{T}^{r_2}$ are given as follows:



A feasible schedule for these tours is for example given by:

$$\begin{aligned} \bar{I}^{r_1} &= \{[0,4), [4,6), [6,7)\} \\ \bar{I}^{r_2} &= \{[0,2), [2,4), [4,6), [6,10)\}\,. \end{aligned}$$

Here, the second arc of robot r_2 and the first arc of robot r_1 are in use at the same time. By setting $\bar{I}^{r_2} = \{[0,6), [6,8), [8,10), [10,14)\}$ we get a feasible schedule where the second arc of robot r_2 is used after the the second arc of robot r_1; the robot r_2 now “waits” longer on its first arc.

In this model waiting is only possible on the arcs, meaning that for two successive arcs a_1 and a_2 in a tour $\sup(\bar{I}_{a_1}) = \min(\bar{I}_{a_2})$ must hold, but the length of $\bar{I}_{a_1}$ can be arbitrarily longer than τ_{a_1}. So, instead of waiting in a certain position, the robot has to use its last arc for a longer period of time. It is however possible to start a tour later and thus effectively leave the start position s^r later. By definition, this is always possible as there cannot be any collisions for the start nodes.
This design decision allows us to restrict to collisions between arcs only. A slightly more flexible option would be to also allow waiting in the nodes together with an additional collision set referring to collisions between nodes and arcs. However, as this collision definition complicates the analysis without providing any further theoretical insights, we restrict to arc collisions only.

Remark: In the given model, collisions can be either avoided by rescheduling, i.e. introducing waiting times so that the conflict resolves, or by rerouting or reassigning the jobs. Therefore, the problem corresponds to a combination of a routing and a scheduling problem.

A different possibility for avoiding conflicts is to take “detours” over other nodes that are also visited at a later point in time by either the same tour or by completely different tours. However, if a node is visited more than once, also its processing time is added up multiple times, as it is part of the arc cost. This could in some sense be avoided by introducing additional “Steiner nodes”. Those nodes can be visited, for example, in order to avoid collision, but they do not correspond to actual jobs and thus do not need to be visited and have no processing time. As the first option leads to inconsistent solutions and the addition of Steiner nodes makes the problem harder to solve, we restrict the solution to elementary tours.

2.2.1 Related Routing Problems

One of the best-known routing problems, that is also closely related to the discrete WCP, is the *Vehicle Routing Problem (VRP).*
The basic version of the VRP can be defined as follows: Let S be a set of vehicles or servers and let $G = (V, A)$ be a directed graph with arc costs $c_a \geq 0$. Each vehicle $s \in S$ has a dedicated depot $d^s \in V$, the remaining nodes $V^C = V \setminus \{d^s : s \in S\}$ represent the customers. The task is to find for each vehicle $s \in S$ a tour $\bar{T}^s = (d^s, v_2, \ldots, v_{n^s-1}, d^s)$ such that the tours $\bar{T}^s$ are a partition of V and the cost $\sum_{s\in S} \sum_{a\in\bar{T}^s} c_a$ is minimized.
To represent the WCP as closely as possible in the VRP setting, the robots can be considered as vehicles. This concept has also been sketched in Figure 2.1 on page 12. So far, the only objective of the VRP is to minimize the overall cost and the concept of an upper limit for the tours, such as a maximal cycle time T^{limit} in the WCP, is not given.

This idea is considered for the so-called *Capacitated Vehicle Routing Problem (CVRP)*. Here, every customer $v \in V^C$ has a specific demand $d_v \geq 0$, while every vehicle only has a capacity of D.

Furthermore, for the WCP an explicit schedule is required to resolve possible conflicts. Scheduling in the context of vehicle routing problems is usually related to time windows or other resources that are consumed along the tour of a vehicle. The *Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)* is a CVRP where every customer $v \in V^C$ also has a certain time interval $[x_v, y_v]$ in which it must be visited. This problem was first introduced in Desrosiers et al. [21].
In Chapter 3 we discuss the concept of *resource constraints* in the context of shortest paths. This concept can also be used to generalize time windows, capacity constraints and other tour-structural constraints, such as elementary tours, for the VRP. However, the big difference between the discrete WCP and all the other resource constraints mentioned so far is that collisions lead to global renewable resources, while the other resources are only local for one vehicle.

Combinatorial problems where routing is combined with global scheduling constraints between multiple servers have not been discussed often. Some of the very few examples are the results in Gawrilow et al. [39], where conflict- and deadlock-free routes of automated guided vehicles through a given network have to be calculated.
Another example is the work by Gellert and König [40]. Here, a system of rail-mounted cranes is considered for which a set of transportation requests needs to be performed. The task is to minimize the total makespan, while one-dimensional collisions are avoided. The third problem, which is actually very similar to the WCP, is the *Laser Sharing Problem (LSP)*. Its first version including collision avoidance was introduced in Rambau and Schwarz [69] and then further extended in [76] and [70].

The Laser Sharing Problem

The basic situation of the laser sharing problem is similar to the situation for the WCP as they are both motivated by car body manufacturing: In a welding cell where certain welding tasks have to be performed by laser welding robots, the goal is to find a way to dispatch the robots so that the expensive laser sources can be shared among them and the cycle time is minimized.

An instance of the LSP, as defined in Schwarz [76], consists of a set of robots R, a set J of jobs and a set L of laser sources. Each job $j \in J$, however, corresponds to a welding arc and therefore has two job positions j_a and j_b. It can be processed either in both directions or in one fixed direction. The LSP has a set of line-line collisions defined identical as for

the discrete WCP, but further also line-point collisions $\mathcal{C}_{lp}$ that express collisions between a robot residing in a certain position while another robot is moving.
The task in the LSP is to compute a *scheduled dispatch* $\bar{D} = (\bar{T}, \bar{I}, \bar{\ell})_{r \in R}$. Thus, just as in the WCP each robot $r \in R$ gets assigned a tour and a schedule, but further also a function that assigns a laser source $\bar{\ell}^r(j) \in L$ to each job j in a tour. The scheduled dispatch should further minimize the makespan.

We will now discuss the similarities to the discrete WCP and explain under which circumstances it is possible to model and solve the LSP as an instance of the discrete WCP. There are several differences between the two problems. The first one is that for the LSP waiting in certain positions is possible while on the other hand, it is not allowed to vary the velocity between two positions to resolve conflicts.
However, every scheduled tour $(\bar{T}, \bar{I})$ for the WCP that respects the line-line collisions can also be interpreted as feasible tour and schedule for the LSP: Arc usages longer than τ_a are hereby interpreted as waiting in the head of a for the LSP. Since line-point collisions $\mathcal{C}_{lp}$ of the LSP induce line-line collisions for all corresponding incoming and outgoing arcs, this solution also respects the line-point collisions of the LSP.
However, as our collision definition is a little more conservative, we might loose some cycle time in this process.

For the case where the direction in which a job needs to processed is fixed (we assume in the following that j_a has to be visited before j_b) it is possible to represent the welding arcs in the WCP: As preemption is not allowed for the LSP, we only have the possibility to leave j_a via j_b. By interpreting j_a as well as j_b as own jobs with processing times of zero and assigning only one valid outgoing arc to j_a. It is therefore possible to directly integrate these welding arcs into the WCP.

The key difference between those two problems is the existence of laser sources. They can only be modeled as part of the WCP, if either $|L| = 1$ or $|L| \geq |R|$. If a laser source is available for each robot, the entire scheduling of the lasers can be omitted, as it can never reduce the solution quality to use distinct laser sources for each robot. If exactly one laser source exists its distribution can be model by additional line-line collisions so that it is not possible to process two different jobs at the same time.
Finally, one can calculate the optimal makespan by performing a binary search over all possible cycle times.

The discrete Welding Cell Problem as well as the Laser Sharing Problem are related problems both combining routing and scheduling in similar ways. However, they focus on different aspects of robot optimization: While the focus of the LSP is on the sharing of laser sources, the discrete WCP has been designed as a subproblem of the integrated WCP.

2.2.2 Solving the Discrete WCP

The discrete WCP itself, as described in Section 2.2, can also be modeled as an extended *Set Partitioning Problem* and formulated as an integer linear program with additional constraints for the collision avoidance. For this representation we first assume that all time values are discretized: The travel times for all arcs $a \in A$ should therefore be in $\{1, \ldots, L\}$, where L is a natural number representing the time limit. Such a discretization of the continuous distances and times can be obtained easily. This can be performed with arbitrary precision, but the more resulting time steps we introduce, the bigger the problem will get. Now, the discrete WCP can be modeled as the following constraint integer program:

$$\min \sum_{T\in\Omega} c_T x_T \tag{MP}$$

$$\text{s.t.} \sum_{T\in\Omega} \delta_{vT} x_T = 1 \qquad \forall v \in V \tag{2.1a}$$

$$x \text{ is collision-free} \tag{2.1b}$$

$$x_T \in \{0,1\} \qquad \forall T \in \Omega \tag{2.1c}$$

The set Ω contains all feasible scheduled tours $(\bar{T}^r, \bar{I}^r)$ for all the robots $r \in R$. The coefficient δ_{vT} specifies, whether node v is visited by the scheduled tour T or not:
For $T = (\bar{T}^r, \bar{I}^r)$, we have

$$\delta_{vT} = \begin{cases} 1 & \text{if } v \in \bar{T}^r, \\ 0 & \text{otherwise.} \end{cases}$$

The binary variables x_T denote whether this particular scheduled tour is used or not. The set partitioning constraints (2.1a) guarantee that every node is visited exactly once and since s^r is also contained in V, exactly one tour will be assigned to each robot $r \in R$. All other problem constraints are implicitly given by Ω, as it, by definition, only contains elementary tours with feasible schedules that are within the given time limit.

The MP finds tours so that the total sum of their costs c_T gets minimized. This cost was not present in our original definition of the discrete WCP and these tour costs can be interpreted in the following ways:

- By setting the costs c_T to zero for all $T \in \Omega$ we convert the program MP into a pure feasibility problem, which exactly represents the problem defined in Section 2.2.
- From a computational point of view it is usually favorable to assign some costs that help to guide the solution process into the right direction. For example, one

could set the tour costs c_T in relation to the actual length of the tour with respect to the given distance information $\mathcal{D}^r$ or even combine this with some penalties for arcs that occur in many elements of the collision set $\mathcal{C}$. In both cases the solution process can be stopped after the first feasible solution has been found.

- As a third option, this model can be used to find a minimum cost solution under the given constraints. This makes it for example possible, to find a feasible solution that minimizes the energy consumption without loosing anything from the given time limit.

In any case, the program MP contains a huge number of variables: Already the number of possible elementary tours in the underlying graph G is exponential in the number of nodes $|V|$. In fact, as the WCP combines routing with scheduling, this number even grows since there is in general a big possible number of feasible schedules for one particular tour. Therefore, our solution procedure is based on a column generation approach. An introduction about this technique can be found, e.g., in Desrosiers and Lübbecke [20]. We consider as the *restricted master problem* the continuous relaxation of the MP model where the constraints (2.1b) and (2.1c) have been replaced by $x_T \geq 0$. The tour variables only correspond to a subset $\Omega' \subseteq \Omega$ of tours. Now, we iteratively solve the restricted master problem and search for new columns having negative reduced cost that is computed using the optimal dual solution.

Let the dual variables corresponding to the constraints (2.1a) be π_v. We determine whether a scheduled tour in $\Omega \setminus \Omega'$ could improve the fractional solution by solving the following associated *pricing problem*:

$$\bar{c}^* = \min_{T \in \Omega} \left\{ c_T - \sum_{v \in V} (\delta_{vT} \pi_v) \right\} . \tag{2.2}$$

If $\bar{c}^*$ is negative, the current solution is not optimal for the restricted master problem and the column corresponding to the optimal solution for the subproblem (2.2) should be added. The new restricted master problem is re-solved and the process is iterated as long as the pricing problem has negative solutions. As the set Ω is finite after the discretization, this process will eventually terminate and an optimal solution for the master problem is returned.

The pricing problem corresponds to the combinatorial optimization problem of finding the shortest tour T^* with respect to the arc costs c_a and node prices of π_v. Tour T^* must be a valid tour for robot $r \in R$, which means that it must visit the starting position s^r and that no vertex is visited more than once. Additionally, the traversal time of the tour must be bounded by L.

As the dual variables π_v correspond to equality constraints in the primal problem, they can be positive and it is therefore possible that in the graph corresponding to the pricing problem negative cost cycles exist.

We notice that the restricted master problem may be infeasible during the loop mentioned above, either in the beginning when no columns have been generated yet or due to branching constraints, which are introduced later. In this case, we can use Farkas' Lemma to add variables that gradually restore the feasibility or detect that there is no such variable and the current master is infeasible. This method has been called *Farkas Pricing* and is very similar to the pricing problem. It can be solved by the same problem for which simply a zero cost function is used and the dual solution values are replaced by the farkas multipliers.

So far, we have just solved the fractional relaxation of MP and the constraints (2.1b) and (2.1c) are in general not satisfied. These constraints then need to be enforced in a branch-and-bound framework.

Integrality Constraints

Let us assume the optimal solution returned for the restricted master problem is fractional. We then choose a tour with fractional value and on this tour we select an arc whose value, i.e. the sum of the values of all tours using this arc, is less than one and therefore fractional. Since the start positions s^r cannot be visited by any other robot than r and due to the equality constraints for the nodes, such an arc always exists. We then branch on this selected arc a:
For the first branch, we force arc $a = (u, v)$ to be in the tour of one robot. This can be done by simply removing all other arcs leaving the node u and all other arcs entering v. As the node v always needs to be visited due to the constraints (2.1a), the only possibility now is to use the arc a, which enforces a value of one along this arc.
For the second subproblem we remove arc a in all graphs of the corresponding sub-tree.

If the values of all arcs are either zero or one, this induces binary values for all tour variables and the constraints (2.1c) are fulfilled. All these branching decisions can be modeled by changes in the problem graph of the corresponding pricing problem, while the problem itself stays unchanged as no explicit constraints are added.

Remark: This branching rule relies on the fact that every node needs to be visited and that it is therefore possible to force an arc by removing all other incoming arcs. To get around this requirement, one could follow the so-called path-splitting branching rule used for multi-commodity flow in Barnhart et al. [6]. In this rule two branches are created for the point where two fractional tours differ. They accordingly partition the set of edges adjacent from this node into two subsets E_1 and E_2. The arcs of each set are then forbidden in one branch. Now, the integrality can be enforced without the need of forced arcs.

Avoiding Collisions

Also the constraints (2.1b) can be handled with branch-and-bound. Assume we are using the two incompatible arcs a_1 and a_2 simultaneously in the time interval $[t_a, t_e]$. We then select one time step $t_b \in [t_a, t_e]$ and branch on this time step. For time step t_b, there are three compatible possibilities for the arcs a_1 and a_2:
(a) a_1 is used and a_2 is not
(b) a_2 is used and a_1 is not
(c) neither a_1 nor a_2 are used
So we create three subproblems where in each one of them one of these possibilities is ensured by forcing or forbidding the corresponding arcs in time step t_b. We say an arc a is forced in time step t_b, if the arc must be used during t_b. It does not matter whether a is used in other time steps or not. This branching scheme is valid because every collision-free integer solution always fits into one of the branches and, therefore, no solution is lost. Moreover, the branching scheme will find the optimal collision-free integer solution in finitely many steps since there are only finitely many possible time steps to branch on and only finitely many arcs.

To prevent the generation of duplicate tours and to ensure the optimality of the pricing problem, these constraints must also be imposed for the pricing. This then leads to the problem of finding the shortest elementary tour with additional multiple forbidden time windows on the arcs and some time intervals for which a certain arc must be visited. The pricing problem is in general $\mathcal{NP}$-hard and can be interpreted as a special variation of a *Shortest Path Problem with Resource Constraints.* As the pricing itself presents a very interesting and new combinatorial optimization problem it is discussed in more detail in Chapter 3.

This branch-and-price approach has been implemented using the modular constraint integer programming solver SCIP [1] developed at the Konrad-Zuse-Zentrum in Berlin.

2.3 Collision-Aware Preprocessing

In the described solution approach collisions are resolved via branching alone. They do not have an impact on the LP bound used in the branch-and-price process. To improve this behavior we developed a preprocessing step that takes collisions into account. This step eliminates arcs which can never occur in any feasible solution as they introduce so

many collisions that resolving these collisions via waiting would lead to tours longer than T^{limit}.
Hereby, we check for each arc $a \in A$ and robot $r_1 \in R$ whether feasible solutions exist in which robot r_1 is using a and all other nodes $n \in V$ can still be visited. If this is not the case, then we know that arc a will never be used by robot r_1.
We will hereby use the fact that conflicting arcs cannot be used at the same time to derive lower bounds on the tour length for this particular situation.

In the following we say that a pair of a robot and an arc $(r_1, a) \in R \times A$ is *in conflict* with $(r_2, n) \in R \times V$, if for all incoming and outgoing arcs $a' \in \delta^+(n) \cup \delta^-(n)$ the arcs a and a' are in conflict, i.e. $(r_1, a, r_2, a') \in \mathcal{C}$. This definition can also be extended to say that two nodes u and v are in conflict, if all the incoming and outgoing arcs of u are in conflict with v.

Let now $(r_1, \bar{a})$ and $(r_2, \bar{n})$ be such a conflict. It is clear that node $\bar{n}$ must either be visited before or after arc $\bar{a}$. This observation can be used to derive a lower bound on the end time of any solution where robot r_1 uses $\bar{a}$ and robot r_2 visits $\bar{n}$.
We will also need the shortest distance $d^r(v, v')$ between any two positions v and v' of robot r. The resulting distances $d^r(v, v')$ can be obtained by solving an all-pairs shortest paths problem for each robot $r \in R$. Therefore, the distances d^r are always metric for a particular robot r.

Let us first consider the case where the arc $\bar{a} = (u, v)$ is visited before node $\bar{n}$. The situation arising for this case is also depicted in Figure 2.3:
The earliest time at which node $\bar{n}$ can be visited is after r_1 reaches v. This time must be at least as large as $d^{r_1}(s^{r_1}, u) + \mathcal{D}^{r_1}(u, v)$. After node $\bar{n}$ has been reached, a time of at least $d^{r_2}(\bar{n}, s^{r_2})$ is required to finish the tour of robot r_2. Together, this leads to the conclusion that in this case the end time t^{end} of the scheduled tour for robot r_2 must be at least

$$t^{end} \geq d^{r_1}(s^{r_1}, u) + \mathcal{D}^{r_1}(u, v) + d^{r_2}(\bar{n}, s^{r_2})\,. \tag{2.3}$$

We can now further improve this bound using the following observation:
It is in general not possible to visit node $\bar{n}$ right after $\bar{a}$. Robot r_1 needs to be on an arc that is not in conflict with $\bar{n}$ before r_2 can visit $\bar{n}$ and the robot r_2 can only start using an arc leading to $\bar{n}$ when r_1 is done with $\bar{a}$. This means that robot r_1 must first reach a node which is not in conflict with $\bar{n}$ and r_2 must start from a node not in conflict with $\bar{a}$. Therefore, the shortest distance to such a node can be added to (2.3):
The shortest distance from node v to such a node that is not in conflict with $\bar{n}$ is denoted by d_v^+:

$$d_v^+ = \min \left\{ d^{r_1}(v, v') : v' \in V, (r_1, v') \text{ and } (r_2, \bar{n}) \text{ are not in conflict} \right\}\,.$$

Analogously, we have $d^-_{\bar{n}}$ as the shortest distance from a safe node to $\bar{n}$:

$$d^-_{\bar{n}} = \min\left\{d^{r_2}(v', \bar{n}) : v' \in V, (r_1, \bar{a}) \text{ and } (r_2, v') \text{ are not in conflict}\right\}\,.$$

This can now be combined to get a better bound on the end time:

$$t^{end} \geq d^{r_1}(s^{r_1}, u) + \mathcal{D}^{r_1}(u, v) + \max\left\{d^+_v, d^-_{\bar{n}}\right\} + d^{r_2}(\bar{n}, s^{r_2})\,. \tag{2.4}$$

The situation used to derive (2.4) is visualized in Figure 2.3.

An analogous bound can also be calculated for the situation where $\bar{n}$ is visited before $\bar{a}$. By then taking the minimum of those two bounds we get a lower bound on the makespan of any solution, where r_1 uses $\bar{a}$ and r_2 uses $\bar{n}$.

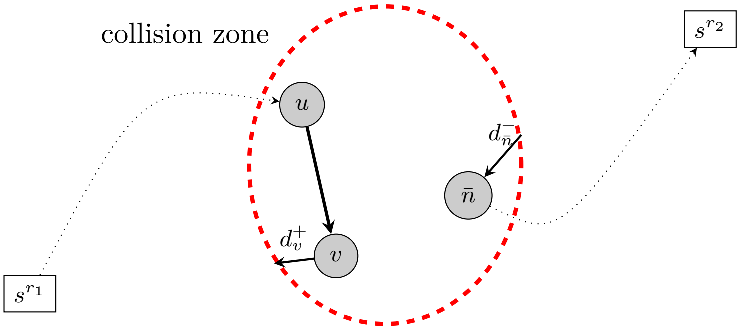


Figure 2.3: Conflicting arc-node pair

Now, for all the conflicting pairs $(r_1, a) \in R \times A$ and $(r_2, n) \in R \times V$ with $r_1 \neq r_2$ we calculate the bounds corresponding to the inequality (2.4). If for one combination the corresponding end time is greater than the time limit T^{limit}, we know that there will never be a feasible solution in which r_1 uses a and r_2 visits n. And further, if there exists a pair (r_1, a) and a node n such that n cannot be visited by any robot in W_n, we know that arc a can never be used for robot r_1 and it can thus be removed.

2.3.1 Fixed Jobs

The same idea can be extended if there are jobs j in the discrete WCP instance that can only be visited by a single robot, i.e. $|W_j| = 1$. The corresponding nodes are called *fixed nodes*.

The general idea is exactly the same as in the last section: We again try to give a bound on the makespan of the solution where robot r_1 uses $\bar{a}$ and r_2 uses $\bar{n}$, but this time we also take fixed nodes into account:

Consider the same situation as in Figure 2.3, but now there are also fixed nodes F of r_2, i.e. for each $j \in F$ we have $W_j = \{r_2\}$, that are in conflict with $\bar{a}$. None of the nodes in $F \cup \bar{n}$ can be visited while robot r_1 is on the arc $\bar{a}$. It is therefore only possible for robot r_2 to visit a subset $F' \subset F$ of nodes before we have to leave the collision zone and then (after $\bar{a}$ is done) enter the collision zone again to finish the remaining nodes in F. This "switching" to arc $\bar{a}$ however costs additional time, as robot r_1 has to reach a node that is not in conflict with the next node of r_2 before that robot can visit its next node. By using the same notation as for (2.4), we have that the switching time from arc $\bar{a}$ to a node $f \in F$ is at least

$$t^s(\bar{a}, f) := \max\{d_v^+, d_f^-\}\,.$$

Now, the lower bound for the makespan consist of the shortest possible time frame to visit a subset $F' \subset F$ in any order, then $\bar{a}$ and then $F \setminus F$. Even for a small set F the number of all possibilities grows too quickly to be able to solve this via "brute force" enumerations. Instead, we convert this into a well-studied combinatorial optimization problem which can be solved quickly for these smaller sizes – the *Traveling Salesman Problem*. A general overview of the TSP and its solution methods can be found in Applegate et al. [4], Gutin and Punnen [48].

Hereby, we build the following directed graph $G' = (V', A')$, with $V' = \{s_0\} \cup \{u, v\} \cup F$, where s_0 is a dummy start node. We now choose the costs of the arcs in such a way that the cost of each Hamiltonian cycle corresponds to the time of the particular order in which the nodes in F and the arc $\bar{a}$ is visited:

- The arcs $a' \in F \times F$ get the original distance of r_2 between those two nodes.
- The arcs $a' \in F \times \{u\}$ correspond to a switch from f to $\bar{a}$ and, thus, gets a cost of $t^s(\bar{a}, f)$.
- The arcs $a' \in \{v\} \times F$ correspond to a switch from $\bar{a}$ to f.
- For all $f \in F$ the arcs (f, s_0) get a cost of $d^{r_2}(f, s^{r_2})$, while the arcs (s_0, f) get $d^{r_2}(s^{r_2}, f)$.
- Lastly, (v, s_0) has a cost of $d^{r_1}(v, s^{r_1})$ and (s_0, u) gets the cost $d^{r_1}(s^{r_1}, u)$.

All the arcs not mentioned in this list get a cost higher than T^{limit} and will therefore not be part of any feasible Hamiltonian cycle. The graph G' together with the cost corresponds to an asymmetric TSP instance where the optimal solution corresponds to a lower bound on the makespan of any solution where r_1 uses $\bar{a}$ and r_2 uses $\bar{n}$. An example of the resulting graph G' is also shown in Figure 2.4.

Remark: If there are no fixed nodes then the graph G' only consists of four nodes $s_0, u, v, \bar{n}$ and there are only two different Hamiltonian cycles: One where $\bar{n}$ is visited

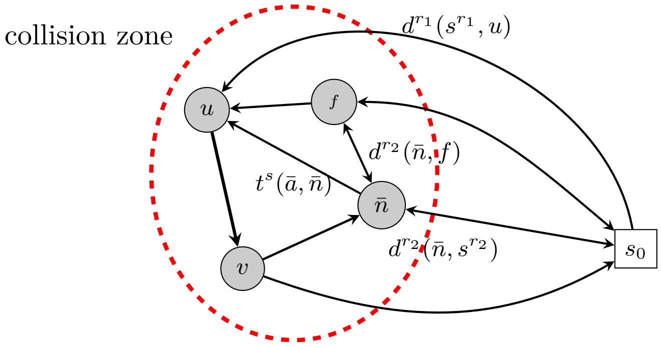


Figure 2.4: Conflicting arc-node pair with a fixed job and the resulting TSP graph G'

before u and the other one where $\bar{n}$ is visited after v. This is then corresponds to exactly the same situation as in Figure 2.3 and the TSP based preprocessing approach can also correct for situations where there are no forced arcs.

During the execution of the preprocessing rule, one TSP instance for all the conflicting pairs $(r_1, a) \in R \times A$ and $(r_2, n) \in R \times V$ with $r_1 \neq r_2$ needs to be solved. Although each instances only consists of very few nodes (even for WCP instances with many conflicting arcs, the number is typically less than 15) many different instances need to be solved, which can still be very time consuming. We note, however, that for fixed r_1, r_2, a the distances between identical nodes do not change. We can therefore construct a graph G'' that contains all the nodes and then hide the nodes which are not in F. This approach has the advantage that information from other calculations can be reused. The TSP instances are hereby solved as integer linear programs with subtour elimination. (The details of such an ILP based implementation are provided for a slightly different context in Section 6.2.1. However, the IP used for the reprocessing is identical to ATSP-IP in that section.)
For each fixed r_1, r_2, a one new ILP is created. Then, for every node n, visiting of all nodes will be forbidden that are not in G' by setting the corresponding degree constraints to zero. As our implementation only generates inequalities that are valid no matter which nodes are forbidden, they can be reused and they are thus kept for the next n. This idea considerably reduces the number of new separation steps.

Remark: For WCP instances that do not contain that many conflicts, the TSP instances will be very small and they only contain less than 6 nodes. In this case even the warm-started ILP approach is an “overkill”, as creating and solving the underlying LPs for the bounds introduces unnecessary overhead. Therefore, for such small instances we use a very simple from of bounded enumeration in which closest neighbors are preferred (Algorithm 2).

Algorithm 2: Solving TSP with bounded enumeration

```
Input: a directed graph $G = (V, A)$ with arc cost
Output: a shortest Hamiltonian Cycle
foreach node $v \in V$ do
    sort all outgoing arcs of $v$ in ascending order based on their cost;
$T^{best} \leftarrow$ null;
recursion($\{s\}$);
return $T^{best}$;

Function recursion($T$)
    if cost of $T$ is lower than the cost of current best tour $T^{best}$ then
        if $T$ is Hamiltonian Cycle of $G$ then
            $T^{best} \leftarrow T$;
        else
            $u \leftarrow$ last node in $T$;
            foreach outgoing arc $a = (u, v) \in \delta^+(u)$ do
                if $v$ is not contained in $T$ then
                    recursion($T \cup \{v\}$);
```

2.4 Trajectory Planning and Collision Detection

The optimal trajectory planning for the welding cell problem can be solved as a nonlinear optimization problem. The approaches and algorithms used in this context have been developed by Gerdts, Henrion, Hömberg, and Landry [42].
As these methods belong to a different field of research, explaining them in full detail would exceed the scope of this thesis. Instead, we use this section to give a very brief introduction about how the trajectories can be calculated.

To keep the situation as simple as possible, we only consider the very small case with a two dimensional robot composed of exactly one convex polygon. We are then interested in finding the fastest trajectory from the initial position $q_I \in \mathbb{R}^2$ to the goal position $q_G \in \mathbb{R}^2$ that does not collide with the obstacle represented by another convex polygon P.

To describe the motion from q_I to q_G, we need to define the following variables:
Let r, v and a denote the position, velocity and acceleration, respectively, of the robot in the two dimensional space. Further, let t_G be the travel time between the two positions. The motion of the robot for $t \in [0, t_G]$ can then be described by the following dynamic laws:

$$r'(t) = v(t) \quad \text{and} \quad v'(t) = a(t)\,. \tag{2.5}$$

A trajectory is collision-free, if the robot represented by

$$R(t) = \left\{ \mathbf{x} \in \mathbb{R}^2 : \mathbf{A}(t)\mathbf{x} \leq \mathbf{b}(t) \right\}$$

does not intersect with the obstacle described by the polygon

$$P = \left\{ \mathbf{x} \in \mathbb{R}^2 : \mathbf{C}\mathbf{x} \leq \mathbf{d} \right\}\ .$$

For fixed time t the robot $R(t)$ does not collide with P, if and only if the following linear system does not have a solution:

$$\begin{pmatrix} \mathbf{A}(t) \\ \mathbf{C} \end{pmatrix} \mathbf{x} \leq \begin{pmatrix} \mathbf{b}(t) \\ \mathbf{d} \end{pmatrix}\ .$$

By applying Farka's lemma (see e.g. Bertsimas and Tsitsiklis [9]) we get that $R(t)$ and P do not intersect, if and only if there exists a vector $\mathbf{w} \geq 0$ such that

$$\begin{pmatrix} \mathbf{A}(t) \\ \mathbf{C} \end{pmatrix}^T \mathbf{w} \ = \ \mathbf{0} \text{ and } \begin{pmatrix} \mathbf{b}(t) \\ \mathbf{d} \end{pmatrix}^T \mathbf{w} \ < \ 0. \tag{2.6}$$

This gives us a nice algebraic representation of collisions, which can then be used as constraints in the actual optimization problem.

The fastest trajectory of the robot is the solution of an *Optimal Control Problem (OCP)* where the system of ordinary differential equations is given by (2.5). The collision avoidance is guaranteed as soon as the vector w of (2.6) is found for each time t. However, for the actual OCP the strict inequality in (2.6) needs to be sharpened to include a safety margin $\varepsilon > 0$ and boundary conditions need to be added.
To solve the OCP numerically, the package OCPID-DAE1[1] developed by M. Gerdts is used. A good overview of optimal control problems with ordinary differential equations can be found for example in Gerdts [41].

---

[1] See `www.optimal-control.de`

Using the same idea, this approach can also be extended to work for more than one obstacle and three dimensional robots, their position being represented by the vector of joint angles. However, the resulting optimization problem contains a lot of constraints. Details on the full problem as well as the introduced methods to reduce the number of variables and constraints can be found in Landry et al. [59].

One crucial aspect for the performance of the optimization algorithm is the quality of the initial trajectory used to start the optimization process. Our method combines discrete and continuous optimization concepts. First, a shortest path algorithm is used to determine a list of via points. Then, a much smaller optimal control problem is used to determine the fastest trajectory that passes through the vicinity of the via points. This trajectory is then used as the initial solution.
As this first OCP does not take collisions into account, it is essential that the via points lead to close to optimal, collision-free trajectories. The combinatorial techniques used to find such good via points are discussed in the next section.

To check resulting scheduled tours for their feasibility, the algorithm as described in Schwarzer et al. [77] has been used to detect a collision between robots moving along specified trajectories.

2.5 Path Computations for the WCP

In this section, we discuss two aspects of the continuous WCP – the distance initialization as well as the computation of via points for the kinodynamic motion planning.
Both approaches rely on robot path-planning, which is one of the key problems in robotics: Path-planning involves searching a collision-free path in the workspace between two given positions q_I and q_G. Several techniques exist such as graph search algorithms, cell decomposition, potential field methods, probabilistic roadmaps or rapid exploring random trees. See for example LaValle [63] or Goerzen et al. [43] for an exhaustive review.

2.5.1 Calculating Initial Distances

The first step that needs to be performed for the integrated approach is the initialization of the travel times: As the initial heuristic distance computations need to be performed for all possible point pairs, it is essential that these can be calculated very efficiently. There are several ways to achieve this:

Position Distances

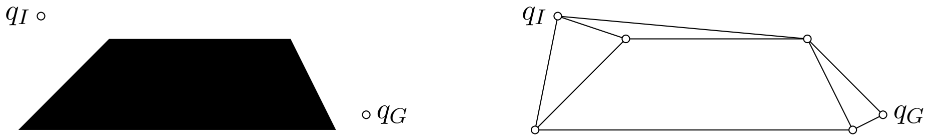
The easiest way is to completely ignore the workpiece and just calculate the distance between the two points q_I and q_G. The corresponding travel time $\tau^r_{(q_I, q_G)}$ of robot r can then be calculated by applying the maximal acceleration and deceleration of that robot. The acceleration might of course vary for different joints and directions so that the corresponding norms for the distance computation should be selected. These times are very easy to compute, obviously even for instances with very complex geometry, but as they do not take any special structure of the component into account they do not represent the actual differences very well.

Shortest-Path Roadmaps

One efficient approach that relies on the obstacles $\mathcal{C}_{obs}$ to be composed of convex polygons is the *shortest-path roadmap* (see e.g. LaValle [63, Section 6.2]). Here, a roadmap $\mathcal{R}$ is constructed that consists of all the vertices of the polygons. If a *bitangent* line can be drawn through a pair of vertices, then a corresponding edge is made in $\mathcal{R}$. A bitangent line is a line that is incident to two vertices and does not traverse the interior of $\mathcal{C}_{obs}$. This completes the roadmap $\mathcal{R}$. To now find a solution to a query of q_I and q_G, the roadmap $\mathcal{R}$ is first extended by connecting q_I and q_G to all roadmap vertices that are visible, i.e. where the direct line does not intersect with $\mathcal{C}_{obs}$. The minimum collision-free distance from q_I to q_G can now be found using a simple shortest path computation on the extended roadmap, where each edge is given a cost that corresponds to its path length.

Again, this distance needs to be converted into the corresponding travel time. In order to guarantee that the derived travel times represent a lower bound on the actual trajectory lengths, we again have to assume that the returned paths are always traversed with maximum acceleration.

The shortest-path roadmap approach is also shown in Figure 2.5.



(a) Obstacle **(b)** Extended roadmap $\mathcal{R}$

Figure 2.5: The shortest path in the extended roadmap is the shortest collision-free path between q_I and q_G.

It is in principle possible to extend this approach for three dimensions, but in 3D the queries are usually given in joint space, i.e. the space defined by all the vectors that describe the displacement of each joint, while the obstacles are usually given in Cartesian space. Given the joint angels, it is easy to calculate the position of the end-effector in Cartesian space. But the other direction, the so-called *inverse kinematics* is more complicated to calculate and usually more than one possible joint configuration for one position of the end-effector exists.

The idea of calculating the distances in joint space can be generalized to the concept of *configuration space*: Motions of robots are represented as a path in configuration space, which is the set of all possible configurations for a particular robot.

- If the robot has a two-dimensional shape, each pose of the robot can be represented as a translation combined with a rotation. This leads to a three-dimensional configuration space for a two-dimensional workspace.
- For a three-dimensional robot arm each pose can be described with the angle of each of the k joints. In this case the configuration space is equal to the joint space and is also k-dimensional.

This leads to the following general approach using configuration space:

Grid-Based Search

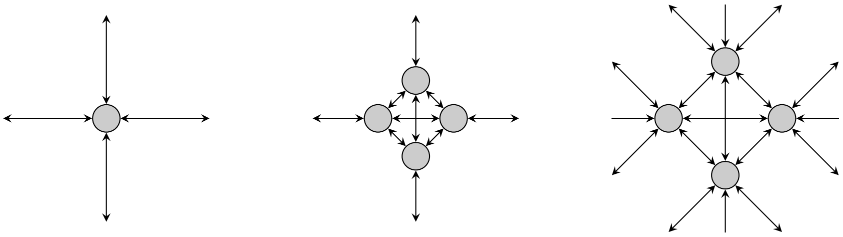
Grid-based approaches cover the configuration space by a regular grid. Each point of the grid $\mathcal{G}$ then corresponds to one configuration of the robot. Since the obstacles are usually given in Cartesian space, they have to be converted into configuration space first:
Grid points are removed from $\mathcal{G}$, if the respective configurations lead to a collision or to a violation of safety margins. Adjacent grid points are then connected in $\mathcal{G}$, which then leads to the so-called *grid graph*. Hereby, it is crucial that the discretization is chosen in such a way that the grid is fine enough so that every connection in $\mathcal{G}$ indeed represents a feasible collision-free motion of the robot. Unfortunately, the number of points on the grid grows exponentially in the number of dimensions of the configuration space. To create valid courser grids, not only the nodes need to be checked for collisions but also the entire motion of the robot between two nodes.
A trajectory between the two grid nodes q_I and q_G can now be found by a shortest path computation in the grid $\mathcal{G}$. Although the initial creation of the grid might be very time consuming, the advantage of this very flexible approach is that after this initial step, the actual path computation are fairly easy and can be calculated rather fast. The length of the shortest path corresponds to the Manhattan distance between q_I and q_G in the configuration space. This distance again needs to be transformed into the travel time using the maximal possible acceleration in that direction.

