FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Optimierungsansätze zur Planung von Freizeit-Fahrradrouten

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Benedikt Klocker B.Sc.
Matrikelnummer 0926194

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. DI Dr. Günther Raidl
Mitwirkung: Mag. DI Dr. Matthias Prandtstetter

Wien, 22. April 2015

Benedikt Klocker

Günther Raidl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Optimization Approaches for Recreational Bicycle Tour Planning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Benedikt Klocker B.Sc.
Registration Number 0926194

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. DI Dr. Günther Raidl
Assistance: Mag. DI Dr. Matthias Prandtstetter

Vienna, 22nd April, 2015

Benedikt Klocker | Günther Raidl

Erklärung zur Verfassung der Arbeit

Benedikt Klocker B.Sc.
Neustiftgasse 31/27, 1070 Wien

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

Wien, 22. April 2015

Benedikt Klocker

Danksagung

Diese Arbeit ist Teil eines Projektes des AIT Austrian Institute of Technology, welchem ich für diese Möglichkeit danken will. Einen besonderen Dank möchte ich meinen Betreuern Prof. Günther Raidl und Dr. Matthias Prandstetter für ihre stetige Unterstützung, ihre professionellen Ratschläge, ihre Geduld und ihre konstruktive Kritik aussprechen. Weiters möchte ich Alec Hager von der Radlobby Wien dafür danken, dass ich deren Popularitätsdaten zum Testen verwenden darf.

Des Weiteren möchte ich meinen Eltern dafür danken, dass sie mir die Möglichkeit gegeben haben zu studieren. Ich danke ihnen und meiner Freundin Stefanie Posch für ihre durchgängige Unterstützung. Zusätzlich danke ich Johann Klocker, Ruth Crevis und Adam Tynas dafür, dass sie Teile meiner Arbeit Korrektur gelesen haben.

Acknowledgements

This thesis is based on a project of the AIT Austrian Institute of Technology, which I want to thank for this opportunity. Especially, I would like to thank my advisors Prof. Günther Raidl and Dr. Matthias Prandstetter for their constant support and advice, their patience and their constructive feedback. Moreover I want to thank Alec Hager from the "Radlobby Wien" for letting me use their popularity data for testing purposes.

Furthermore, I want to thank my parents for giving me the possibility to study and I also want to thank them and my girlfriend Stefanie Posch for their continuous support. Additionally I want to thank Johann Klocker, Ruth Crevis and Adam Tynas for proof reading parts of this thesis.

Kurzfassung

Körperliche Aktivität ist wichtig, um gesund zu bleiben. Deshalb entwickeln wir einen Algorithmus, um schöne Freizeit-Fahrradrouten zu berechnen, mit dem Ziel, das Fahrradfahren attraktiver zu machen.

Wir formulieren die Aufgabe als ein mathematisches Optimierungsproblem, welches dem „arc orienteering problem“ (AOP) ähnlich ist. Das Problem ist auf einem gerichteten Multigraphen definiert und hat als Ziel, die Attraktivität einer Route zu maximieren, während die Länge der Route beschränkt ist. Die mehrfache Verwendung einer Straße ist erlaubt, führt aber zu einer Verringerung der Gesamtattraktivität. Das Problem ist NP-schwer und daher ist die Entwicklung von Algorithmen, die in der Praxis in akzeptabler Zeit gute Lösungen liefern, besonders wichtig.

Drei gemischt-ganzzahlige lineare Programme werden entwickelt, um das Problem exakt zu lösen. Das erste Programm verwendet als Subtour-Eliminationsbedingungen eine klassische Schnittformulierung, das zweite einen Flussansatz und das dritte Programm die Kombination der ersten beiden Formulierungen.

Das Testen der Implementierungen der drei gemischt-ganzzahligen linearen Programme unter der Benützung von CPLEX ergibt, dass die Flussformulierung für die meisten Testinstanzen schneller ist als die anderen zwei Formulierungen. Beim Vergleich unserer Implementierung der Flussformulierung mit anderen exakten Algorithmen, welche ähnliche Probleme wie zum Beispiel das AOP behandeln, war unsere Implementierung bis zu 1000 Mal schneller. Verglichen mit heuristischen Lösungsansätzen ähnlicher Probleme erhalten wir, dass für manche Instanzen unsere Implementierung in kurzer Zeit die optimale Lösung findet, während die Heuristiken nur suboptimale Lösungen finden. Allerdings skalieren die Heuristiken besser auf sehr große Instanzen. Unsere Implementierung ist in ländlichen Gegenden für Routen bis zu 60 km und in städtischen Bereichen für Routen bis zu 13 km gut anwendbar, was für die meisten praktischen Zwecke ausreichend erscheint.

Abstract

Exercising is important to stay healthy. Therefore, we develop an algorithm for finding nice recreational bicycle tours with the goal of making exercising by cycling more attractive.

We formulate this challenge as a mathematical optimization problem similar to the arc orienteering problem (AOP) on a directed multigraph. The objective is to maximize the attractiveness of a route under the condition of not exceeding a maximal tour length. It allows multiple usage of streets, but penalizes it such that the attractiveness or score of the route decreases. The problem is NP-hard and developing practically effective algorithms running in reasonable time is therefore crucial.

Three mixed integer linear programs are provided for solving the given problem exactly. The first program uses a classical cut formulation as sub tour elimination, the second a flow formulation and the third the combination of the first two.

Testing the implementations of the three mixed integer linear programs using CPLEX reveals that the flow formulation is for our purposes more efficient than the other two formulations. Compared to other exact algorithms solving similar problems like the AOP, our implementation of the flow formulation is faster up to a factor of 1000. If we compare our implementation with heuristic approaches for similar problems, we get the result that for some instances our implementation finds the optimal solution in short time and the heuristic approaches do not find the optimal solution. However, the heuristics scale better for large instances. On the countryside the algorithm is applicable for routes up to 60 km and in urban areas for routes up to 13 km, which seems to be enough for our intended practical purposes.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

It is commonly known that exercising is important to stay healthy and fit. Especially for people in rehabilitation exercising is important since they have to rebuild their muscles.

This work has been done as part of the project Fit2Trike of the AIT Austrian Institute of Technology. The German title of the project is " Machbarkeitsstudie zur Entwicklung eines altersgerechten E-Tricycles, Routings und Sharing-Systems für Wohneinrichtungen" and it is funded by the Austrian Research Promotion Agency (FFG) with the project number 835903. Its goal is to support people with special needs to do more exercising by cycling. With people with special needs we mean here a very diverse population group of old people, people in rehabilitation or people with chronic diseases like multiple sclerosis. One part of this project is to find suitable recreational bicycle routes for those people. The main goal of this thesis is to find computational approaches to solve this problem fulfilling the practical needs.

We characterize the suitability of a route by the following properties:

- The route starts and ends at a user defined point.
- The length of the route has to be within a custom interval.
- No street is used more than two times in the same direction.
- The route stays within a given radius around the starting point.
- The route should be as attractive as possible which is influenced in a positive way by the following properties:
  - Streets are not used more than once.
  - Cycleways are used instead of heavily trafficked roads.
  - The route leads through nice areas.
  - It includes some points of interest.