Grid-based search offers an easy but also very flexible approach to find short and collision free paths. Other techniques commonly used in robot motion planning are *triangulation* or different forms of *cell decomposition*. By using a triangulation of the free configuration space it is possible to obtain a roadmap for the shortest path computation that contains considerably fewer edges and vertices than the grid and does not depend on the discretization. The main difficulty using such approaches is to convert the workspace obstacles into forbidden areas of the configuration space. And even when an appropriate cell decomposition of the configuration spaces is at hand, it does not necessarily provide safe paths, if we are interested in paths that also maximize clearance or deal with addition objectives as presented in the next section.

2.5.2 Computation of Via Points

The path planning approach as presented in Section 2.4 relies on local-search and thus requires good initial trajectories that are derived from certain via points. In our approach, these points are calculated based on the shortest path in the grid graph $\mathcal{G}$, created as described in the last section. Such a path can be represented by its start point q_I, its end point q_G and its turning points, i.e. nodes whose position in joint space is not collinear with the position of its predecessor and its successor. The turning points of the resulting path are then used as the via points. These via points should ideally lead to collision-free and "smooth" trajectories. Therefore, we are looking for a path that is not necessary the shortest. In addition, the path must have a small number of turns. If many changes of direction occur (Figure 2.7a illustrates such a case), then the second step will be more complicated and could misbehave (see Landry et al. [60]). To find such a path, we use an adaptation of classical roadmap methods such as Dijkstra's algorithm. To this end, the workspace is covered by a regular grid. Here, grid nodes only exist, if they do not correspond to a coordinate lying on an obstacle.
For simplicity, we only allow horizontal and vertical movements and the distance between two grid points is set to the constant δ. Since many turns may pose a problem, one solution is to calculate the shortest q_I-q_G path with the least amount of turns. This is a concept which is a common approach in many path planning problems, see for example Bohlin [11] or Maheshwari et al. [65]. This can be modeled by using a large enough turn cost M to penalize paths containing turns.

Unfortunately, even if M is chosen larger than the number of nodes in the grid, it is not sufficient to modify Dijkstra's algorithm to just add the corresponding turn cost in the extension step: Dijkstra's algorithm only keeps the shortest distance corresponding to one subpath per node. In the turn cost setting, however, there might be different subpaths and each leading to different turn costs depending on the next arc. Therefore, the optimality would no longer be guaranteed.



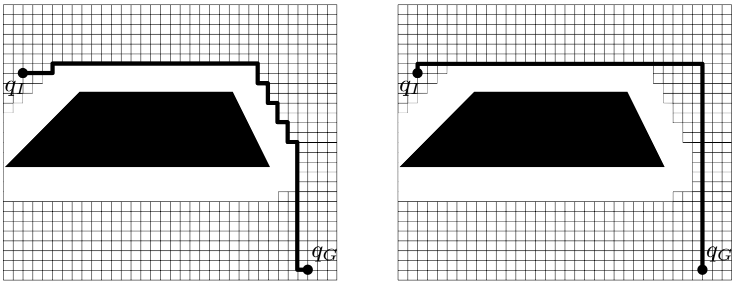
(a) Grid graph **(b)** Node expansion **(c)** Pseudo dual graph

Figure 2.6: Transformations for handling turn costs

The easiest way to deal with such turn minimal paths is a node expansion: One original grid node is split up into four nodes, one for every possible direction. These nodes are then connected in such a way, that edges corresponding to turns have cost M and the other edges have cost 0. This blows up the size of graph by a factor of four, but Dijkstra's algorithm can still be used on this extended graph.

Another way of representing turn costs is by a so-called pseudo-dual graph G' (see e.g. Winter [87]). In such a graph the edges of the original primal graph correspond to the nodes in G'. The arcs connecting two nodes in G' now correspond to the turns in the original graph. We can now give the respective turn-costs for these arcs. Then, an unmodified Dijkstra's algorithm on G' can be used to find the shortest path with turn costs. The turn cost transformations for a grid graph are sketched in Figure 2.6.

Figure 2.7b shows the result of such a turn-minimal computation for a very simple two dimensional example with one obstacle.



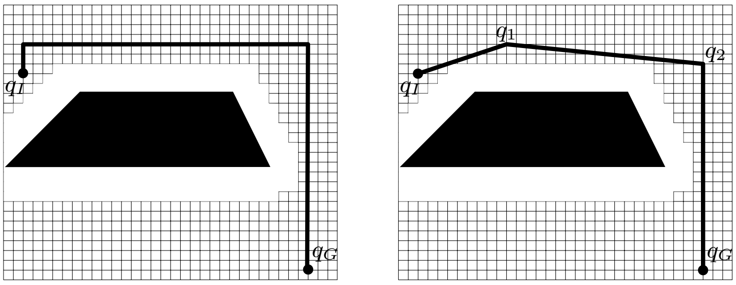
(a) **(b)**

Figure 2.7: (a) One possible path from q_I to q_G that uses the least amount of edges in the grid. (b) This path uses the same amount of edges, i.e. has the same length, as the path in (a), but contains only two turns

By definition, all paths on the grid are feasible. All nodes that could possibly conflict with any obstacle are not included in the graph. So far, we only performed shortest path computations. Due to the nature of such paths, trajectories are often favored which pass by an obstacle as close as possible. However, these parts of the path make it harder for the exact trajectory computation. In some situations it would be much better to take a path that is slightly longer, but on the other hand keeps a larger distance to the obstacles. This can be achieved by adding specific costs to the nodes. These costs should depend on the distance of a node to its nearest obstacle and should drastically decrease with increasing obstacle distance.

However, in this context it makes no longer sense to find the shortest path with least turns. We want to explicitly allow some additional turns, if that helps to avoid close obstacles. Thus, we introduce the concept of *turn costs* to penalize paths with a higher amount of turns without forbidding them explicitly.

The shortest path problem with turn costs can be solved using the presented graph modifications and Dijkstra's algorithm. Turn costs together with the distance depended node penalties now give us good options to alter the resulting path in such a way that it is balanced and will most probably also lead to a good starting solution.



(a) **(b)**

Figure 2.8: (a) Path that avoids close obstacles without taking to many turns. However, his path is slightly longer than the length of the shortest path in Figure 2.7b Smoothened version of the path in (a)

So far, the resulting path only contains right angles, which is also not very well suited for future trajectory calculations. It is necessary to smoothen the path even further. One approach, that showed significant improvements in practical experiments, is the following: Starting in q_I we move along the so far calculated path and check for each node v_i on this path, whether the direct motion from the point corresponding to q_I to the point corresponding to v_i collides with the obstacles. When a node v_i was reached where such a collision occurs, we remove all turning points between q_I and v_i and replace them with v_{i-1}. We then continue along the original path further until we find a node

v_j where the motion form v_{i-1} to v_j is not feasible. In this case we again remove all the turning points between v_{i-1} and v_j by replacing them with v_{j-1}. See Figure 2.8b for an example of such a smoothened path. The inner nodes of the smoothened path are the via points that will then be used in the next sub-section.

Applying this smoothening process we do no longer explicitly avoid trajectories which are too close to the obstacles. However, by using the nodes of a shortest path that takes this distance into account, we assure that at least the start and end nodes of the line segments are farther away. This fact leads to sufficiently good trajectories

To speed up the path computation process the grid is created dynamically: The node obstacle penalties are only calculated when the specific node is visited for the first time in the shortest path calculation. There is also no need to generate and store the edges in advance, since they are implicitly given by the node coordinates.

The idea presented in this section represents an heuristic approach that leads to good initial trajectories. All of its aspects, such as turn cost, obstacle clearance and smoothening, are very reasonable as they represent properties of a good initial solution.

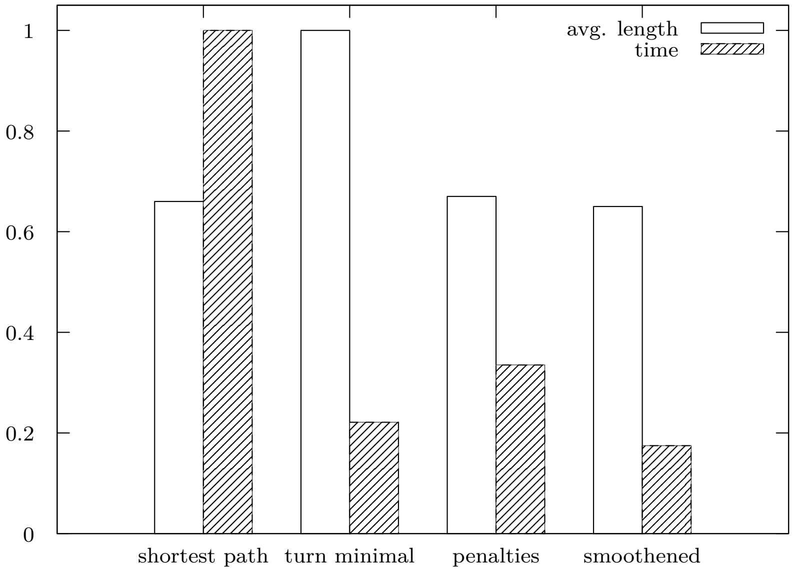


Figure 2.9: Comparison of initial trajectories

To further justify these aspects we give in Figure 2.9 a short overview how these steps improve the solution quality and computation speed of the trajectory computations. Here, the trajectories between all possible pairs of positions in the first small instance 4.1 on page 72 have been calculated. In Figure 2.9 the average length of the trajectories as well as the total computation time are visualized. They have been normalized so that the largest values correspond to a value of one in this diagram. Even for this small and simple instance we can see, that the approach combining turn costs, obstacle penalties and smoothening leads to the best solutions in the shortest amount of time.

Chapter 3

Pricing – Shortest Path Problem with Forced Time Windows

Column generation is a state-of-the-art method for optimally solving most vehicle routing and crew scheduling problems. Their master problem usually consists of a generalized set covering formulation where the generated variables consist of routes, schedules or combinations of both. In these cases the subproblem corresponds to a *Shortest Path Problem with Resource Constraints (SPPRC)*. The SPPRC provides a very general framework for finding a shortest path that satisfies certain constraints with respect to resource consumption and path-structural constraints. In this context a resource is a quantity that is consumed along a path, while the resource constraints limit the feasible values of this resource throughout the path. Typical examples of resources include time, cost or load. Although the exact structure of the subproblem greatly varies for different applications, the concept of SPPRC is designed with the intention in mind to provide a basis for as many of these subproblems as possible. However, this generality comes at the price of the lack of a precise algorithm that can be used to solve the problem. Some techniques, such as the label setting and label correcting algorithms, can be used as a framework, but then still require a fair amount of adaption for the individual problem. A great overview of the general concept of resource constrained shortest paths and their solution approaches can be found in Irnich and Desaulniers [53].

Also the column generation approach for the discrete Welding Cell Problem described in Chapter 2 is no exception here: For the underlying pricing problem it is required to find shortest paths with respect to arc costs by also taking certain time related constraints into account. One key component of the problem are the given time windows in which certain arcs *must* be used exclusively, i.e. they are forced.
Since the correctness and performance of the pricing problem is a crucial aspect of the entire column generation approach, we analyze the pricing problem in the context of the SPPRC by providing efficient problem dependent approaches and implementations for the pricing.

In this chapter we characterize and solve the pricing problem of the WCP in terms of an SPPRC. In the first section we compare the problem with other well-known variations of resource constrained shortest path problems. In particular, the differences to the well-studied *Shortest Path Problem with Time Windows (SPPTW)* are discussed. To deal with these difficulties, the *Shortest Path Problem with Strict Time Windows (SPPSTW)* is introduced and a transformation of the pricing problem is given.

In Section 3.2 we present an algorithm for the SPPSTW based on dynamic programming. This algorithm fundamentally differs from the approaches used for the SPPTW or other similar problems due to its special time window requirements. By introducing the concept of so-called *gridlocks* in Section 3.3, these differences can now be formalized and even eliminated. In the next section some of the most relevant implementation aspects are discussed before a summary of the computational results is given in Section 3.5.

3.1 Classification of the Pricing Problem

The pricing problem, which was introduced as the minimization problem (2.2) in Section 2.2, is solved separately for each robot. Hereby, it consists of the following:

For a given directed graph we have arc costs together with node prices λ_v, corresponding to the values of the dual variables, and integer travel times denoted by τ_a. For simplicity we combine the costs and the prices into one arc cost $c_{(u,v)}$ by subtracting the price λ_v of the target node v.

Further, due to branching constraints, some arcs are forced or forbidden for certain time windows, while waiting is only allowed on the arcs and not in nodes. The actual pricing problem consist of finding elementary scheduled tours that are feasible with respect to the given time windows and have negative cost or show that no such tours exists.

Before we can formally define the combinatorial problem which represents the pricing problem, we first have to clarify what exactly a scheduled path is. This definition extends the concept of scheduled tours, introduced in Section 2.2, by also including general and not elementary paths.

Definition 3.1.1 (Scheduled path)

A *scheduled path* is a pair (P, I). Here, $P = (a_1, a_2, \dots, a_p)$ is a path in the underlying graph and I is a schedule that assigns each of the arcs $a \in P$ a left-closed interval whose length is at least τ_a, i.e. I_i is the interval in which arc a_i is used. Thus, $\min(\mathcal{I}_i)$ corresponds to the departure time in the tail of a_i.

The intervals must further have the following properties:

1. They must not intersect
2. and there must not be any gap between two successive intervals, i.e. if a_i and a_{i+1} are two successive arcs along the path and x is contained in I_i and $y \in I_{i+1}$, then

every $\theta \in [x, y]$ must be either contained in I_i or in I_{i+1}.
If the path is elementary, the interval corresponding to arc a can also be denoted by I_a.

As there cannot be any gaps between the intervals, waiting in the nodes is not allowed. Scheduled tours are a special case of scheduled paths, it is therefore possible to interpret Example 2.2.1 on page 16 as an example for scheduled paths.
In the case of the discrete WCP the travel times are always integral and positive. Thus, also the scheduled paths should only have integer intervals. We further assume that the intervals are always represented as closed intervals $[x, y] \subset \mathbb{N}_{\geq 0}$ (as the ground set only contains natural numbers, this is always possible). If an arc has the travel time $\tau_a \geq 1$, then the length of the corresponding interval must be at least $\tau_a - 1$.
Let (P, I) be a scheduled path, the lower bound of the first interval I_1 is called *start time* of the scheduled path (P, I), while the upper bound of the last interval I_p is called *end time*.

We can now use this definition of scheduled paths to give the *Elementary Shortest Path Problem with Forced Time Windows (ESPPFTW)*, which represents the combinatorial optimization problem corresponding to the pricing problem:

Definition 3.1.2 (ESPPFTW)
Let $G = (V, A)$ be a directed graph with a source node s and a destination node t. Each arc $a \in A$ has a positive duration $\tau_a \in \mathbb{N}_{\geq 1}$ corresponding to the travel time and a cost $c_a \in \mathbb{Q}$. Further, each arc has a (possibly empty) forced time interval F_a and a set of time windows $\mathcal{T}_a$.
The goal is to compute a shortest elementary scheduled path (P, I) (w.r.t. the costs c_a) with the following properties:

- For every arc $a \in P$ the corresponding interval I_a lies in one of the free time windows given by $\mathcal{T}_a$.
- Every arc $a \in P$ is used at least during F_a, i.e. $F_a \subseteq I_a$ holds.

The pricing problem now consists of solving an instance of the ESPPFTW, where F and $\mathcal{T}$ encode the current branching decisions as well as the time limit T^{limit}, to find the minimum cost scheduled s^r-s^r path. If the cost of this path is negative, the corresponding column is added to the problem.

The ESPPFTW is related to the well-studied *Shortest Path Problem with Time Windows (SPPTW)*. It was first introduced in Desrosiers et al. [21] and can be described as follows:
Let $G = (V, A)$ be a directed graph with source node s and sink node t. Each node $v \in V$ has a time window $[x_v, y_v]$ when the node v can be visited. Each arc $a \in A$ has a positive duration τ_a corresponding to the travel time and a cost c_a.

The objective of the SPPTW is to find the minimum cost path between s and t while respecting the time window condition in all the visited nodes, i.e. each node is visited within the given time window. In this model waiting in the nodes is allowed, as the visiting time at a node can be greater than the arrival time.
One of the best solution approaches for the SPTW (and also for more general shortest path problems with resource constraints containing more than one restricted resource) is the so-called *Label Setting Algorithm (LSA)*, see Irnich and Desaulniers [53] and Dumitrescu and Boland [25].
There is also a variation of the shortest path problem with time windows called the elementary SPPTW, where the task is to find elementary paths. This version was introduced by Desrochers et al. [19] as a subproblem of the vehicle routing problem with time windows and Dror [23] showed that in general, i.e. on graphs that contain negative cost cycles, the elementary SPPTW is $\mathcal{NP}$-hard in the strong sense.
Since the ESPPFTW (and thus the pricing problem) is a special case of the elementary SPPTW, it must therefore also be $\mathcal{NP}$-hard.

However in a sense, the ESPPFTW is more of an arc routing problem than the SPPTW, as time windows and also the concept of waiting are only given for arcs: Gamache et al. [37] suggested a variation of the SPPTW, where the arrival time in each node must always be identical to the visiting time. But even this variation does not quite cover our problem, where waiting is allowed but only within the given time windows.
Gawrilow et al. [39] introduced a variation of the SPPTW, which is more closely related to the ESPPFTW: They considered a problem called the *Quickest Path Problem with Time Windows (QPPTW)*, which is exactly based on time windows and waiting only on the arcs, but they consider the cost of a path to be the sum of the transit times plus the waiting times. This assumption then makes the problem considerably easier. For the ESPPFTW the costs are explicitly given and also some of the arcs can be forced.
Nevertheless, their proposed algorithm can still be used as a basis for the following problem, which we call the *Shortest Path Problem with Strict Time Windows (SPPSTW)*:

Definition 3.1.3 (Shortest path problem with strict time windows)
Let $G = (V, A)$ be a directed graph with a source node s and a destination node t and let I_s be an interval of allowed start times in s. Each arc $a \in A$ has a positive duration τ_a corresponding to the travel time, a cost c_a and a set of time windows $\mathcal{F}_a$. The goal is to compute a shortest scheduled path (with respect to the arc costs c_a) that starts at a time step contained in I_s and respects the given time windows, i.e. for every arc a in the path the corresponding interval I_a must lie in one of the free time windows given by $\mathcal{F}_a$.

We always assume that each of the feasible time windows in $\mathcal{F}_a$ are large enough to support the travel time of arc a. Smaller time windows will never be used and can thus be discarded.

The Figure 3.1 shows an example of such an SPPSTW instance. Here, all arcs have the same travel time $\tau_a = 1$ and their cost c_a and time windows $\mathcal{F}_a$ (green) are given as shown. All resulting feasible scheduled s-t paths in this instance are visualized by their cost and end time.

In this instance, the only feasible scheduled path with minimum cost uses s-b-t and has a cost of 1 and an end time of 4. The path s-a-b-t is not feasible, as the arc (b, t) can only be used after the path (a, b) is already forbidden and, thus, no scheduled path along these nodes exists that stays within the feasible time windows. There are two feasible schedules for the path s-a-t, but this path has a cost of 3.

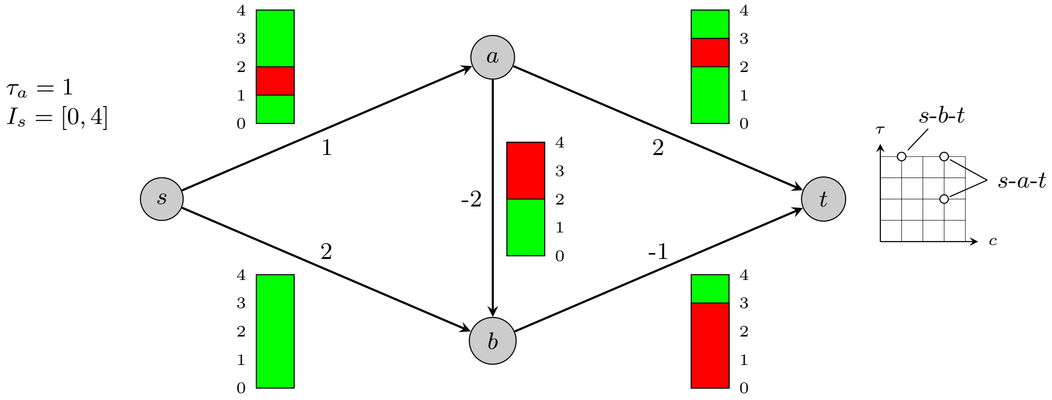


Figure 3.1: Example of an SPPSTW instance

In the presented definition of the SPPSTW only feasibility intervals are given. They directly correspond to the time windows $\mathcal{T}_a$ as they occur in the ESPPFTW. Forcing of arcs at certain time windows, however, is not explicitly possible. Fortunately, this can be integrated:

Proposition 3.1.4 *For any instance $\mathcal{I}_{forced}$ of the ESPFTW there exists an instance $\mathcal{I}_{strict}$ of the elementary SPPSTW, such that*

1. *any feasible solution of $\mathcal{I}_{strict}$ corresponds to a solution of $\mathcal{I}_{forced}$ with the same cost and*
2. *$\mathcal{I}_{forced}$ is infeasible if and only if $\mathcal{I}_{strict}$ is infeasible.*

PROOF:
Given an instance of the ESPPFTW we construct an SPPSTW instance, such that its optimal elementary solution corresponds to an optimal solution of the original instance. To do this, we have to represent the forced time intervals F_a of the ESPPFTW instance as feasibility time windows $\mathcal{F}_a$.

In the following we only show how a single forced interval can be incorporated. This

approach, however, can analogously be extended for more than one forced arc.
So, let $F_{\bar{a}} = [x_{\bar{a}}, y_{\bar{a}}]$ be the nonempty forced interval of arc $\bar{a}$. This can be modeled by using the following time windows:

$$
\begin{aligned}
I_s &:= [0, x_{\bar{a}}] && && \text{(3.1a)}\\
\mathcal{F}_a &:= \{W \setminus [0, y_{\bar{a}}] : W \in \mathcal{T}_a\} && \forall a \in \delta^+(\text{head}(\bar{a})) && \text{(3.1b)}\\
\mathcal{F}_a &:= \{W \setminus [0, y_{\bar{a}}] : W \in \mathcal{T}_a\} && \forall a \in \delta^-(t) \setminus \{\bar{a}\} && \text{(3.1c)}\\
\mathcal{F}_{(u,v)} &:= \left\{W \setminus [x_{\bar{a}}, y_{\bar{a}}] : W \in \mathcal{T}_{(u,v)}\right\} && \forall (u,v) \in A \setminus \{\bar{a}\}, v \neq t && \text{(3.1d)}\\
\mathcal{F}_{\bar{a}} &:= \{W \setminus [0, \theta] : W \in \mathcal{T}_{\bar{a}}\}\ , && && \text{(3.1e)}
\end{aligned}
$$

where θ is the last forbidden time step in $\mathcal{T}_{\bar{a}}$ with $\theta < x_{\bar{a}}$.

To prove that this is correct, we show the two following facts:

1. Every elementary scheduled path (P, I) that respects the time windows $\mathcal{T}$ and where the arc $\bar{a}$ is in use during $I_{\bar{a}} \supseteq F_{\bar{a}}$ does not violate any of the constraints (3.1a) - (3.1e).
2. Every feasible solution of the SPPSTW with the transformed $\mathcal{F}$ and I_s uses the arc $\bar{a}$ in the time interval $[x_{\bar{a}}, y_{\bar{a}}]$.

1. Let (P, I) be such a feasible elementary scheduled path respecting the time windows. We know from the fact that arc $\bar{a}$ is in use during $[x_{\bar{a}}, y_{\bar{a}}]$ that no other arc is used during this time period (3.1d) and that the start time of (P, I) is at most $x_{\bar{a}}$, while the end time is at least $y_{\bar{a}}$. Since also no node in P is visited more than once, the scheduled path (P, I) also respects the intervals given in (3.1a) and (3.1c). Further, the arc $\bar{a}$ is used exactly once and it is therefore not possible that it is visited before θ (3.1e) and all outgoing arcs can only be used after $\bar{a}$ (3.1b).

2. Every scheduled path that respects the previously defined $\mathcal{F}$ and I_s cannot not use any arc $a \in A \setminus \{\bar{a}\}$ during $[x_{\bar{a}}, y_{\bar{a}}]$. So, the only two possibilities remaining where $\bar{a}$ is not used during $[x_{\bar{a}}, y_{\bar{a}}]$ are that either s is left after $y_{\bar{a}}$ or that t is reached before $x_{\bar{a}}$. The first situation cannot occur as this would violate the give start times I_s. If t is reached before $x_{\bar{a}}$, this means that, due to (3.1c), the arc $\bar{a}$ must be the last used arc in P. However, in this case we know that between the end time of P and $x_{\bar{a}}$ the arc $\bar{a}$ is allowed, due to (3.1e). Thus, by increasing the upper bound of $I_{\bar{a}}$ to $y_{\bar{a}}$ we get a new scheduled tour that has the same cost but now also respects the forced time interval $F_{\bar{a}}$. ■

The proof of Proposition 3.1.4 also shows that an optimal solution of the constructed elementary SPPSTW can be transformed into an optimal solution of the ESPPFTW in $\mathcal{O}(1)$. It is therefore possible to solve the ESPPFTW by solving an instance of the elementary SPPSTW.

Remark: Another approach for solving the ESPPFTW is to add an additional binary resource for every forced arc. Initially all these resources are set to 1 and they are reduced to 0 only when the corresponding forced arc a is visited *within* its given time interval F_a. With these resources we assured that no path that has not yet visited the forced arc in the right time interval dominates a path that has. In this approach (3.1a) and (3.1d) are the only modifications needed. But since the addition of further resources makes the extension step more complicated and the domination weaker, we use the approach as described in Proposition 3.1.4 in our implementation.

3.2 Solving the ESPPFTW

In the last section we showed that the ESPPFTW can be solved as an elementary SPPSTW. In this section we, therefore, focus on how the SPPSTW can be solved as efficiently as possible.

Although the SPPSTW cannot be directly solved using the framework of the so-called generalized *Label Setting Algorithm (LSA)* described in Irnich and Desaulniers [53], it still offers a good foundation that we will later extend to incorporate the SPPSTW:
The generalized LSA can be used for many shortest path problems with resource constraints, it is described in Algorithm 3 and its basic idea is the following:
The algorithm uses a set of labels for each node. Each of these labels represents a path from the source s to that respective node. It consists of a vector of values, that represent the consumption of each of the resources along that particular path.
Starting with a trivial label representing the initial consumption in s (*initialization step*), the algorithm processes each label by extending the corresponding path along all outgoing arcs (*extension step*). The algorithm only stores efficient labels for every node, i.e. labels that do not lead to feasible or optimal s-t-paths can be discarded (*dominance step*).
The algorithm terminates when all labels have been processed and then the labels in t correspond to optimal s-t-paths.

By combining the algorithm for the QPPTW [39] with the generalized LSA as described in Algorithm 3 we can now construct an algorithm for the SPPSTW:
A label $L = (c_L, I_L)$ in the node v_L consists of the cost value c_L and a time interval I_L. Each label L represents a scheduled subpath from s to v_L with the corresponding cost of c_L. The Interval $I_L = [x_L, y_L]$ represents the interval of possible start times for the next arc leaving v_L. We further assume that it is always possible to reconstruct the corresponding path from a label L. The additional information for such a reconstruction (e.g., predecessor label, previous arc) is not stated as explicit properties of L. In each extension step we try to extend the current label L along the corresponding arc a for all

Algorithm 3: Generalized label setting algorithm

```
Input: an SPPRC instance
Output: shortest $s$-$t$ paths respecting all resource constraints
// initialization step:
create initial label $L_0$ in $s$;
$\mathcal{U} \leftarrow \{(L_0)\}$;
$\mathcal{P}_s \leftarrow \{(L_0)\}$;
foreach $v \in V \setminus \{s\}$ do
    $\mathcal{P}_v \leftarrow \varnothing$;
while $\mathcal{U} \neq \varnothing$ do
    choose label $L \in \mathcal{U}$;
    $v \leftarrow$ the node of the label $L$;
    $\mathcal{U} \leftarrow \mathcal{U} \setminus \{L\}$;
    // dominance step:
    remove all dominated labels from $\mathcal{P}_v$;
    // extension step:
    forall the arcs $a \in \delta^+(v)$ do
        $L_{new} \leftarrow$ the label $L$ extended over the arc $a$;
        if $L_{new}$ is feasible then
            $\mathcal{U} \leftarrow \mathcal{U} \cup (L_{new})$;
            add $L_{new}$ to the set $\mathcal{P}_w$ corresponding to its end node $w$;
if $\mathcal{P}_t \neq \varnothing$ then
    return paths corresponding to min-cost paths in $\mathcal{P}_t$;
else
    return $\varnothing$;
```

of the time windows on this arc. The resulting label L_{new} has the cost of $c_L + c_a$ and its interval is chosen as large as possible.
The dominance is then defined as follows:
We say label L dominates another label L' if and only if

$$v_L = v_{L'} \quad \text{and} \quad c_L \leq c_{L'} \quad \text{and} \quad I_{L'} \subseteq I_L\,. \tag{3.2}$$

This leads to the algorithm described in Algorithm 4.

Such an algorithm is correct and can be used to solve the SPPSTW for arbitrary arc

Algorithm 4: Label setting algorithm for the SPPSTW

```
    Input: data for the SPPSTW instance
    Output: shortest $s$-$t$ paths respecting the time windows
 1  create initial label $L_0 = (0, I_s)$ in $s$;
 2  $\mathcal{U} \leftarrow \{(L_0)\}$;
 3  while $\mathcal{U} \neq \varnothing$ do
 4      choose label $L = (c_L, [x_L, y_L]) \in \mathcal{U}$ in node $v_L$;
 5      $\mathcal{U} \leftarrow \mathcal{U} \setminus \{L\}$;
 6      foreach arcs $a = (v_L, w) \in \delta^+(v_L)$ do
 7          foreach time window $W_a = [x, y] \in \mathcal{F}_a$ do
 8              if there is a $\theta \in [x_L, y_L]$ such that $[\theta, \theta + \tau_a - 1] \subseteq W_a$ then
 9                  $I_{new} \leftarrow [x_L + \tau_a, \infty) \cap [x + \tau_a, y + 1]$;
10                  $L_{new} \leftarrow (c_L + c_a, I_{new})$;
11                  if there is no $L' \in \mathcal{U}$ that dominates $L_{new}$ then
12                      $\mathcal{U} \leftarrow \mathcal{U} \cup \{L_{new}\}$;
13                      remove all labels from $\mathcal{U}$ dominated by $L_{new}$;
14  if there is a label in the node $t$ then
15      return corresponding paths of min-cost labels in $t$;
16  return $\varnothing$;
```

costs.

Proposition 3.2.1 *The Algorithm 4 together with the dominance rule* (3.2) *finds a path with minimal cost for the SPPSTW.*

PROOF:
First, we note that in every extension step of a label the lower bound of its interval of feasible starting times strictly increases. Therefore, since every time window in $\mathcal{F}$ is finite, the algorithm will eventually terminate.
To show the correctness of the algorithm, we have to show

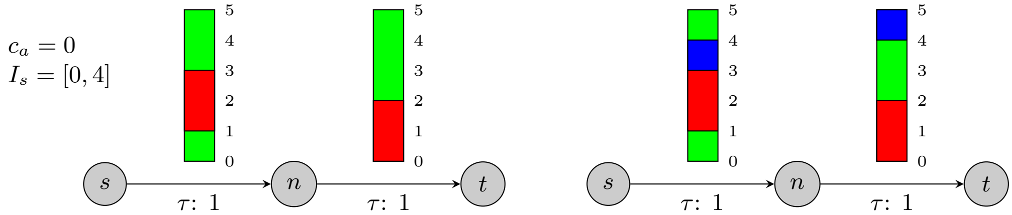
1. that labels representing all possible subpaths are generated and
2. that only labels are dominated which cannot lead to an optimal solution.

In line 8 of Algorithm 4 we check, if the label L can be extended along arc a in the feasibility interval W_a. For the actual extension we have to calculate new values for the label: The new starting time in the tail of a must be at least $x_L + \tau_a$, as we cannot start before x_L and traversing of a takes τ_a. But we cannot use the arc a longer than

W_a permits, which leads to the latest possible start time for the next arc of $y+1$. Thus, L_{new} contains the largest possible interval of start times resulting from W_a. Since this step is repeated for all $W_a \in \mathcal{F}_a$, all feasible extensions are generated. Therefore, Algorithm 4 without the dominance step computes all feasible scheduled paths (represented in a condensed version).

It remains to show that no optimal subpath (label) is dominated. This is true, as from $I_{L'} \subseteq I_L$ it follows that $I_{L'_{new}} \subseteq I_{L_{new}}$ holds. Also for the cost we have that $c_L \leq c_{L'}$ leads to new labels with $c_{L_{new}} \leq c_{L'_{new}}$. ■

Remark: It is in general not sufficient, to just create one new label for the next feasible time window in $\mathcal{F}_a$:
Consider the situation as described in Figure 3.2a, where the feasible time intervals are marked green. For this example we only need to consider the travel times and the time intervals but not the costs of the arcs. The interval of start times is $I_s = [0, 4]$. If Algorithm 4 only creates the earliest feasible label in every extension step, then only the label L in the intermediate node with the interval $[1, 1]$ would be generated. This label L, however, cannot be extended over the next arc. Thus, the algorithm would not return a feasible solution, although this instance has a feasible s-t path as visualized in blue in Figure 3.2b.
We call this situation, in which a feasible subpath/label cannot be extended along an arc, although another label arriving later can, *gridlock*.
The problem of gridlocks is special for the SPPSTW as it can only occur, if also the waiting is limited by time windows. Gridlocks, especially in the context of the pricing problem, are further discussed in Section 3.3.



(a) SPPSTW instance **(b)** Feasible scheduled path

Figure 3.2: Gridlock example

By combining Proposition 3.1.4 and Proposition 3.2.1 we are now able to solve the pricing problem, if there are no negative costs cycles in G.
Unfortunately, due to the structure of the underlying LP negative costs and negative cost

cycle will occur. It is therefore necessary to modify our approach so that only elementary paths are generated.

For the elementary shortest path problem Beasley and Christofides [7] proposed to add an additional binary resource to every node. Visiting a node in a path increases the corresponding resource by one and since it has an upper limit of one, nodes cannot be visited more than once. This approach allows us to solve the *elementary SPPSTW (ESPPSTW)* in the general setting of shortest paths with resource constraints, but it also drastically weakens the dominance step, as now many more labels have to be kept in every node.
To incorporate this in Algorithm 4, a label $L = (c_L, I_L, \mathbf{R}_L)$ now also contains the vector $\mathbf{R}_L \in \{0,1\}^{|V|}$, which contains a one for each of the nodes previously visited by L. Of course, this vector $\mathbf{R}_L$ must also be taken into account for the dominance:
A label L dominates a label L', if and only if

$$v_L = v_{L'} \quad \text{and} \quad c_L \leq c_{L'} \quad \text{and} \quad I_{L'} \subseteq I_L \quad \text{and} \quad \mathbf{R}_L \leqq \mathbf{R}_{L'}\,.$$

As one can see, this drastically weakens the dominance step, since now not nearly as many labels can be discarded. However, due to the time windows and the costs, some nodes might never be visited more than once and introducing a resource for them is therefore not necessary. One approach to dynamically detect which nodes might be critical and which nodes are not, is the so-called *decremental state-space relaxation* (DSSR) as introduced by Righini and Salani [73].

Instead of solving the ESPPSTW we only solve the Θ-*ESPPSTW* where Θ is a set of nodes. The Θ-ESPPSTW is the problem of finding a (not necessary elementary) solution to the SPPSTW, where every node in Θ is only visited at most once. (The nodes in the set $V \setminus \Theta$ can still be visited more than once.) The Θ-ESPPSTW can be modeled by adding the binary variables explained in the last paragraph only for the nodes $v \in \Theta$.
The idea is described in Algorithm 5 and by starting with Θ being equal to the empty set, we solve the Θ-ESPPSTW. If one of the paths that have minimal cost is elementary, then we are done as this path also is an optimal solution to the ESPPSTW. If all min-cost paths contain cycles, we have to to add some of the vertices visited more than once to Θ and thereby tightening the relaxation provided by Θ-ESPPSTW. When Θ contains the entire set of vertices, we solve the exact ESPPSTW and therefore Algorithm 5 will terminate eventually. However, in most cases, it is sufficient to solve the Θ-ESPPSTW for a Θ that contains considerably fewer nodes. Therefore, the DSSR approach is usually much faster, than solving the ESPPSTW directly [12].