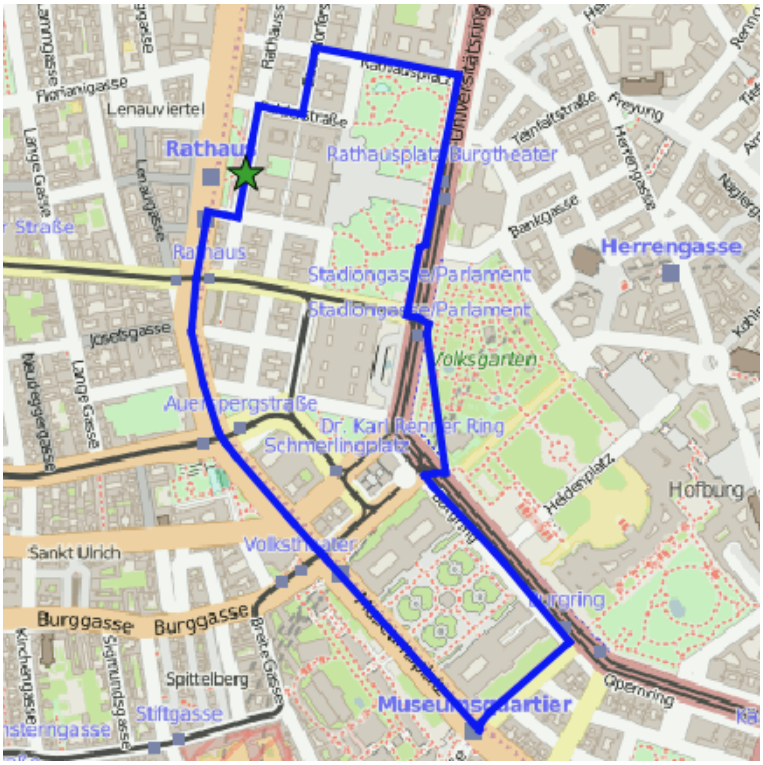

Figure 1.1: Small suitable recreational bicycle route with between 2 km and 3 km length starting and ending at the town hall of Vienna (green star). (©OpenStreetMap contributors)

Figure 1.1 illustrates such a suitable recreational bicycle route with length between 2 km and 3 km, starting and ending at the town hall of Vienna. There are many reasons why computing well suited bicycle routes could improve the attractiveness of bicycling for people with special needs.

- It can give them a safer feeling since they do not have to worry where to drive anymore.
- The fear of not finding home is taken from them.
- Letting the route stay in a small radius has the benefit that the rider can turn around and ride home at any point, for example if he or she is not feeling well. Another positive aspect is that medical help is not far away.
- Avoiding busy roads increases the safety of the route.
- An attractive route can be an additional motivation factor.

1.1 Methodological Approach

To formalize the characterizations from above we will define a mathematical optimization problem, which we will call Recreational Tour Planning Problem(RTPP). It is similar to the arc orienteering problem (AOP) introduced in [22] and even more similar to an extension of the AOP, the so called cycle trip planning problem (CTPP), which is defined in [25]. To model the street system we use a directed multigraph. Every arc has a score, which represents its attractiveness, and a length. The objective of the optimization problem is now to maximize the total score of the route with the restriction that the tour length must be within a given interval. To avoid using a street multiple times we penalize it by reducing the total score. Our defined problem is NP-hard since there is a polynomial time reduction from the traveling salesman problem (TSP) to our problem.

To reduce instances we apply a preprocessing phase before solving the RTPP. To exactly solve the optimization problem we will formulate three mixed integer linear programs. Three different approaches for subtour elimination will be presented. The first subtour elimination is a classical cut formulation, the second a flow formulation and the third the combination of the first two. The relaxation of the first and the second formulations are not comparable. Therefore the relaxation of the third formulation is stronger than the relaxations of the first two. To solve the first and the third formulation, we will use branch-and-cut to dynamically generate violated constraints, since the number of the constraints is exponential.

To test the three formulations we implemented them in C++ using CPLEX. Comparing the three implementation for different test instances will show, that the flow formulation is faster than the other two formulations. To compare our implementation with other approaches from the literature we will use benchmark instances which were used in [22] and [25] and compare our flow formulation with their approaches. The approaches described in [22] and [25] are two mixed integer programs solved with CPLEX, a greedy randomized adaptive search procedure (GRASP) and an iterated local search procedure (ILS). For most instances our implementation is with up to a factor 1000 faster than their mixed integer programs solved with CPLEX. There are also instances where the heuristics GRASP and ILS do not find the optimal solution and our implementation does within a short time. However, the two heuristic approaches scale better for large instances. To test the practical applicability of our implementation we will use test instances representing parts of Josefstadt in Vienna, Kittsee in Austria and East-Flanders in Belgium. The results are that on the countryside the implementation is applicable for tours up to 60 km and in urban areas for tours up to 13 km.

The preparation of map material and attractiveness values is not part of this thesis. For testing we use map material provided by the AIT Austrian Institute of Technology and benchmark instances from the literature.

1.2 Structure of the Work

This thesis is structured as follows.

- After this introduction we specify the details of the route planning problem and come up with a mathematical problem formulation. Several problem transformations will be introduced, yielding new problem variants, to simplify certain aspects and show relationships.
- In Chapter 3 we will discuss complexity issues of our problem variants.
- Related problems are discussed in Chapter 4.
- Chapter 5 contains the description of the algorithms which we are proposing for solving the problem. A general preprocessing is discussed and and the three different MIP-models are introduced.
- Chapter 6 gives an overview on our specific implementation.
- In Chapter 7 we will present some test results for different test instances. Furthermore we will compare one of our algorithms with algorithms for similar problems and we will test the practical applicability of this algorithm.
- Finally, in Chapter 8, we will draw conclusions and outline possible future interesting work.

CHAPTER 2

Recreational Tour Planning Problem

In this chapter, we will specify our problem in mathematical terms. After defining a basic version of the problem we will use some transformations to get a more general formulation which will be then used throughout the whole document.

2.1 Towards a Problem Formulation

In this part we will discuss some requirements for attractive routes and how we can mathematically define them.

To model a street system we will be represented by a directed graph where each intersection corresponds to a node and each part of a street between intersections corresponds to an arc. Since there are street systems where two distinct street parts start and end at the same intersections, we will formally need a multigraph.

We already talked about a few benefits of well suited bicycle routes for people with special needs in Chapter 1. It is clear that our definition of an attractive route should support these benefits.

When referring to bicycle routes we are referring to a closed tour, where a closed tour refers to a route which ends at the same point from which it started.

One of the benefits of well suited bicycle routes is that the user can decide how long a route should be or how long it should last. Since we cannot always find a route of an exact length or duration we will model this by a minimal/maximal route length or duration and then search for routes with lengths or durations in between this interval.

Another benefit is that the user can specify that the route should not lead too far away from the start point. For this we need the restriction that the distance of every point of the route to the start point should not be larger than some given maximal distance. Since we can ensure this property by removing all streets which are farther

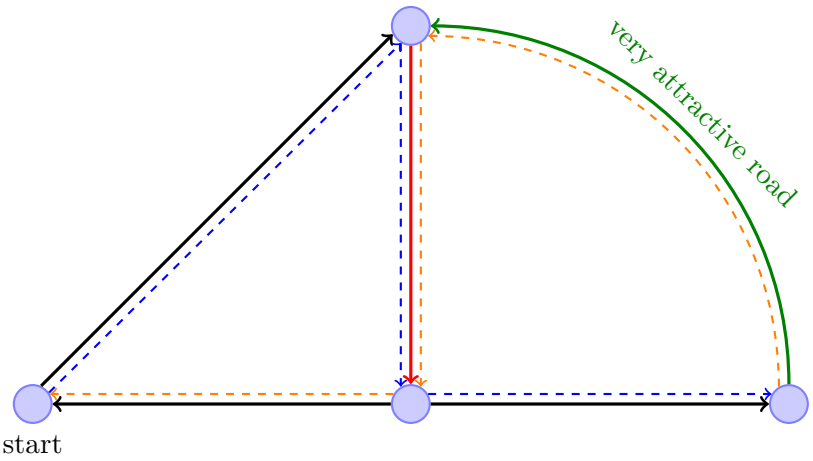

Figure 2.1: Street system where the only way to use the attractive green road and come back to the start point would be to use the red street two times.

away from the start point than the maximal distance, we will not consider this restriction in the problem formulation. This removal can be done in a pre-processing step and is therefore not relevant for our solution methods.

The next benefit of well planned bicycle routes is that we can prepare nice and attractive bicycle routes. So, first we have to think about what "nice" and "attractive" means for a route. Clearly, it is not attractive to drive one street multiple times and therefore it would be nice to forbid this. But the problem is that in some street systems, especially in street systems with many one-way streets, it will be necessary to drive one street multiple times, even in the same direction. Figure 2.1 shows an example of a street system where we need to use a street two times in the same direction to be able to use a very attractive street. The dashed lines represent a tour from the start over the very attractive road back to the start. The blue part is the part of the tour before we reach the attractive road and we can see that it uses the red street once. The orange part is the part from the attractive road back to the start and it also uses the red street once, therefore all in all the complete tour uses the red street twice.

There are also constructions where you need to use a street more than two times to get to all nice roads. Figure 2.2 shows a street system where we need to use one street $n + 1$ times in the same direction in order to use all attractive green streets R_1 to R_n. Although we just showed that it is possible such street systems will appear in the real world only rarely. Also if it would happen in a real street system the question would be if the usage of the attractive green roads would compensate the fact that driving one street three times in the same direction is barely interesting. This leads us to the restriction that we can use one street at most two times in the same direction.

Using the same road more than once is generally less attractive. We penalize multiple road traversals. Therefore we want to give penalties for traversing one road two times in the same direction but we also want to give penalties for using one road two times in

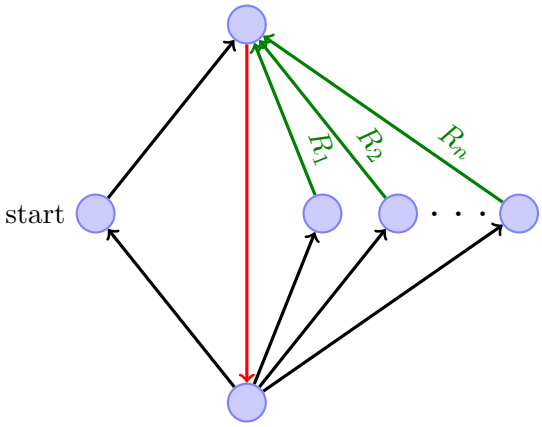

Figure 2.2: Street system where the only way to use all attractive green roads R_1 to R_n is to use the red road $n+1$ times.

opposite directions, since this could also be a little boring and since we want to get nice cycle tours at the end and not only a path to some point and then the same way back. But in complex street systems it could be that a street is split into two parts for each direction and then it could happen that on the right side is an intersection and on the left side not. Now it is not clear anymore which parts of the street usages should get penalized how. How should we penalize it if the route uses the left side and only half of the right side until the intersection? A solution to this problem is to allow penalizing of any two street parts on the map. That means in our example that we can independently penalize the usage of the left side of the street together with the first part of the right side and the usage of the left side of the street together with the second part of the right side. In general this means that we can define for any two street parts s_1 and s_2 that, if we use the street part s_1 and the street part s_2 both at least once in our route, the attractiveness of our route will decrease by some penalty. If $s_1 = s_2$ we define that this penalty is only given if we use the street part two times.

Lastly, it is preferable that busy streets are used only rarely, so as to avoid heavy traffic. It is also assumed that roads with less traffic are more likely to lie in attractive regions (e.g. recreational areas).

To get all this we will provide an attractiveness value for every arc (street part). A high attractiveness value would indicate that the street is maybe a cycle way and that it lies in a nice area. A low attractiveness, which can also be negative, would indicate that the road has maybe heavy traffic or does not lie in a nice area. There could be many more factors which can affect the attractiveness of a street and we will not discuss in detail how to calculate such attractiveness values, which can also heavily depend on the preferences of the user. To calculate an attractiveness value of the whole route we will sum up all attractiveness values of the used arcs and subtract the penalties.

The goal is now to find a closed route with length or duration in the given interval and with a maximal attractiveness.

2.2 First Problem Formulation

We can now formulate the problem introduced in Section 2.1 in a mathematical manner. Before we do this we need a few mathematical definitions. The graph theoretical notations are mostly from the book [12] with a view adaptations.

Definition 2.2.1 (Directed Multigraph)**.** Let V be a set of nodes and A be a set of arcs. Let further be $s : A \to V$ and $t : A \to V$ two functions. Then $G = (V, A, s, t)$ is called a *directed multigraph*. For an arc a we call $s(a)$ its *source node* and $t(a)$ its *target node*. So the arc a describes an arc from the node $s(a)$ to the node $t(a)$.

Remark 2.2.1. We did not forbid that $s(a) = t(a)$ for some arc a, that means we also allow loops in a directed multigraph.

Definition 2.2.2 (Walk)**.** Let $G = (V, A, s, t)$ be a directed multigraph. A *walk* is a finite sequence $w = (w_1, ..., w_k) \in A^k$ for some $k \in \mathbb{N}$ such that $t(w_i) = s(w_{i+1})$ for all $1 \leq i \leq k - 1$. w is a *closed* walk if additionally $s(w_1) = t(w_k)$.

Remark 2.2.2. In [12] a walk is defined as a sequence of arcs and nodes, but since we can get start and end node of an arc very easily by using the functions s and t we omit the nodes in the sequence to get an easier notation.