One important remaining question is what nodes should be chosen for the inclusion in the set Θ. This is obviously an important question for the performance of the algorithm:

Algorithm 5: Solving the ESPPSTW with decremental state-space relaxation

```
Input: an ESPPSTW instance
Output: a set of resource feasible elementary paths with negative reduced costs,
        $\varnothing$ if no such path exists
$\Theta \leftarrow \varnothing$;
while true do
    let $\mathcal{P}$ be the paths returned as solution of the $\Theta$-ESPPSTW;
    if $\mathcal{P}$ contains only paths that have a cycle then
        // determine critical nodes
        $\Theta \leftarrow \Theta \cup$ MultipleVisits($\mathcal{P}$);
    else
        return elementary paths in $\mathcal{P}$;
```

If we make Θ unnecessarily big, then the computation time in each iteration will increase. If, on the other hand, only one vertex is added in every iteration, then this might lead to many iterations which is also a negative factor for the overall running time. Several possible strategies for `MultipleVisits` exist, the approach used in our implementation is explained in Section 3.4.

The concept of DSSR can also easily combined with 2-cycle elimination:
Typically, when solving the SPPSTW there will be a lot of paths cycling between two “good” vertices. Of course, it is possible to eliminate those cycles using the DSSR-approach. But cycles of length two can be detected and avoided by a modified dominance step. Larsen [62] showed a dominance rule that does not eliminate potentially optimal paths, even if in the extension step 2-cycle are never generated. This, however, comes at the cost that now twice as many labels are necessary. Fortunately, this approach is still faster then having more iterations in the DSSR.

3.3 Dealing with Gridlocks

As we saw in Figure 3.2a, situations exist in which a feasible label cannot be extended along the next arc as this would introduce additional waiting times not allowed for the given time windows. Those situations are prevented in Algorithm 4 by generating new labels for all the time windows in $\mathcal{F}_a$. In this section we present a different approach that introduces additional dummy arcs to resolve gridlocks. To show that this approach is correct, we first formally define gridlocks:

Definition 3.3.1 (Gridlock)
For a given SPPSTW instance a *gridlock* is a situation in which there exist two feasible scheduled subpaths (P, I) and (P, I'), both corresponding to the same path (but having different intervals) with the following properties:

- (P, I') has a strictly higher end time than (P, I)
- and (P, I') can be extended along an adjacent arc a' into a feasible s-t path, while (P, I) cannot be extended along a', i.e. it cannot wait on its previous arc long enough.

It now follows that for SPPSTW instances without gridlocks no more than one label needs to be generated in the extension step of Algorithm 4:

Lemma 3.3.2 *In an SPPSTW instances where no gridlocks can occur, the Algorithm 4 can be modified to exit the extension step after one feasible label L_{new} has been found and remains correct.*

PROOF:
In a situation where the second label is needed, there exists a label L whose feasible extension L^1 along W_a^1 cannot lead to an optimal s-t path, while the extension of L along a later time window W_a^2 on the same arc is part of an optimal solution. Let P be the path corresponding to this optimal solution.
We know that there must be an arc $a \in P$ that cannot be used by any extension of label L^1. Let a' denote the first of these arcs. This means that we have a gridlock in a', since the extension of L^1 arrives earlier but cannot be extended along a'. ■

Reducing the number of labels during the execution of an LSA is vital for its performance. Therefore, analyzing the circumstances under which gridlocks occur and avoiding the generation of labels, when they are not necessary will improve the performance of the shortest path computation. Next, we give a very simple necessary condition to check for the existence of gridlocks. This condition should be easy to verify, so that it can be efficiently implemented, but it should still be as precise as possible, so that "false positives" do not occur too often. Such a necessary condition is given in the following:

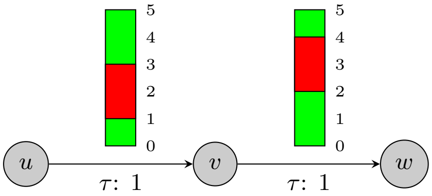
Proposition 3.3.3 *For a gridlock to occur in an SPPSTW instance, it is necessary that there are two adjacent arcs $a = (u, v)$ and $a' = (v, w)$ with the following properties:*

1. *There exists a $\theta \in [x, y]$ with $\theta \notin \mathcal{F}_a$ and $\theta \notin \mathcal{F}_{a'}$, where x is the earliest time at which node v can be reached and y is the last time step at which arc a can be used, i.e. $x \in \mathcal{F}_a$.*
2. *There exists a feasible scheduled s-t path that uses both a and a'.*

Proof:
A gridlock occurs for arc a, if there exists a scheduled subpath P that reaches node v at time step θ_1 and there also is a different scheduled subpath P' at time step $\theta_2 > \theta_1$ where path P' can be extended along a' while P cannot. In order for that to be true, arc a' must be forbidden for time step θ_1 and waiting on the arc a must not be allowed at some point θ before a' is allowed again and after θ_1, i.e. at time step θ both arcs a and a' are forbidden. And since P' can be extended, $\theta < \theta_2$ must hold.
As P as well as P' both use the arc a, we get $x \leq \theta_1 \leq \theta < \theta_2 \leq y + 1$.
By the definition of a gridlock there must also exist a feasible scheduled path that uses arc a as well as a'. ■

This proposition also shows that for SPPSTW instances where $\mathcal{F}_a$ contains at most one time window for each arc, not more than one extension along the same arc of a label needs to be stored. This is not very surprising as the problem in this case corresponds to a simple SPPTW, which can be solved by storing only the best possible start time in the labels (see e.g., Desrosiers et al. [21]). If Algorithm 4 is performed on such instances it will also never generate more than one label per extension step.

Remark: The Proposition 3.3.3 only states a very simple necessity, but this condition is definitely not sufficient for the existence of a gridlock. Consider for example the situation as described in the following picture:



Here, the situation as defined in Proposition 3.3.3 occurs, but still there is no gridlock. Nevertheless, Proposition 3.3.3 gives us an easy to verify criterion to identify possible arc pairs which could lead to gridlocks.

However, we also learn from Proposition 3.3.3 that a gridlock is a local problem, in the sense that it can only occur if there are two critical adjacent arcs.
So what can be done to handle gridlocks without keeping the labels for each time window? There are two possibilities:

1. Modify the dominance rule to the following:
   A label L dominates a label L' if and only if
   a) $v_L = v_{L'}$,

   b) $c_L \leq c_{L'}$
   c) and $I_{L'} \subseteq I_L$ if node v has two critical arcs as described in Proposition 3.3.3, or $\inf(I_L) \leq \inf(I_{L'})$ otherwise.
2. Resolve all possible gridlock situation by modifying the underlying graph

It is easy to see that the first option is indeed correct and only labels that do not lead to better solutions are dominated. From a practical standpoint this is sufficient for the implementation of an algorithm that (at the cost of more expensive dominance checks) requires fewer labels than Algorithm 4. However, in the remainder of this section we focus on the second option as it allows us to solve the SPPSTW in the framework of an unmodified generalized LSA, very similar to the classical SPPTW:
After such pair of critical adjacent arcs $a = (u, v)$ and $a' = (v, w)$ has been identified, the situation can be resolved by creating a dummy arc d from u to w. This arc gets the resource consumption of a plus a' in all resources and $\mathcal{F}_d = \mathcal{F}_a \cap \mathcal{F}_{a'}$. Figure 3.3 shows how the gridlock arising in the example in Figure 3.2 can be avoided by introducing such a dummy arc.

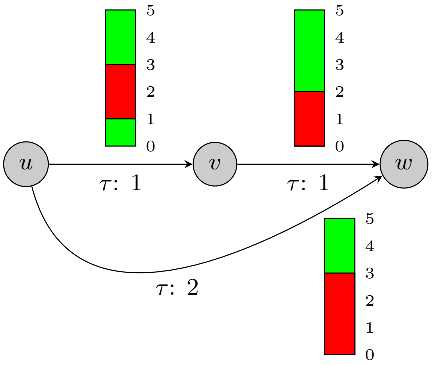


Figure 3.3: Dummy arc to resolve gridlocks

The dummy arc d basically represents using a and a' successively, but prevents the gridlock situation as it tracks the waiting back to node u. A scheduled path using the arc d can be easily transformed into a path in the original graph, by using the arcs a and a' in the corresponding interval instead.
Unfortunately, it could be that this newly added dummy arc now introduces a new gridlock situation in node u. Therefore, to remove all possible gridlocks one has to iteratively add dummy arcs until the situation as described in Proposition 3.3.3 no longer occurs for arc pairs without a dummy arc.

The number of arcs added in this approach could get very big. It is therefore not suited for all SPPSTW instances. Let us, for example, consider the instances resulting from the transformation of an ESPPFTW into an SPPSTW: For each forced arc in the original instance, we get that all the other arcs are forbidden for that specific time window. Therefore, all possible pairs of these arcs are critical and dummy arcs would be created.

Algorithm 6: Solving the SPPSTW by eliminating gridlocks

```
Input: an SPPFSW instance on the graph $G = (V, A)$
Output: shortest feasible scheduled path

$d \leftarrow$ the shortest distance from $s$ to all nodes w.r.t. $\tau_a$;
foreach $a = (u, v) \in A$ do
    $\theta \leftarrow$ last allowed time step in $\mathcal{F}_a$;
    foreach outgoing arc $a' = (v, w) \in A$ do
        if $\exists x \in [d(v), \theta]$ with $x \notin \mathcal{F}_a$ and $x \notin \mathcal{F}_{a'}$ and no dummy $d = (u, w)$ then
            create new arc $d = (u, w)$ with the combined resources of $a$ and $a'$;
            restart the algorithm;
solve resulting problem with modified Algorithm 4;
```

In Algorithm 6 the SPPSTW is solved by first removing all gridlocks, so that the problem then can be solved with a modification of Algorithm 4 that only generates the first feasible label. In Algorithm 6 the shortest distance between s and v with respect to the travel times τ_a is used as a lower bound for the earliest possible arriving time in v.

Remark: If there are no gridlocks, it is possible to solve the SPPSTW with labels structured like $L = (c_L, t_L)$, where t_L corresponds to the earliest possible start time in the node v_L. In the extension step, we then have to check whether the label can be extended along arc a and whether the hereby induced waiting times on the predecessor arc are feasible. Now, this algorithm fits exactly in the framework of a generalized LSA described in Algorithm 3.

Thus, by analyzing the concept of gridlocks we were able to further reduce the number of labels in Algorithm 4 and by introducing dummy arcs, we manged to transform the SPPSTW into a problem that can now be solved using the same label structure and the same algorithm framework as for the regular SPPTW.

3.4 Implementing the Pricing Problem

As already indicated in the last section, we solve an instance of the shortest path problem with forced time windows defined in Definition 3.1.2 by transforming it into an SPPSTW instance and then using the algorithm described in Algorithm 4. However, in

our implementation a label $L = (c_L, \ell_L, I_L, \mathbf{R}_L, pred_L)$ consists of the cost c_L, the length of the path ℓ_L, i.e. the number of nodes, interval of feasible start times I_L, the resource vector $\mathbf{R}_L$ for already visited nodes and the last used arc $pred_L$, which is used for the 2-cycle elimination. We further assume that the label L also contains some reference to the node v_L to which it belongs as well as its predecessor label. But as this information does not correspond to some resource values, we did not explicitly include it as a property of the label.
Following the outline of Algorithm 4, the first implementation aspect that needs to be considered is how the next label L is chosen from the queue $\mathcal{U}$:

The Priority Queue $\mathcal{U}$

The label L that is picked in every iteration is selected with respect to the length ℓ_L of the corresponding path, i.e. preferring shorter paths. If we always pick one of the labels with shortest path length, the maximum length difference between any labels throughout the algorithm will never be more than one. Therefore, such a priority queue can be very efficiently implemented using a *bucket-based priority queue*. A bucket queue for our problem is a circular array B of two linked lists. All possible path length are now wrapped around the circular array in such a way that a label L with path length ℓ_L is stored in $B[\ell_L \bmod 2]$. This way all labels in one bucket always have the same length.
The only two operations that need to be supported by B are `Insert` and `ExtractMin`. As the buckets are implemented using linked lists, `Insert`(B, L) takes constant time. A call of `ExtractMin` looks at the first element of the list in $B[m \bmod 2]$. If this list is empty, m is incremented and the other list is checked. The variable m will never be incremented more than once per call.
In general, any value out of the cost, the length, or the feasible start times could have been used for the sorting criterion of the priority. The cost, however, is the least favorable parameter, as selecting labels with the smallest cost would prefer paths that use negative cycles. Between the other two there is no real quality difference, but since a priority queue for the start times would be much more complicated to implement, we used the length in out implementation.

Since the underlying algorithm is a label setting algorithm, we have to make sure that no label that has been extended gets dominated and thus deleted afterwards. This would lead to a situation where the resulting path could no longer be easily reconstructed from the last label. The following two facts guarantee that such a situation does not occur:

1. The length of a path is strictly increasing and since an upper bound on the length is given, the algorithm will eventually terminate.

2. By extending labels with smaller length first, we assure that all the labels that might dominate $L \in \mathcal{U}$ have already been created before L is extended.

Implementation of Dominance

In order to make the dominance step as efficient as possible, the labels for each vertex are stored in two lists, while one of them is sorted. In the sorted list the labels are stored with respect to a linear extension of $\prec$, where $L_1 \prec L_2$ holds, if L_1 is dominated by L_2. A newly generated label is first added to the unsorted list. Only if a label has been extracted from $\mathcal{U}$, we check all of the new labels in the corresponding node for dominance: For each label L of the unsorted labels, we run through the sorted list until a correct position for L has been found. Due to the property of the linear order it is now sufficient to check, whether L is dominated by any of the labels prior to L. And, if this is not the case, whether L dominates any of the labels after it. No other dominance checks are necessary. If one of the new labels is dominated, it is not discarded right away but only marked as "dominated". Dominated labels are then finally discarded and deleted from the memory when they are picked from $\mathcal{U}$.

Determining Critical Nodes

Several different approaches exist how the critical nodes in the decremental state-space relaxation (Algorithm 5) should be selected. The following approaches were also proposed in Boland et al. [12]:

HMO Highest multiplicity node on the optimal path: From all feasible *s*-*t* paths one with the lowest cost P is picked. Now, add one of the nodes to the critical set that are visited most often in P. In every iteration of DSSR exactly one node is added to the critical set.

HMO-All Highest multiplicity on the optimal path – all nodes: In contrast to HMO, *all* the nodes with highest multiplicity on one min-cost path P are added.

MO-ALL Multiplicity greater than one on the optimal path – all nodes: This strategy adds all nodes visited at least twice in the first feasible min-cost path.

All these augmentation strategies have been compared numerically in Boland et al. [12]. The authors described a clear degradation in the performance as the algorithm becomes more aggressive and adds more nodes during the state space augmentation. Therefore, the HMO strategy was adopted for our implementation.

The strategy is combined with an early termination rule: After a certain threshold of generated labels has been reached (50,000 for our tests), the min-cost label in t is evaluated whether it is elementary or not. If it contains a cycle, the solution process for this state-space relaxation can be stopped and the HMO strategy is applied for the path corresponding to this label. Although the min-cost label at that point might not correspond to the optimal solution, it can still be used for the next state space augmentation. If the threshold number is chosen large enough, it is very likely that, even after all the labels have been processed, no elementary path with minimum costs would have been generated.

Extending the Labels

The extension step consists of the following: Given a label L and an arc $a \in A$ over which we extend, create a feasible label L_{new} that corresponds to the path that is constructed by appending a to the path of L and its respective resource consumption.
Calculating the new resource consumption for the cost, length and visited nodes and checking for feasibility can be trivially done by adding the corresponding resource consumption along arc a. However, we will take a closer look how the new intervals of starting times can be computed. In Algorithm 4 we formally described the extension of the time interval of label L along arc a by using $[x_L + \tau_a, \infty) \cap [x + \tau_a, y + 1]$ for the new label, where x_L is the lower bound of the interval of L and $[x, y]$ is the considered time window.
To implement this idea as efficiently as possible, Algorithm 7 is used. This algorithm uses two functions `NextForbidden` and `NextAllowed`. Here, `NextForbidden`(a, x) returns the earliest time $\theta \geq x$ with $\theta \notin \mathcal{F}_a$ and `NextAllowed`(a, x) returns the earliest time $\theta \geq x$ with $\theta \in \mathcal{F}_a$ or ∞ if no such time window exists in $\mathcal{F}_a$. As the travel times are assumed to be always integer for the SPPSTW, these functions are well defined.

Since we assumed that there are no time windows in $\mathcal{F}_a$ that are smaller than the travel time τ_a, Algorithm 7 is indeed sufficient to generate all label with the largest possible time intervals:

Proposition 3.4.1 *Let $[x_L, y_L]$ be the start time interval of label L and let a be the arc along which we extend. The intervals returned by Algorithm 7 cover all possible start times of the next arc in* $\operatorname{tail}(a)$ *after $x_L + \tau_a$.*

PROOF:
Since $\mathcal{F}_a$ does not contain time windows having a smaller size than the time that is needed to travel along arc a, `NextStart` returns the earliest time where a could be used after label L. If this time is within the interval $[x_L, y_L]$, then we will definitely have one

Algorithm 7: Extension step for the SPPSTW

```
Input: a label $L$ with interval $[x_L, y_L]$ and an outgoing arc $a$
Output: a list of feasible extensions of $L$ over $a$

$\mathcal{L} \leftarrow \emptyset$;
$\theta_1 \leftarrow$ NextStart($a, x_L$);
while $\theta_1 \leq y_L$ do
    $\theta_2 \leftarrow$ NextForbidden($a, \theta_1 + \tau_a$);
    create new label $L'$ with interval $[\theta_1 + \tau_a, \theta_2]$;
    $\mathcal{L} \leftarrow \mathcal{L} \cup L'$;
    $\theta_1 \leftarrow$ NextAllowed($a, \theta_2$);
return $\mathcal{L}$;

Function NextStart($a, x_L$)
    $\theta \leftarrow$ NextForbidden($a, x_L$);
    if $\theta \geq x_L + \tau_a$ then
        return $x_L$;
    return NextAllowed($a, \theta$);
```

extension. By making the resulting interval as big as possible, we assure that all possible start times for the current time window are included.
Starting in θ_2 represents the last possible start time in v as it is the next forbidden time step for arc a. Using arc a any longer than θ_2 is thus not possible, which leads to a latest start time for the next arc of exactly θ_2. We will then iteratively create the largest possible intervals for all future time windows. ■

As those functions are called for every label and each outgoing arc, it is very important for the performance of the entire algorithm that those calls are as efficient as possible. In this context, we developed three data structures that support the efficient execution of `NextForbidden` and `NextAllowed`. We explain the data structures and their different advantages and disadvantages in the next section. A computational evaluation is then discussed as part of Section 3.5.

3.4.1 Time Window Data Structures

Algorithm 7 is used during the extension step of Algorithm 4 to calculate all possible extensions of an label. `NextForbidden` and `NextAllowed` are the only two operations that need to be supported for data structures representing multiple time windows in $\mathcal{F}_a$.

We further assume that the time is discretized and that the highest integer contained in $\mathcal{F}_a$ of all the arcs $a \in A$ is denoted by T^{limit}. This allows us to store the information (whether a certain time step is forbidden or not) for every possible point in time and thereby making the possible data structures much more flexible:

List of Intervals: The feasibility time windows $\mathcal{F}_a$ for each arc are represented by a list of pairs (x, y), where x and y correspond to the lower and upper bound of the intervals. Here, `NextAllowed` has to loop through all the intervals, to check whether the given argument is either contained in one of the intervals, or to find the next interval. The function `NextForbidden` can be implemented analogously.
This can be improved, however, by storing the intervals in an array sorted by their lower bound and then performing binary search. However, in both cases the running time of `NextAllowed` as well as of `NextForbidden` depends on the number of intervals in $\mathcal{F}_a$ but not on the discretization T^{limit}.

Lookup Vector: This data structure consists of an array of T^{limit} integers. A negative value at a position θ means that the corresponding arc is forbidden for time step θ, while a positive value means that it is allowed. Further, the absolute value of the element at position θ corresponds to the next time step which is forbidden if we are currently allowed, or the other way around. From a theoretical point of view it is therefore possible to identify whether a time step is forbidden in $\mathcal{O}(1)$. Also, the return values of `NextForbidden` and `NextAllowed` can directly be derived from the elements of the array in $\mathcal{O}(1)$. Unfortunately, the size of the array depends on the discretization T^{limit}. It is necessary to store a full machine integer for every time step, which leads to a larger creation time. Due to the larger size of the data structure, it is also not very "cache-friendly" and some of the memory access operations will not be as fast, as the access pattern is hard to predict for the CPU.

Bit Array: The third approach is in some sense a compromise between time and space complexity. In this data structure each bit represents one time step. The bit is either set to 1, if the corresponding time step is forbidden or set to 0, if it is not. Although the size of the array still obviously depends on T^{limit}, it is a much more compact representation than the lookup vector. However, to find the next forbidden time step we have to traverse the array until the next set bit is found. This leads to a running time of $\mathcal{O}(T)$ which is much worse than for the previous two structures. Fortunately, most modern processor architectures include instructions to identify the position of the least significant bit set in a given unsigned machine word. For example in the Intel 386 and later architecture this instruction is called `bsf`, see [52, Vol. 2A 3-80]. Finding the first set bit is not so different from finding the next set bit. Using this instruction the function `NextForbidden`(a, x) for can be implemented as

$$\texttt{NextForbidden}(a,x) = \texttt{bsf}(\mathcal{F}_a \text{ AND} \left(-1 \ll (x-1)\right)), \tag{3.3}$$

where $\mathcal{F}_a$ is represented as a bit array, -1 is the word that has all bits set and $\ll$ performs a logical left shift in the direction of the most significant bit. All the bitwise operations assure that the first $x - 1$ bits of $\mathcal{F}_a$ are set to zero before `bsf` is called. Therefore, performing the first set bit operation now gives us the first set bit starting from position x, i.e. `NextForbidden`.

Unfortunately, the function as presented in (3.3) only works, if T^{limit} is not more than the number of bits in a machine word. (The length of a machine word on the CPU we used for the computational experiments is 64 bits.) If the bit array is larger than this number, more than one word needs to be combined. However, even in this case it is never necessary to call `bsf` more than once:

1. First, jump to the word containing the x-th element.
2. Then, check whether this word has a bit set after x by shifting right and then comparing with zero.
3. If it has a set bit, perform the operation as described in (3.3), if not goto 2 and consider the next word.

The entire function can therefore be implemented using only high performance binary operations with very few memory accesses. The function `NextAllowed` can be implemented analogously, as calling `bsf` on the negated word, gives the position of the first bit equal to zero. Computational results comparing the different data structures can be found in Section 3.5.

3.4.2 Preprocessing for the ESPPFTW

Although data preprocessing is often one of the first steps introduced to increase the performance of a practical solution approach, preprocessing for the resource constrained shortest path has not been discussed very often in the literature. Aneja et al. [3] tried to reduce the number of arcs in the underlying graph by finding lower bounds on the resource consumption on paths between s and all nodes $v \in V$ as well as from each node to the destination t. These bounds are then used to identify arcs which cannot be used in a feasible s-t path without violating resource limits, as their current consumption plus the bound exceeds the total limit. Such arcs will never be used and can be deleted from the network. These bounds on the resource consumption can also be used in the extension step to identify labels that will never lead to a feasible s-t path. The approach of Aneja et al. can be extended by using Lagrangean relaxation to obtain lower and upper

bounds on the cost, this method is, e.g., used in Beasley and Christofides [7]. Grötschel et al. [45] proposed an approach based on integer programming that is then solved using Lagrangean relaxation techniques. Such a preprocessing method, however, is quite time consuming as solving the Lagrangean relaxation is computationally rather expensive. It therefore only pays of for larger networks.
In this section we focus on some simple preprocessing ideas for the ESPPFTW that can be derived from forced arcs. They help to remove unnecessary arcs or they tighten the time windows and the arcs, so that fewer labels will be generated:

Revisiting forced arcs: The first rule can be derived directly from (3.1e) that was used in Proposition 3.1.4 to transform forced arcs into feasibility time windows. Each forced arc a needs to be visited for its time interval F_a and is visited exactly once. This means that if a is not allowed for some time step θ then it also cannot be visited in $[0, \theta]$ if $\theta < \inf(F_a)$ holds or $[\theta, \infty)$ if $\theta > \sup(F_a)$. Applying this to the time windows contained in $\mathcal{T}_a$ helps to eliminate paths that contain the arc a more than once and thus lower the number of iterations needed in the DSSR approach.

Nodes on forced arcs: If an arc $a = (u, v)$ is forced for $[x, y]$ in the elementary problem, this means that no other arc leaving u or entering v can be used at any point in time. Tours that contain (u, v) and any of these arcs are not elementary. Therefore, the corresponding arcs can be directly removed from the graph. Further, all incoming arcs $a \in \delta^-(u)$ can only be used before arc a and can thus be forbidden for $[x, \infty]$.

Two forced arcs: If the arc $a_1 = (u, v)$ is forced for $[x_1, y_1]$ and there is also a different forced arc a_2 with $F_{a_2} = [x_2, y_2]$ which is forced later, then the following rule can be formulated:
Each outgoing arc $b = (v, w) \in \delta^+(v)$ can also be forbidden, if $y_1 + \tau_b + d(w) \geq x_2$, where $d(w)$ corresponds to the shortest distance (w.r.t. to the travel times) from w to the tail of a_2. The corresponding rule can also be formulated for all the incoming arcs of the second forced arc a_2.

3.5 Computational Results

The performance of the label setting algorithm used for the pricing problem has been thoroughly tested isolated from the integer program and the column generation itself. Three sets of test instances have been considered. The first two sets are derived from the

well-known Solomon's CVRPTW benchmark and they are essentially the same instances that have been used by Feillet et al. [33] and Righini and Salani [73] for their experiments on the elementary SPPRC:
Solomon's benchmark instances are divided into three groups – clustered ("C"), random ("R") and random-clustered ("RC") – according to the location of the customers. As all of these instances have been originally designed for the *Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)*, we first have to convert them into SPPSTW instances: Hereby, we only consider the coordinates of the customers and their time windows; the remaining data, such as vehicle capacities and customer demands, is ignored. This makes the problems considerably harder, as more feasible paths and thus more labels exist. To reduce the size of the instances, only the first 25 nodes of each instance have been considered. This number matches the usual instance size when used as a pricing problem for the discrete WCP. The distance between two nodes is set to be the euclidean distance between the respective customers. The travel times are then derived from the distances by always rounding up the nearest integer. The time windows on the nodes have been converted into time windows on the arcs in the following way: Let $a = (u, v)$ be an arc and let $[x_u, y_u]$ and $[x_v, y_v]$ be the time windows for its nodes. We now set the feasible time windows of arc a to be $[x_u, y_v - 1]$. For each instance, we generate the prizes λ_v by randomly picking integers uniformly distributed in $[0, \dots, 20]$. The cost of an arc (u, v) then corresponds to $c_{(u,v)} - \lambda_v$. This data generation technique was devised by Feillet et al. [33] to have a reasonable number of negative cycles for elementary shortest path problems. Due to the special structure of the random instances, they rarely contain negative cycles. Therefore, only the clustered and random-clustered instances have been considered. Finally, the SPPSTW consists of finding the shortest elementary tour that respects the time windows and starts and ends in the first node given in the instance.

The third benchmark set is taken directly from the subproblems occurring in the solution process of the discrete WCP and they therefore represent ESPPFTW instances. This set has been selected to represent the biggest and most difficult pricing problems that have to be solved during the solution processes of the discrete WCP instances considered in [85]. In order to get comparable results of the performance of our LSA, we always calculate an elementary path with minimum cost even for these instances.

All of the computations have been performed on an Intel Xeon E5-2630 CPU clocked at 2.6 GHz.

Time Windows

The following experiments correspond to the observations made in Section 3.4.1. Three data structures have been introduced that are especially well suited for problems with many disretized time windows. The data structures have been tested on the set derived from the Solomon instances as well as the WCP set. Table 3.1 gives the average number of forbidden time windows on each arc and the total computation time for each set of instances. The complete test results for every single instance can be found in the appendix in Section A.1.

set	inst.	intervals	List (s)	Lookup (s)	BitArray (s)
SOL C	15	1.52	1928.3	1984.4	1860.3
SOL RC	16	1.53	234.6	225.9	230.1
WCP	51	0.76	1274.5	1253.1	1176.7
total			3437.5	3463.3	3267.1

Table 3.1: Summarized results for the data structures

The performance differences of the data structures are of course not nearly as large as when only random `NextAllowed` or `NextForbidden` queries (and no actual shortest path computation) are considered. In the context of the SPPSTW, the BitArray data structure performs in total 6 % faster on the tested WCP instances.

Preprocessing for Forced Time Windows

Since the preprocessing strategies as described in Section 3.4.2 rely on forced arcs, these approaches have only been tested on WCP instance that contain at least one forced arc. In the following we compare the impact of the preprocessing on these instances and compare it with the results without preprocessing. Here, only the modifications as introduced in Proposition 3.1.4 are used.

The total run time for all the WCP shortest path instances with forced edges, as well as the average number of DSSR iterations is given in Table 3.2. The detailed overview can again be found in the appendix in Section A.2.

		TW preprocessing		no preprocessing	
set	inst	iter	time (s)	iter	time (s)
forced WCP	35	8.00	1135.4	9.31	3800.6

Table 3.2: Summarized results for the time window preprocessing

The preprocessing reduces the number of iterations considerably. This results in a speed-up of more than a factor three.

Chapter 4

Combining Discrete Optimization with Nonlinear Optimization

So far, we stated in Chapter 2 a solution approach for the discrete WCP and gave an example how an optimal trajectory between two positions can be calculated. The actual key algorithm, however, combining those two parts has only been shortly introduced to motivate the two subproblems.
In this chapter we develop the idea from Section 2.1 further by presenting the concrete algorithm that is capable of computing an optimal solution to the integrated WCP by avoiding unnecessary trajectory calculations.

The integrated algorithm is formally introduced in Section 4.1, before the actual implementation is discussed in Section 4.2. The last section gives some information about the software that has been developed in the context of this research project and visualizes some two-dimensional instances and the solutions obtained.

4.1 Combination Algorithm

The basic idea behind the combination is the following: Instead of solving the exact WCP with all trajectory and collision information, we only solve the relaxation WCP$(\mathcal{D}', \mathcal{C}')$ with different collisions and distances. For that, we first initialize the set of conflicting trajectory pairs $\mathcal{C}'$ to the empty set. The distances $\mathcal{D}'$ are initialized with lower estimates for all distances between two positions. Any solution to the so formulated WCP$(\mathcal{D}', \mathcal{C}')$ yields a lower bound on the optimal solution of the actual discrete problem WCP$(\mathcal{D}, \mathcal{C})$, where $\mathcal{D}$ and $\mathcal{C}$ contain the exact and complete information. Furthermore, this solution gives us a promising assignment and sequencing, which can then be used to calculate the exact distances and collisions for the tours in that solution and thus, to refine the relaxation. This idea leads to Algorithm 8.

Algorithm 8: Integrated algorithm for the WCP

```
Input: a WCP instance
Output: a feasible solution
initialize distances $\mathcal{D}'$;
initialize conflict set $\mathcal{C}' = \varnothing$;
solve the WCP($\mathcal{D}', \mathcal{C}'$);
while feasible solution found do
    foreach arc a in found tours do
        if distance for a is estimated then
            calculate exact trajectory for $a$;
            in $\mathcal{D}'$ replace length of $a$ with exact value;
    if solution with exact distances is still feasible then
        check collisions for tours;
        if collisions present then
            add conflicting arc pair to $\mathcal{C}'$;
        else
            return found solution;
    resolve the WCP($\mathcal{D}', \mathcal{C}'$);
return $\varnothing$;
```

If the discrete WCP (with complete distance and collision information $\mathcal{D}$ and $\mathcal{C}$) has a feasible solution, then Algorithm 8 returns a solution. Due to the very conservative collision handling in the discrete WCP, it might even be the case that the algorithm finds a solution that is not feasible for the discrete WCP. But as only tours without exact collisions are returned, this solution is still feasible with respect to the original WCP.

The correctness of Algorithm 8 can be shown with the following observation. The only requirement is that the distances $\mathcal{D}$ are always *underestimated*, i.e. that calculating the exact distances never decrease the distance information $\mathcal{D}$.

Proposition 4.1.1 *Let $\mathcal{D}$ be the exact distance information and let $\mathcal{C}$ bet the complete collision set. Any feasible solution $(\bar{T}, \bar{I})$ of the corresponding discrete $WCP(\mathcal{D}, \mathcal{C})$ is also feasible in any iteration of Algorithm 8.*

PROOF:
Since the distances throughout the execution of Algorithm 8 will never be larger than the exact distances in $\mathcal{D}$, the solution $(\bar{T}, \bar{I})$ is always a feasible scheduled tour for $WCP(\mathcal{D}', \varnothing)$. However, it might very well be that the scheduled tour $(\bar{T}, \bar{I})$ in the

context of the $WCP(\mathcal{D}', \varnothing)$ has more and longer waiting times on the arcs than for the exact $\mathcal{D}$, as the differences in the travel times are compensated via waiting.
If the scheduled tour $(\bar{T}, \bar{I})$ is feasible for the discrete WCP with complete collision information $\mathcal{C}$, it will also be feasible for any subset $\mathcal{C}' \subseteq \mathcal{C}$. ■

As the number of jobs and thus also the $|\mathcal{C}|$ is finite, the algorithm will eventually terminate. However, usually only a small subset of the trajectories needs to be computed.

The computational expensive part, however, remains in line 7 of Algorithm 8, where the initial distances are *updated* with the correct ones.
The approach used in Algorithm 8 to reduce the number of these updates can be generalized for linear combinatorial optimization problems. Such a generalization is shown in Algorithm 9 for the following problem:

Definition 4.1.2 (Linear combinatorial optimization problem)
A *linear combinatorial optimization problem* is a tuple $(E, \mathcal{S}, c)$ consisting of a finite ground set E, a subset $\mathcal{S} \subseteq 2^E$ of feasible solutions and a cost function $c : E \to \mathbb{R}$.
The task is to find a feasible solution $S \in \mathcal{S}$ such that $c(S) := \sum_{e \in S} c(e)$ is minimized.

Many classical combinatorial optimization problems, such as the shortest path problem, minimum spanning trees or the TSP, correspond to linear combinatorial optimization problems.

Algorithm 9: Optimizing a linear problem with estimated distances

```
Input: a linear combinatorial optimization problem $P = (E, \mathcal{S}, c)$
Output: an optimal solution of $P$
initialize cost function $c' : E \to \mathbb{R}$ such that $c'(e) \leq c(e)$ for all $e \in E$;
$S \leftarrow$ optimal solution of $P$ with respect to the cost $c'$;
while solution found do
    if solution $S$ only consists of elements with $c'(e) = c(e)$ then
        return $S$;
    else
        foreach $e \in S$ do
            $c'(e) \leftarrow c(e)$;
    resolve $P$ with new $c'$;
return null;
```

It can be shown that Algorithm 9 indeed finds an optimal solution for the correct cost function c as long as the costs never decrease during updates:

Proposition 4.1.3 *The Algorithm 9 returns an optimal solution for the corresponding linear combinatorial optimization problem or shows that no feasible solution exists.*

Proof:
Let S be the solution returned by Algorithm 9 and let c' corespond to the cost function in the last iteration. As S contains only elements with the exact cost c, we have $c'(S) = c(S)$. Since S is an optimal solution with respect to c', we get $c'(S) \leq c'(S')$ for all $S' \in \mathcal{S}$. And thus

$$c(S) = c'(S) \leq c'(S') \leq c(S')$$

holds for all $S' \in \mathcal{S}$, i.e. S is an optimal solution with respect to c.
If not solution is returned by Algorithm 9, then the original problem is also infeasible. ■

In general, both Algorithm 8 and 9 do not need to use all of the original distance information for their computation of the optimal solution. But the question remains, how many updates do they need and how "good" is this?

The number of updates depends on the problem instance itself as well as the quality of the estimated costs. It is therefore in general not possible to give a nontrivial upper bound on the number of updates required.
A better evaluation criterion for the number of updates can be obtained by comparing them with the lowest number possible in this situation, i.e. the minimal number of updates that an algorithm knowing the exact information needs in this situation. Exactly these considerations are taken into account for optimization problems *under uncertainty*.

As Part I of this thesis has been designed to explain the practical and implementation aspects of the Welding Cell Problem, we do not go more into detail about these rather theoretical aspects of the problem. The corresponding concept of uncertainty is instead discussed in full detail in Part II and in particular in Chapter 5. In that chapter the foundations are developed which allow us to analyze and formulate algorithms with guaranteed update efficiency. There, we analyze a problem closely related to the WCP – the traveling salesman problem – as it corresponds to a WCP problem with only one robot.