Definition 2.2.3 (Usage Function)**.** Let $G = (V, A, s, t)$ be a directed multigraph. A *usage function* is a function $u : A \to \mathbb{N}$ where $u(a)$ represents the number of times the arc a is used. Let $w = (w_1, ..., w_k)$ be a walk in G, then we define the *usage function* u^w of w by

$$u^w(a) := |\{j : w_j = a\}|$$

Definition 2.2.4 (Total Costs)**.** Let $G = (V, A, s, t)$ be a directed multigraph and $c : A \to [0, \infty)$ be a cost function ($c(a)$ is the cost of the arc a). Let $u : A \to \mathbb{N}$ be a usage function.

We define the *total costs* of u by

$$c(u) := \sum_{a \in A} u(a) \cdot c(a).$$

We define the *total costs* of a walk w by $c(w) = c(u^w)$ where u^w is the usage function of w.

Remark 2.2.3. In our context a cost value $c(a)$ can be interpreted as the length of a or the duration to drive along the arc a.

Since streets can make curves we cannot assume in general, regardless if $c(a)$ is the length of a or the duration to drive along a, that the triangular inequality holds for c. Figure 2.3 is an example of a street system violating the triangle inequality. When the streets modeled by the arcs a_1 and a_2 are straight and the street modeled by the arc a_3 is curved it is clear that the street a_3 is longer than a_1 and a_2 together. Also if the

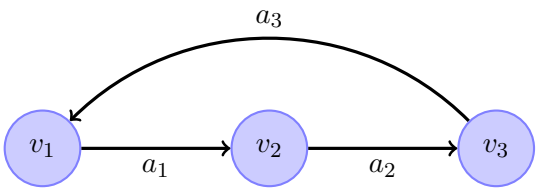

Figure 2.3: Reasonable realistic street system where the triangle inequality does not hold for street lengths

streets are flat the time duration to ride along it will be proportional to the length and therefore the triangular inequality will also not hold in this case for our situation.

Another situation where the triangular inequality does not hold could be when a street is bumpy and therefore the duration to drive along it may be longer than driving along another street even if that would be a detour in terms of length.

That means it can be that there are three arcs a_1, a_2, a_3 with $s(a_2) = t(a_1)$, $s(a_3) = t(a_2)$, $s(a_1) = t(a_3)$ and $c(a_1) > c(a_2) + c(a_3)$. An easy example is if $s(a_2)$ lies on a line between $s(a_1)$ and $s(a_3)$, the streets a_1 and a_2 are straight and the street a_3 is a curved street directly from $s(a_3)$ to $t(a_3) = s(a_1)$.

Definition 2.2.5 (Total Profit)**.** Let $G = (V, A, s, t)$ be a directed multigraph and $p : A \to \mathbb{R}$ be a profit function ($p(a)$ is the profit of the arc a). Let further be $P : A \times A \to \mathbb{R}$ a symmetric penalty function. Let $u : A \to \mathbb{N}$ be a usage function.

We define the *total profit* of u by

$$p(u) := \sum_{a \in A} u(a) \cdot p(a) - \sum_{\substack{a_1, a_2 \in A \\ a_1 \neq a_2 \\ u(a_1) \geq 1 \wedge u(a_2) \geq 1}} P(a_1, a_2) - \sum_{\substack{a \in A \\ u(a) \geq 2}} P(a, a).$$

We define the *total profit* of a walk w by $p(w) = p(u^w)$ where u^w is the usage function of w.

Remark 2.2.4. In our context a profit value $p(a)$ can be interpreted as the attractiveness of a. P represents the penalties. That means $P(a_1, a_2)$ is the penalty if we use the arcs a_1 and a_2 both at least once. $P(a, a)$ is the penalty if we use the arc a at least twice.

Problem (RTPP1)**.** Let $G = (V, A, s, t)$ be a multigraph and $v_{\text{start}} \in V$ be the start node of the searched route. Let further be $[C_{\min}, C_{\max}] \subseteq \mathbb{R}$ a cost interval and let $p : A \to \mathbb{R}$ be a profit function and $c : A \to [0, \infty)$ be a cost function. Let $P : A \times A \to \mathbb{R}$ further be a symmetric penalty function.

The problem is now to find a closed walk $w = (w_1, ...w_k)$ in G with maximal total profit $p(w)$ satisfying the following conditions:

$$s(w_1) = s(w_k) = v_{\text{start}} \tag{2.1}$$
$$u^w(a) \leq 2 \quad \forall a \in A \tag{2.2}$$
$$C_{\min} \leq c(w) \leq C_{\max} \tag{2.3}$$

We call the tuple $(G, v_{\text{start}}, C_{\min}, C_{\max}, p, c, P)$ an instance of RTPP1.

2.3 Problem Transformations

In this section we will present some problem transformations, and reach our final problem formulation which will be used for our algorithms.

2.3.1 Adding an end node

For some solution approaches it is easier if the starting and ending point are explicitly not the same, because it makes it easier to distinguish between the start node or the end node and the other nodes used in a walk. Therefore the first transformation will be splitting the start node into two nodes, a start node and an end node such that in general we have two distinct nodes for starting and ending.

Problem (RTPP2)**.** Let G, $C_{\min}$, $C_{\max}$, p, c and P as in the problem definition of RTPP1. Let now be $v_{\text{start}} \in V$ and $v_{\text{end}} \in V$ two distinct nodes, that means $v_{\text{start}} \neq v_{\text{end}}$.

The problem is now to find a walk $w = (w_1, ..., w_k)$ in G with maximal total profit $p(w)$ satisfying (2.2) and (2.3) and the following new condition:

$$s(w_1) = v_{\text{start}}, \quad s(w_k) = v_{\text{end}} \tag{2.4}$$

We call the tuple $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ an instance of RTPP2.

We now present that every instance of RTPP1 can be transformed into an instance of RTPP2 in linear time and vice versa. The figures 2.4 and 2.5 illustrate these transformations.

Theorem 2.3.1. *Let $(G, v_{\text{start}}, C_{\min}, C_{\max}, p, c, P)$ be an instance of RTPP1. Then we can construct in linear time an instance $(G', v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p', c', P')$ of RTPP2 such that the solution w' of RTPP2 can be transformed into a solution w of RTPP1 in constant time.*

Proof. We will formalize the transformation illustrated in figure 2.4. Let $G = (V, A, s, t)$ then we define

$$V' := V \cup \{v_{\text{end}}\} \text{ with } v_{\text{end}} \notin V$$
$$A' := A \cup \{a_{end}\} \text{ with } a_{end} \notin A$$
$$s'(a) := s(a) \quad \forall a \in A \quad \text{and} \quad s'(a_{end}) = v_{\text{start}}$$

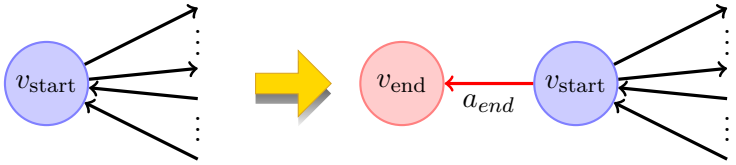

Figure 2.4: Problem transformation from RTPP1 to RTPP2 by inserting a new end node.

$$t'(a) := t(a) \quad \forall a \in A \quad \text{and} \quad t'(a_{end}) = v_{\text{end}}$$

With that we get a new multigraph $G' = (V', A', s', t')$. We further define

$$p'(a) = p(a) \quad \forall a \in A \quad \text{and} \quad p'(a_{end}) = 0$$

$$c'(a) = c(a) \quad \forall a \in A \quad \text{and} \quad c'(a_{end}) = 0$$

$$P(a_1, a_2)' = P(a_1, a_2) \quad \forall a_1, a_2 \in A \quad \text{and} \quad P(a_{end}, a) = P(a, a_{end}) = 0 \quad \forall a \in A'$$

Let now $w' = (w'_1, ..., w'_k)$ be a solution according to the new instance. We know $t(w_k) = v_{\text{end}}$ and therefore it follows that $w_k = a_{end}$ since a_{end} is the only arc with $t(a_{end}) = v_{\text{end}}$. For all $i < k$ we know $w'_i \neq a_{end}$ since there is no arc a satisfying $s(a) = v_{\text{end}}$. We define $\mathbf{w} := (w_1, ..., w_{k-1})$ where $w_i := w'_i$. This is a walk in G and since $s(w_1) = v_{\text{start}}$ and $t(w_{k-1}) = s(w_k) = s(a_{end}) = v_{\text{start}}$ it is a closed walk. It satisfies (2.1) and since $u^w(a) = u^{w'}(a)$ for all $a \in A$ also (2.2) is satisfied. We get

$$c(w') = \sum_{a \in A'} u^{w'}(a) \cdot c'(a) = \sum_{a \in A} u^{w'}(a) \cdot c'(a) = \sum_{i=1}^{m} u^w(a) \cdot c(a) = c(w)$$

for the total cost. Therefore (2.3) is satisfied. Assume now that there exists a closed walk $\hat{w} = (\hat{w}_1, ...\hat{w}_\ell)$ satisfying (2.1)-(2.3) with $p(\hat{w}) > p(w)$. Then $\hat{w}' := (\hat{w}_1, ...\hat{w}_\ell, a_{end})$ is a walk satisfying (2.2)-(2.4) and $p(\hat{w}') = p(\hat{w})$. But this would mean $p(\hat{w}') > p(w')$ which is a contradiction.

Therefore $p(w)$ has to be maximal and therefore w is the solution according to the original instance. □

For the other direction the idea is to add a new artificial node which is the new start and end point. But to avoid that this new artificial node is used more often we have to forbid using an arc going there twice. To achieve this we will set the penalty for using this arc so high that it will not be profitable using it twice.

Definition 2.3.1 ($\|\cdot\|_\infty$)**.** Let $f : M \to \mathbb{R}$ for some set M. Then the maximum norm $\|f\|_\infty$ is defined by

$$\|f\|_\infty = \sup_{x \in M} |f(x)|$$

Definition 2.3.2 ($P_{\max}$)**.** Let A be a set of arcs, $p : A \to \mathbb{R}$ be a cost function and $P : a \times A \to \mathbb{R}$ a symmetric penalty function. Then we define $P_{\max}(p, P)$ by

$$P_{\max}(p,P) = 4|A|\|p\|_\infty + 2|A|^2\|P\|_\infty + 1$$

where $\|\cdot\|_\infty$ is the maximum norm defined in Definition 2.3.1 and $|A|$ is the number of arcs in the set A.

Lemma 2.3.2. *Let $G = (V, A, s, t)$ be a directed multigraph, $p : A \to \mathbb{R}$ a profit function and $P : A \times A \to \mathbb{R}$ a symmetric penalty function. Then for any usage function $u : A \to \{0, 1, 2\}$ we get*

$$|p(u)| < \frac{P_{\max}(p,P)}{2}$$

Proof.

$$\begin{aligned} 2|p(u)| &= 2\left|\sum_{a\in A} u(a)\cdot p(a) - \sum_{\substack{\{a_1,a_2\}\subseteq A \\ a_1\neq a_2 \\ u(a_1)\geq 1,\, u(a_2)\geq 1}} P(a_1,a_2) - \sum_{\substack{a\in A \\ u(a)\geq 2}} P(a,a)\right| \\ &\leq 2\left(|A|\cdot 2\|p\|_\infty + |A|^2\|P\|_\infty\right) \\ &= 4|A|\|p\|_\infty + 2|A|^2\|P\|_\infty < P_{\max}(p,P) \end{aligned}$$

□

Theorem 2.3.3. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ be an instance of* (RTPP2)*. Then we can construct in linear time an instance $(G', v'_{\text{start}}, C_{\min}, C_{\max}, p', c', P')$ of* (RTPP1) *such that the solution w' of* (RTPP1) *can be transformed into a solution w of* (RTPP2) *in linear time.*

Proof. We will formalize the transformation illustrated in figure 2.5. Let $G = (V, A, s, t)$ then we define

$$V' := V \cup \{v'_{\text{start}}\} \text{ with } v'_{\text{start}} \notin V$$

$$A' := A \cup \{a_{\text{start}}, a_{end}\} \text{ with } a_{\text{start}}, a_{end} \notin A$$

$$s'(a) := s(a) \quad \forall a \in A \quad \text{and} \quad s'(a_{\text{start}}) = v_{\text{start}},\ s'(a_{end}) = v_{\text{end}}$$

$$t'(a) := t(a) \quad \forall a \in A \quad \text{and} \quad t'(a_{\text{start}}) = v'_{\text{start}},\ t'(a_{end}) = v'_{\text{start}}$$

With that we get a new multigraph $G' = (V', A', s', t')$. We further define

$$p(a)' = p(a) \quad \forall a \in A \quad \text{and} \quad p'(a_{\text{start}}) = p'(e_{end}) = 0$$

$$c'(a) = c(a) \quad \forall a \in A \quad \text{and} \quad c'(a_{\text{start}}) = c'(e_{end}) = 0$$

$$P'(a_1, a_2) = P(a_1, a_2) \quad \forall a_1, a_2 \in A$$

$$P'(a_{\text{start}}, a) = P'(a, a_{\text{start}}) = P'(a_{end}, a) = P'(a, a_{end}) = 0 \quad \forall a \in A \cup \{a_{\text{start}}\}$$
$$P'(a_{end}, a_{end}) = P_{\max}(p, P)$$

Let now $w' = (w'_1, ..., w'_k)$ be a solution according to the new instance. We know that $s(w'_1) = v'_{\text{start}}$ and therefore $w'_1 = a_{\text{start}}$. In the same way we get $t(w'_k) = v'_{\text{start}}$ and therefore $w'_k = a_{end}$.

We have now two cases, either a_{end} is used twice or a_{end} is used only once.