4.2 Resolving the Discrete WCP

After new and updated distances have been computed, the discrete WCP needs to be solved again for the new sets $\mathcal{D}'$ and $\mathcal{C}'$.

Fortunately, the chosen branch-and-price based approach is especially well suited for this procedure. Explicit restarts are not necessary and the continuous part can be seamlessly integrated in the discrete solution process.

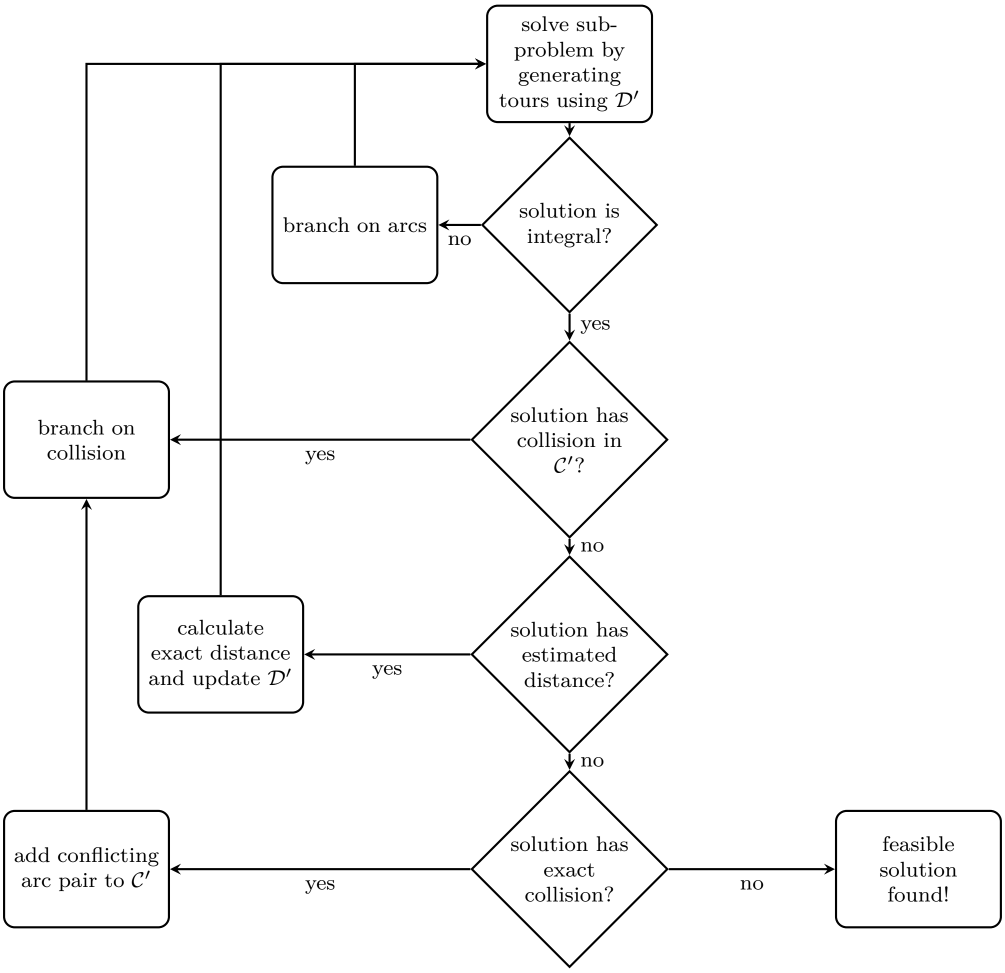

solve sub-problem by generating tours using $\mathcal{D}'$
solution is integral?
no
branch on arcs
yes
solution has collision in $\mathcal{C}'$?
yes
branch on collision
no
solution has estimated distance?
yes
calculate exact distance and update $\mathcal{D}'$
no
solution has exact collision?
yes
add conflicting arc pair to $\mathcal{C}'$
no
feasible solution found!

Figure 4.1: Flowchart of the integrated algorithm

If a new distance of arc a has been calculated, we first remove all variables from the restricted master problem representing tours that contain the updated arc a. Now, the corresponding value in $\mathcal{D}'$ is updated so that the pricing algorithm generates new tours with respect to the updated distance. If the tour is still feasible, it might be the case that the same tour (but now containing the exact distance) is generated again. As all other tours remain in the problem, this can be interpreted as a *warm start* of the discrete WCP. An optimal solution to the relaxed master problem can therefore be determined

much faster than when the pricing is started from scratch.
When an integral solution containing only updated exact distances has been found, we check these tours for exact collisions. If no such collisions exist the tours represent a feasible solution for the integrated WCP. Otherwise, we add the collision to the set $\mathcal{C}'$ and branch on that particular collision. A simplified flowchart of this approach is given in Figure 4.1.

Due to the “power of column generation” these ideas can be integrated without any modifications of the code for the discrete WCP. Since SCIP is build as a solver for constraint integer programs, the integrated parts can very conveniently be implemented as custom *constraint handlers*. When the constraint handler is called to check the current solution for feasibility, we call the programs of the continuous part and update the distances and collision information accordingly. The priorities of the constraint handlers must be set in such a way that they are called in the order as described in Figure 4.1.
Following this approach, not a single line of existing code needs to be changed!

4.3 The 2D-Demonstrator

In the context of this thesis, we implemented a program that integrates all the aspects discussed in Part I. The program takes as input a WCP instance, containing information about the workpiece $\mathcal{W}$, the weld points J and the robots R, and calculates a collision free solution within the given cycle time.
In the following we restrict to the two-dimensional WCP problem, where the obstacles as well as the robots are polygons in the 2D plain. This has the great advantage that the instances and solutions can be better visualized “on paper”. Furthermore, in 2D the continuous problem, which is not the main focus of this thesis, is much easier and faster to solve so that the actual impact of the integration and the discrete part becomes much clearer.

The actual program relies on the following input:

- Obstacles can be represented as any set of convex polygons.
- The robots are also represented by convex polygons, that move through 2D space by translation and rotation: Let therefore r,v and a, respectively, denote the position, the velocity and acceleration of the center of gravity of the robot in $\mathbb{Q}^2$. Since the robot can rotate, let θ denote the angle of ration of the robot and let μ be the velocity of the angle of rotation.
  The kinematic properties of each robot can be specified with the following parameters:

  - $\underline{a}, \overline{a} \in \mathbb{Q}^2$ limit the acceleration a of the center of gravity of the robot in the two-dimensional plane, i.e. $\underline{a} \leq a(t) \leq \overline{a}$ for all t.
  - $\underline{\mu}, \overline{\mu} \in \mathbb{Q}$ limit the velocity of the angle of rotation, i.e. $\underline{\mu} \leq \mu(t) \leq \overline{\mu}$.

- Each job $p \in \mathbb{Q}^2$ is given as a point in the two-dimensional space. For a robot to reach job p and start processing, the following must hold for the fixed time point t:

  $$r(t) = p \quad \text{and} \quad v(t) = \mathbf{0} \quad \text{and} \quad \theta(t) = 0\,.$$

  Thus, the robots must completely stop at p and must also have a fixed rotation. Then, the processing time of p can start. Further, a job can either be exclusive for one particular robot or it can be equally processed by all robots.

- Also a cycle time T^{limit} needs to be given.

Remark: The program only checks for a feasible solution and the first found feasible solution is returned. This solution is not necessarily optimal with respect to the makespan.

Test Instances

In the following, we present some of the 2D-instances that have been solved using our implementation. The trajectories visualized in the following only represent the trace of the center of gravity. For clarity reasons, the rotation along the trajectories is not shown. Also the actual timing of the robots is not presented explicitly, but it has been taken into account for the calculations. All the weld points are assumed to have a processing time of 0 s; if they are exclusive to one particular robot, they are drawn in that robots color.

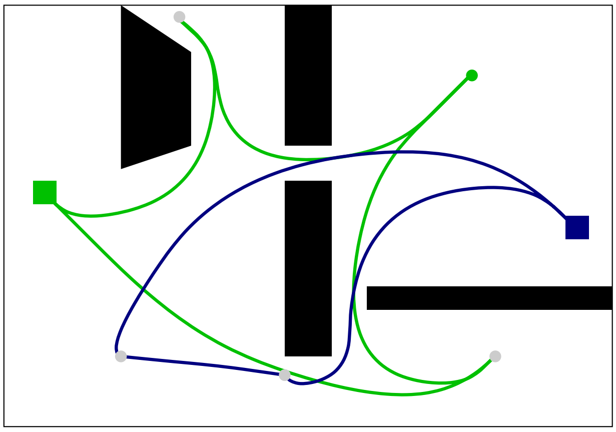
For each robot r we have

$$-\underline{a}_r = \overline{a}_r = \begin{pmatrix} A_r \\ A_r \end{pmatrix}$$

and

$$-\underline{\mu}_r = \overline{\mu}_r = M_r\,,$$

where A_r and M_r differ for the robots and the instances and are explicitly given. A higher acceleration A_r corresponds to a faster robot $r \in R$. We assume that for all the distances the triangle inequality holds, but symmetricity is not assumed.
For each solution we give the number of arcs, i.e. the number of possible pairs of position for all robots, and the number of exact trajectories that have been calculated. Furthermore, also the total computation time as well as the time that has been spent for the continuous calculations is specified. For each robot we also give the length of its tour including waiting times.
The process of our program is showcased for the five following examples:

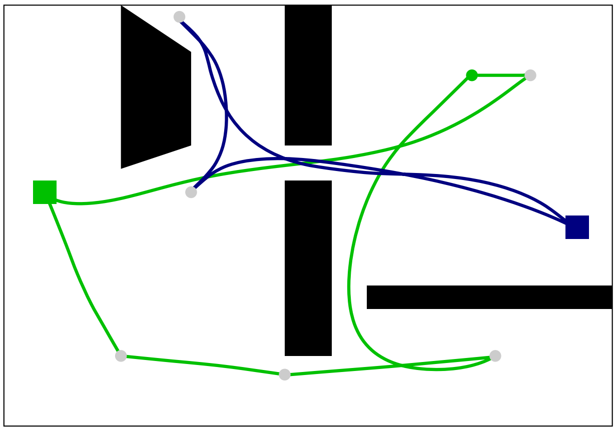


robot	A_r	M_r	tour length
🟩	2.0	$\frac{\pi}{10}$	24.76
🟦	1.0	$\frac{\pi}{10}$	24.85

cycle time: 25.5

arcs	updates	cont. time	total time
50	13	19.9 s	21.0 s

Instance 4.1

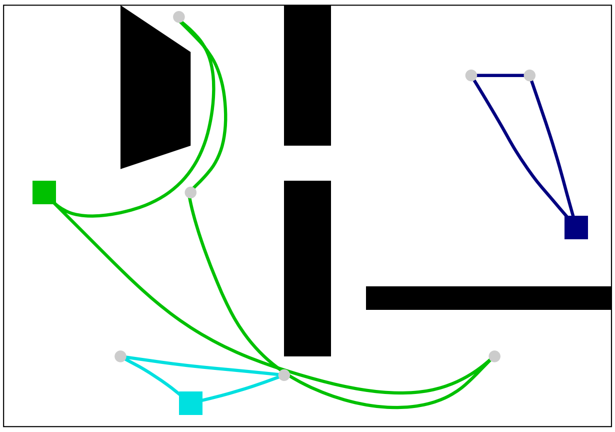


robot	A_r	M_r	tour length
🟩	2.0	$\frac{\pi}{10}$	27.15
🟦	1.0	$\frac{\pi}{10}$	23.16

cycle time: 27.5

arcs	updates	cont. time	total time
98	33	28.2 s	33.0 s

Instance 4.2

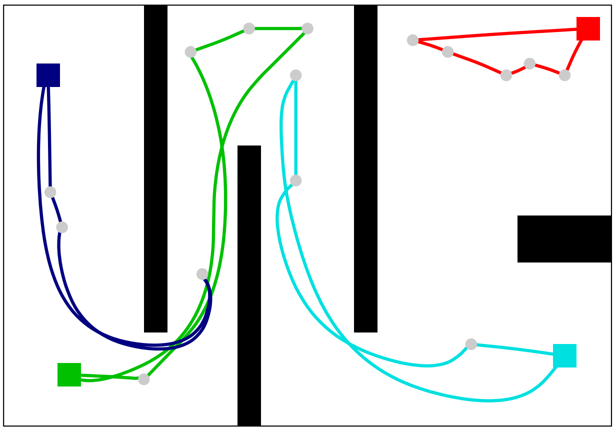


robot	A_r	M_r	tour length
🟩	2.0	$\frac{\pi}{10}$	22.04
■	1.0	$\frac{\pi}{10}$	13.58
🟦	0.5	$\frac{\pi}{10}$	18.26

cycle time: 23.5

#arcs	#updates	cont. time	total time
168	34	27.1 s	29.5 s

Instance 4.3

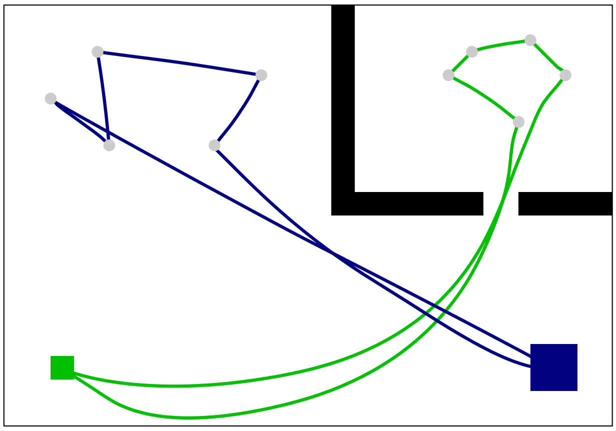


robot	A_r	M_r	tour length
🟩	1.0	$\frac{\pi}{10}$	26.59
■	1.0	$\frac{\pi}{10}$	26.28
🟦	1.0	$\frac{\pi}{10}$	25.95
🟥	0.5	$\frac{\pi}{10}$	26.52

cycle time: 26.7

arcs	updates	cont. time	total time
788	62	56.9 s	66.0 s

Instance 4.4



robot	A_r	M_r	tour length
🟩	1.0	$\frac{\pi}{10}$	32.69
🟦	1.0	$\frac{\pi}{10}$	31.10

cycle time: 34.0

arcs	updates	cont. time	total time
220	45	38.9 s	844.7 s

Instance 4.5

All of these examples show that our integrated approach works and indeed helps to drastically reduce the number of computed distances. As expected, the quality of the estimated distances has a great influence on the number of required updates. The best ratio is achieved for Instance 4.4. This instance has a very tight cycle time and contains rather large distances. Thus, many tours can already be ruled out using the estimated distances. However, even for the weaker instances the update ratio is always better than three.

Part II

Theory

Chapter 5

Aspects of Uncertainty in Optimization Problems

Many combinatorial optimization problems exist that can be used to represent and then solve a variety of real-world applications such as routing, production scheduling or network design. All those applications require input data that is derived from measurements or observations of actual goods and processes. However, this data can often only be estimated or may even vary from time to time. Therefore, it is crucial to model these uncertainties in real-world phenomenons.

There are many possible ways to deal with such difficulties and the two most common approaches are *stochastic programming* and *robust optimization*.
In stochastic programming it is assumed that a probability distribution, which describes certain input parameters, is either known or can at least be estimated. For a detailed description of stochastic programming see e.g., Birge and Louveaux [10]. The task in robust optimization, on the other hand, is to find solutions that are feasible even if uncertainties up to a certain extent occur. An overview of this topic can for example be found in Ben-Tal et al. [8].

In this chapter we pursue a third approach. In our setting data is given within certain uncertainty limits but an algorithm can obtain the exact value of an input parameter by performing a so-called *update* operation.
We further assume that this operation is very expensive so that in this setting the number of updates required is far more important than time or space requirements of the actual algorithm.

There are a number of applications where this uncertainty setting is reasonable: One example would be the Welding Cell Problem described in Part I of this thesis. Here, the underlying combinatorial optimization problem relies on the exact trajectories that can be computed in a computationally very expensive process, but certain bounds on the trajectories are known or can easily be obtained.

Another field of application contains problems where the agents, and thus the data, are “in motion”, i.e. there are changes over time within certain limits. This can for example occur in ad hoc wireless sensor networks (see e.g. Dargie and Poellabauer [18] or Sohraby et al. [80] for an introduction about sensor networks), where multi-hop communications takes place. Here, data should be routed along the shortest path (or along the minimum spanning tree for broadcasts). The exact location can be determined, but this should be avoided when possible due to required high energy consumption. This uncertainty aspect in ad hoc networks has also been considered in Bruce et al. [14].

To evaluate the performance of an algorithm, we compare the number of updates required with the optimal number of updates that an algorithm knowing the exact weights would need. This is called *update competitive ratio* and in this context a constant update competitive ratio is pursued.
This notion of complexity was first used in Kahan [55] where the author presented several geometric uncertainty problems and gave strategies that lead to an optimal ratio. Bruce et al. [14] discussed the problems of computing maximal points or the points on the convex hull of a set of points in the stated uncertainty setting. The authors also introduced the general concept of *witness algorithms* to tackle such uncertainty problems with a provable update competitive ratio. Erlebach et al. [29] used this concept to compute minimum spanning trees under uncertainty.
A different approach to assesses the quality of the selected updates was proposed in Feder et al. [30, 31]. Here, the updates have different update costs and the goal is to minimize the total update cost. Within this framework, Feder et al. considered the problem of computing the median of n numbers within a given tolerance and a shortest s-t path problem.

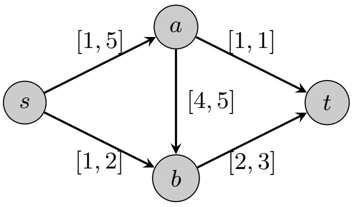
In Chapter 4 we described an integrated approach for the WCP that tries to minimize the number of trajectory computations and still guarantees an optimal solution. In this chapter we give the generalized theoretical foundation for this observation. Since the WCP – even without uncertainty – is a very complex problem that also takes scheduling aspects into account, we focus our observations on the classic traveling salesman problem. The TSP is a special case of the WCP as it represents the situation where only one robot exists.
In Section 5.1 we formally define the used uncertainty concept. In the next section we then discuss the TSP and several related problems, namely the shortest path problem and the minimum spanning tree problem, in this uncertainty setting. As the TSP and many of the so far considered problems do not have a constant update competitive ratio, we further extend the model over the following two sections to incorporate more aspects to the special situation we have for the WCP. This includes metric data that is not too far away from its given limits (with certain probability). This finally allows us to give an algorithm for the TSP that only requires order of n updates in expectation.

Finally, in Section 5.5 we give a tabular overview of the complexity of all the considered optimization problems in the different uncertainty settings.

5.1 Preliminaries

The uncertainty model used in this chapter is based on *interval data*. This means that the weight c_e of each element e of the finite ground set of a combinatorial optimization problem lies within a certain given interval I_e, but the exact weight is initially not known.

Before we formally define the model, let us first look at an intuitive example:
Consider the shortest s-t path problem of the following graph, where the actual arc weights are unknown. Only intervals are given for each arc, which define a lower and an upper bound on the weights:



The task is to find the shortest s-t path. We note that the graph contains exactly three such paths:

1. Path s-a-t with a cost in $[2,6]$,
2. Path s-a-b-t with a cost in $[7,13]$,
3. Path s-b-t with a cost in $[3,5]$.

Thus, it is clear that the second path can never be a shortest path no matter what the exact weights may be. However, to decide whether the first or the last path is actually shorter we need to *update* some arcs. If an update of the arc (s,a) reveals an exact weight of 1, we know after just one update that the path s-a-t is definitely the shortest.

For general combinatorial optimization problems this can formally be defined by using the uncertainty model described in Erlebach et al. [29]:

Definition 5.1.1 (Uncertainty problem with interval data)
Each problem instance P is a triple $(\mathcal{C},\mathcal{I},\phi)$, where

- the *configuration* $\mathcal{C}$ is an ordered set of data $\mathcal{C}=\{c_1,\dots,c_n\}$, with $c_i\in\mathbb{R}$,
- $\mathcal{I}$ is an ordered set of intervals $\mathcal{I}=\{I_1,\dots,I_n\}$, with $c_i\in I_i$ for all $c_i\in\mathcal{C}$,
- and a function ϕ such that $\phi(\mathcal{C})$ is the set of solutions for P.

As the function ϕ returns the solutions for a given configuration $\mathcal{C}$, it does not depend on the uncertainty intervals. It is thus the same for all instances of a problem and can therefore be used to represent the problem itself. For an instance P the goal then is to find an element of $\phi(\mathcal{C})$.

The intervals I_i are called *uncertainty intervals* or *uncertainty areas* and we further say that an uncertainty interval is *trivial*, if it consists of a single element. An ordered set of data $\{w_1, \ldots, w_n\}$ (not necessarily the actual configuration) is called a *realization* of $\mathcal{I}$, if $w_i \in I_i$ holds for all its elements.
In the following chapter we only consider uncertainty problems that are based on optimization problems on graphs, i.e. given the graph $G = (V, A)$ the configuration $\mathcal{C}$ consists of G and its actual arc weights. However, the ordered set of intervals specifies the graph G exactly, i.e. complete knowledge of the graph can be assumed:

Example 5.1.2 Shortest path problem with interval data: In the corresponding version of the shortest path problem the configuration $\mathcal{C}$ consists of the given directed graph $G = (V, A)$ and a value c_a for all arcs $a \in A$ that represents the actual weight of that arc a. The set $\mathcal{I}$ now contains an arbitrary interval I_a for each arc such that $c_a \in I_a$. Then $\phi(\mathcal{C})$ is the set of all shortest *s*-*t* paths in G, where a path is represented by an ordered set of arcs.

Given such an uncertainty problem with interval data we are interested in an algorithm that computes an element of $\phi(\mathcal{C})$. At the beginning only the interval set $\mathcal{I}$ and not the set $\mathcal{C}$ is known to the algorithm. However, in contrast to other uncertainty models, we are not trying to find a "good" solution with respect to the uncertainty intervals, but our goal is to calculate an optimal solution for the underlying configuration. Therefore, there must exist the possibility for the algorithm to gain additional information, namely to obtain some of the exact values c_i. This process is called *update*:

Definition 5.1.3 (Update)
Let $P = (\mathcal{C}, \mathcal{I}, \phi)$ be an instance of an uncertainty problem with interval data. For the given set of uncertainty intervals $\mathcal{I} = \{I_1, \ldots, I_n\}$ an update of interval I_i reveals the corresponding exact value c_i, i.e. after the update the new set of uncertainty intervals is $\{I_1, \ldots, I_{i-1}, [c_i, c_i], I_{i+1}, \ldots, I_n\}$.

A trivial algorithm that solves any instance $P = (\mathcal{C}, \mathcal{I}, \phi)$ of an uncertainty problem with interval data is described in Algorithm 10. It consists of a loop updating all arcs with nontrivial uncertainty intervals. After the loop the set of uncertainty intervals corresponds to the exact configuration $\mathcal{C}$ and (assuming that $\phi(\mathcal{C})$ is computable) it is therefore possible to find an element of $\phi(\mathcal{C})$.

Algorithm 10: Trivial algorithm for an uncertainty problem with interval data

```
Input: instance $P = (\mathcal{C}, \mathcal{I}, \phi)$
Output: a solution of $P$
foreach nontrivial $I_i \in \mathcal{I}$ do
    update $I_i$;
return an element of $\phi(\mathcal{C})$;
```

Note that our primary goal is to minimize the number of updates necessary to calculate a solution. In this context we do not consider the running time or space requirement of an algorithm. The quality of an algorithm is only determined by the number of updates, i.e. by comparing it to the minimum number of updates for that specific uncertainty problem:

Definition 5.1.4 (Update strategy)
Let $P = (\mathcal{C}, \mathcal{I}, \phi)$ be an instance of an uncertainty problem with interval data. We say that the set $S(P) \subseteq \mathcal{I}$ is an *update strategy* $S(P)$, if updating the intervals of $S(P)$ leads to a new problem instance $P' = (\mathcal{C}, \mathcal{I}', \phi)$ where $\phi(\mathcal{C})$ contains a solution $\mathcal{S}$ that can be verified using $\mathcal{I}'$, i.e. $\mathcal{S}$ is a solution for all possible realizations of $\mathcal{I}'$.

The update strategy of a problem instance that contains the fewest elements is called *optimal update strategy*:

Definition 5.1.5 (Optimal update strategy)
Let $P = (\mathcal{C}, \mathcal{I}, \phi)$ be an instance of an uncertainty problem with interval data. We say that an update strategy $S_{OPT}(P)$ is an optimal update strategy for P, if $|S_{OPT}(P)|$ is the smallest number of updates required to verify an element of $\phi(\mathcal{C})$.
The number of updates $|S_{OPT}(P)|$ is denoted by OPT.

Now, we can use OPT to evaluate the quality of an algorithm:

Definition 5.1.6 (k-update competitive algorithm)
We say that an algorithm $\mathcal{A}$ is *k-update competitive* for a given uncertainty problem ϕ, if for every problem instance P of ϕ the algorithm needs at most $k \cdot OPT + c$ updates, where c is a constant.
In this context k is also called the *update competitive ratio.*

Definition 5.1.7 (k-update competitive problem)
We say that an uncertainty problem with interval data ϕ is *k-update competitive*, if there exists a deterministic algorithm for ϕ that is k-update competitive.

A problem is called *update competitive*, if it has a constant update competitive ratio.

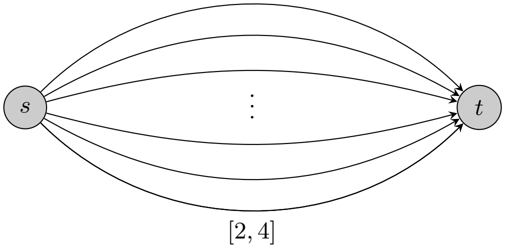
Remark: Note that the optimal update strategy is defined per problem instance, while an update competitive algorithm needs to work for all possible problem instances. In this sense OPT is somehow the perfect adversary.

After all the basic concepts have been introduced, we can now take a closer look at several combinatorial optimization problems in the context of uncertainty with interval data.

5.2 Uncertainty Problems with Interval Data

In this section we use the uncertainty model that we defined in the last section.

Remark: We also assume throughout the entire chapter that all the intervals are either trivial or open, as this gives us a much stronger foundation for deterministic algorithms. Erlebach et al. [29] showed for example that in their context update competitivity results can only be achieved, if the intervals are not closed (and not half-open). The general idea behind this assumption is that for many equivalent and indistinguishable closed intervals revealing one at its lower bound is sufficient to rule out the others without any additional updates. Any deterministic algorithm, on the other hand, has to update all of them in the worst case if all revealed weights but the last are strictly larger than the common lower bound. This idea is visualized in the following example with m parallel arcs, all having the same closed interval $[2, 4]$:



If the lowest exact weight is 2, the perfect algorithm only needs to update any arc with this weight. However, all of these arcs are indistinguishable for deterministic algorithms. Thus, for any deterministic algorithm there exists an instances where it needs $m - 1$ updates to find the one arc with weight 2.

A possible way to avoid this problem has been proposed by Gupta et al. [47]. They introduce the notion of a lexicographically smallest solution. Here, any algorithm must not only find an optimal solution but the lexicographically smallest optimal solution (if multiple solutions exist). However, as this introduces an artificial ordering on the input data, which is normally not present in the considered combinatorial optimization problems, we do not follow this approach and restrict ourselves to open intervals instead.

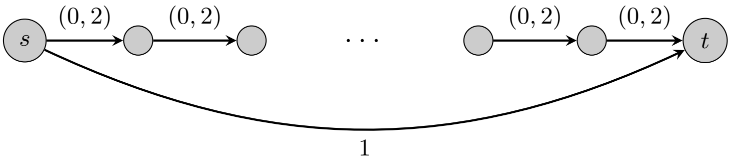
5.2.1 Shortest Paths

The first problem we analyze is the *Shortest Path Problem (SPP)*, as it is one of the most simple optimization problem on graphs and it still helps us understand the specific difficulties of uncertainty problems with interval data. In Section 5.1 we already used the shortest path problem to motivate our uncertainty framework.
We can now formally introduce the EDGE-UNCERTAINTY-SPP:

Configuration: a directed graph $G = (V, A)$
a weight $w_a \geq 0$ for every arc $a \in A$
a start node s and a target node t
Intervals: an open or trivial interval I_a for every arc $a \in A$
Goal: a path P from s to t that minimizes the sum $\sum_{a \in P} c_a$

Unfortunately, it is easy to see that the EDGE-UNCERTAINTY-SPP does not have a constant update competitive ratio.

Example 5.2.1 The EDGE-UNCERTAINTY-SPP is not update competitive:



One of the successive arcs has weight 1.5 and the others have weight ε small enough. Now, in the optimal update strategy for that instance it is sufficient to just update the arc with weight 1.5. After that, the path $P = (s, t)$ is obviously the shortest. Any deterministic algorithm, however, cannot distinguish between the edges with uncertainty interval $(0, 2)$. Thus, it has to query $n-1$ edges before finding the edge with weight 1.5 in the worst case.

Since the regular problem is not update competitive, we look at the *approximate* EDGE-UNCERTAINTY-SPP next:

Configuration: a directed graph $G = (V, A)$
a weight $w_a \geq 0$ for every arc $a \in A$
a start node s and a target node t
Intervals: an open or trivial interval I_a for every arc $a \in A$
Goal: a path P from s to t whose cost $\sum_{a \in P} c_a$ is at most $(1 + \delta)$
times the cost of an optimal s-t path, where $\delta > 0$ is given

Just as approximation algorithms in the regular, not uncertain setting, are considered as they improve the running time at the cost of solution quality, their purpose in the uncertainty setting is to reduce the number of updates at the cost of solution quality. Therefore, to asses an algorithm $\mathcal{A}$ for the approximate EDGE-UNCERTAINTY-SPP, we compare the number of updates required by $\mathcal{A}$ with OPT_{exact} of the corresponding exact problem, i.e. EDGE-UNCERTAINTY-SPP.

Example 5.2.2 The approximate EDGE-UNCERTAINTY-SPP is not update competitive when compared to OPT_{exact}:
As a counterexample we use the graph from Example 5.2.1 with the same uncertainty intervals and exact weights. Let n be the number of nodes in this graph and let $\varepsilon < \frac{1}{2n}$. We have $OPT_{\text{exact}} = 1$, but even after $n-2$ updates of arcs with nontrivial uncertainty interval, the uncertain s-t path could still have an actual cost of less than $\frac{1}{2}$ or more than 2. Thus, for $\delta < 1$ any deterministic algorithm still needs $n-1$ updates to find a $(1+\delta)$-approximation in the worst case.

The last example shows that even the approximate version of the shortest path problem cannot have an update competitive algorithm. In order to obtain such an algorithm for the SPP, we have to restrict the problem even further.
This leads to the *approximate* EDGE-UNCERTAINTY-SPP *with restricted intervals*:

Configuration:	a directed graph $G = (V, A)$ a weight c_a for every arc $a \in A$ a start node s and a target node t
Intervals:	each arc $a \in A$ either has a trivial interval or an open interval I_a where the upper bound of I_a is at most twice the lower bound
Goal:	a path P from s to t whose cost $\sum_{a \in P} c_a$ is at most $(1+\delta)$ times the cost of an optimal s-t path, where $\delta > 0$ is given

Under these restrictions it is now finally possible to given an update competitive algorithm. First, we consider the case where the set $\mathcal{P}$ of all possible s-t paths only consists of two elements:

Theorem 5.2.3 *Let $\mathcal{P}$ be the set of all paths for an instance of the approximate* EDGE-UNCERTAINTY-SPP *with restricted intervals. If $|\mathcal{P}| = 2$, then the problem is $\left(\frac{2}{\delta} + 2\right)$-update competitive, when compared to OPT_{exact}.*

To prove this theorem we will first state Algorithm 11 before showing that this algorithm (and thereby the problem itself) is update competitive.
As a first step, we define an easy criterion that can be used by Algorithm 11 to verify that a solution is indeed a $(1+\delta)$-approximation:

In the following, let P_U for any path $P \in \mathcal{P}$ denote the upper bound of this path, i.e. the sum of all upper bounds on all its arcs $a \in P$ and let P_L denote its lower bound.

Proposition 5.2.4 *A path $P' \in \mathcal{P}$ is a verifiable solution to the approximate* EDGE-UNCERTAINTY-SPP*, if and only if $P'_U \leq (1+\delta) \cdot P''_L$ holds for all other paths $P'' \in \mathcal{P}$.*

PROOF:
If $P'_U \leq (1+\delta) \cdot P''_L$ holds for all $P'' \in \mathcal{P} \setminus \{P\}$, then we know that for any possible realization P' is a $(1+\delta)$-approximation.
If the path P' is a verifiable solution, it is a solution for all possible realization and therefore $P'_U \leq (1+\delta) \cdot P''_L$ holds for all other paths $P'' \in \mathcal{P}$. ■

By using Proposition 5.2.4 as a stopping criterion we can now formulate an algorithm for the EDGE-UNCERTAINTY-SPP with restricted intervals. The resulting Algorithm 11 is obviously correct, as it updates nontrivial intervals until a $(1+\delta)$-approximation can be verified.

```
Algorithm 11: Approximate EDGE-UNCERTAINTY-SPP with restricted intervals
Input: an instance of the approximate EDGE-UNCERTAINTY-SPP with restricted
       intervals and $|\mathcal{P}| = 2$
Output: an $s$-$t$ path
while there is no path $P$ with $P_U \leq (1+\delta) \cdot P'_L$ for the other path $P'$ do
    $I \leftarrow$ interval with the largest width out of $\bigcup_{P \in \mathcal{P}} P$;
    update $I$;
return approximating path;
```

To provide a better structure for the prove of Theorem 5.2.3 we provide one additional lemma:
Given k elements with weights, we take largest-weight elements into a set S until the total weight of the elements in S is at least an α-fraction of the weight of all elements. Lemma 5.2.5 gives us a bound on the weight of the smallest element in S.

Lemma 5.2.5 *Given a set E of k elements with weights $w_e \in \mathbb{R}$ for all $e \in E$ and a constant $\alpha \in (0,1)$, let $W = \sum_{e \in E} w_e$ denote the sum of all these weights and let $S \subseteq E$ be a subset with smallest cardinality for which $\min_{e \in S} w_e \geq w_{e'}$ holds for all $e' \in E \setminus S$ and $\sum_{e \in S} w_e \geq \alpha W$.*
It now holds that $w_e \geq \frac{(1-\alpha)W}{k}$ for all $e \in S$.

PROOF:
Assume that $k > 1$. If $k = 1$ holds, the claim is trivially true.
The worst-case situation, i.e. the situation that results in the smallest $\min_{e \in S} w_e$, occurs when there is exactly one element with weight $\alpha W - \varepsilon$ and $k-1$ elements of the same

small weight of $\varepsilon' := \frac{W-(\alpha W-\varepsilon)}{k-1}$. In this case S contains exactly two elements, one with the weight $\alpha W - \varepsilon$ and the other one with weight ε'. Therefore, we have

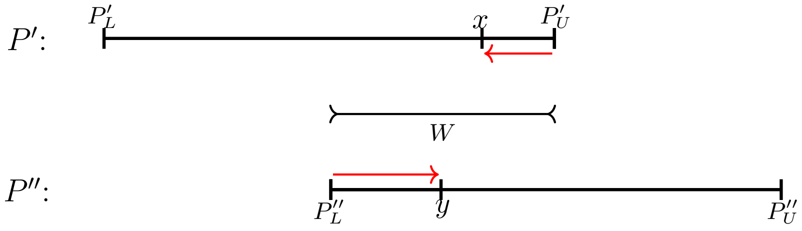
$$\min_{e\in S} w_e \geq \frac{W-(\alpha W-\varepsilon)}{k-1} = \frac{(1-\alpha)W+\varepsilon}{k-1} > \frac{(1-\alpha)W}{k}\,.$$

In all other situations the weight of the smallest element in S will only increase. ■

Proof of Theorem 5.2.3 We prove that Algorithm 11 is $(\frac{2}{\delta}+2)$-update competitive when compared with OPT_{exact}.
We can assume without loss of generality that there are no nontrivial arcs that lie on both paths. In this case, the weight of such arcs would just be an offset to both paths and as such never updated for OPT_{exact}. Hence, by setting them to their respective lower bound we can assure that any update strategy that proves a cost within a factor of $(1+\delta)$ for this situation is also true for any weight on these arcs.

Let in the following the two paths P', P'' of $\mathcal{P}$ be ordered in such a way that $P'_L \leq P''_L$. The proof consists of two cases, distinguishing whether P' or P'' is the actual shortest path. In both cases the basic idea of the proof is the following: To verify that a path is the shortest, any optimal update strategy has to raise (or lower) the bounds of the paths so that the bounds no longer overlap. This can be achieved by updating the intervals that will lead to the largest progress in the desired direction. Lemma 5.2.5 gives us a bound on the smallest progress made to cover only a certain fraction of the overall progress. By updating all intervals whose width is at least as large as this bound we can therefore assure that this fraction is reached.