1. If a_{end} is used twice we will prove that there exists no solution for the original instance. Assume that there exists a walk $w = (w_1, ..., w_\ell)$ in G satisfying the conditions (2.2) to (2.4). Then we can construct a walk $w'' = (a_{\text{start}}, w_1, ..., w_\ell, a_{end})$ in G'. This walk satisfies (2.1) to (2.3) and it holds $p(w'') = p(w)$. We calculate

$$\begin{aligned}
p(w') &= \sum_{a \in A'} u^{w'}(a) \cdot p'(a) - \sum_{\substack{\{a_1, a_2\} \subseteq A' \\ a_1 \neq a_2 \\ u^{w'}(a_1) \geq 1, u^{w'}(a_2) \geq 1}} P'(a_1, a_2) \\
&\quad - \sum_{\substack{a \in A' \\ u^{w'}(a) \geq 2}} P'(a, a) \\
&= \sum_{a \in A} u^{w'}(a) \cdot p(a) - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^{w'}(a_1) \geq 1, u^{w'}(a_2) \geq 1}} P(a_1, a_2) \\
&\quad - \sum_{\substack{a \in A \\ u^{w'}(a) \geq 2}} P(a, a) - P'(a_{end}, a_{end}) \\
&< \frac{P_{\max}(p, P)}{2} - P_{m+2,m+2}
\end{aligned}$$

   The last inequality follows from Lemma 2.3.2 applied to the usage function $u^{w'}|_A$ of the graph G. Now we can apply Lemma 2.3.2 again to the usage function u^w and get

$$p(w') < \frac{P_{\max}(p, P)}{2} - P_{m+2,m+2} = -\frac{P_{\max}(p, P)}{2} < p(w) = p(w'')$$

   But this is a contradiction to the fact that the profit of w' is maximal. Therefore, the original instance has no solution.

2. If a_{end} is only used once we also know that a_{start} is only used once since if $w'_i = a_{\text{start}}$ with $i > 1$ the previous arc w'_{i-1} must be a_{end}. Therefore, we can define the walk

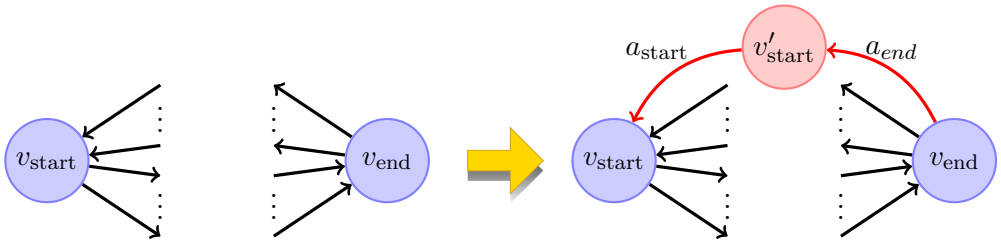

Figure 2.5: Problem transformation from RTPP2 to RTPP1 by inserting a new node, which is the new start and end node at the same time.

$w := (w'_2, ..., w'_{k-1})$ and this is now a walk in G. It holds $p(w) = p(w')$ and $c(w) = c(w')$ and therefore w satisfies the conditions (2.2) to (2.4). Assume there is another walk $\hat{w} = (\hat{w}_1, ..., \hat{w}_\ell)$ in G satisfying (2.2) to (2.4) and with $p(\hat{w}) > p(w)$. Then we could construct the walk $\hat{w}' = (a_{\text{start}}, \hat{w}_1, ..., \hat{w}_\ell, a_{end})$ which would satisfy the conditions (2.1) to (2.3). It holds

$$p(\hat{w}') = p(\hat{w}) > p(w) = p(w')$$

but this is a contradiction since the total profit of w' was maximal. Therefore, the total profit of w has to be maximal too, and we have found a solution of the original problem.

□

2.3.2 Duplicating Arcs

As we will see in Chapter 4 there exist already many solution approaches for similar problems to ours. Many of those similar problems have in common that each arc of the graph can only be used once. Therefore it would maybe make it easier to find good solution approaches if we would have a problem formulation where every arc of the graph can only be used once. Therefore our next goal will be to transform the problem such that every arc can only be used once. To achieve this we will copy each arc. After copying every arc we can restrict the usage of an arc to one, but we still have to simulate the penalty of using an arc twice. We can either lower the attractiveness of the copied arc or set a penalty between the original arc and its copy. To simplify the penalty matrix we will reduce the attractiveness of the copied arc.

Another problem is how to define penalties between different arcs after the transformation. If we keep the penalties between the original arcs and do not penalize any of the copied arcs it could happen that it is more profitable to use the copied arc instead of the original arc at some point. To avoid this we introduce dependencies between arcs. We therefore introduce the concept that one arc depends on another, that means it can only be used in the solution walk if the other arc is also used.

Problem (RTPP3)**.** Let G, $C_{\min}$, $C_{\max}$, p, c, v_{start}, v_{end} and P be as in the problem definition of RTPP2. The penalties $P(a,a)$ will now be unimportant and therefore can be set to 0.

Let further be $D \subseteq A \times A$ a dependency relation.

The problem is now to find a walk $w = (w_1, ..., w_k)$ in G with maximal total profit $p(w)$ satisfying (2.3), (2.4) and the following two new conditions:

$$u^w(a) \in \{0,1\} \quad \forall a \in A \tag{2.5}$$

$$u^w(a_1) \leq u^w(a_2) \quad \forall a_1, a_2 \in A, a_1 D a_2 \tag{2.6}$$

We call the tuple $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ an instance of RTPP3.

If an arc a_1 stands in relation with an arc a_2 ($a_1 D a_2$, read "a_1 depends on a_2") this means that a_1 can only be used if a_2 is used. This is ensured by (2.6) since if a_1 is used that means that $u^w(a_1) = 1$ and therefore also $u^w(a_2) = 1$ and also a_2 is used, this means that a_1 can only be used if a_2 is used.

We will now present a transformation of instances of RTPP2 to RTPP3, which is illustrated in figure 2.6.

Theorem 2.3.4. *Let* $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ *be an instance of* (RTPP2)*. Then we can construct in linear time an instance* $(G', v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p', c', P', D)$ *of* (RTPP3) *such that the solution* w' *of* (RTPP3) *can be transformed into a solution* w *of* (RTPP2) *in linear time.*

Proof. We will formalize the transformation illustrated in figure 2.6. Let $G = (V, A, s, t)$ then we define

$$A' = A \cup \{a' : a \in A\}$$

$$s'(a) := s'(a') := s(a) \quad \forall a \in A$$

$$t'(a) := t'(a') := t(a) \quad \forall a \in A$$

where a' is a new copy of a. With that we get a new graph $G' := (V, A', s', t')$. We further define

$$p'(a) = p(a) \quad \forall a \in A \quad \text{and} \quad p'(a') = p(a) - P(a,a) \quad \forall a \in A$$

$$c'(a) = c'(a') = c'(a) \quad \forall a \in A$$

$$P'(a_1, a_2) = P(a_1, a_2) \quad \forall a_1, a_2 \in A, a_1 \neq a_2 \quad \text{and} \quad P'(a,a) = 0 \quad \forall a \in A$$

$$P'(x, a') = P'(a', x) = 0 \quad \forall x \in A', a \in A$$

$$D = \{(a', a) | a \in A\}$$

Let now $w' = (w'_1, ..., w'_k)$ be a solution according to the new instance. Then we define a walk $w = (w_1, ...w_k)$ in G by

$$w_i = \begin{cases} w'_i & \text{if } w'_i \in A \\ a & \text{if } w'_i = a' \notin A \end{cases}$$

The only difference between w' and w is that copies a' of arcs a are replaced by the originals a. Since $s'(a') = s'(a) = s(a)$, $t'(a') = t'(a) = t(a)$ and $c'(a') = c'(a) = c(a)$ for all arcs $a \in A$ we get that w satisfies (2.3) and (2.4). It is also clear that

$$u^w(a) = u^{w'}(a) + u^{w'}(a') \leq 1 + 1 = 2$$

and therefore also (2.2) holds.

The next step will be to prove that $p(w') = p(w)$. For that we need that $u^{w'}(a) + u^{w'}(a') = u^w(a)$ and therefore $u^w(a) \geq u^{w'}(a)$ and that

$$u^{w'}(a') = 1 \Leftrightarrow u^w(a) = 2 \tag{2.7}$$

holds. The direction from right to left of (2.7) is true since $2 = u^w(a) = u^{w'}(a) + u^{w'}(a')$ implies that $u^{w'}(a') = 1$. For the other direction we use that if $u^{w'}(a') = 1$ then also $u^{w'}(a) = 1$ since w' satisfies (2.6), but that means $u^w(a) = u^{w'}(a) + u^{w'}(a') = 2$. From (2.7) and $u^w(a) \geq u^{w'}(a)$ we get that $u^{w'}(a) \geq 1$ if and only if $u^w(a) \geq 1$.

$$\begin{aligned}
p(w') &= \sum_{a \in A} u^{w'}(a) \cdot p'(a) + u^{w'}(a') \cdot p'(a') - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^{w'}(a_1) \geq 1, u^{w'}(a_2) \geq 1}} P'(a_1, a_2) \\
&= \sum_{a \in A} (u^{w'}(a) + u^{w'}(a')) \cdot p(a) - \sum_{a \in A} u^{w'}(a') \cdot P(a, a) \\
&\quad - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^w(a_1) \geq 1, u^w(a_2) \geq 1}} P(a_1, a_2) \\
&= \sum_{a \in A} u^w(a) \cdot p(a) - \sum_{\substack{a \in A \\ u^{w'}(a') = 1}} P(a, a) - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^w(a_1) \geq 1, u^w(a_2) \geq 1}} P(a_1, a_2) \\
&= \sum_{a \in A} u^w(a) \cdot p(a) - \sum_{\substack{a \in A \\ u^w(a) = 2}} P(a, a) - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^w(a_1) \geq 1, u^w(a_2) \geq 1}} P(a_1, a_2) \\
&= p(w)
\end{aligned}$$

Let us now assume that there exists a closed walk $\hat{w} = (\hat{w}_1, ...\hat{w}_\ell)$ in G satisfying (2.2) to (2.4) with $p(\hat{w}) > p(w)$. We construct from this closed walk a closed walk $\hat{w}' = (\hat{w}'_1, ...\hat{w}'_\ell)$ in G' by defining

$$\hat{w}'_i = \begin{cases} a & \text{if } \hat{w}_i = a \text{ and } \forall j < i : \hat{w}_j \neq a \\ a' & \text{if } \hat{w}_i = a \text{ and } \exists j < i : \hat{w}_j = a \end{cases}$$

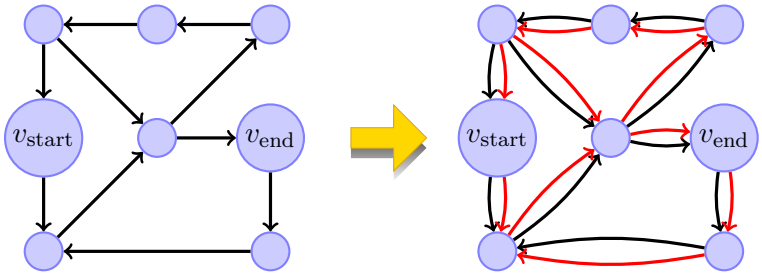

Figure 2.6: Problem transformation from RTPP2 to RTPP3 for an example graph by copying each arc.

$\hat{w}'$ still satisfies (2.3) and (2.4). It also satisfies (2.5) since $\hat{w}$ satisfies (2.2). By definition $\hat{w}'$ contains only a copied arc a' if it also contains a and therefore $\hat{w}'$ satisfies also (2.6). It is easy to see that $\hat{w}$ stands in the same relation with $\hat{w}'$ as w does with w' and therefore we can use our calculation and get also $p(\hat{w}) = p(\hat{w}')$. Now we have

$$p(\hat{w}') = p(\hat{w}) > p(w) = p(w')$$

which is a contradiction since w' has maximal profit. Therefore, w has also maximal profit and is therefore a solution of RTPP2.

□

In the following chapters we will mainly use problem formulation RTPP3 for further discussions and algorithms.

2.3.3 Removing Dependencies

In this section we will present a transformation to remove the dependency conditions from the problem formulation First we will prove that we can assume without loss of generality that the dependency relation D is transitive.

Definition 2.3.3 (Transitive)**.** Let $R \subseteq A \times A$ be a binary relation. R is called *transitive* if the following holds

$$(R(a_1, a_2) \wedge R(a_2, a_3)) \rightarrow R(a_1, a_3) \quad \forall a_1, a_2, a_3 \in A. \tag{2.8}$$

Definition 2.3.4 (Transitive Closure)**.** Let $R \subseteq A \times A$ be a binary relation. The *transitive closure* R^+ of R is the smallest transitive relation on $A \times A$ containing R.

Remark 2.3.1. For finite relations like we deal with the transitive closure can be constructed by finding triples (a_1, a_2, a_3) such that (2.8) is unsatisfied and add the pair (a_1, a_3) to the relation until (2.8) is satisfied for all triples. This algorithm stops at the latest when the relation is the whole $A \times A$ which is transitive.

Theorem 2.3.5. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3 then the solutions of this instance are exactly the same as the solutions of the instance*

$$(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D^+).$$

Proof. Since the only difference between the two instances are the dependencies it is sufficient to show that a walk w satisfies (2.6) for D if and only if it satisfies (2.6) for D^+.

Since $D^+ \supseteq D$ it is clear that every walk satisfying (2.6) for D^+ also satisfies (2.6) for D.

For the other direction let w be a walk satisfying (2.6) for D. We define the relation

$$R := \{(a_1, a_2) | u^w(a_1) \leq u^w(a_2)\}$$

on $A \times A$. It is clear that $R \supseteq D$ and the next step is to show that R is transitive. Let therefore be $a_1, a_2, a_3 \in A$ such that $a_1 R a_2$ and $a_2 R a_3$. That means $u^w(a_1) \leq u^w(a_2)$ and $u^w(a_2) \leq u^w(a_3)$ which implies $u^w(a_1) \leq u^w(a_3)$ and therefore $a_1 R a_3$. So we get that the transitivity of $\leq$ implies the transitivity of our relation R. Now we know that R is transitive and contains D and therefore by definition we know R contains D^+. But this exactly means that w satisfies (2.6) for D^+.