Case I: The optimal update strategy reveals that P' is the shortest path. The situation for this proof is sketched in the following graphic:



In this case the optimal update strategy has to cover at least $W = P'_U - P''_L$ by either raising the lower bound of P'' or by lowering the upper bound of P'. Let us assume that this requires exactly k updates.
For a $(1+\delta)$-approximation only a smaller part needs to be covered: Here, it is sufficient

to move the upper bound of P' down to x and the lower bound of P'' up to y, where $x = (1+\delta)y$ is satisfied. Thus, only $W - (x - y)$ needs to be covered. This distance is maximized when y is as close as possible to P''_L. Therefore, the algorithm needs to cover at least $W - (1+\delta)P''_L + P''_L = P'_U - P''_L(1+\delta)$, which is denoted by A.

We already know that the optimal update strategy consists of k updates, it selects the k intervals where an update leads to the largest possible progress in the desired direction. By applying Lemma 5.2.5, where the elements in E correspond to the k intervals updated, their weights being the respective progress and $\alpha = \frac{A}{W}$, we now get that it is possible to improve the bounds by $A = \alpha W$ with intervals whose progress is at least $\frac{(1-\alpha)W}{k}$.
As the progress cannot be larger than the actual width of the interval, it must be possible to cover A only with intervals of width greater than $\frac{(1-\alpha)W}{k}$. Since

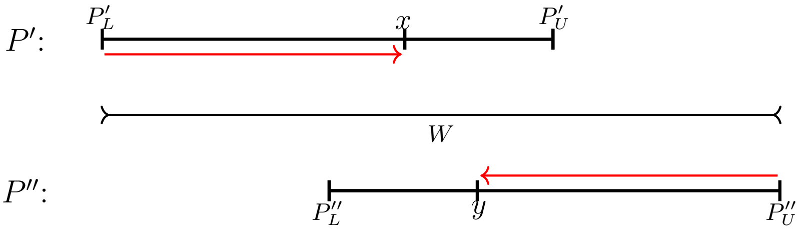
$$\frac{(1-\alpha)W}{k} = \frac{W-A}{k} = \frac{P''_L \cdot \delta}{k}$$

holds, updating all intervals with width of at least $\frac{P''_L \cdot \delta}{k}$ guarantees a progress of at least A. Next, we bound the number of such intervals using the fact that $P_U \leq 2 \cdot P_L$:

$$\frac{(P'_U - P'_L)}{\frac{P''_L \cdot \delta}{k}} + \frac{(P''_U + P''_L)}{\frac{P''_L \cdot \delta}{k}} \leq \frac{P'_L}{\frac{P''_L \cdot \delta}{k}} + \frac{P''_L}{\frac{P''_L \cdot \delta}{k}} \leq \frac{2 \cdot P''_L}{\frac{P''_L \cdot \delta}{k}} = \frac{2k}{\delta}$$

Therefore, the algorithm terminates after at most $\frac{2k}{\delta}$ updates and only needs a factor of $\frac{2}{\delta}$ more updates than OPT.

Case II: The optimal update strategy reveals that P'' is the shortest path.



This case is similar to the last situation, but here the optimal update strategy has to cover at least $W' = P''_U - P'_L$. Let us assume that this requires exactly k' updates.
For a $(1+\delta)$-approximation we have to raise the lower bound of P' to x and lower the upper bound of P'' to y, where $y = (1+\delta)x$ holds. Thus, a distance of $W' - (y - x)$ needs to be covered. In the worst-case, i.e. when W' is maximized, the value x is as small as

possible. Thus, W' is maximized when y is as close as possible to P''_L and the algorithm has to raise P'_L up to $\frac{P''_L}{1+\delta}$. The total distance it has to cover is called A'. It now holds that $A' = W' - (y-x) = W' - \left(P''_L - \frac{P''_L}{1+\delta}\right)$.
We apply Lemma 5.2.5 with $\alpha = \frac{A'}{W'}$. Thus, the smallest progress made by the optimal update strategy that is sufficient to cover A' is at least

$$\frac{(1-\alpha)W'}{k'} = \frac{W'-A'}{k'} = \frac{P''_L - \frac{P''_L}{1+\delta}}{k'} = \frac{P''_L\left(\frac{\delta}{1+\delta}\right)}{k'} .$$

For the maximum number of such intervals we have

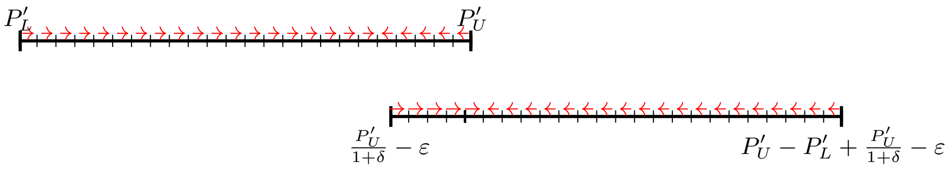
$$\frac{2 \cdot P''_L}{\frac{P''_L\left(\frac{\delta}{1+\delta}\right)}{k'}} = \frac{2k(1+\delta)}{\delta} .$$

Hence, the algorithm requires at most a factor of $(\frac{2}{\delta}+2)$ more updates than OPT. ■

To evaluate the quality of the update competitive ratio of $(\frac{2}{\delta}+2)$, we will now give a lower bound on this ratio:

Theorem 5.2.6 *There are instances of the approximate* EDGE-UNCERTAINTY-SPP *with restricted intervals and two paths where any deterministic algorithm not knowing the exact weights has a competitive ratio arbitrarily close to* $(\frac{1}{2\delta}-1)$ *when compared with* OPT_{exact}.

PROOF:
Consider the following situation, where we have two paths with the following bounds:


P'_L
P'_U
$\frac{P'_U}{1+\delta} - \varepsilon$
$P'_U - P'_L + \frac{P'_U}{1+\delta} - \varepsilon$

Assume there are k intervals in each path all with the length of $a+\varepsilon$ and the exact weights being ε away from either their lower or their upper bound. Thus, updating any interval will contribute a in the corresponding direction. For any interval with an arrow pointing right, the exact weight is ε away from upper bound, and for any interval pointing left it is ε away from the lower bound.

Let there be exactly $\lceil (P'_U - \frac{P'_U}{1+\delta} + \varepsilon)/a \rceil$ intervals in P' where updating lowers the upper bound of P' and exactly $\lceil (P'_U - \frac{P'_U}{1+\delta} + \varepsilon)/a \rceil$ intervals in P'' that raise the lower bound.

An optimal update strategy will update exactly $\lceil (P'_U - \frac{P'_U}{1+\delta} + \varepsilon)/a \rceil$ of those intervals and thus having $P'_U \leq P''_L$, which proves that P' is the shortest path.

For ε small enough updating any one interval out of the optimal update strategy will lead to a situation in which P' is a $(1+\delta)$-approximation. As the intervals on one path are identical, the best option is to pick a path first and then only update intervals on this path. In this case there are instances where any deterministic algorithm has to cover a little more than $\frac{P'_U}{1+\delta} - P'_L$ before finding a interval that contributes to the desired direction.

If we now let $k \to \infty$ and $\varepsilon \to 0$, we can ignore the rounding for the optimal update strategy as well as for our algorithm and get the following factor:

$$\frac{\frac{P'_U}{1+\delta} - P'_L}{P'_U - \frac{P'_U}{1+\delta}} = \frac{\frac{2p'_L}{1+\delta} - P'_L}{2p'_L - \frac{2p'_L}{1+\delta}} = \frac{\frac{2}{1+\delta} - 1}{2 - \frac{2}{1+\delta}} = \frac{2-(1+\delta)}{2(1+\delta)-2} = \frac{1+2\delta}{1-2\delta} = \frac{1}{2\delta} - 1 \qquad \blacksquare$$

The result from the last Theorem 5.2.6 shows that the update competitive ratio of the approximate EDGE-UNCERTAINTY-SPP with restricted intervals and two paths has to depend on $\frac{1}{\delta}$. It also shows that for small δ the update competitive ratio of Algorithm 11 is roughly a factor of 4 higher than the best possible ratio.

Constant number of paths

Next, we consider the case where there are more than two but still constantly many paths in $\mathcal{P}$. To obtain an algorithm for this variation, it is sufficient to modify Algorithm 11 in such a way that the largest width interval in each path is updated. The correctness of the resulting Algorithm 12 also shows that the approximate EDGE-UNCERTAINTY-SPP with restricted intervals and a constant number of paths is update competitive.

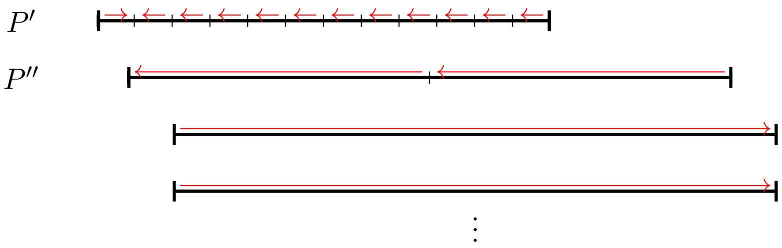
Theorem 5.2.7 *Let $\mathcal{P}$ be the set of all paths for an instance of the approximate* EDGE-UNCERTAINTY-SPP *with restricted intervals. If $|\mathcal{P}| = \ell$ constant and there are no shared arcs between paths, then Algorithm 12 is $\ell\left(\frac{2}{\delta} + 2\right)$-update competitive, when compared to OPT_{exact}.*

Algorithm 12: Approximate EDGE-UNCERTAINTY-SPP with restricted intervals

```
Input: an instance of the approximate EDGE-UNCERTAINTY-SPP with restricted
       intervals with $|\mathcal{P}| = k$ constant
Output: an $s$-$t$ path
while there is no path $P$ with $P_U \leq (1+\delta) \cdot P'_L$ for all $P' \in \mathcal{P} \setminus \{P\}$ do
    foreach $P \in \mathcal{P}$ do
        $I \leftarrow$ interval with the largest width in $P$;
        update $I$;
return approximating path;
```

PROOF:
Let P^* be the shortest path. Since we always update one interval in each path, we will always update P^* and the path with smallest lower bound P'. Thus, the same analysis as for the two-path case holds and we will terminate after at most $k \cdot \left(\frac{2}{\delta} + 2\right)$ iterations. As ℓ updates are performed in each iteration, this leads to an update competitive ratio of $\ell \left(\frac{2}{\delta} + 2\right)$. ■

Remark: Unfortunately, the approach of always picking the largest width interval does not work for a graph with an arbitrary number of paths. This is true even when we modify Algorithm 12 so that in each iteration only one largest width interval in $\bigcup_{P \in \mathcal{P}} P$ is updated. Consider the following situation:



Again, the red arrows indicate the direction of the progress, that an update of that particular interval would make.
In this case, the algorithm that in each iteration updates the largest width interval, would only update an interval in P' if all the other paths are completely updated. An optimal update strategy of the exact problem, on the other hand, only updates P'' and one interval out of P' to show that P'' is the shortest path, i.e. $OPT_{exact} = 3$.
Since the number of paths is not bounded, the competitive ratio of the algorithm can be arbitrarily bad.

5.2.2 Minimum Spanning Tree

In the last section we discussed shortest path problems in the context of edge uncertainty. Now, we consider the minimum spanning tree problem in the same uncertainty setting.

As it turns out, the MST is a problem which is very well suited for this notion of uncertainty. Erlebach et al. [29] gave a 2-update competitive algorithm for the EDGE-UNCERTAINTY-MST problem and showed that this is the best ratio that can be achieved by any deterministic algorithm.

The EDGE-UNCERTAINTY-MST problem can be defined as follows:

Configuration:	an connected undirected graph $G = (V, E)$ a weight $w_e \geq 0$ for every edge $e \in E$
Intervals:	an open or trivial interval I_e for every edge $e \in E$
Goal:	a spanning tree T with weight less than or equal to the weight of every other spanning tree

Here, for each edge $e \in E$ the value L_e denotes the lower bound of I_e, i.e. $\inf(I_e)$, and U_e denotes the corresponding upper bound, i.e. $\sup(I_e)$.

We now present a 2-update competitive algorithm, that is very similar to the algorithm U-RED described in Erlebach et al. [29] and uses the concept of *witness sets* as first introduced by Bruce et al. [14]. A witness set for an uncertainty problem with interval data is defined in the following way:

Definition 5.2.8 (Witness set)
The set $\mathcal{W} \subseteq \mathcal{I}$ is called a *witness set* of $(\mathcal{I}, \phi)$, if for every feasible configuration $\mathcal{C}$, i.e. a configuration with $c_i \in I_i$, no element of $\phi(\mathcal{C})$ can be verified without updating an element of $\mathcal{W}$.

Algorithm 13: The general witness algorithm

```
Input: problem instance $(\mathcal{C}, \mathcal{I}, \phi)$
while no element of $\phi(\mathcal{C})$ can be verified using $\mathcal{I}$ do
    find a witness set $\mathcal{W}$;
    foreach interval $I$ in $\mathcal{W}$ do
        update $I$;
return element of $\phi(\mathcal{C})$ that can be verified using $\mathcal{I}$;
```

The concept of witness sets can be used to formulate a generic algorithm for uncertainty problems. This algorithm is called *witness algorithm* and is described in Algorithm 13. It

can be used to show general update competitivity results: Any algorithm of this structure that only uses witness sets with $|\mathcal{W}| \leq k$ is k-update competitive.
This property of the witness algorithm was first shown in Bruce et al. [14] for a particular uncertainty problem. In the following, we will extend this proof for our definition of uncertainty problems with interval data:

Theorem 5.2.9 *If there exists a constant k such that every witness set that gets updated by Algorithm 13 is of size at most k, then this algorithm is k-update competitive.*

PROOF:
Assume that the general witness algorithm needs n iterations to find a solution. Let $\mathcal{W}_1, \dots, \mathcal{W}_n$ be the witness sets in the same order as found by the witness algorithm. Let $\mathcal{I}_j$ for $j \leq n$ be the uncertainty intervals after updating $\mathcal{W}_1, \dots, \mathcal{W}_j$ and let $\mathcal{I}_0$ denote the original set of uncertainty intervals $\mathcal{I}$. Further, let $\mathcal{R}_i$ be the set of all possible realizations of $\mathcal{I}_i$.
Updating arcs corresponds to narrowing the respective interval. Therefore, updates never introduce new realizations and so $\mathcal{R}_0 \supset \mathcal{R}_1 \supset \cdots \supset \mathcal{R}_n$ holds. The set $\mathcal{W}_i$ in each iteration $i \leq n$ is chosen in such a way that it represents a witness set of $\mathcal{I}_{i-1}$ and thus $\mathcal{W}_i$ is also a witness set of $\mathcal{I}_j$ for all $j < i$. As the witness sets $\mathcal{W}_i$ are disjoint, the optimal update strategy has to contain at least n elements, i.e. $OPT \geq n$. Since further all witness sets are of size at most k, the witness algorithm updates at most $k \cdot n$ intervals and is therefore k-update competitive. ■

The algorithm for the EDGE-UNCERTAINTY-MST problem proposed in this section is such a witness algorithm, but before we can formulate this algorithm a few additional definitions for the EDGE-UNCERTAINTY-MST problem are necessary:

Definition 5.2.10 (Edge order)
Let $(\mathcal{C}, \mathcal{I}, \phi)$ be an instance of the EDGE-UNCERTAINTY-MST problem, where the underlying graph is denoted by $G = (V, E)$.
We say that $e \prec f$, if one of the following is true

1. $L_e < L_f$,
2. $L_e = L_f$ and $U_e \leq U_f$.

We assume in the following that all the edges $e_1, \dots, e_m$ are ordered in such a way, that we have $e_1 \prec e_2 \prec \cdots \prec e_m$, where edges with the same upper and lower weight limit are ordered arbitrarily.

This order is also used when comparing paths:

A path P can be characterized by the following vector $v_P = \left(x_1^P, \dots, x_m^P\right)$, where

$$x_i^P = \begin{cases} 1 & \text{if } e_i \in P, \\ 0 & \text{otherwise}. \end{cases}$$

Definition 5.2.11 (Smallest path)
We say that path P is *smaller* than P' with respect to $\prec$, if and only if $v_P < v_{P'}$ holds with respect to the reverse lexicographic order.
A path P is said to be the *smallest path*, if there does not exist a path P' which is smaller than P.

Definition 5.2.12 (Always maximal edge)
Let C be a cycle in the corresponding graph. We say the edge $e \in C$ is an *always maximal* edge in C, if $L_e \geq U_f$ for all $f \in C \setminus \{e\}$.

In contrast to the algorithm U-RED described in [29], our Algorithm 14 tries to identify for each edge, whether this edge will be for sure part of an MST, not part of an MST or whether it needs to be updated, since it is part of a witness set. This approach has the advantage that it does not require any restarts.

In the following we show that Algorithm 14 is correct and 2-update competitive. Before showing the update competitivity under the restriction to open or trivial intervals (Theorem 5.2.18), we discuss some technical preliminaries about MSTs first. Then, we introduce Lemma 5.2.15 to 5.2.17 that help to structure the actual proof of Theorem 5.2.18.

The first simple fact about MSTs, which we will need, is also known as the *cycle property*:

Proposition 5.2.13 *Let G be a connected weighted graph and let C be a cycle in G. If there exists an edge $e \in C$ with weight larger than the weights of all other edges of C, then e is not in any MST of G.*

This can be extended to the following proposition:

Proposition 5.2.14 *Let G be a connected weighted graph and let e be an edge in G. If for any cycle C containing e there always exists an edge on C which has larger weight than e, then e belongs to all MSTs of the graph.*

PROOF:
Without loss of generality we assume that the edge weights are distinct so that there is only one unique minimum spanning tree in G. If the weights are not distinct, one can

Algorithm 14: Update competitive algorithm for EDGE-UNCERTAINTY-MST

```
Input: an undirected graph $G = (V, E)$ with uncertain interval data on the edges
Output: a minimum spanning tree
index all edges such that $e_1 \prec e_2 \prec \cdots \prec e_m$;
let $\Gamma = (V, \varnothing)$ be without any edges;
$i \leftarrow 1$;
while $i \leq m$ do
    if $e_i$ closes a cycle in $\Gamma$ then
        $e_i$ is not in the MST;
        $i \leftarrow i + 1$;
    else
        let $C$ be the best cycle, wrt $\prec$, containing $e_i$;
        if $U_{e_i} > L_e$ holds $\forall e \in C \setminus \{e_i\}$ then
            if $\exists f \in C$ with $f \neq e_i$ and $U_f = \max_{e \in C} U_e$ then
                update $e_i$ and $f$;
            else
                update $e_i$;
            reindex the edges;
        else
            add $e_i$ to $\Gamma$;
            $i \leftarrow i + 1$;
return $\Gamma$
```

number the edges arbitrarily and then consider a lexicographic order of the pairs (edge weight, edge number) instead of the weight alone.

Now, let us assume there is an MST T that does not contain the edge $e = \{u, v\}$.
We know that for every cycle C containing e the unique edge with maximum weight is different from e. By applying the cycle property we therefore get that every possible u-v path contains an edge that is not in T, i.e. u and v are not connected in T. This contradicts the assumption that T is a spanning tree. ■

Using these propositions it is now easy to identify witness sets of size one:

Lemma 5.2.15 *Let C be the smallest cycle w.r.t. $\prec$ containing edge f. If $U_f > U_e$ holds for all $e \in C \setminus \{f\}$ and if we can further find an edge $e \in C \setminus \{f\}$ with $L_e > L_f$, then $\{f\}$ is a witness set.*

PROOF:
Since C is the smallest cycle, we know that for any cycle C' containing f there will always exist an edge $e' \in C'$ with $L_{e'} > L_f$. It now follows directly from Proposition 5.2.14 that there is a realization with $w_f < L_{e'}$ where f is part of the MST.
On the other hand, it follows from Proposition 5.2.13, that there also exists a realization where w_f is the largest edge in C and thus not part of any MST.
Hence, it cannot be decided whether the edge f is part of an optimal MST or not without updating edge f. ■

The following lemma is technical and it is solely needed in the proof of Lemma 5.2.17:

Lemma 5.2.16 *If the condition in line 10 of Algorithm 14 is reached and evaluates to true, then C contains no always maximal edge and e_i is nontrivial and not yet updated.*

PROOF:
We know, since e_i did not close a cycle in Γ, that C must contain at least one edge e' with $e_i \prec e'$, which also means that we have $L_{e_i} \leq L_{e'}$. Together with the fact that $U_{e_i} > L_{e'}$ it now follows that $U_{e_i} > L_{e_i}$, i.e. e_i is nontrivial.

As the condition in line 10 is true, only e_i remains as a possible candidate for an always maximal edge. This, however, cannot be the case since we have an edge e' with $e_i \prec e'$ but $U_{e_i} > L_{e'}$. ■

Lemma 5.2.17 *Assume that all uncertainty areas are open or trivial. The edges e_i and f as described in Algorithm 14 form a witness set in every iteration.*

PROOF:
This proof is very similar to the proof in [29]. The slight difference is due to the fact that the set Γ in Algorithm 14 is constructed differently.

Assume the edge set $\{e_i, f\}$ is not a witness set. This means that we can update some edges, but not f or e_i, to obtain the uncertainty graph $\mathcal{U}'$ that has a provable MST T'. Let U'_e and L'_e denote the upper and lower uncertainty limits of an edge e after all the updates have been performed. It now holds:

$$\begin{aligned}
&U'_{e_i} = U_{e_i}, L'_{e_i} = L_{e_i}\\
&U'_f = U_f, L'_f = L_f\\
&U'_e \leq U_e, L'_e \geq L_e \quad \text{for all other edges } e\,.
\end{aligned}$$

We will bring this assumption to a contradiction proving the following claims: We show that f is not in the MST T', that there exists a path P in $\mathcal{U}'$ connecting the two ends of edge f without using the edges e_i and f and that for all path edges $p \in P$ we have $p \prec f$

with respect to the current iteration. The existence of such a path P means that the cycle C could be modified in such a way that it still contains e_i but instead of f only edges that are ordered before f. This would lead to cycle $C' \ni e_i$ which is lexicographically smaller than C. However, since C is the smallest cycle containing e_i, these claims lead to a contradiction to the assumption that e_i and f are not a witness set.

We know from Lemma 5.2.16 that there is no always maximal edge in C. This means that there exists a realization where the weight of f is greater than the weight of any other edge in C. By Proposition 5.2.13 the edge f is not in any MST of that realization and therefore not in T'.

Let P be the unique path in T' connecting the ends of f. The path P does not contain the edge f as f is not part of T' and we have $U'_p \leq L_f$ for all $p \in P$. Since $U_{e_i} > L_f$, the edge e_i is also not in P.

By combining the fact that f is always nontrivial with the observation that $U'_p \leq L_f$ for all $p \in P$, we get

$$L_p \leq L'_p \leq U'_p \leq L_f < U_f$$

for all $p \in P$. Now, there are two cases: either p was nontrivial in the beginning, and thus $L_p < U'_p$, or p was trivial, i.e. $L_p = L'_p = U'_p = U_p$. Thus, we either have $L_p < L_f$ or $L_p = U_p \leq L_f < U_f$ and therefore in both cases $p \prec f$.
This means that there exits a cycle containing e_i which is smaller than C. Since this cannot be the case, we have a contradiction to our assumption. ■

By applying the last three lemmata we can now easily conclude the following result:

Theorem 5.2.18 *Algorithm 14 is 2-update competitive.*

Proof:
If e_i and f exist, we know from Lemma 5.2.17 that they form a witness set of size two.

In line 14 we have the situation that e_i is the unique element with maximum upper limit in C and since e_i did not close a cycle in Γ, there exists at least one other element $e' \in C$ with $L_{e'} > L_{e_i}$. Lemma 5.2.15 states that in such cases e_i forms a witness set of size one.

Due to Lemma 5.2.16 and the fact that after updating and reindexing the new index of edge e_i will again be at least i, the restarted algorithm would perform exactly the same for all edges in positions 1 to $i-1$. Thus, an explicit restart is not necessary. ■

It remains to show that Algorithm 14 is also correct:

Theorem 5.2.19 *Algorithm 14 terminates and returns an MST of G.*

Proof:
Again, without loss of generality we assume that the exact edge weights are distinct so that there will only be one unique minimum spanning tree.

In the algorithm we only add an edge e_i, if there exists another edge $c \in C$ with $U_{e_i} \leq L_c$, where C is the best cycle containing e_i. Since C is the smallest cycle w.r.t. $\prec$, we know that for each cycle containing e_i there always has to be an edge e' whose lower limit is at least the upper limit of e. We will now show using Proposition 5.2.14 that this means that edge e_i belongs to the MST for every realization. Here, two cases need to be distinguished:

1. $U_{e_i} = L_{e_i} = w_{e_i}$ holds and we have $w_{e_i} \leq L_{e'}$. Due to the fact that the weights are distinct we further have $w_{e_i} < w_{e'}$ whether e' was trivial or not.
2. $L_{e_i} < w_{e_i} < U_{e_i} \leq L_{e'}$ holds and Proposition 5.2.14 can be applied for all realizations.

We omit an edge e_i, if adding e_i closes a cycle in Γ. Since Γ only consists of edges where it is guaranteed, that they belong to the MST, the edge e_i cannot be part of the MST.

The number of edges is finite and due to Lemma 5.2.16 an edge is considered at most twice (once uncertain and once updated), the algorithm will eventually terminate and the MST is returned. ■

Just as the algorithm proposed by Erlebach et al. [29], Algorithm 14 is 2-update competitive. Its advantages over the former are discussed in Section 5.3 as it can be easily extended to also work in the probabilistic setting introduced in that section.

Approximate MST

After the last few pages focused on the uncertain MST problem, we now analyze these results in the context of our actual goal – the traveling salesman problem:
If the inputs are metric, an approximate TSP solution can be easily constructed from an optimal MST solution (see e.g. Williamson and Shmoys [86, Theorem 2.12]). As the EDGE-UNCERTAINTY-MST has a constant update competitive ratio, it seems reasonable to use EDGE-UNCERTAINTY-MST as the basis of an update competitivity result for the (approximate) TSP. In this case we have to compare the number of updates for the MST problem with OPT of the corresponding TSP instance. Unfortunately, such a comparison is much harder due to the structural differences between those two problems. However, if there exists a general upper bound on the number of updates for any instance of the EDGE-UNCERTAINTY-MST, this bound can directly be used to obtain such a bound for corresponding TSP. Therefore, we will analyze next under which conditions such a bound for the EDGE-UNCERTAINTY-MST exists:

Lemma 5.2.20 *There exist instances of the* EDGE-UNCERTAINTY-MST *problem where* $OPT = \Omega(n^2)$ *for any possible configuration.*

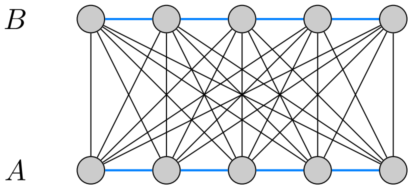
PROOF:
Consider the complete Graph G with the uncertainty area $(0, 1)$ for all edges. For any possible feasible configuration even the optimal algorithm needs to update all edges. ■

The EDGE-UNCERTAINTY-MST problem does not have a constant bound on the number of updates and it is therefore not easily suited as the basis of an approximate TSP algorithm as described in the last paragraph. Therefore, as a next step we will further relax the problem such that an approximation of the MST is sufficient – the *Approximate* EDGE-UNCERTAINTY-MST problem:

Configuration:	an connected undirected graph $G = (V, E)$ a weight $w_e \geq 0$ for every edge $e \in E$
Intervals:	an open or trivial interval I_e for every edge $e \in E$
Goal:	a spanning tree T whose weight is at most $(1 + \delta)$ times the weight of an MST, where $\delta > 0$ is given

For the analysis of the number of updates, we first consider the following example:

Example 5.2.21 The approximate EDGE-UNCERTAINTY-MST problem is not update competitive:



The blue edges have a fixed known cost of 1. All edges between the sets A and B have the open uncertainty interval $(1, M)$. Only one edge e has weight $1 + \varepsilon$ while all the others have weight $M - \varepsilon$, with $\varepsilon < \delta$.

In this case the optimal update strategy only contains edge e, while any deterministic algorithm not knowing the exact distances has to update all $\frac{n}{2} \cdot \frac{n}{2}$ edges to find a $(1 + \delta)$-approximation in the worst case.

The graph from Example 5.2.21 can also be used as a worst-case instance with respect to the total number of updates required:

Corollary 5.2.22 *There are approximate* EDGE-UNCERTAINTY-MST *instances where any deterministic algorithm needs* $\Omega(n^2)$ *updates to verify a solution.*

In this context it does not make much sense to compare the updates required for the approximate EDGE-UNCERTAINTY-MST problem with OPT_{exact}: Algorithm 14 for the exact EDGE-UNCERTAINTY-MST problem is update competitive and returns an optimal solution which is obviously also a $(1+\delta)$-approximation.
However, we will revisit this idea in Section 5.3 within a slightly different uncertainty setting.

Although the exact MST can be efficiently tackled in the uncertainty model, it still does not provide us with a framework that can be easily extended for the TSP. It is therefore necessary to also thoroughly analyze the traveling salesman problem itself.

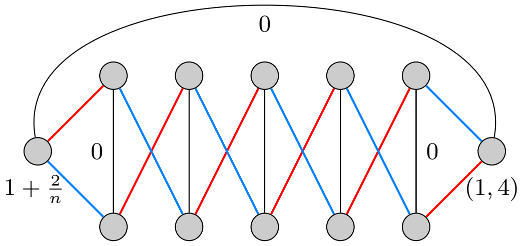
5.2.3 Traveling Salesman Problem

The final problem we consider in this context is the *Traveling Salesman Problem (TSP)*. This leads to the following EDGE-UNCERTAINTY-TSP:

Configuration:	a connected undirected graph $G=(V,E)$ a weight $w_e \geq 0$ for every edge $e \in E$
Intervals:	an open or trivial interval I_e for every edge $e \in E$
Goal:	a tour T visiting every node exactly once whose cost $\sum_{e \in T} w_e$ is less than or equal to the cost of every other tour that visits all the nodes

We first look at an example that shows that for the EDGE-UNCERTAINTY-TSP an update competitive algorithm cannot exist.

Example 5.2.23 The EDGE-UNCERTAINTY-TSP is not update competitive:



The black edges have a fixed weight of 0. The red edges have uncertainty intervals of $(1, 4)$ while all the blue edges have a fixed weight of $1 + \frac{2}{n}$. Exactly one red edge e has weight 3 and all the others have weight $1 + \varepsilon$. This graph has exactly three Hamiltonian Cycles:

1. The *blue cycle* using all the blue edges and then using the long black loop
2. The *red cycle* using all the red edges and the loop
3. The cycle that uses all blue and red edges but not the loop

Since the loop has a length of 0 and the colored edges have positive weight, the length of the third cycle will always be longer than the length of the first two cycles. Thus, we only need to compare the length of the red and the blue cycle. The blue cycle has a fixed length of $\ell_{BLUE} = \frac{n}{2} + 1$. For an optimal update strategy it is sufficient to only update edge e, as this shows that the cycle using only the blue edges is the shortest: $\ell_{BLUE} < \frac{n}{2} + 2 < \ell_{RED}$. By choosing $\varepsilon < \frac{1}{n}$ we assure that even after updating any number of non-expensive edges the red cycle could still be the shortest.

Furthermore, it is impossible to give a nontrivial upper bound on the number of updates required:

Theorem 5.2.24 *There are instances of the* EDGE-UNCERTAINTY-TSP *where* $OPT = \Omega(n^2)$ *for all possible configurations.*

PROOF:
Let K_n be the complete undirected graph with $n \geq 5$. All the uncertainty intervals are $(1, 2)$ and have arbitrary weights. Let one optimal solution be denoted by

$$C := (v_1, v_2, \dots, v_{i-1}, v_i, v_{i+1}, \dots, v_{j-1}, v_j, v_{j+1}, \dots, v_{n-1}, v_n)$$

To prove that C actually is a shortest cycle, any optimal update strategy has to rule out all other possibly shorter Hamiltonian cycles, i.e. it has to show that $U_C \leq L_{C'}$ holds for all other Hamiltonian cycles C'. Since each uncertainty area is an open interval, this is only possible by raising the lower bound $L_{C'}$ on all other cycles. Next, we consider the following cycles:

$$(v_1, v_2, \dots, v_{i-1}, v_j, v_{j+1}, \dots, v_{n-1}, v_n, v_i, v_{i+1}, \dots, v_{j-1}) \quad \forall 2 < i < j < n$$

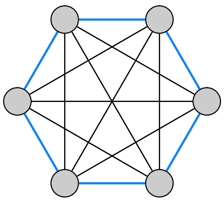
All of these cycles differ from C in exactly three edges, namely (v_{i-1}, v_j), (v_n, v_i) and (v_{j-1}, v_1) and these edges are different for every possible i, j with $2 < i < j < n$. So, to raise the lower bounds the optimal strategy needs to update at least one of the three edges. Since there are exactly $\binom{n-3}{2}$ possible ways of choosing i and j, at least $\binom{n-3}{2}$ updates will be required. ∎

Very similar to the uncertainty problems we discussed so far, we will now take a closer look at the approximated version, i.e. the *approximate* EDGE-UNCERTAINTY-TSP:

Configuration: a connected undirected graph $G = (V, E)$
a weight $w_e \geq 0$ for every edge $e \in E$
Intervals: an open or trivial interval I_e for every edge $e \in E$
Goal: a tour T visiting every node exactly once whose cost is within a factor of $(1 + \delta)$ of the cost of a shortest tour that visits all the nodes

Unfortunately, even the approximate version of the EDGE-UNCERTAINTY-TSP does not have a usable upper bound on the number of updates. To show this, we give an instance where any deterministic algorithm has to update order of n^2 edges in the worst case, while $OPT = n$.

Example 5.2.25 The *approximate EDGE-UNCERTAINTY-TSP* is not update competitive:

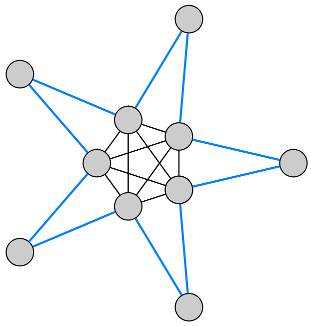


All edges have uncertainty intervals of $(1, M)$, where the blue edges have a weight of $1 + \varepsilon$, while the black edges are close to M. For ε small enough it is clear that the tour consisting of only the blue edges is a $(1 + \delta)$-approximation. Therefore, the optimal update strategy only needs to update n edges, while an deterministic algorithm needs to update all $\binom{n}{2}$ edges in the worst case. This leads to an update competitive ratio of $\frac{\Omega(n^2)}{n}$.

Corollary 5.2.26 *There are instances of the approximate* EDGE-UNCERTAINTY-TSP*, where any deterministic algorithm in the worst case needs $\Omega(n^2)$ updates to verify a solution.*

As we already mentioned in the last section, that it is a very natural idea to find an approximation for the TSP on the basis of an MST computation. We also showed that due to the fact that no upper bound on the number of updates exits, it would be very difficult to analyze the update competitive ratio. But the actual situation is even worse, as there are instances where the EDGE-UNCERTAINTY-MST is much harder than the approximate EDGE-UNCERTAINTY-TSP in terms of updates:

Example 5.2.27 The EDGE-UNCERTAINTY-MST can require more updates than the approximate EDGE-UNCERTAINTY-TSP:



The blue edges have a weight of 2 and all other edges have an uncertainty interval of $(1, M)$. The graph contains exactly one Hamiltonian cycle. For it the weight of all edges is known and no updates are necessary. Any MST algorithm, on the other hand, would need to update all $\Omega(n^2)$ black edges, as connecting two nodes using any black edge could be cheaper than a blue edge.

This results makes it clear that the uncertainty framework introduced in this section is not sufficient to provide an update competitive algorithm for the TSP, as in this framework the "perfect adversary" OPT is used to evaluate the number of updates.
In the next section we slightly weaken this criterion: By introducing probabilities on the edges it allows us to talk about the *expected* number of updates. Now, the overall performance of an algorithm no longer depends that much on one certain worst-case instance.

5.3 Probabilistic Uncertainty

The observations in the last section showed that for every problem but the MST and some very restricted shortest path problems we have no update competitivity and no upper bound on the number of updates. In this section we therefore introduce a certain notion of probability into our uncertainty. This leads to a slightly less general model, but it still is a realistic version especially in the context of the WCP. It allows us to make stronger observations about the number of required updates.

Definition 5.3.1 (Uncertainty problem with constant probability)
Each instance P of an uncertainty problem with constant probability for a given $\delta > 0$ is a tuple $(\mathfrak{X}, \mathcal{I}, p_\delta, \phi)$, where

- $\mathcal{I}$ is an ordered set of intervals $\mathcal{I} = \{I_1, \ldots, I_n\}$, with $I_i \subset \mathbb{R}$ for all $I_i \in I$,
- $\mathfrak{X}$ is an ordered set of independent random variables $\mathfrak{X} = \{X_1, \ldots, X_n\}$ with X_i being a random variable with values in I_i and $\mathbb{P}(X_i \leq \inf(I_i) \cdot (1 + \delta)) \geq p_\delta$,
- and $\phi(\mathfrak{X})$ is the set of solutions for P.

The pair (p_δ, ϕ) is the same for all instances of a problem and can therefore be used as a representation of the problem. The function ϕ is identical to the function of the corresponding uncertainty problem with interval data.

For this setting we change the definition of updates slightly:

Definition 5.3.2 (Update)
Let $P = (\mathfrak{X}, \mathcal{I}, p_\delta, \phi)$ be an instance of an uncertainty problem with constant probability for a given $\delta > 0$. For the set of uncertainty intervals $\mathcal{I} = \{I_1, \ldots, I_n\}$, an update of interval I_i, reveals the corresponding realization x_i of the random variable X_i, i.e. after the update the new set of uncertainty intervals is $\{I_1, \ldots, I_{i-1}, [x_i, x_i], I_{i+1}, \ldots, I_n\}$.

The goal now is to find an algorithm that is able to verify an element of $\phi(\mathfrak{X})$ by using a certain number of updates. To assess the quality of such an algorithm for an uncertainty problem with constant probability, we use the following four classifications. All of them only take the number of updates into account, as in the last section, running times are not considered.

Definition 5.3.3 (Expected number of updates)
Let P be an instance of an uncertainty problem with constant probability. Further, let the random variable $Y_\mathcal{A}$ represent the number of updates required by algorithm $\mathcal{A}$ to verify an element of $\phi(\mathfrak{X})$. Now, $\mathbb{E}[Y_\mathcal{A}]$ corresponds to the expected number of updates for $Y_\mathcal{A}$.

Definition 5.3.4 (Worst-case number of updates)
Let P be an instance of an uncertainty problem with constant probability. The number of updates required by algorithm $\mathcal{A}$ in the worst case corresponds to the maximum number of updates required for any instance of the corresponding uncertainty problem with interval data.

Definition 5.3.5 (Expected update competitive ratio)
Let P be an instance of an uncertainty problem with constant probability. Further, let the random variable $Y_\mathcal{A}$ represent the number of updates required by algorithm $\mathcal{A}$ to verify an element of $\phi(\mathfrak{X})$, while Y_{OPT} represents the number of updates needed by an algorithm that knows every realization in advance. Now, $\mathbb{E}\left[\frac{Y_\mathcal{A}}{Y_{OPT}}\right]$ corresponds to the expected update competitive ratio of $\mathcal{A}$.

Definition 5.3.6 (Worst-case update competitive ratio)
Let P be an instance of an uncertainty problem with constant probability. Here, the worst-case update competitive ratio of algorithm $\mathcal{A}$ is equal to the highest update competitive ratio of algorithm $\mathcal{A}$ for any instance of the corresponding uncertainty problem with interval data.

As uncertainty problems with constant probability are closely related to their corresponding uncertainty problem with interval data, it is easy to see the following facts:

- Any correct algorithm $\mathcal{A}$ for an uncertainty problem with interval data can also be used as an algorithm for the corresponding uncertainty problem with constant probability.
- Any update competitivity result of $\mathcal{A}$ carries over as a worst-case result in this model.

MST

The first problem we revisit is the EDGE-UNCERTAINTY-MST. For the exact case we know that EDGE-UNCERTAINTY-MST with constant probability is worst-case 2-update competitive (Theorem 5.2.18) and that there are instances where the number of updates is always $\Omega(n^2)$ (Lemma 5.2.20). Due to the fact that the problem is worst-case update competitive we also know that it is update competitive in expectation.

Next, we take a look at the approximate EDGE-UNCERTAINTY-MST problem. We already know that the problem is not worst-case update competitive (Example 5.2.21) and requires $\Omega(n^2)$ updates in the worst-case (Corollary 5.2.22). However, the probabilistic setting now allows us to show some results on the expected number of updates.
For this purpose, we use a new algorithm as described in Algorithm 15. It is essentially very similar to Kruskal's algorithm (see for example Cormen et al. [17, Chapter 23]), but only edges whose exact weights are close enough to their lower bounds are added.

Theorem 5.3.7 *The Algorithm 15 terminates and returns a spanning tree whose cost is within a factor of* $(1+\delta)$ *of the cost of an MST of* $G=(V,E)$.

PROOF:
Let us consider the following modified edge weights w'_e for each edge $e \in E$:

$$w'_e := \begin{cases} L_e & \text{if } w_e \leq L_e \cdot (1+\delta), \\ w_e & \text{otherwise.} \end{cases}$$

By adding a small enough ε it is still possible to maintain open intervals. But in order to avoid handling unnecessary ε, they have been omitted in the following.

Algorithm 15: Approximation MST-Algorithm

```
Input: an undirected graph $G = (V, E)$ with uncertain interval data on the edges
Output: a spanning tree in $G$
sort all edges such that $e_1 \prec e_2 \prec \cdots \prec e_m$;
let $\Gamma = (V, \varnothing)$ be without any edges;
$i \leftarrow 1$;
while $i \leq m$ do
    if $e_i$ closes a cycle in $\Gamma$ then
        $e_i$ is not in the MST;
        $i \leftarrow i + 1$;
    else
        update $e_i$;
        if $w_{e_i} \leq L_{e_i} \cdot (1 + \delta)$ then
            add $e_i$ to $\Gamma$;
            $i \leftarrow i + 1$;
        else
            reindex remaining edges;
return $\Gamma$
```

We note that, when the weights are modified to w' and all other input data stays the same, Algorithm 15 returns an MST with respect to w': For w' the algorithm only adds edges whose weight is at their lower bound and the edges added are exactly the same edges that Kurskal's algorithm would add for the regular MST problem with weights w'. The set of edges returned by Algorithm 15 is an MST with respect to w' and thus definitely a spanning tree of G. The weights w' have further been constructed in such a way that Algorithm 15 returns the same tree T no matter whether the input has the weights w or the weights w'.

It remains to show that T is also a $(1 + \delta)$-approximation of the MST M (w.r.t. w). As the weights w' only decrease the weights, we know that $\sum_{e \in T} w'_e \leq \sum_{e \in M} w_e$ holds. This leads to the following conclusion

$$\sum_{e \in T} w_e \leq (1 + \delta) \sum_{e \in T} w'_e \leq (1 + \delta) \sum_{e \in M} w_e \,,$$

which proves our claim. ■

Algorithm 15 finally allows us to formulate an upper bound on the number of updates for the MST problem under uncertainty:

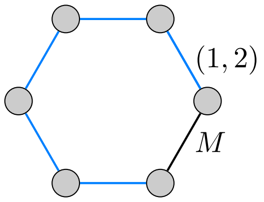
Theorem 5.3.8 *In expectation Algorithm 15 finds a $(1+\delta)$-approximation with no more than $\mathcal{O}(n)$ updates.*

PROOF:
Let Y_k be the random variable which denotes the number of iterations that the algorithm needs to find the k-th edge of the MST. If the algorithm picks an edge with trivial or already update uncertainty area we have $U_e = w_e = L_e$ and $\mathbb{P}(w_e \leq L_e \cdot (1+\delta)) = 1 \geq p_\delta$. If we assume that the algorithm always picks a not yet updated edge, we have

$$\mathbb{E}[Y_k] \leq \sum_{i=1}^{\infty} i \cdot (1-p_\delta)^{i-1} p_\delta = \frac{1}{p_\delta}.$$

Since all Y_k are independent under these assumptions, we have that the algorithm needs at most $\frac{n-1}{p_\delta}$ updates in expectation. If the algorithm happens to pick an already updated edge, this edge is definitely part of the returned tree and no further iterations are necessary for the k-th edge. Hence, the number of updates will only decrease. ■

Although Theorem 5.3.8 guarantees a linear number of updates in expectation, the update competitive ratio, when compared with OPT_{exact} for the exact MST problem, can still be arbitrarily bad. Consider the following instance:



This cycle graph has exactly one edge with large weight M, while all the other edges have the uncertainty interval $(1,2)$. For M large enough, the MST always consist of all edges but the edge with weight M and we have $OPT = 0$. Algorithm 15 on the other hand updates all $n-1$ smaller edges before adding them.

In the following we will therefore combine Algorithm 15 with the update competitivity result from the last section. The resulting Algorithm 16 uses the same criteria as Algorithm 14 to identify whether an edge is in a witness set or not. But if an updated edge has a reveled weight close enough to the lower bound, it can directly be added to Γ.
As the 2-update competitivity follows directly from Theorem 5.2.18, we first show the correctness of the algorithm.

Algorithm 16: Approximation MST-Algorithm

```
Input: an undirected graph $G = (V, E)$ with uncertain interval data on the edges
Output: a minimum spanning tree in $G$
sort all edges such that $e_1 \prec e_2 \prec \cdots \prec e_m$;
let $\Gamma = (V, \varnothing)$ be without any edges;
$i \leftarrow 1$;
while $i \leq m$ do
    if $e_i$ closes a cycle in $\Gamma$ then
        $e_i$ is not in the MST;
        $i \leftarrow i + 1$;
    else
        let $C$ be the best cycle, wrt $\prec$, containing $e_i$;
        if $U_{e_i} > L_e$ holds $\forall e \in C \setminus \{e_i\}$ then
            let $f \in C$ such that $U_f = \max_{e \in C} U_e$;
            update $e_i$ and $f$;
            if $w_{e_i} \leq L_{e_i} \cdot (1 + \delta)$ then
                add $e_i$ to $\Gamma$;
                $i \leftarrow i + 1$;
            else
                reindex remaining edges;
        else
            add $e_i$ to $\Gamma$;
            $i \leftarrow i + 1$;
return $\Gamma$
```

Theorem 5.3.9 *Given an undirected uncertainty graph G, the Algorithm 16 terminates and returns a spanning tree whose cost is within a factor of $(1+\delta)$ of the cost of an MST of G.*

PROOF:
The proof is very similar to the one of Theorem 5.3.7. We first consider the same modification w' of the edge weights:

$$w'_e := \begin{cases} L_e & \text{if } w_e \leq L_e \cdot (1+\delta), \\ w_e & \text{otherwise.} \end{cases}$$

We assume without loss of generality that the weights w'_e are distinct for every edge $e \in E$.

We now have to show that Algorithm 16 returns an MST for the modified weights and that it returns the same set of edges for the unmodified weights. After that we can apply the same argumentation as in the proof of Theorem 5.3.7 to show that the algorithm finds a $(1+\delta)$-approximation.

The algorithm adds an edge e, if its weight w'_e is at its lower bound (and does not close a cycle) or if $U_e \leq \max_{e' \in C \setminus \{e\}} L_{e'}$ holds for all cycles C containing e. Due to the correctness of Kruskal's algorithm the first condition always adds edges which are in the MST for the edge weights w'. For the second condition we have the same situation as in the proof of Theorem 5.2.19 and by using the same argumentation, edge e must be in the MST for the weights w'.

It is easy to see that Algorithm 16 returns the same spanning tree T whether the exact weights are w or w'. Therefore, we can compare the weight of T with the weight of the MST M with respect to w:

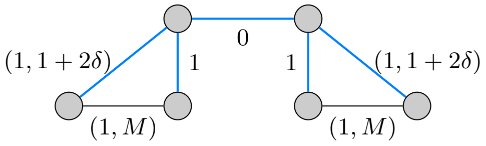
$$\sum_{e \in T} w_e \leq (1+\delta) \sum_{e \in T} w'_e \leq (1+\delta) \sum_{e \in M} w_e \,.$$

■

Corollary 5.3.10 *For any $\delta > 0$ Algorithm 16 will never need more than twice as many updates as OPT_{exact} needs to solve the exact MST problem.*

PROOF:
Algorithm 16 never requires more updates than Algorithm 14, since it uses the same criterion to identify edges that need to be update. As Algorithm 14 is a 2-update competitive algorithm for the EDGE-UNCERTAINTY-MST, the claim holds. ■

Remark: Unfortunately, Algorithm 16 is not update competitive:



In this example the spanning tree containing only the blue edges has cost less than $4 \cdot (1+\delta)$, while any spanning tree will cost at least 4.
Algorithm 16, on the other hand, updates at least one of the edges with interval $(1, 1+2\delta)$.

This complication can be avoided by adding a preprocessing step to the algorithm in which it checks whether any updates are necessary or not.
As in the uncertainty setting, we are not concerned about running times, this can be

achieved by enumerating all spanning trees and then comparing their bounds. If $OPT = 0$ holds, there must be a spanning tree whose upper bound is not more than $(1 + \delta)$ times the lower bound of all other trees.
After the addition of such a step, the resulting algorithm is $\mathcal{O}(n)$-update competitive in expectation.

Using Algorithm 16 we showed that in this setting the approximate EDGE-UNCERTAINTY-MST problem in expectation only needs a linear number updates and it is also $\mathcal{O}(n)$-update competitive in expectation. This gives us the foundation we need to find and update competitive algorithm for the approximate TSP.

TSP

The last problem we consider in the probabilistic setting is the EDGE-UNCERTAINTY-TSP. We already know that there are instances of the EDGE-UNCERTAINTY-TSP with constant probability where the number of updates is $\Omega(n^2)$ (Theorem 5.2.24) and that the problem has a worst-case update competitive ratio of $\Omega(n)$ (Example 5.2.23).

For the approximate EDGE-UNCERTAINTY-TSP with constant probability we have that it is not worst-case update competitive (Example 5.2.25) and also $\Omega(n^2)$ updates are required in the worst-case (Corollary 5.2.26).

The graph used in Example 5.2.23 to show that the TSP is not update competitive only contains one edge with large weight, while all the other edge weights are very close to their respective lower bounds. This fact can be used to derive results on the expected update competitive ratio:

Lemma 5.3.11 *In expectation the* EDGE-UNCERTAINTY-MST *with constant probability has an update competitive ratio which is at least arbitrarily close to* $\frac{1}{1-p_\delta}$.

PROOF:
In Example 5.2.23 we constructed an instance for the general EDGE-UNCERTAINTY-MST where the algorithm needs to find one edge with large weight, while the other edges are arbitrarily close to their lower bound. We can now convert this instance into an instance with constant probability:
The red edges have weight $1+\varepsilon$ with a probability of p_δ and a weight of 3 with a probability of $1 - p_\delta$. To find a verifiable solution, any algorithm, either has to update all the $\frac{n}{2}$ red edges or reveal an edge weight of 3. As all the red edges are indistinguishable for any algorithm $\mathcal{A}$ not knowing the exact weights, it has to update edges until an edge with high weight is found. On the other hand, if there exists at least one edge with weight 3,

we have $OPT = 1$; if there is no such edge exists, we have $OPT = \frac{n}{2}$.
Let the random variable $Y_{\mathcal{A}}$ represent the number of updates required by algorithm $\mathcal{A}$ and let Y_{OPT} be the random variable for the number of updates of an algorithm knowing the weights. For the expected update competitive ratio we have

$$\mathbb{E}\left[\frac{Y_{\mathcal{A}}}{Y_{OPT}}\right] = \mathbb{E}\left[Y_{\mathcal{A}}\right] \cdot \mathbb{E}\left[\frac{1}{Y_{OPT}}\right] .$$

Since

$$\mathbb{E}\left[\frac{1}{Y_{OPT}}\right] = 1 \cdot \left(1 - p_\delta^{\frac{n}{2}}\right) + \frac{2}{n} \cdot p_\delta^{\frac{n}{2}}$$

converges against 1, while $\mathbb{E}\left[Y_{\mathcal{A}}\right]$ goes to $\frac{1}{1-p_\delta}$, we also get that the expected update competitive ratio $\mathbb{E}\left[\frac{Y_{\mathcal{A}}}{Y_{OPT}}\right]$ converges (from below) against $\frac{1}{1-p_\delta}$ for $n \to \infty$.
However, there might be instances where the update competitive ratio is worse than this.■

This uncertainty setting with constant probability is related to the well-known field of *random graphs*. Here, certain properties of graphs based on different random models are analyzed. An overview of the theory behind such graphs can for example be found in Bollobás [13].

In the following we analyze the approximate EDGE-UNCERTAINTY-MST with constant probability in the context of graphs where an edge only exist with a certain probability. To connect an uncertainty problem with constant probability with random graphs, we assume that an edge only exists if it is within a factor of $(1 + \delta)$ of its lower bound:

Theorem 5.3.12 *Let G be the complete undirected graph with n nodes. Let the intervals on all arcs be $[L, \infty]$ and for every edge e let its random variable X_e have $\mathbb{P}(X_e = L) = p$ and $\mathbb{P}(X_e = \infty) = 1 - p$.*
Then there exists an algorithm which either finds a Hamiltonian path with finite cost between two specified vertices of G, or shows that no such path exists. The algorithm runs in expected time $\frac{cn}{p}$ and uses storage cn, where c is an constant.

PROOF:
Thomason [84] shows the existence of an algorithm that finds a Hamiltonian path in $\mathcal{O}(\frac{n}{p})$ in expectation on random graphs where an edge exists with the probability p. ■

Corollary 5.3.13 *Considering the following uncertainty problem with constant probability:*

Configuration:	*the complete undirected graph $G = (V, E)$ with n nodes, a random variable X_e for every edge $e \in E$ that fulfills the conditions for an uncertainty problem with constant probability*
Intervals:	*open or trivial intervals with identical lower bounds, i.e. $L = \inf(I_e)$ for all the edges $e \in E$*
Goal:	*a Hamiltonian cycle whose cost is within a factor of $(1+\delta)$ of the weight of a shortest Hamiltonian cycle*

There exists an algorithm which finds a Hamiltonian cycle only containing edges which are at most a factor of $(1+\delta)$ away from their lower bound or shows that no such cycle exists. It updates $\mathcal{O}(n)$ edges in expectation.
If the algorithm succeeds, the resulting Hamiltonian cycle is a $(1+\delta)$-approximation.

PROOF:
The algorithm from Theorem 5.3.12 can be used to find such a Hamiltonian cycle. Since the algorithm has an expected linear running time, this results in $\mathcal{O}(n)$ updates, even if the algorithm requires constantly many update in every step. ■

Since in our case we have an edge probability that does not depend on n, the following holds for the uncertainty problem described in Corollary 5.3.13 according to Komlós and Szemerédi [57]:

$$\lim_{n\to\infty} \mathbb{P}(G \text{ has a Hamiltonian cycle that only uses cheap edges}) = 1\,.$$

These findings suggest that there might be special cases of the approximate EDGE-UNCERTAINTY-TSP with constant probability that can be solved performing $\mathcal{O}(n)$ updates in expectation. Such an algorithm is introduced in the next section.

5.4 Uncertainty in Metric Space

By introducing the probability model we could give an algorithm that is able to find a $(1+\delta)$-approximation of the optimal MST with order of n updates in expectation. This is an important prerequisite to our goal of being able to approximate the TSP under uncertainty.

It is known for approximation algorithms for the TSP in the regular setting, i.e. without uncertainty, that for any $\alpha > 1$ there does not exist an α-approximation algorithm for the TSP on n cities, if we assume that $\mathcal{P} \neq \mathcal{NP}$ (see e.g. Williamson and Shmoys [86, Theorem 2.9]). It is therefore necessary to restrict the input to metric instances in order

to obtain approximation results. To make use of these results in our uncertainty setting, we also have to apply this restriction to the corresponding uncertainty problems with constant probability.
Hence, we only consider uncertainty problems with interval data on undirected metric graphs, i.e. weighted complete graphs $G = (V, E)$ where for each triple $i, j, k \in V$

$$w_{\{i,k\}} \leq w_{\{i,j\}} + w_{\{j,k\}}$$

holds. This restriction can easily be added to problems with interval data, but it gets a little more complicated for uncertainty problems with constant probability. Here, the main difficulty consists in the fact that due to the required metric structure the random variables are no longer independent. This leads to the following problem definition:

Definition 5.4.1 (Metric uncertainty problem with constant probability)
Each problem instance P for a given $\delta > 0$ is a tuple $(G, \mathcal{I}, p_\delta, \phi)$, with

- $G = (V, E)$ being a complete undirected graph,
- $\mathcal{I} = \{I_1, \ldots, I_m\}$ being an ordered set of intervals, one for each edge $e \in E$, such that for each I_i and all $w_i \in I_i$ the set $I_1 \times \cdots \times I_{i-1} \times \{w_i\} \times I_{i+1} \times \cdots \times I_m$ contains an element which corresponds to metric edge weights on G,
- $p_\delta \in [0, 1]$,
- and $\phi(W)$ being a function that for every ordered set of weights $W = \{w_1, \ldots, w_m\}$ returns the set of solutions.

The goal is to find a solution that is contained in $\phi(W)$ for each possible set of edge weights $W = \{w_1, \ldots, w_m\}$ such that W is metric and $w_e \in I_e$ for all edges $e \in E$. The pair (p_δ, ϕ) is the same for all instances of a problem and thus represents the problem itself.

In the metric setting the definition of updates differs from the definition in the general uncertainty setting. In addition to revealing an edge weight, we also have to adapt the other edge uncertainty intervals accordingly in order to assure that for a certain interval I_e their does not exist a realization w_e which violates the triangle inequalities. In particular, this assures that the probability $\mathbb{P}(w_e \leq \inf(I_e)(1 + \delta))$ can always be positive.

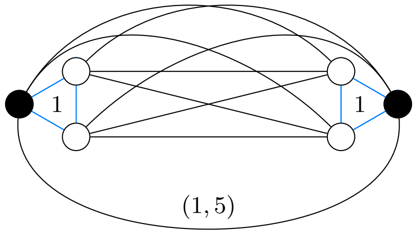
Definition 5.4.2 (Update)
Let $P = (G, \mathcal{I}, p_\delta, \phi)$ be an instance of a metric uncertainty problem with constant probability for a given δ. An update of an edge interval I_e transforms P into a new feasible instance P' of the same problem where the weight of edge e is “revealed”. More formally, this means an update of I_e leads to $P' = (G, \mathcal{I}', p_\delta, \phi)$. Here, the interval I'_e is set to $[w_e, w_e]$, for w_e chosen from I_e at random with respect to $\mathbb{P}(w_e \leq \inf(I_e)(1+\delta)) \geq p_\delta$. For all other edges $f \in E \setminus \{e\}$ the interval $I'_f \subseteq I_f$ is defined in such a way that $\mathcal{I}' = \{I'_1, \ldots, I'_m\}$ is a feasible set of intervals for the given graph G.

The criteria which we use to measure the quality of a given algorithm for a certain problem are the same as in the last section, namely *expected/worst-case number of updates* and *expected/worst-case competitive ratio.*

Remark: Since metric uncertainty problems with constant probability are in some sense a special case of the uncertainty models introduced in the previous section, any correct algorithm for those models can also be applied in this setting. However, the update competitive ratio might change, because in the metric setting updates are more “powerful” as they reveal information not only about the updated arc.
Also the worst-case results on the number of updates can change, as the worst case might no longer be feasible in the metric setting.

In the next paragraphs we therefore revisit the problems introduced in Section 5.2 in the metric context and show that in this setting they are not considerably easier to solve in terms of updates. Although the knowledge of an underlying metric graph might seem to provide additional structure, a problem can even get substantially harder in the metric setting:

Example 5.4.3 The *metric EDGE-UNCERTAINTY-MST problem with constant probability* is not worst-case update competitive:



This complete undirected graph contains two complete subgraphs each having $\frac{n}{2}$ nodes and blue edges. Each of these subgraphs has exactly one *black* node while the other nodes are *white*. The blue edges have a fixed weight of 1, all the other edges have an uncertainty interval of $(1, 5)$.
The actual weight w_e for each edge $e \in E$ depends on its end points:

$$w_{\{u,v\}} = \begin{cases} 4 & \text{if } u \text{ and } v \text{ are black,} \\ 3 & \text{if either } u \text{ or } v \text{ is black,} \\ 2 & \text{otherwise.} \end{cases}$$

The weights of the graph are metric and there exist exactly $(\frac{n}{2}-1)\cdot(\frac{n}{2}-1)$ edges with weight 2. Updating an edge has an influence on the bounds of other edges:

- When the edge with weight 4 is updated, all edges connecting a black and a white node now get uncertainty intervals of $[3,5)$ and all edges connecting two white nodes get the interval $[2,4]$.
- When an edge with weight 3 is revealed, only the uncertainty intervals of the adjacent edges changes and they become $[2,4]$.
- When an edge with weight 2 is revealed, all adjacent edges get uncertainty intervals of $(1,3]$, the other edges get $(1,4]$.

The closed upper and lower bounds of the new uncertainty intervals can be interpreted as a result of the triangle inequalities and thus do not contradict our general assumption of open uncertainty intervals.

The optimal update strategy consists of the edge with weight 4 and any edge with weight 2 as this leads to a situation where all other edges implicitly need to have a weight greater than 2. Since furthermore updating any single edge does not lead to an instance where a solution can be verified, we have $OPT = 2$.

For any deterministic algorithm there is an instance in which the first $\frac{n}{2}-1$ updates always reveal weights of 2. As these updates do not change any lower bounds of other edges, the algorithms needs to update at least $\frac{n}{2}$ edges in the worst case.

The bound on the number of updates, however, stays the same in the metric case:

Lemma 5.4.4 *There are instances of the metric* EDGE-UNCERTAINTY-MST *problem with constant probability where $OPT = \Omega(n^2)$ holds for any possible configuration.*

PROOF:

Consider the complete graph G with uncertainty interval $(1,2)$ for every edge. The optimal algorithm needs to update all edges, since the information that can be derived from the triangle inequality does not strengthen any bound. ■

Example 5.2.21 (which was used to show that the approximate EDGE-UNCERTAINTY-MST problem with constant probability is not worst-case update competitive) is not metric and can therefore not be used in this setting. However, the main result for this problem – the expected update competitive – still holds.

Corollary 5.4.5 *The* approximate metric EDGE-UNCERTAINTY-MST problem with constant probability *has an expected update competitive ratio of $\mathcal{O}(n)$ and there exists an deterministic algorithm that only needs $\mathcal{O}(n)$ updates in expectation.*

PROOF:

Algorithm 16 can also be used in this setting and the number of required updates will only decrease. Therefore, the algorithm, together with the preprocessing step to check whether any updates are necessary, still is $\mathcal{O}(n)$ update competitive. ■

Again, the TSP is considered next. The example that we used in the last section to show that it is not worst-case update competitive relies on the existence of nonmetric weights. Therefore, we have to find new examples to show this property in the metric setting:

Example 5.4.6 The *metric EDGE-UNCERTAINTY-TSP with constant probability* is not worst-case update competitive:
We use exactly the same graph with the same weights and uncertainty intervals as in the last Example 5.4.3. For this graph we know that after the edge with weight 4 has been updated, all edges connecting the two blue subgraphs have an actual weight of at least 2. Therefore, it is now sufficient to reveal this edge and two further edges with weight 2. This gives us a Hamiltonian cycle with the shortest possible length of $2 \cdot \left(\frac{n}{2} - 1\right) + 2 \cdot 2$ and we have $OPT = 3$.
Any deterministic algorithm, on the other hand, has to update at least $\frac{n}{2}$ edges in the worst case to find an edge whose weight is not 2.

The requirement that the graph must be metric does not considerably improve the update competitive ratio. Moreover, there are metric instances where even an algorithm knowing the exact weights has to perform order of n^2 updates:

Corollary 5.4.7 *There are instances for the metric* EDGE-UNCERTAINTY-TSP *with constant probability where an optimal update strategy contains $\Omega(n^2)$ updates for all possible configurations.*

Proof:
The instance used in the proof of Theorem 5.2.24 is a complete graph only containing uncertainty intervals of $(1, 2)$, i.e. no additional information can be derived from the triangle inequalities for any update and the argumentation stays the same. ■

Thus, even in the metric setting there does not exist a nontrivial bound on the number of updates required to solve the exact TSP and focusing on the approximate metric EDGE-UNCERTAINTY-MST is reasonable:
In the metric setting we can finally use the previous results on MST problems to formulate an update competitive algorithm for the approximate metric TSP:

Theorem 5.4.8 *For the* EDGE-UNCERTAINTY-TSP *with constant probability we can find a $(2 + \delta)$-approximation with no more than $\mathcal{O}(n)$ updates in expectation.*

Proof:
According to Corollary 5.4.5 we can find a spanning tree T with only $\mathcal{O}(n)$ updates in expectation, whose cost is within a $(1 + \frac{\delta}{2})$ factor of the cost c_{MST}^{opt} of an optimal MST. We now replace each edge of T with two copies of itself. The resulting graph has a

cost of at most $2 \cdot (1 + \frac{\delta}{2}) \cdot c_{MST}^{opt}$. This Eulerian graph can now be transformed into a Hamiltonian cycle H by only decreasing its cost. Since the cost of an optimal TSP tour is not larger than the cost of any Hamiltonian cycle, it follows that this cycle has a cost of $c_H \leq (2 + \delta) \cdot c_{MST}^{opt} \leq (2 + \delta) \cdot c_{TSP}^{opt}$.
Since the graph is metric, the transformation of the MST into the Hamiltonian cycle H can be performed without any additional updates. ■

Theorem 5.4.8 shows that for the metric setting there exists an algorithm that calculates a $(2 + \delta)$-approximation with $\mathcal{O}(n)$ updates in expectation. By adding a preprocessing step that checks whether for the given instance $OPT \geq 1$ holds (e.g. by enumerating all Hamiltonian cycles and then comparing their bounds), this approach gives us an expected update competitive ratio of $\mathcal{O}(n)$. This competitive ratio is best possible, if we assume that not only the edges but also their weights of an optimal solution need to be returned by the algorithm. This would be the case, if the algorithm is used as part of the WCP, where the calculated tours also need to be scheduled afterwards.

5.5 Overview

The following table summarizes the results of this chapter:
The first line of every cell denotes the competitive ratio, e.g., 1-$\Omega(n)$-competitive means that $OPT = 1$, while for any deterministic algorithm there are instances for which $\Omega(n)$ updates are needed.
In the models with probability WC indicates worst-case competitive ratio and E indicates results in expectation.

The last line gives an upper bound on the number of updates that an algorithm needs for any instance, i.e. there are instances where any algorithm not knowing the exact weights is guaranteed to need that many updates.

	model		
problem	uncertainty	probability	metric
exact MST	1-2-competitive #updates $\Omega(n^2)$ Theorem 5.2.18, Lemma 5.2.20	WC 1-2-competitive #updates $\Omega(n^2)$ Theorem 5.3.8, Lemma 5.2.20	WC 2-$\Omega(n)$-competitive #updates $\Omega(n^2)$ Example 5.4.3, Lemma 5.4.4
approx MST	1-$\Omega(n^2)$-competitive #updates $\Omega(n^2)$ Example 5.2.21, Corollary 5.2.22	E 1-$\mathcal{O}(n)$-competitive WC 1-$\Omega(n^2)$-competitive E #updates $\mathcal{O}(n)$ Theorem 5.3.8	E 1-$\mathcal{O}(n)$-competitive E #updates $\mathcal{O}(n)$ Corollary 5.4.5
exact TSP	1-$\Omega(n)$-competitive #updates $\Omega(n^2)$ Example 5.2.23, Theorem 5.2.24	WC 1-$\Omega(n)$-competitive E $\geq$ 1-$\frac{1}{1-p_\delta}$-competitive #updates $\Omega(n^2)$ Lemma 5.3.11, Theorem 5.2.24	WC 3-$\Omega(n)$-competitive #updates $\Omega(n^2)$ Example 5.4.6, Corollary 5.4.7
approx TSP	n-$\Omega(n^2)$-competitive #updates $\Omega(n^2)$ Example 5.2.25, Corollary 5.2.26	WC n-$\Omega(n^2)$-competitive #updates $\Omega(n^2)$ Example 5.2.25, Corollary 5.3.13	$(2+\delta)$-approximation in E #updates $\mathcal{O}(n)$ Theorem 5.4.8

Chapter 6

Subway Challenge

Guinness World Records is probably the world's most famous reference book for curious and impressive world records for human achievement. It lists the following record:

> **Fastest time to travel to all New York City Subway stations**
>
> The fastest time to travel the entire New York City Subway is 22 hr 26 min 02 sec and was achieved by Andi James, Steve Wilson, Peter Smyth, Martin Hazel, Glen Bryant and Adham Fisher (all UK) between 18 and 19 November 2013 (Guinness World Records [46]).

This world record shows:
Visiting all subway stations of a metropolitan area in the shortest time possible is something so extraordinary that it deserves "mankind's" recognition and an impressive world record certificate by Guinness World Records. Although its practical relevance and application might not be obvious at first, it still is an interesting optimization problem. At second glance, however, it becomes obvious that for different variations of this challenge many well-known routing problems can be applied.

The *Subway Challenge* or *Rapid Transit Challenge* is a challenge in which the participants have to traverse an entire subway system in the shortest time possible. However, there are several possible interpretations on how a subway network should be traversed.

In the following chapter we will always represent this problem by a graph. The time needed to go from one part of the network to another is modeled as arc weights. Therefore, only the average traveling or changing time for this particular situation is taken into account and not the actual subway schedule. Fortunately, the results showed that due to the size of such subway systems the average waiting times come very close to these expected times.

A variety of routing problems related to the subway challenge exist: First, there is the whole class of *Arc Routing Problems (ARPs).* In contrast to "regular" routing problems, arc routing problems focus more on the properties of the used arcs and the route as a whole.

Probably, the two most central arc routing problems are (including their variants):

- *Chinese Postman Problem (CPP)*, where one has to find the shortest tour visiting all arcs of a graph,
- *Rural Postman Problem (RPP)*, a generalization of the CPP, where only a subset of arcs needs to be visited.

Variants of ARPs arise naturally in many practical applications such as mail delivery, garbage collection, milk delivery, network maintenance or road de-icing (e.g., Eiselt et al. [27, 28], Eglese [26] or Dror [24]). Unfortunately, most of these applications cannot be modeled as pure CPP or RPP instances as various additional problem dependent characteristics are involved (see e.g., Kim et al. [56]).

Also, closely related to the Subway Challenge is the *Traveling Salesman Problem (TSP)* and in particular the *Generalized Traveling Salesman Problem (GTSP)*, in which the nodes are partitioned into clusters and the problem is to find the shortest tour visiting every cluster exactly once. This problem was introduced by Henry-Labordere [49] and Srivastava et al. [82] and the authors presented a simple dynamic programming formulation for it. The GTSP is a useful model for many practical problems, e.g., network location, or post-box collecting, and can be used to transform certain arc routing problems into node routing problems. For an overview see e.g. Laporte et al. [61].

In this chapter we discuss a model for the Subway Challenge, that takes traveling and changing/waiting times into account. Based on this model we will then formulate two basic problem types: In the first problem the rider is required to cover all lines, i.e. traverse every distinct segment in any direction. For the second – which models the world record challenge – every station complex needs to be visited.
These two variants also correspond to the classes defined by the Amateur New York Subway Riding Committee [74] created by Peter Samson in 1966. These rules were the foundation for the many world record attempts in the following years. Peter Samson also introduced a third class. Its only difference is the definition of a station in contrast to a station complex. From a mathematical point of view, however, this can be modeled as the same problem.

In the first section we formally introduce the two different problems. In the next sections we will present different solution approaches for the special structure induced by the subway challenge. Finally, all of these approaches are tested and evaluated on the subway network of Berlin and the results are given in the last section.

6.1 Basic Model

Since the underlying graph corresponds to a transportation network, we have the following structure:

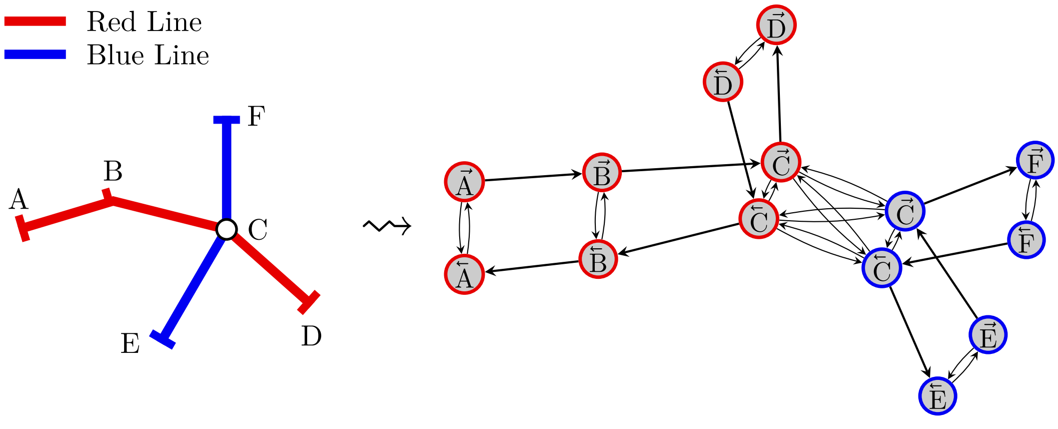
- A set of stations S.
- A set of subway lines $\mathcal{L}$, where each line $L \in \mathcal{L}$ is an ordered set of stations. We assume that each line is operated in both directions and that each line consists of at least two stations.

Now, this problem can be modeled as a directed graph $G = (V, A)$ with the following properties: For every line $L \in \mathcal{L}$ and all stations $s \in L$ we always have a forward node $\vec{v}_{s,L} \in V$ and a backward node $\overleftarrow{v}_{s,L} \in V$. We further define $V_s := \bigcup_{L \in \mathcal{L}: s \in L} \{\vec{v}_{s,L}, \overleftarrow{v}_{s,L}\}$ to be the set of all nodes belonging to the station $s \in S$.

Next, we add the arcs corresponding to the segments used by a subway line between two stations. For all $s \in L$ we have the arc $(\vec{v}_{s,L}, \vec{v}_{s',L})$, if s has the successor s' in L. Analogously, we also add the arc $(\overleftarrow{v}_{s,L}, \overleftarrow{v}_{s'',L})$ for s'' being the predecessor of s. This set of arcs that only connects nodes of the same line and direction is denoted by R and its elements are called *segment arcs.*

Furthermore, we add the arc (u, v) for every possible pair of nodes in each V_s. These arcs correspond to line or direction changes and they are called *changing arcs.*

Every arc $a \in A$ has a weight $t_a > 0$. If a is contained in R, this represents the travel time and if $a \notin R$ holds, the value t_a corresponds to the time needed for changing in this particular station. The resulting graph is called *subway graph* of $\mathcal{L}$ and S.



(a) Subway map with two lines **(b)** Corresponding subway graph

Figure 6.1: Example for the construction of a subway graph

An example of such a subway graph for a very small instance can be found in Figure 6.1, where we have two lines and one changing station.

Using this setting, we will now consider the following two problems:

Definition 6.1.1 (Segment problem)
Let $G = (V, A)$ be a subway graph with positive arc weights $t_a \in \mathbb{Q}$. The *segment problem* is to find the cheapest (not necessarily elementary) tour T containing each arc $a \in R$ at least once.

Definition 6.1.2 (Station problem)
Let $G = (V, A)$ be a subway graph with positive arc weights $t_a \in \mathbb{Q}$. The *station problem* is to find the cheapest tour T so that for every station $s \in S$ at least one node in V_s is contained in T.

In the next two sections we will take a closer look at those two problems and present efficient solution approaches for all of them.

6.2 Segment Problem

The *segment problem* corresponds to the following situation: Given a subway transportation network, every segment between two stations, i.e. every line and every direction, needs to be visited at least once. Changing between two different lines and between different directions of the same line takes a certain amount of changing time. The goal is to minimize the total time needed for such a tour.

This problem is a special case of the *Directed Rural Postman Problem (DRPP)*. The DRPP is $\mathcal{NP}$-hard [64] and so is the segment problem.

Theorem 6.2.1 *The segment problem is $\mathcal{NP}$-hard.*

PROOF:
The decision problem corresponding to the asymmetric TSP is $\mathcal{NP}$-complete, e.g. Schrijver [75, Theorem 58.1]. We will now reduce the ATSP to our segment problem.
Let the directed graph $G = (V, A)$ and cost function $c : A \to \mathbb{Q}_{\geq 0}$ be the input for the ATSP, we now construct the subway graph G' that has $|V| + 1$ stations and $|V|$ lines, one for each $v \in V$. Each line consists of exactly two stations, one that is exclusive for this line and one station h that is shared among all lines. For every arc $(u, v) \in A$ we assign the arc in G' that connects the line corresponding to u with the line corresponding to v a weight of $c_{(u,v)}$. All other arcs in G' that connect nodes of h get a weight higher than the limit L of the ATSP instance. The remaining arcs, i.e. the changing arcs in

the stations $s \in S \setminus \{h\}$, each get a weight of 1. Those arcs will always be included in any feasible solution of the segment problem. This transformation can be performed in polynomial time and we know that, if the segment problem corresponding to G' has a solution with value less than $3 \cdot |V| + L$, this gives us a solution to the initial ATSP instance with value less than L. ■

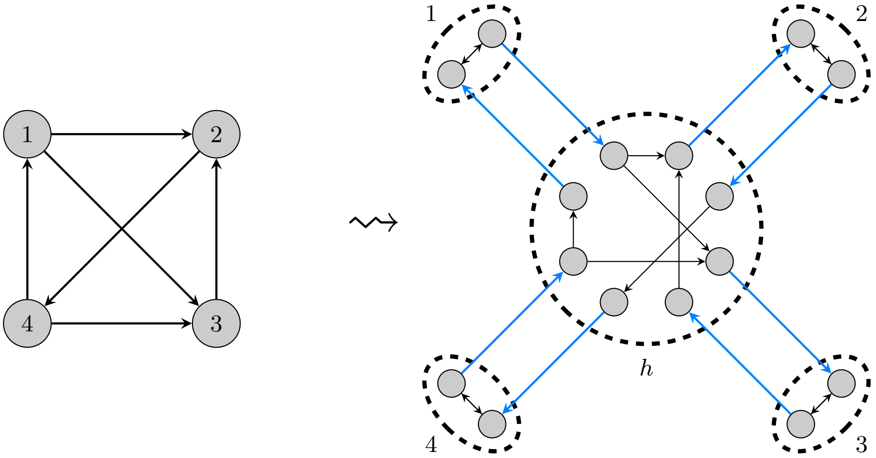


Figure 6.2: Example for the reduction of the ATSP to the segment problem. The segment arcs are given in blue and arcs with large weight are not shown.

We will now present an integer programming formulation that uses dynamic constraint generation and subtour elimination to efficiently solve the segment problem.

6.2.1 ILP Formulation

The segment problem is closely related to the asymmetric TSP. But there are two major differences: There is a certain subset of arcs that needs to be visited and it is in general allowed that some nodes (and arcs) are visited more than once. To accommodate to these differences we will use a very flexible IP method. Our approach is roughly based on a method proposed by Fischetti et al. [35], where the authors present an IP formulation which is used to solve real-world ATSP problems. An overview of different inter programming formulations for the TSP can also be found in Orman and Williams [67].

The following IP-formulation corresponds to such a standard model for the ATSP:

$$\begin{aligned}
\min \sum_{a\in A} c_a x_a & & & \text{(ATSP-IP)}\\
\text{s.t.} \sum_{a\in\delta^+(v)} x_a &= 1 & \forall v \in V & \quad (6.1a)\\
\sum_{a\in\delta^-(v)} x_a &= 1 & \forall v \in V & \quad (6.1b)\\
\sum_{\substack{(u,v)\in A\\ u\in C, v\in C}} x_{(u,v)} &\le |C| - 1 & \forall C \subset V, C \neq \varnothing & \quad (6.1c)\\
x_a &\in \mathbb{N}_{\ge 0} & \forall a \in A &
\end{aligned}$$

In ATSP-IP we have $2^{|V|} - 2$ subtour elimination constraints (6.1c). Since this number gets huge even for a small set of nodes, we have to generate and add those constraints dynamically. This step is called *separation*: Let x^* be an optimal solution to the linear relaxation of ATSP-IP without the constraints (6.1c). Now we have to check whether x^* violates any of the constraints. If this is the case, this particular constraint is added to the problem and the optimization process is started again. Once no violated constraint can be found the solution is feasible.
The separation problem induced by (6.1c) can be solved as a minimal cut problem:

Lemma 6.2.2 *Let* $x \in \mathbb{Q}^A_{\ge 0}$ *be a solution that satisfies the constraints* (6.1a) *and* (6.1b), *then* $\sum_{\substack{(u,v)\in A\\ u\in C,v\in C}} x_{(u,v)} \le |C| - 1$ *holds for a valid subset* C, *if and only if it satisfies* $\sum_{a\in\delta^+(C)} x_a \ge 1$.

PROOF:
We have

$$|C| = \sum_{\substack{(u,v)\in A\\ u\in C}} x_a = \sum_{\substack{(u,v)\in A\\ u\in C,v\in C}} x_{(u,v)} + \sum_{a\in\delta^+(C)} x_a \,.$$

By applying this for (6.1c), we get

$$\sum_{\substack{(u,v)\in A\\ u\in C,v\in C}} x_{(u,v)} \le \sum_{\substack{(u,v)\in A\\ u\in C,v\in C}} x_{(u,v)} + \sum_{a\in\delta^+(C)} x_a - 1$$

and thus

$$\sum_{a\in\delta^+(C)} x_a \ge 1 \,. \tag{6.2}$$

Since all the steps are equivalent transformations, the other direction can be shown analogously. ■

Fortunately, it is possible to separate these inequalities in polynomial time using multiple max-flow computations (see e.g. Applegate et al. [5], Hong [51]).

We will now use ATSP-IP as a basis to formulate a model for our segment problem:

$$\begin{aligned}
\min \; & \sum_{a \in A} t_a x_a && && \text{(Segment-IP)} \\
\text{s.t.} \; & \sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a = 0 && \forall v \in V && (6.3a) \\
& \sum_{a \in \delta^+(C)} x_a + \sum_{a \in \delta^-(C)} x_a \geq 2 && \forall C \subset V, C \neq \varnothing && (6.3b) \\
& x_a \geq 1 && \forall a \in R && (6.3c) \\
& x_a \in \mathbb{N}_{\geq 0} && \forall a \in A &&
\end{aligned}$$

We first have to add the very simple constraints (6.3c) to ensure that all segment arcs $a \in R$ are visited. As for the segment problem some nodes can be visited more than once, we need constraints (6.3a) to enforce that the indegree of each node equals the outdegree. Due to the degree constraints we further have that $\sum_{a \in \delta^+(C)} x_a = \sum_{a \in \delta^-(C)} x_a$ and thus the constraints (6.3b) are equivalent to (6.2).

After Segment-IP has been solved we still need to convert the resulting variable assignment into a tour. This can be done using the following graph:

Definition 6.2.3 (Solution graph)
Let $\hat{x} \in \mathbb{N}^A_{\geq 0}$ be a feasible integral solution of Segment-IP corresponding to the subway graph $G = (V, A)$. We construct the directed multigraph $G_{\hat{x}} = (V', A')$, where $V' = V$ and for each $a \in A$ with $\hat{x}_a \geq 1$ the set A' contains $\hat{x}_a$-many copies of that arc. The graph $G_{\hat{x}}$ is also called *solution graph* of G.

Such a solution graph has the following properties:

Lemma 6.2.4 *For any feasible integral solution $\hat{x}$ of Segment-IP, the resulting solution graph $G_{\hat{x}}$ is*
(a) Eulerian,
(b) connected,
(c) and contains every required arc.

Proof:

(a) A directed graph is Eulerian, if every vertex has equal indegree and outdegree [58, Chapter 2.4]. As the constraints (6.3a) enforce exactly this property, the graph $G_{\hat{x}}$ will definitely be Eulerian.

(b) Since in any subway graph every node v has exactly one outgoing or incoming segment arc, every node of the solution graph $G_{\hat{x}}$ is the head or tail of at least one arc. Analogously to ATSP-IP, the constraints (6.3b) eliminate all subtours. Hence, the graph $G_{\hat{x}}$ is connected.

(c) follows directly from the constraints (6.3c). ■

Therefore, every Euler tour of $G_{\hat{x}}$ corresponds to a feasible solution to the segment problem. All Euler tours in $G_{\hat{x}}$ have the same cost, which is equal to the value of the objective function in Segment-IP. These observations can now be combined to show the correctness of the given IP model.

Theorem 6.2.5 *The model described as Segment-IP is a correct formulation for the segment problem on directed subway graphs.*

Proof:

It only remains to show, that any feasible tour T of the segment problem also corresponds to a feasible integer solution to Segment-IP: Let $x \in \mathbb{N}^A_{\geq 0}$ be the variable assignment where x_a is set to be the number of times arc a has been visited in T or zero if a is not in T. The assignment x is integral and fulfills all the constraints in Segment-IP. ■

Remark: Since the station problem is a special case of the directed rural postman problem, any algorithm for the DRPP can also be used to solve the station problem. E.g. Christofides et al. [15] described a branch-and-bound algorithm using Lagrangean Relaxation for the DRPP.

6.2.2 Implementation Details

To efficiently solve the separation problem we first transform the directed subway graph G into an undirected graph G^{sep}. Let x^* be the current fractional solution to Segment-IP, we set the capacities in G^{sep} to $c_{\{u,v\}} = x^*_{(u,v)} + x^*_{(v,u)}$. Using this, we have that the value of a cut C in G^{sep} is exactly $\sum_{a\in\delta^+(C)} x^*_a + \sum_{a\in\delta^-(C)} x^*_a$ with respect to the original graph G. Therefore, the most violated constraint of type (6.3b) can be identified by finding a *global minimum cut* C in G^{sep}, i.e. a subset of nodes $C \subset V, C \neq \varnothing$ such that $\sum_{e\in\delta(C)} c_e$ is minimized.

There are several ways to find a global minimum cut: The easiest is to compute a minimum s-t cut for all choices of $s, t \in V$ using the max-flow min-cut theorem. Thus, the separation can be performed in polynomial time.

Remark: Since the constraints of the LP relaxation of Segment-IP can be separated in polynomial time, the entire LP relaxation can be solved in polynomial time. This is due to the fact that optimization is equivalent to separation, a fundamental result by Grötschel, Lovász, and Schrijver [44]. However, as for the segment problem we solve an integer program, this does not contradict Theorem 6.2.1.

For our implementation we use the algorithm of Stoer-Wagner [83] for the global min-cut computation, which performs in $\mathcal{O}(n^3)$.

After an optimal solution $\hat{x}$ has been found, we construct $G_{\hat{x}}$. Any Euler tour in $G_{\hat{x}}$ corresponds to an optimal solution of the segment problem. To find such an Euler tour we use Algorithm 17, which was first described by Hierholzer [50] in 1873. The algorithm runs in linear time.

Algorithm 17: Finding an Euler tour

```
Input: a directed Eulerian graph $G = (V, A)$
Output: an Euler tour in $G$

Algorithm EulerCircuit()
    $E \leftarrow \emptyset$;
    $u \leftarrow$ any node in the graph $G$;
    FindCircuit($u, E$);
    return $E$;

Procedure FindCircuit($u, E$)
    foreach outgoing arc $(u, v)$ do
        remove $(u, v)$ from $G$;
        FindCircuit($v, E$);
    $E \leftarrow E \cup \{u\}$;
```

6.3 Station Problem

The *station problem* corresponds to the following situation: Given a subway transportation network every station complex needs to be visited at least once. However, in this problem it does not matter which line and direction is used when the station is visited. Changing between two different lines and between different directions of the same line takes a certain amount of changing time. The goal is to minimize the total time needed for such a tour.

This problem is strongly related to the asymmetric generalized TSP (see e.g. Noon and Bean [66]) and is also $\mathcal{NP}$-hard.

Theorem 6.3.1 *The station problem is $\mathcal{NP}$-hard.*

PROOF:
We reduce the $\mathcal{NP}$-complete metric TSP (Schrijver [75]) to the station problem:
Let the undirected complete graph $G = (V, E)$ and cost function $c : E \to \mathbb{Q}_{\geq 0}$ be the input for the TSP, we now construct the subway graph G', that has $|E|$ lines and $|V|$ stations, one for each $v \in V$. For every edge $\{u, v\} \in E$ we have a line consisting of exactly two stations, one corresponding to u and the other one corresponding to v. For this line we set the travel time to $c_{\{u,v\}}$ in both directions. All changing arcs in G' get a changing time of 1. We note that, since the length function satisfies the triangle inequality, any optimal solution of G' will be an elementary tour, containing no node more than once and changing exactly once in every station.
This transformation can be performed in polynomial time and if the station problem corresponding to G' has a solution with value less than $|V| + L$, it gives us a solution to the initial TSP instance with value less than L. ∎

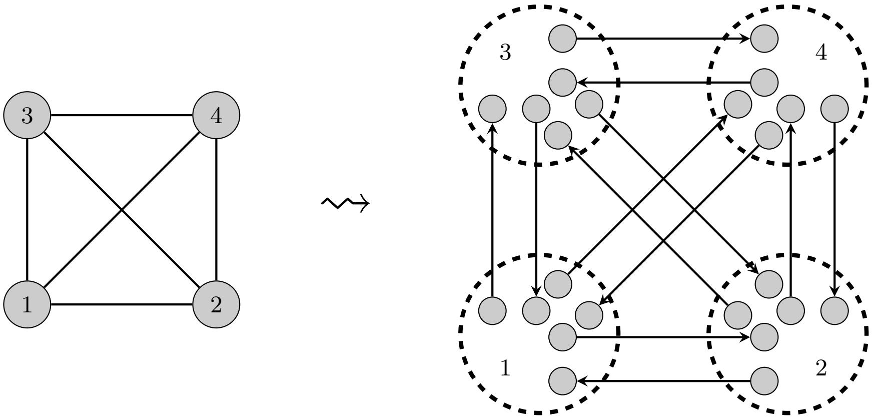


Figure 6.3: Example for the reduction of the metric TSP to the station problem. Changing arcs are not shown.

We will now introduce three formulations for the station problem: The first formulation transforms the problem into a regular symmetric TSP instance, which can then be solved using a generic solver like Concorde [4]. The other two are integer programming formulations, one with exponentially many subtour elimination constraints, which need to be separated, and the last one is a flow-based formulation with polynomially many constraints.

6.3.1 TSP transformation

A station problem is very similar to the generalized traveling salesman problem where the stations correspond to clusters. The crucial difference between those two problems is that in the GTSP every cluster needs to be visited exactly once. Due to the special structure of a subway graph this is not the case for stations in the station problem. Changing in a station requires us to visit at least two nodes in that station and thus two nodes of a cluster are visited.

Nevertheless, Gamrath [38] explained how it is possible to transform the station problem into an equivalent GTSP instance. The main idea is to construct a directed graph G' with the same set of stations and nodes and then add an arc (u, v) for every pair of nodes $u, v \in V$ with a weight that corresponds to the length of the shortest u-v path in the original subway graph G. After that, the arcs that stay in one cluster are removed from G'. Line changes in G' are now implicitly given in the newly introduced arcs and every cluster needs to be visited exactly once. Next, we can apply the transformation described in Noon and Bean [66] to solve the GTSP instance as an ATSP. By applying this transformation the number of nodes stays the same, but the cluster arcs will get a very large weight, weakening the bounds derived from the LP-relaxation.

The resulting ATSP can either be solved directly or transformed into a symmetric TSP first and then solved by Concorde. Computational results of this approach can also be found in [38].

6.3.2 Subtour ILP Formulation

In this section we present an integer programming formulation that operates directly on the subway graph and does not require any transformation.

The station problem corresponds to a variation of the GTSP, where every cluster can be visited more than once and the underlying graph corresponds to a subway graph. Fischetti et al. [34] studied the symmetric generalized traveling salesman problem, where each cluster needs to be visited at least once, and described a branch-and-cut algorithm that can solve instances involving up to 400 nodes. Our approach is similar but focuses on the special structure of the subway graph.

We will give the formulation first and then discuss under which assumptions this model is correct:

$$
\begin{aligned}
\min \; & \sum_{a\in A} t_a x_a && && \text{(Station-IP)}\\
\text{s.t.} \; & \sum_{a\in\delta^+(v)} x_a - \sum_{a\in\delta^-(v)} x_a = 0 && \forall v \in V && (6.4a)\\
& \sum_{a\in\delta^+(V_s)} x_a \geq 1 && \forall s \in S && (6.4b)\\
& \sum_{a\in\delta^+(C)} x_a + \sum_{a\in\delta^-(C)} x_a \geq 2 && \forall C \subset V, C \text{ and } \overline{C} \text{ contain a station} && (6.4c)\\
& x_a \in \mathbb{N}_{\geq 0} && \forall a \in A &&
\end{aligned}
$$

The formulation is somehow similar to the Segment-IP: The first sets of constraints assure that the solution only consists of a collection of subtours and due to constraints (6.4b) every station is visited at least once by some subtour. The constraints (6.4c) are also valid, since they make sure that every station is connected to all the other stations.

Let $\hat{x} \in \mathbb{N}^A_{\geq 0}$ be a feasible integral solution for Station-IP. We say a set of tours corresponds to the solution $\hat{x}$, if it contains exactly one Euler tour for each of the connected components of $G_{\hat{x}}$ introduced in Definition 6.2.3. Such tours always exist, since the outdegree of all nodes equals the indegree.

Remark: There are several different possibilities for an Euler tour of one component, but they always have the same costs and use the same arcs. Therefore, they are all in some sense equivalent.

Unfortunately, the constraints (6.4c) in Station-IP are in general not sufficient. Figure 6.4 shows such an example: The optimal solution of Station-IP uses all three lines in both directions but never changes to different lines. Therefore, we have to make the additional assumption, that there is a line L which has a terminal station that is not visited by any other line.

Remark: For the general case the approach described by Fischetti et al. [34] can be used. Here, the authors show that for the symmetric case the following inequalities are sufficient to eliminate all the subtours:

$$\sum_{e\in\delta(C)} x_e \geq 2\cdot(y_i + y_j - 1) \quad \forall C \subset V, C \neq \varnothing, \forall i \in C, j \in \overline{C}\,, \tag{6.5}$$

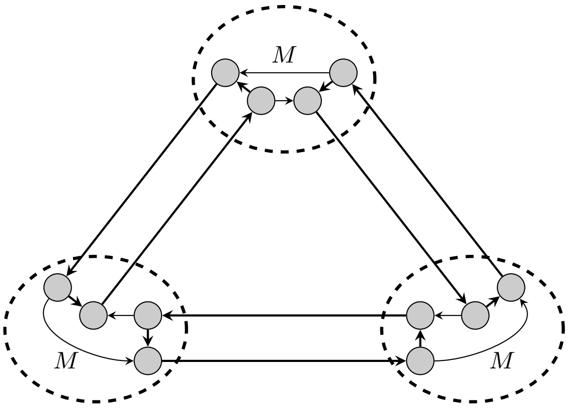


Figure 6.4: Example of the station problem without exclusive stations

where y_v is the binary variable indicating whether node $v \in V$ is visited. It is easy to see that the constraints (6.4c) of Station-IP are contained in (6.5): Every station needs to be visited and the set C as well as $\overline{C}$ in (6.4c) always contain at least on complete station. Thus, there will always be an $i \in C$ and a $j \in \overline{C}$ such that both y_i and y_j are one and the inequality corresponds to the constraint in (6.4c).

Theorem 6.3.2 *The model described as Station-IP is a correct formulation for an instance of the station problem, if there is at least one terminal station $t \in S$ that is not a changing station, i.e. t is only contained in one line.*

To prove this we first show that the existence of such a terminal station t guarantees that in any solution there always exists a station that is only visited by a single tour. This observation is then sufficient to show the theorem.

Lemma 6.3.3 *If there is at least one terminal station $t \in S$ that is not a changing station, then for any feasible solution $\hat{x}$ of Station-IP there exist tours corresponding to $\hat{x}$ such that the station t is only visited by a single tour.*

PROOF:
Let in the following L denote the only line that contains t. The station t needs to be visited, i.e. $\sum_{a \in \delta^+(V_t)} \hat{x}_a \geq 1$ holds. Since V_t only consists of of the nodes $\overleftarrow{v}_{t,L}$ and $\vec{v}_{t,L}$, the arc $(\overleftarrow{v}_{t,L}, \vec{v}_{t,L})$ connecting these nodes must have an $\hat{x}$-value of at least one. Hence, all the nodes of V_t are in the same component of $G_{\hat{x}}$ and belong to the same tour. ∎

Remark: It also follows from the proof of Lemma 6.3.3 that the constraints (6.4c) are not required for the observation to be true. Thus, it also holds for an optimal solution to Station-IP containing only an arbitrary subset of the constraints (6.4c).

This fact can now be used to proof the theorem. But first, we state the following technical lemma, which is solely used in the proof of Theorem 6.3.2:

Lemma 6.3.4 *If for any feasible solution $\hat{x}$ of Station-IP the solution graph $G_{\hat{x}}$ has one station $s \in S$ that lies in only one of its components, then $\hat{x}$ corresponds to a single tour visiting all stations.*

PROOF:
Let us assume this is not the case and we always have multiple subtours. Let us denote the tour that visits s by T. The tour T does not visit all stations in G. If this was the case, then the graph $G_{\hat{x}}$ would consist of exactly one component and it would thus be possible to represent the solution $\hat{x}$ by one single tour. Therefore, we also have a station s' that is not visited by T.
By taking the graph G with capacities of $\hat{x}_a$ for all $a \in A$ and by further introducing a super source node for the cluster s and a super sink for s', let now C denote the minimum s-s' cut between those two super nodes. Since s and s' are only visited by different subtours, the capacity of C will be zero and therefore violate one of the constraints (6.4c). This contradicts the assumption that $\hat{x}$ is a feasible solution for Segment-IP. ■

Using this, we can finally show the theorem:

Proof of Theorem 6.3.2: We have to show the three following facts: 1. Let T be a tour that visits all stations, then there exists a feasible solution to Station-IP that represents the tour T. 2. Any optimal solution of Station-IP corresponds to a feasible tour for the station problem. 3. The objective value of Station-IP corresponds to the time needed for that particular tour.

1. Finding the variable assignment can be trivially achieved by setting x_a to the number of times arc $a \in A$ appears in T or setting it to zero if $a \notin T$. Since T is a single tour visiting all stations we have that every station is connected to any other station via at least two arcs and, thus, the constraints (6.4c) are always fulfilled. Also all the other constraints are valid for this solution.
2. According to Lemma 6.3.3, there exists a station $s \in S$ that is only visited by single tour. Now, Lemma 6.3.4 can be applied to conclude that such a solutions corresponds to one tour visiting all stations.
3. Since in the objective function we sum up the times (traveling or changing) needed for every edge that is used, this corresponds to the time needed for a tour. ■

Although Station-IP is not valid for general subway graphs (e.g. Figure 6.4), the restriction is very reasonable for real-world subway networks. For example, the subway networks of Tokyo, New York City, London, and Berlin all have such a terminal station.

The most realistic issue that might occur, are so-called “express lines”, i.e. lines that only visit a subset of stations of other lines. If every regular line also has an express version, this formulation could not be applied, unless some of the express lines are removed from subway graph.

Implementation Details

The number of constraints for the Station-IP is exponential. The constraints (6.4c) should therefore be separated. Following the proof of Lemma 6.3.4 this can be achieved in the following way:

- Given an integer solution $\hat{x}$, build the corresponding graph $G_{\hat{x}}$ and compute its connected components.
- Identify a station s that only lies in one component of $G_{\hat{x}}$.
- If there exists another station s' that has none of its nodes in that component, find the minimum cut C that contains s entirely but no node of s' using a super source and super sink and a max-flow algorithm. This cut will have a capacity of less than two.
- Create the constraint corresponding to C and reoptimize.

The first step can be performed in $\mathcal{O}(|V|+|A|)$, after that the cut can be computed using generic push-relabel max-flow computations in $\mathcal{O}(|V|^3)$. However, for the implementation of this formulation we use an approach that comes closer to the separation of the segment problem so that more of the code base can be reused.

Our separation approach again relies on minimum global cut computations:
For an integer solution $\hat{x}$, we first construct the undirected graph G' with capacities equal to the sum of the $\hat{x}$ value of the corresponding arcs. Then, in order to avoid generating cuts containing only unvisited nodes, these unvisited nodes are connected to one visited node of their station. A global min-cut computation on G' results to a set of nodes C that violates a constraint in (6.4c). This approach is also formally described in Algorithm 18.

Lemma 6.3.5 *Let $\hat{x} \in \mathbb{Q}^A_{\geq 0}$ be a solution that satisfies the constraints* (6.4a) *and* (6.4b), *then $\hat{x}$ violates* (6.4c) *for a subset C, if C is returned by Algorithm 18 and it satisfies all the constraints* (6.4c)*, if Algorithm 18 returns the empty set.*

PROOF:
We consider the case where Algorithm 18 returns a cut C first: Since unused nodes of any station have been connected (with capacity two) to one visited node of that station, we know that the set C as well as $\overline{C}$ each contain at least one complete station. As this cut C also has a capacity of less then two, the constraint in (6.4c) corresponding to that subset C is violated.

Algorithm 18: Separation of the station problem using global min-cut

```
Input: a subway graph $G = (V, A)$ and a solution $\hat{x}$
Output: a violated cut $C$
construct the directed graph $G' = (V, E)$ of $G$;
foreach $\{u, v\} \in E$ do
    $c_{\{u,v\}} \leftarrow \hat{x}_{(u,v)} + \hat{x}_{(v,u)}$;
foreach station $s \in S$ do
    find a node $v_s \in V_s$ that has an arc $(v_s, u)$ in $G$ with $u \notin V_s$ and $\hat{x}_{(v_s,u)} > 0$;
foreach node $v \in V$ do
    if $v$ has no outgoing arc in $G$ with $\hat{x}_a > 0$ then
        $c_{\{v,v_s\}} \leftarrow 2$, where $s$ is the station of $v$;
$C \leftarrow$ global minimum cut in $G'$ with capacities $c$;
if neither $C$ nor $\overline{C}$ is empty and $c(C) < 2$ then
    return $C$;
else
    return $\varnothing$;
```

If Algorithm 18 returns the empty set, this means that all stations are connected and thus none of the constraints (6.4c) can be violated. ■

Therefore, the separation approach described in Algorithm 18 is correct and can be used to dynamically generate subtour constraints until an optimal solution of Station-IP has been found.

6.3.3 Flow-based ILP Formulation

In this section we present a polynomial size formulation for the station problem. The advantage of such a formulation is that it can be solved using state-of-the-art IP solvers without any problem specific implementations. Here, the subtour constraints are replaced by a flow based concept. Similar approaches have already been applied for a variety of other combinatorial problems, e.g., for the Steiner Tree Problem [68], the TSP [16] or for a distance constrained Vehicle Routing Problem [72]. In Reuther [72] the author also uses a flow base approach to solve many of the instances of the TSPLIB [71], where this approach leads to very good LP-bounds.

As a prerequisite we have to identify a node d that needs to be visited in any feasible solution. For a clearer notation, we now define the set $N := V \setminus \{d\}$ and the set $S' := \{s \in S : d \notin V_s\}$. If the subway graph contains a terminal station t that is not a changing station, we can choose any node of V_t for d. If no such node exists, one has to formulate and solve the IP for any node of a fixed station.

The idea of the flow-based model is the following:
For every node $v \in N$ we find a flow $f^{v,d}$ from v to d. Since every station $s \in S$ needs to be visited at least once, the combined outflow out of V_s needs to be at least 1 and as flows the $f^{v,d}$ have to satisfy the flow conservation. A feasible tour must now visit at least all the arcs that carry flow. This can be formulated in the following mixed integer program:

$$\begin{aligned}
\min \; & \sum_{a \in A} t_a x_a && && \text{(FIP)}\\
\text{s.t.} \; & \sum_{a \in \delta^+(u)} f_a^{v,d} - \sum_{a \in \delta^-(u)} f_a^{v,d} = 0 && \forall v \in N, u \in N \setminus \{v\} && \text{(6.6a)}\\
& \sum_{v \in V_s} \sum_{a \in \delta^+(V_s)} f_a^{v,d} - \sum_{v \in V_s} \sum_{a \in \delta^-(V_s)} f_a^{v,d} \geq 1 && \forall s \in S' && \text{(6.6b)}\\
& x_a \geq f_a^{v,d} && \forall v \in N, a \in A && \text{(6.6c)}\\
& \sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a = 0 && \forall v \in V &&\\
& \sum_{a \in \delta^+(V_s)} x_a \geq 1 && \forall s \in S &&\\
& x_a \in \mathbb{N}_{\geq 0} && \forall a \in A &&\\
& f_a^{v,d} \in \mathbb{Q}_{\geq 0} && \forall a \in A, v \in N &&
\end{aligned}$$

FIP has $\mathcal{O}(|V|^3)$ variables and $\mathcal{O}(|V|^3)$ constraints.
The first constraints (6.6a) and (6.6b) ensure that the outflows are indeed flows and behave as stated before. The constraints (6.6c) link the flow with the actual arc variables and the remaining constraints are then exactly the same as in Station-IP.

This model implicitly contains all subtour constraints used in Station-IP:

Lemma 6.3.6 *The LP-Relaxation of FIP implies the subtour elimination constraints*

$$\sum_{a \in \delta^+(C)} x_a \geq 1$$

for every cut $C \subset V$, where C and $\overline{C}$ each contain at least one complete station.

Proof:
Assume that (x, f) with $x \in \mathbb{N}^A_{\geq 0}$ and $f \in \mathbb{Q}^{A\times N}_{\geq 0}$ is a feasible solution of the LP-relaxation of FIP, but

$$\sum_{a\in\delta^+(C)} x_a < 1$$

holds for a feasible cut $C \subset V$, where C as well as the complement $\overline{C}$ contain at least one complete station.
Without loss of generality, let $d \in \overline{C}$. As C contains at least one complete station s, we have due to (6.6b) that f contains flow from nodes in V_s to $d \notin C$ with combined value of at least one. As the variables x_a are at least as big as the individual flow values on a, they also have to sum up to at least one along the cut. This contradicts the assumption.■

Theorem 6.3.7 *The model described as FIP is a correct formulation for the station problem on directed subway graphs.*

Proof:
To show: 1. Let T be a tour that visits all stations, then there exists a feasible solution to FIP that represents T. 2. Any optimal solution of FIP corresponds to a feasible tour for the station problem. 3. The objective value of FIP corresponds to the time needed for that particular tour.

1. Finding the variable assignment can be trivially achieved by setting x_a to the number of times arc $a \in A$ appears in T or setting it to zero if $a \notin T$. For every node $v \in N$ we set the corresponding $f^{v,d}$ so that it matches the tour T with a flow of one. Since T visits every station, (6.6b) is also satisfied.
2. According to Lemma 6.3.6, FIP implicitly contains all the constraints (6.4c). All the other constraints of Station-IP are explicitly included and since Station-IP is a correct formulation, so is FIP.
3. The objective function is exactly the same as in Station-IP. ■

Although FIP is a correct formulation and can thus be used to solve the station problem, its LP relaxation does not lead to nearly as good bounds as the flow based approach for the TSP described by Reuther [72]. For the station problem the constraints for the flow always involve sums over many nodes and arcs, while for the TSP a flow to and from a node nicely corresponds to the resulting tour. Thus, this approach takes considerably longer (many hours for the subway network used in our experiments in Section 6.5) to solve than Station-IP with dynamic subtour elimination.

6.4 Preprocessing

The subway graph constructed in Section 6.1 contains many arcs and nodes that are never used or redundant, therefore a preprocessing step to remove those arcs is favorable. We will first consider the arcs that represent changes from one direction of the line L to the other direction:

Lemma 6.4.1 *In any optimal solution to the segment problem or the station problem a change of direction will never occur*

- *between the first station of a line and the first changing station*
- *or between the last changing station and the terminal.*

PROOF:
In both problems the last station in every direction needs to be visited. The only way to get there from a different line is via the closest changing station. Any change of direction between those two stations, therefore, means that the solution contains a cycle on which every node is visited more than once. Removing this cycle would thus improve the solution. ■

A changing arc a will only be used, if $\mathrm{tail}(a)$ has a incoming segment arc and $\mathrm{head}(a)$ has an outgoing segment arc. Thus, all changing arcs not fulfilling this criterion can be removed as well.

6.5 Computational Results

The computational experiments have been performed for the subway network of Berlin[1] – the largest subway system in Germany. It has a total of 170 stations, of which 19 are changing stations, and it consists of 9 lines. As the line U55 is currently not connected to any other subway lines, its three stations have been omitted.

After preprocessing this results in an graph with 286 nodes and 666 arcs. The travel times in the graph correspond to the original times of the subway network (as given on `www.bvg.de`). The time required for changing into a new line or for changing the direction of one line has been set identically on all changing arcs and has been varied to test different scenarios. All the instances have been solved using CPLEX 12.04 on an Intel Xeon E5-2630 CPU clocked at 2.6 GHz using only one thread.
The results of the integer programming models with dynamic constraint generation can be found in the tables 6.1 and 6.2. The first line corresponds to a realistic changing time

---

[1] Berliner Verkehrsbetriebe (BVG) `www.bvg.de`

of 5 minutes which comes very close to the actual time needed on average. For the second instance, the changing times are set to a large value, so that the number of changes gets minimized. Similar, in the last line the changing times are set to ε which minimizes the actual time spend riding in trains.

changing time	tour length	riding time	changes	computation
5 min	595 min	460 min	27	203 ms
1000 min	23,494 min	494 min	23	70 ms
0.001 min	440 min	440 min	41	53,885 ms

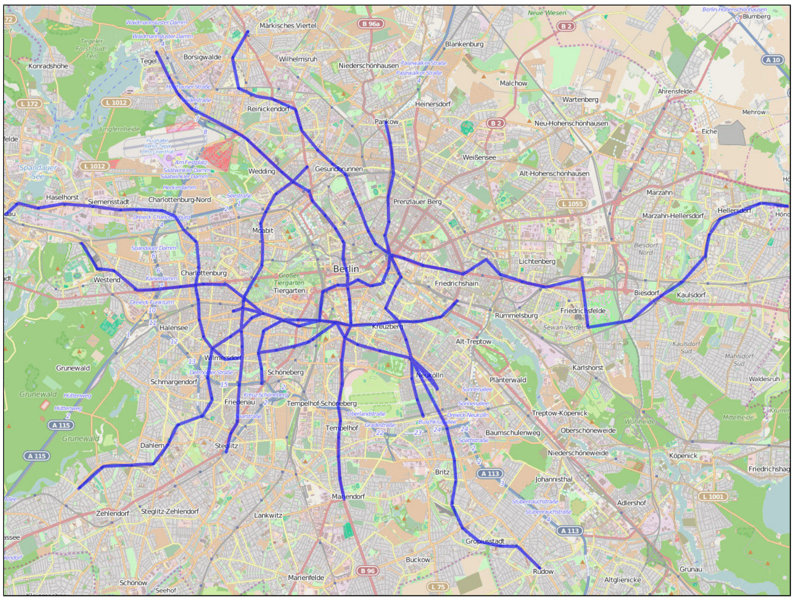
Table 6.1: Results for the station problem of the Berlin subway network

changing time	tour length	riding time	changes	computation
5 min	708 min	578 min	26	2405 ms
1000 min	24,614 min	614 min	24	47 ms
0.001 min	574 min	574 min	28	33 ms

Table 6.2: Results for the segment problem of the Berlin subway network

The optimal tours for the realistic changing times (5 min) are shown on pages 139 and 140. The segments used in each of these tours have also been visualized using map data provided by the OpenStreetMap project. Segments used once are shown in blue, while segments used multiple times are red.

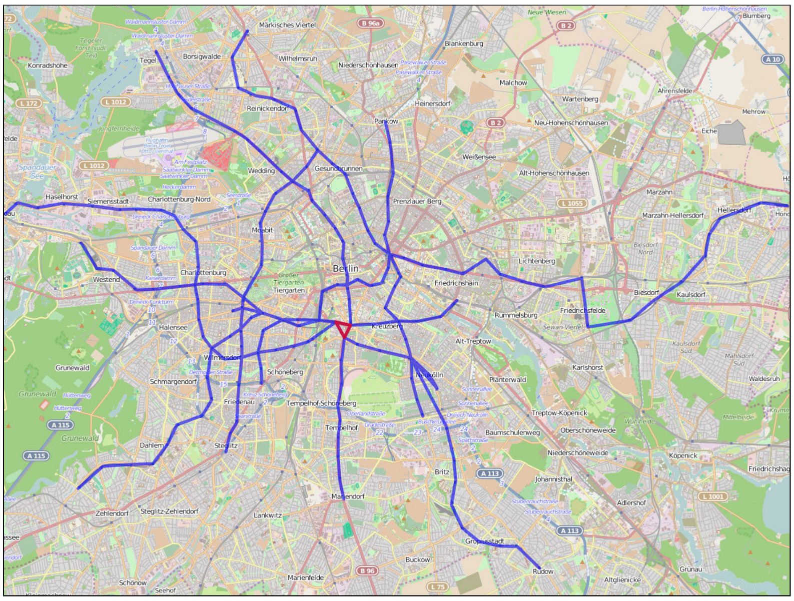
These results show, that with the given approaches it is possible to solve Subway Challenge problems for real-world networks in a matter of seconds. The rapid transit challenge for Berlin, where every station complex needs to "touched" at least once, can be performed in about 595 min. If the goal is the minimization of the changes – either between directions or between different lines – a value of 23 changes can be achieved. If only the time spend on a moving train is considered, 440 min are required.



Map data © OpenStreetMap contributors, CC-BY-SA

Start	Line	Destination	Time (m)
Zoologischer Garten	U9	Nauener Platz	11
Nauener Platz	U9	Leopoldplatz	18
Leopoldplatz	U6	Alt-Tegel	36
Alt-Tegel	U6	Alt-Mariendorf	80
Alt-Mariendorf	U6	Hallesches Tor	97
Hallesches Tor	U1	Warschauer Straße	112
Warschauer Straße	U1	Nollendorfplatz	133
Nollendorfplatz	U4	Innsbrucker Platz	144
Innsbrucker Platz	U4	Bayerischer Platz	151
Bayerischer Platz	U7	Rudow	186
Rudow	U7	Hermannplatz	209
Hermannplatz	U8	Hermannstraße	218
Hermannstraße	U8	Wittenau	259
Wittenau	U8	Alexanderplatz	286
Alexanderplatz	U5	Hönow	324
Hönow	U5	Alexanderplatz	362
Alexanderplatz	U2	Pankow	378
Pankow	U2	Ruhleben	431
Ruhleben	U2	Bismarckstraße	446
Bismarckstraße	U7	Rathaus Spandau	469
Rathaus Spandau	U7	Berliner Straße	499
Berliner Straße	U9	Rathaus Steglitz	510
Rathaus Steglitz	U9	Spichernstraße	524
Spichernstraße	U3	Krumme Lanke	548
Krumme Lanke	U3	Wittenbergplatz	575
Wittenbergplatz	U1	Uhlandstraße	583
Uhlandstraße	U1	Kurfürstendamm	589
Kurfürstendamm	U9	Zoologischer Garten	595

Instance 6.1: Optimal solution for the **station problem**



Map data © OpenStreetMap contributors, CC-BY-SA

Start	Line	Destination	Time (m)
Zoologischer Garten	U9	Osloer Straße	12
Osloer Straße	U8	Alexanderplatz	28
Alexanderplatz	U2	Pankow	44
Pankow	U2	Nollendorfplatz	78
Nollendorfplatz	U4	Innsbrucker Platz	89
Innsbrucker Platz	U4	Nollendorfplatz	100
Nollendorfplatz	U3	Krumme Lanke	129
Krumme Lanke	U3	Nollendorfplatz	158
Nollendorfplatz	U1	Uhlandstraße	168
Uhlandstraße	U1	Hallesches Tor	184
Hallesches Tor	U6	Alt-Mariendorf	201
Alt-Mariendorf	U6	Alt-Tegel	245
Alt-Tegel	U6	Mehringdamm	278
Mehringdamm	U7	Rathaus Spandau	317
Rathaus Spandau	U7	Rudow	379
Rudow	U7	Möckernbrücke	409
Möckernbrücke	U1	Warschauer Straße	425
Warschauer Straße	U1	Nollendorfplatz	446
Nollendorfplatz	U2	Ruhleben	470
Ruhleben	U2	Alexanderplatz	512
Alexanderplatz	U5	Hönow	550
Hönow	U5	Alexanderplatz	588
Alexanderplatz	U8	Hermannstraße	607
Hermannstraße	U8	Wittenau	648
Wittenau	U8	Osloer Straße	664
Osloer Straße	U9	Rathaus Steglitz	692
Rathaus Steglitz	U9	Zoologischer Garten	708

Instance 6.2: Optimal solution for the **segment problem**

Chapter 7

Conclusion

We proposed an exact approach for the welding cell problem that combines two classical fields of mathematical optimization: discrete and continuous optimization. The continuous part – the computation of optimal collision-free trajectories – is computationally much more expensive than most of the aspects of the discrete optimization process and only estimated distance can be found efficiently. This aspect also incorporates *uncertainty* into our problem, as the number of exact trajectory computations needs to minimized while the optimality of our solution approach should still be maintained.

This thesis aims to build a bridge between the practical and theoretical aspects of the welding cell problem with high determination cost. Although uncertainty in general is an aspect that has been extensively discussed in many different facets, this particular framework of uncertainty is still fairly new. Our main theoretical result explores some of the most famous combinatorial optimization problems within this framework to combine them into an algorithm that is guaranteed to find a $(2 + \delta)$-approximation for the TSP that only requires $\mathcal{O}(n)$ edge updates in expectation.

On the practical side it was our goal to provide an approach that can handle WCP instances of practical relevant scales. For this we developed a program that efficiently combines the continuous optimization routines into a branch-and-price framework. Due to the power and flexibility of column generation, a seamless integration of optimal trajectory control is possible so that no time-consuming restarts or regeneration of existing variables are necessary. Aside from this combination, especially the computational performance of the underlying combinatorial pricing problem has been discussed. We proposed a label setting algorithm for this shortest path problem with forced time windows. By introducing new data structures for the storage of discrete time windows and for the efficient handling of queries, this algorithm performs fast enough for an efficient pricing algorithm.

The results in the context of the WCP are completed by solution approaches for the famous Rapid Transit Challenge, where the entire subway network of a metropolitan area needs to be traversed as fast as possible. Here, techniques that were originally developed for a WCP preprocessing strategy can be used for a branch-and-cut approach that computes the optimal tour for the Berlin subway challenge in less than a second.

We hope that this work helps to show the benefits of “interdisciplinary” projects, as most real-word challenges cannot be solved by one scientific field of research alone.

Bibliography

[1] T. Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, July 2009.

[2] J. Andrews and J. A. Sethian. Fast marching methods for the continuous traveling salesman problem. *Proceedings of the National Academy of Sciences*, 104:1118–1123, 2007.

[3] Y. P. Aneja, V. Aggarwal, and K. P. Nair. Shortest chain subject to side constraints. *Networks*, 13:295–302, 1983.

[4] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde TSP solver, 2006. URL `http://www.math.uwaterloo.ca/tsp/concorde/`.

[5] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem – A Computational Study (Princeton Series in Applied Mathematics).* Princeton University Press, 2007.

[6] C. Barnhart, C. A. Hane, and P. H. Vance. Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems. *Operations Research*, 48: 318–326, 2000.

[7] J. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19:379–394, 1989.

[8] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Robust Optimization.* Princeton University Press, 2009.

[9] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1st edition, 1997.

[10] J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming.* Springer, 2011.

[11] R. Bohlin. *Robot Path Planning.* Chalmers University of Technology, 2002.

[12] N. Boland, J. Dethridge, and I. Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34:58–68, 2006.

[13] B. Bollobás. Random graphs. In *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*, pages 215–252. Springer, 1998.

[14] R. Bruce, M. Hoffmann, D. Krizanc, and R. Raman. Efficient update strategies for geometric computing with uncertainty. *Theory of Computing Systems*, 38:411–423, 2005.

[15] N. Christofides, V. Campos, A. Corberán, and E. Mota. An algorithm for the rural postman problem on a directed graph. In G. Gallo and C. Sandi, editors, *Netflow at Pisa*, volume 26 of *Mathematical Programming Studies*, pages 155–166. Springer, 1986.

[16] A. Claus. A new formulation for the travelling salesman problem. *SIAM Journal on Algebraic Discrete Methods*, 5:21–25, 1984.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press Cambridge, third edition, 2009.

[18] W. Dargie and C. Poellabauer. *Fundamentals of Wireless Sensor Networks – Theory and Practice.* John Wiley & Sons, 2010.

[19] M. Desrochers, J. Desrosiers, and M. Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations research*, 40:342–354, 1992.

[20] J. Desrosiers and M. Lübbecke. A primer in column generation. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, pages 1–32. Springer, 2005.

[21] J. Desrosiers, F. Soumis, and M. Desrochers. Routing with time windows by column generation. *Networks*, 14:545–565, 1984.

[22] B. Donald, P. Xavier, J. Canny, and J. Reif. Kinodynamic motion planning. *Journal of the ACM*, 40:1048–1066, 1993.

[23] M. Dror. Note on the complexity of the shortest path models for column generation in VRPTW. *Operations Research*, 42:977–978, 1994.

[24] M. Dror. *Arc Routing – Theory, Solutions, and Applications.* Springer, 2000.

[25] I. Dumitrescu and N. Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, 42:135–153, 2003.

[26] R. W. Eglese. Routeing winter gritting vehicles. *Discrete applied mathematics*, 48: 231–244, 1994.

[27] H. A. Eiselt, M. Gendreau, and G. Laporte. Arc routing problems, part I: The chinese postman problem. *Operations Research*, 43:231–242, 1995.

[28] H. A. Eiselt, M. Gendreau, and G. Laporte. Arc routing problems, part II: The rural postman problem. *Operations Research*, 43:399–414, 1995.

[29] T. Erlebach, M. Hoffmann, D. Krizanc, M. Mihalák, and R. Raman. Computing minimum spanning trees with uncertainty. In *Proceedings of the 25th Annual Symposium on Theoretical Aspects of Computer Science (STACS '08)*, pages 277–288, 2008.

[30] T. Feder, R. Motwani, R. Panigrahy, C. Olston, and J. Widom. Computing the median with uncertainty. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 602–607, 2000.

[31] T. Feder, R. Motwani, L. O'Callaghan, C. Olston, and R. Panigrahy. Computing shortest paths with uncertainty. *Journal of Algorithms*, 62:1–18, 2007.

[32] Federal Ministry of Education and Research (BMBF), Germany. Project of the future: Industry 4.0, 2012. URL `http://www.bmbf.de/en/19955.php`.

[33] D. Feillet, P. Dejax, M. Gendreau, and C. Gueguen. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks*, 44:216–229, 2004.

[34] M. Fischetti, J. J. Salazar Gonzalez, and P. Toth. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, 45: 378–394, 1997.

[35] M. Fischetti, A. Lodi, and P. Toth. Solving real-world ATSP instances by branch-and-cut. In M. Jünger, G. Reinelt, and G. Rinaldi, editors, *Combinatorial Optimization – Eureka, You Shrink!*, volume 2570 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 2003.

[36] H. Flordal, D. Spensieri, K. Akesson, and M. Fabian. Supervision of multiple industrial robots: optimal and collision free work cycles. In *Proceedings of the 13th annual IEEE International Conference on Control Applications (CCA '04)*, pages 1404–1409, 2004.

[37] M. Gamache, F. Soumis, D. Villeneuve, J. Desrosiers, and E. Gelinas. The preferential bidding system at Air Canada. *Transportation Science*, 32:246–255, 1998.

[38] G. Gamrath. Verallgemeinerung des Chinesischen Postbotenproblems. Bachelor thesis, Technische Universität Berlin, 2013. In German.

[39] E. Gawrilow, E. Köhler, R. H. Möhring, and B. Stenzel. Dynamic routing of automated guided vehicles in real-time. In *Mathematics – Key Technology for the Future*, pages 165–177. Springer, 2008.

[40] T. J. Gellert and F. G. König. 1D vehicle scheduling with conflicts. In *Proceedings of the 13th Workshop on Algorithm Engineering and Experiments (ALENEX '11)*, pages 107–115, 2011.

[41] M. Gerdts. *Optimal Control of ODEs and DAEs*. Walter de Gruyter, 2012.

[42] M. Gerdts, R. Henrion, D. Hömberg, and C. Landry. Path planning and collision avoidance for robots. *Numerical Algebra, Control and Optimization*, 2:437–463, 2012.

[43] C. Goerzen, Z. Kong, and B. Mettler. A survey of motion planning algorithms from the perspective of autonomous UAV guidance. *Journal of Intelligent and Robotic Systems*, 57:65–100, 2010.

[44] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[45] M. Grötschel, R. Borndörfer, and A. Löbel. Duty scheduling in public transit. In W. Jäger and H.-J. Krebs, editors, *Mathematics – Key Technology for the Future*, pages 653–674. Springer, 2003.

[46] Guinness World Records. Fastest time to travel to all New York City Subway stations, 2013. URL `http://www.guinnessworldrecords.com/world-records/travelling-new-york-city-subway-in-shortest-time-(underground)`. Accessed: 2014-12-13.

[47] M. Gupta, Y. Sabharwal, and S. Sen. A generalized query model for uncertain data, 2010. URL `http://www.cse.iitd.ernet.in/~gmanoj/paper/uncertainity.pdf`.

[48] G. Gutin and A. Punnen, editors. *The Traveling Salesman Problem and Its Variations*, volume 12 of *Combinatorial Optimization*. Springe, 2007.

[49] A. Henry-Labordere. The record balancing problem: A dynamic programming solution of a generalized traveling salesman problem. *RIRO*, B-2:43–49, 1969.

[50] C. Hierholzer. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6:30–32, 1873.

[51] S. Hong. *A Linear Programming Approach for the Traveling Salesman Problem.* PhD thesis, Johns Hopkins University, 1972.

[52] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2014. URL `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[53] S. Irnich and G. Desaulniers. Shortest path problems with resource constraints. In G. Desaulniers, J. Desrosiers, and M. Solomon, editors, *Column Generation*, pages 33–65. Springer, 2005.

[54] D. W. Johnson and E. Gilbert. Minimum time robot path planning in the presence of obstacles. In *Proceedings of the 24th annual IEEE Conference on Decision and Control (CDC '85)*, pages 1748–1753, 1985.

[55] S. Kahan. A model for data in motion. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing (STOC '91)*, pages 265–277, 1991.

[56] B.-I. Kim, S. Kim, and S. Sahoo. Waste collection vehicle routing problem with time windows. *Computers & Operations Research*, 33:3624–3642, 2006.

[57] J. Komlós and E. Szemerédi. Limit distribution for the existence of Hamiltonian cycles in a random graph. *Discrete Mathematics*, 43:55–63, 1983.

[58] B. Korte and J. Vygen. *Combinatorial Optimization – Theory and Algorithms.* Springer, 2002.

[59] C. Landry, R. Henrion, D. Hömberg, M. Skutella, and W. A. Welz. Task assignment, sequencing and path-planning in robotic welding cells. In *Proceedings of the 18th International Conference on Methods and Models in Automation and Robotics (MMAR '13)*, pages 252–257, 2013.

[60] C. Landry, W. A. Welz, and M. Gerdts. A coupling of discrete and continuous optimization to solve kinodynamic motion planning problems. Preprint 1900, WIAS, Berlin, 2013.

[61] G. Laporte, A. Asef-Vaziri, and C. Sriskandarajah. Some applications of the generalized travelling salesman problem. *Journal of the Operational Research Society*, pages 1461–1467, 1996.

[62] J. Larsen. *Parallelization of the Vehicle Routing Problem with Time Windows.* PhD thesis, Technical University of Denmark, 1999.

[63] S. M. LaValle. *Planning Algorithms.* Cambridge University Press, 2006.

[64] J. K. Lenstra and A. Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11:221–227, 1981.

[65] A. Maheshwari, J.-R. Sack, and H. N. Djidjev. Link distance problems. *Handbook of Computational Geometry*, pages 519–558, 1999.

[66] C. E. Noon and J. C. Bean. An efficient transformation of the generalized traveling salesman problem. Technical Report 89-36, The University of Michigan, 1989.

[67] A. Orman and H. Williams. A survey of different integer programming formulations of the travelling salesman problem. In E. Kontoghiorghes and C. Gatu, editors, *Optimisation, Econometric and Financial Analysis*, volume 9 of *Advances in Computational Management Science*, pages 91–104. Springer, 2007.

[68] T. Polzin and S. Vahdati Daneshmand. A comparison of Steiner tree relaxations. *Discrete Applied Mathematics*, 112:241–261, 2001.

[69] J. Rambau and C. Schwarz. How to avoid collisions in scheduling industrial robots. Preprint, 2010.

[70] J. Rambau and C. Schwarz. Solving a vehicle routing problem with resource conflicts and makespan objective with an application in car body manufacturing. *Optimization Methods and Software*, 29:353–375, 2014.

[71] G. Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal of Computing*, 3:376–384, 1991.

[72] M. Reuther. Tourenplanung mit Längenbeschränkung. Diploma thesis, University of Applied Sciences Mittweida, 2008. In German.

[73] G. Righini and M. Salani. New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Networks*, 51:155–170, 2008.

[74] P. R. Samson. The rise and fall of the amateur new york subway riding committee, 1980. URL `http://www.gricer.com/anysrc/anysrc.html`.

[75] A. Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency.* Springer, 2003.

[76] C. Schwarz. *Integrated Routing & Scheduling.* PhD thesis, University of Bayreuth, 2011.

[77] F. Schwarzer, M. Saha, and J.-C. Latombe. Adaptive dynamic collision checking for single and multiple articulated robots in complex environments. *IEEE Transactions on Robotics*, 21:338–353, 2005.

[78] M. Skutella and W. A. Welz. Route planning for robot systems. In B. Hu, K. Morasch, S. Pickl, and M. Siegle, editors, *Operations Research Proceedings 2010*, pages 307–312. Springer, 2011.

[79] M. Skutella and W. A. Welz. Conflict-free dispatching of welding robots. Unpublished extended abstract submitted to the 17th European Conference on Mathematics for Industry (ECMI '12), 2012.

[80] K. Sohraby, D. Minoli, and T. Znati. *Wireless Sensor Networks – Technology, Protocols, and Applications.* John Wiley & Sons, 2007.

[81] D. Spensieri, R. Bohlin, and J. S. Carlson. Coordination of robot paths for cycle time minimization. In *Proceedings of the 9th annual IEEE International Conference on Automation Science and Engineering (CASE '13)*, pages 522–527, 2013.

[82] S. Srivastava, S. Kumar, R. Garg, and P. Sen. Generalized traveling salesman problem through n sets of nodes. *CORS Journal*, 7:97–101, 1969.

[83] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44: 585–591, 1997.

[84] A. Thomason. A simple linear expected time algorithm for finding a Hamilton path. In B. Bollobás, editor, *Proceedings of the Cambridge Combinatorial Conference in Honour of Paul Erdös*, volume 43 of *Annals of Discrete Mathematics*, pages 373–379. Elsevier, 1989.

[85] W. A. Welz. Route planning for robot systems. Diploma thesis, Technische Universität Berlin, 2010.

[86] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, 2011.

[87] S. Winter. Modeling costs of turns in route planning. *GeoInformatica*, 6:345–361, 2002.

Appendix A

Computational Results Chapter 3

In Chapter 3 we described and formalized the pricing problem of our column generation approach for the discrete WCP. The overall performance of the underlying shortest path problem with forced time windows depends on many factors that were evaluated in that chapter. In the following we give the complete numerical results for all the tests performed in Section 3.5.

A.1 Time Window Data Structures

First, we give the results of the tests concerning the different data structure of storing and evaluating the discrete time windows. There is a table for each of the three data structures. Each table contains the name of the corresponding instance, the number of nodes and the number of arcs. Further, we give the computation time in seconds, the number of iterations that were required for the DSSR and the value of the min-cost tour. A value of ∞ indicates that the corresponding instance is infeasible.

instance	nodes	arcs	time (s)	it.	solution
SOL_C101	25	600	0.0	1	11.40
SOL_C102	25	600	6.0	6	-39.15
SOL_C103	25	600	370.0	8	-39.28
SOL_C105	25	600	0.0	1	7.12
SOL_C106	25	600	0.0	1	11.40
SOL_C107	25	600	0.0	4	0.13
SOL_C108	25	600	6.4	7	-51.51
SOL_C201	25	600	0.0	2	14.32
SOL_C202	25	600	3.1	9	-15.51
SOL_C203	25	600	264.8	12	-15.51
SOL_C204	25	600	1,261.3	15	-19.10
SOL_C205	25	600	0.2	8	0.82
SOL_C206	25	600	1.8	11	0.82
SOL_C207	25	600	11.8	12	-1.75
SOL_C208	25	600	2.8	11	0.82
SOL_RC101	25	600	0.0	1	∞

instance	nodes	arcs	time (s)	it.	solution
SOL_RC102	25	600	0.0	4	37.07
SOL_RC103	25	600	0.0	4	37.07
SOL_RC104	25	600	0.0	4	37.07
SOL_RC105	25	600	0.0	1	43.88
SOL_RC106	25	600	0.0	1	∞
SOL_RC107	25	600	0.0	1	44.15
SOL_RC108	25	600	0.0	1	43.88
SOL_RC201	25	600	0.0	1	49.25
SOL_RC202	25	600	0.7	6	19.69
SOL_RC203	25	600	1.4	7	19.69
SOL_RC204	25	600	43.2	11	8.58
SOL_RC205	25	600	0.1	5	34.96
SOL_RC206	25	600	0.0	2	38.48
SOL_RC207	25	600	5.1	8	22.41
SOL_RC208	25	600	184.2	16	18.88
A_20_10	13	124	0.2	7	−130.73
A_20_124	15	211	0.1	7	−223.14
A_20_159	16	240	0.4	7	−269.99
A_20_200	15	211	0.3	6	−323.00
A_20_207	16	240	0.9	7	−335.83
A_20_30	13	122	0.3	9	−148.46
A_31_211	14	182	0.1	6	−178.13
A_31_265	21	420	0.9	8	−319.94
A_31_273	21	420	0.9	8	−335.57
A_31_277	21	420	0.7	7	−313.34
A_31_32	16	240	0.4	6	−58.61
A_31_36	16	240	0.2	6	−8.05
A_31_40	16	240	0.2	5	−65.94
A_31_44	16	240	0.4	8	−55.45
A_31_689	21	414	11.1	9	−247.68
A_31_69	21	420	23.7	11	−354.51
A_31_929	21	420	7.1	8	−509.14
A_31_989	21	420	3.3	8	−270.66
D_35_103	27	676	0.9	5	−180.88
D_35_115	27	676	9.6	7	−197.87
D_35_119	27	676	60.8	10	−172.68
D_35_121	22	401	1.1	8	−197.32
D_35_123	27	676	41.0	8	−243.45
D_35_127	27	676	44.5	9	−237.41
D_35_155	27	676	3.5	7	−202.62
D_35_181	22	462	8.4	7	−174.44
D_35_199	27	676	12.0	8	−260.55
D_35_205	22	441	30.8	10	−138.33
D_35_219	27	676	107.1	9	−254.06
D_35_231	27	676	2.4	7	−182.18
D_35_255	27	676	3.8	7	−144.26
D_35_299	27	676	77.1	8	−189.21
D_35_327	27	676	33.0	9	−318.26
D_35_331	27	676	46.8	9	−291.15
D_35_351	27	675	113.8	9	−247.01
D_35_363	27	676	0.6	5	−145.03
D_38_111	23	479	23.8	8	−394.44
D_38_115	23	479	99.5	9	−270.00
D_38_179	23	480	43.5	10	−257.62
D_38_212	26	650	1.2	7	−233.65

instance	nodes	arcs	time (s)	it.	solution
D_38_223	23	480	58.5	9	−364.48
D_38_279	23	479	5.8	8	−382.87
D_38_295	23	480	2.9	7	−216.03
D_38_299	23	480	152.5	10	−340.40
D_38_311	23	480	9.2	7	−278.71
D_38_315	23	479	39.7	8	−326.98
D_38_319	23	479	105.9	10	−734.61
D_38_360	26	650	16.0	7	−593.57
D_38_411	23	478	64.0	9	−258.15
D_38_417	24	552	1.7	7	−198.50
D_38_421	24	552	1.8	6	−235.70

Table A.1: List data structure

instance	nodes	arcs	time (s)	it.	solution
SOL_C101	25	600	0.0	1	11.40
SOL_C102	25	600	5.7	6	−39.15
SOL_C103	25	600	322.3	8	−39.28
SOL_C105	25	600	0.0	1	7.12
SOL_C106	25	600	0.0	1	11.40
SOL_C107	25	600	0.0	4	0.13
SOL_C108	25	600	5.8	7	−51.51
SOL_C201	25	600	0.0	2	14.32
SOL_C202	25	600	2.8	9	−15.51
SOL_C203	25	600	320.6	12	−15.51
SOL_C204	25	600	1,312.7	15	−19.10
SOL_C205	25	600	0.2	8	0.82
SOL_C206	25	600	1.7	11	0.82
SOL_C207	25	600	10.1	12	−1.75
SOL_C208	25	600	2.4	11	0.82
SOL_RC101	25	600	0.0	1	∞
SOL_RC102	25	600	0.0	4	37.07
SOL_RC103	25	600	0.0	4	37.07
SOL_RC104	25	600	0.0	4	37.07
SOL_RC105	25	600	0.0	1	43.88
SOL_RC106	25	600	0.0	1	∞
SOL_RC107	25	600	0.0	1	44.15
SOL_RC108	25	600	0.0	1	43.88
SOL_RC201	25	600	0.0	1	49.25
SOL_RC202	25	600	0.7	6	19.69
SOL_RC203	25	600	1.4	7	19.69
SOL_RC204	25	600	42.3	11	8.58
SOL_RC205	25	600	0.1	5	34.96
SOL_RC206	25	600	0.0	2	38.48
SOL_RC207	25	600	5.1	8	22.41
SOL_RC208	25	600	176.2	16	18.88
A_20_10	13	124	0.2	7	−130.73
A_20_124	15	211	0.1	7	−223.14
A_20_159	16	240	0.4	7	−269.99
A_20_200	15	211	0.3	6	−323.00
A_20_207	16	240	0.9	7	−335.83
A_20_30	13	122	0.3	9	−148.46
A_31_211	14	182	0.1	6	−178.13

instance	nodes	arcs	time (s)	it.	solution
A_31_265	21	420	0.9	8	−319.94
A_31_273	21	420	0.9	8	−335.57
A_31_277	21	420	0.7	7	−313.34
A_31_32	16	240	0.4	6	−58.61
A_31_36	16	240	0.2	6	−8.05
A_31_40	16	240	0.2	5	−65.94
A_31_44	16	240	0.4	8	−55.45
A_31_689	21	414	11.1	9	−247.68
A_31_69	21	420	25.4	11	−354.51
A_31_929	21	420	7.5	8	−509.14
A_31_989	21	420	3.4	8	−270.66
D_35_103	27	676	1.0	5	−180.88
D_35_115	27	676	9.4	7	−197.87
D_35_119	27	676	69.9	10	−172.68
D_35_121	22	401	1.2	8	−197.32
D_35_123	27	676	44.5	8	−243.45
D_35_127	27	676	46.9	9	−237.41
D_35_155	27	676	3.4	7	−202.62
D_35_181	22	462	7.7	7	−174.44
D_35_199	27	676	13.3	8	−260.55
D_35_205	22	441	32.5	10	−138.33
D_35_219	27	676	121.5	9	−254.06
D_35_231	27	676	2.2	7	−182.18
D_35_255	27	676	3.6	7	−144.26
D_35_299	27	676	58.6	8	−189.21
D_35_327	27	676	27.3	9	−318.26
D_35_331	27	676	39.2	9	−291.15
D_35_351	27	675	92.3	9	−247.01
D_35_363	27	676	0.6	5	−145.03
D_38_111	23	479	20.7	8	−394.44
D_38_115	23	479	84.5	9	−270.00
D_38_179	23	480	38.9	10	−257.62
D_38_212	26	650	1.1	7	−233.65
D_38_223	23	480	70.5	9	−364.48
D_38_279	23	479	6.2	8	−382.87
D_38_295	23	480	3.0	7	−216.03
D_38_299	23	480	177.0	10	−340.40
D_38_311	23	480	10.0	7	−278.71
D_38_315	23	479	46.2	8	−326.98
D_38_319	23	479	89.1	10	−734.61
D_38_360	26	650	15.6	7	−593.57
D_38_411	23	478	58.7	9	−258.15
D_38_417	24	552	1.5	7	−198.50
D_38_421	24	552	1.5	6	−235.70

Table A.2: Lookup data structure

instance	nodes	arcs	time (s)	it.	solution
SOL_C101	25	600	0.0	1	11.40
SOL_C102	25	600	5.6	6	−39.15
SOL_C103	25	600	323.7	8	−39.28
SOL_C105	25	600	0.0	1	7.12
SOL_C106	25	600	0.0	1	11.40

instance	nodes	arcs	time (s)	it.	solution
SOL_C107	25	600	0.0	4	0.13
SOL_C108	25	600	5.6	7	−51.51
SOL_C201	25	600	0.0	2	14.32
SOL_C202	25	600	3.2	9	−15.51
SOL_C203	25	600	246.4	12	−15.51
SOL_C204	25	600	1,261.1	15	−19.10
SOL_C205	25	600	0.2	8	0.82
SOL_C206	25	600	1.8	11	0.82
SOL_C207	25	600	9.8	12	−1.75
SOL_C208	25	600	2.9	11	0.82
SOL_RC101	25	600	0.0	1	∞
SOL_RC102	25	600	0.0	4	37.07
SOL_RC103	25	600	0.0	4	37.07
SOL_RC104	25	600	0.0	4	37.07
SOL_RC105	25	600	0.0	1	43.88
SOL_RC106	25	600	0.0	1	∞
SOL_RC107	25	600	0.0	1	44.15
SOL_RC108	25	600	0.0	1	43.88
SOL_RC201	25	600	0.0	1	49.25
SOL_RC202	25	600	0.7	6	19.69
SOL_RC203	25	600	1.4	7	19.69
SOL_RC204	25	600	42.1	11	8.58
SOL_RC205	25	600	0.1	5	34.96
SOL_RC206	25	600	0.0	2	38.48
SOL_RC207	25	600	5.1	8	22.41
SOL_RC208	25	600	180.6	16	18.88
A_20_10	13	124	0.2	7	−130.73
A_20_124	15	211	0.1	7	−223.14
A_20_159	16	240	0.4	7	−269.99
A_20_200	15	211	0.3	6	−323.00
A_20_207	16	240	0.9	7	−335.83
A_20_30	13	122	0.3	9	−148.46
A_31_211	14	182	0.1	6	−178.13
A_31_265	21	420	0.9	8	−319.94
A_31_273	21	420	0.9	8	−335.57
A_31_277	21	420	0.7	7	−313.34
A_31_32	16	240	0.4	6	−58.61
A_31_36	16	240	0.2	6	−8.05
A_31_40	16	240	0.2	5	−65.94
A_31_44	16	240	0.4	8	−55.45
A_31_689	21	414	10.9	9	−247.68
A_31_69	21	420	23.4	11	−354.51
A_31_929	21	420	7.4	8	−509.14
A_31_989	21	420	3.4	8	−270.66
D_35_103	27	676	0.9	5	−180.88
D_35_115	27	676	10.0	7	−197.87
D_35_119	27	676	63.5	10	−172.68
D_35_121	22	401	1.2	8	−197.32
D_35_123	27	676	42.0	8	−243.45
D_35_127	27	676	42.9	9	−237.41
D_35_155	27	676	3.2	7	−202.62
D_35_181	22	462	7.2	7	−174.44
D_35_199	27	676	12.3	8	−260.55
D_35_205	22	441	31.6	10	−138.33
D_35_219	27	676	105.5	9	−254.06

instance	nodes	arcs	time (s)	it.	solution
D_35_231	27	676	2.2	7	−182.18
D_35_255	27	676	3.7	7	−144.26
D_35_299	27	676	60.5	8	−189.21
D_35_327	27	676	28.9	9	−318.26
D_35_331	27	676	39.4	9	−291.15
D_35_351	27	675	89.7	9	−247.01
D_35_363	27	676	0.6	5	−145.03
D_38_111	23	479	20.9	8	−394.44
D_38_115	23	479	84.9	9	−270.00
D_38_179	23	480	40.4	10	−257.62
D_38_212	26	650	1.3	7	−233.65
D_38_223	23	480	60.4	9	−364.48
D_38_279	23	479	5.7	8	−382.87
D_38_295	23	480	3.0	7	−216.03
D_38_299	23	480	153.6	10	−340.40
D_38_311	23	480	9.3	7	−278.71
D_38_315	23	479	40.5	8	−326.98
D_38_319	23	479	86.6	10	−734.61
D_38_360	26	650	14.9	7	−593.57
D_38_411	23	478	55.2	9	−258.15
D_38_417	24	552	1.7	7	−198.50
D_38_421	24	552	1.7	6	−235.70

Table A.3: BitArray data structure

A.2 Time Window Preprocessing

The preprocessing of time windows, described in Section 3.4.2 relies on the existence of forced arcs. Therefore, only the WCP instances containing at least one forced arc have been considered.

In the following we give the results without preprocessing, as the preprocessed results correspond to the outcomes described in Table A.3.

instance	nodes	edges	time (s)	it.	solution
A_20_124	15	211	0.1	9	−223.14
A_20_159	16	240	0.7	8	−269.99
A_20_200	15	211	0.8	7	−323.00
A_20_207	16	240	2.7	8	−335.83
A_31_689	21	414	15.1	9	−247.68
A_31_929	21	420	14.0	9	−509.14
A_31_989	21	420	11.0	9	−270.66
D_35_103	27	676	5.2	7	−180.88
D_35_115	27	676	16.8	8	−197.87
D_35_119	27	676	86.7	11	−172.68
D_35_123	27	676	87.1	9	−243.45
D_35_127	27	676	72.1	10	−237.41
D_35_155	27	676	4.7	8	−202.62
D_35_199	27	676	19.0	9	−260.55

instance	nodes	edges	time (s)	it.	solution
D_35_205	22	441	37.1	10	−138.33
D_35_219	27	676	268.1	10	−254.06
D_35_255	27	676	8.4	9	−144.26
D_35_299	27	676	105.0	8	−189.21
D_35_327	27	676	50.3	10	−318.26
D_35_331	27	676	61.3	10	−291.15
D_35_351	27	675	189.6	10	−247.01
D_35_363	27	676	0.6	5	−145.03
D_38_111	23	479	121.1	10	−394.44
D_38_115	23	479	499.6	11	−270.00
D_38_179	23	480	96.4	12	−257.62
D_38_223	23	480	94.7	10	−364.48
D_38_279	23	479	13.4	9	−382.87
D_38_295	23	480	19.6	10	−216.03
D_38_299	23	480	408.4	11	−340.40
D_38_311	23	480	34.5	9	−278.71
D_38_315	23	479	144.1	10	−326.98
D_38_319	23	479	968.5	13	−734.61
D_38_360	26	650	47.2	9	−593.57
D_38_411	23	478	292.6	11	−258.15
D_38_421	24	552	4.3	8	−235.70

Table A.4: Solving WCP instances without preprocessing