□

The previous theorem shows us that we always can assume that the dependency relation D is transitive without changing the solutions.

In most cases the penalties will be nonnegative but we did not restrict our problem to nonnegative penalties and therefore we can use negative penalties to avoid dependencies. The following theorem removes a dependency of two arcs which have the same start and end point and the same costs.

Theorem 2.3.6. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3 with a transitive dependency relation D and with $G = (V, A, s, t)$ and let $a_1 \in A$, $a_2 \in A$ with $a_1 D a_2$ and $a_2 \not D a_1$, such that $s(a_1) = s(a_2)$, $t(a_1) = t(a_2)$ and $c(a_1) = c(a_2)$.*

Let further be $P_{\text{diff}} \in \mathbb{R}$ such that

$$P_{\text{diff}} > p(a_1) - p(a_2) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a, a_1) - P(a, a_2) < 0}} (P(a, a_1) - P(a, a_2))$$

and

$$p'(a_1) = p(a_1) - P_{\text{diff}} \quad p'(a) = p(a) \quad \forall a \in A \setminus \{a_1\},$$
$$P'(a_1, a_2) = P'(a_2, a_1) = P(a_1, a_2) - P_{\text{diff}}$$
$$P'(a, \hat{a}) = P(a, \hat{a}) \quad \forall (a, \hat{a}) \in A \times A \setminus \{(a_1, a_2), (a_2, a_1)\}.$$

Then the set of solutions S of the instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ is exactly the same as the set of solutions S' of the instance

$$(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p', c, P', D \setminus \{(a_1, a_2)\}).$$

Proof. First of all we prove that for every $w' \in S'$ it holds that if a_1 is used in w' also a_2 is used.

Let us assume there is an optimal walk w' which uses a_1 but not a_2. Then we could generate another walk $\hat{w}'$ by using a_2 instead of a_1 since the two arcs have the same start and end point. $\hat{w}'$ still satisfies (2.3) to (2.5). for $a, a' \in A \setminus \{a_1, a_2\}$ and aDa' we get $u^{\hat{w}'}(a) = u^{w'}(a)$ and $u^{\hat{w}'}(a') = u^{w'}(a')$ and therefore also $u^{\hat{w}'}(a) \leq u^{\hat{w}'}(a')$.

It remains to check the cases where $a = a_2$ or $a' = a_1$. If $a = a_1$ or $a' = a_2$ the equation $u^{\hat{w}'}(a) \leq u^{\hat{w}'}(a')$ trivially holds.

Let now be $a = a_2$, then $a' \neq a_1$ since $a_2 \not{D} a_1$. Then $a_1 D a'$ holds since D is transitive. Since $u^{w'}(a_1) = 1$ it follows that $u^{w'}(a') = 1$ and therefore also $u^{\hat{w}'}(a') = 1$ and from this we get again $u^{\hat{w}'}(a) \leq u^{\hat{w}'}(a')$.

For the last case, let $a' = a_1$ and $a \neq a_2$. Then by transitivity of D we get again aDa_2 and therefore $u^{w'}(a) = 0$. From this we get again $u^{\hat{w}'}(a) = 0$ and therefore $u^{\hat{w}'}(a) \leq u^{\hat{w}'}(a')$.

We proved now that $\hat{w}'$ satisfies (2.6) for D and therefore it is a valid walk for problem RTPP3. We will show that $p'(\hat{w}') > p'(w')$ which is a contradiction to the optimality of w'.

$$
\begin{aligned}
p'(\hat{w}') - p'(w') &= p'(a_2) - \sum_{\substack{a \in A \setminus \{a_2\} \\ u^{\hat{w}'}(a) = 1}} P'(a, a_2) - p'(a_1) + \sum_{\substack{a \in A \setminus \{a_1\} \\ u^{w'}(a) = 1}} P'(a, a_1) \\
&= p'(a_2) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ u^{w'}(a) = 1}} P'(a, a_2) - p'(a_1) \\
&\quad + \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ u^{w'}(a) = 1}} P'(a, a_1) \\
&> p(a_2) + \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a, a_1) - P(a, a_2) < 0}} (P(a, a_1) - P(a, a_2)) \\
&\quad - p(a_1) + P_{\text{diff}} \\
&> 0
\end{aligned}
$$

The last inequality holds by definition of P_{diff}. Therefore, we get a contradiction and therefore every solution w' uses a_1 only if it uses a_2 and therefore satisfies (2.6) for the whole D. In the next step we prove that every walk w satisfying (2.6) for the whole D has the same total profit $p(w) = p'(w)$ regardless which of the two profit and penalty functions we take. If w does not use the arc a_1 this is clear since p and p' and P and P' only differ concerning the arc a_1. Let therefore be w a walk satisfying(2.6) for the whole

D and using a_1. Then it also uses a_2. We calculate now $p'(w)$.

$$\begin{aligned}
p'(w) &= \sum_{a\in A} u^w(a)\cdot p'(a) - \sum_{\substack{\{a,a'\}\subseteq A\\ a\neq a'\\ u^w(a)\geq 1 \text{ and } u^w(a')\geq 1}} P'(a,a') \\
&= \sum_{a\in A} u^w(a)\cdot p(a) - P_{\text{diff}} - \sum_{\substack{\{a,a'\}\subseteq A\\ a\neq a'\\ u^w(a)\geq 1 \text{ and } u^w(a')\geq 1}} P(a,a') + P_{\text{diff}} \\
&= \sum_{a\in A} u^w(a)\cdot p(a) - \sum_{\substack{\{a,a'\}\subseteq A\\ a\neq a'\\ u^w(a)\geq 1 \text{ and } u^w(a')\geq 1}} P(a,a') \\
&= p(w)
\end{aligned}$$

We proved now that all solution walks in S' satisfy (2.6) for the whole D and all walks satisfying (2.6) for the whole D have the same profit regardless of measuring according to p and P or to p' and P'. Since the costs of the walks remain in both instances the same we get that the optimal walks according to the original instance are also optimal according to the new instance with one dependency less and vice versa. Therefore, $S = S'$.

□

Now we know how to remove one dependency under the given conditions, but if we want to use this theorem iteratively we need to always generate the transitive closure of the remaining dependency relation. To ensure that the dependency relation is really getting smaller we therefore need $(D \setminus (a_1, a_2))^+ \subsetneq D$. This is only the case if $(D \setminus (a_1, a_2))^+ = D \setminus (a_1, a_2)$ and therefore if $D \setminus (a_1, a_2)$ is transitive. To achieve this we need that there is no $a_3 \in A$ with $a_1 D a_3$ and $a_3 D a_2$. To ensure this we need that D is acyclic. But first we have to define what that means.

Definition 2.3.5. A binary relation $R \subseteq A \times A$ *contains a cycle* if there exists $a_1, ..., a_k \in A$ with $k \geq 2$ and $a_2 \neq a_1$ such that $a_i R a_{i+1}$ for all $i = 1, ...k-1$ and $a_k R a_1$.

If a binary relation contains no cycles we call it *acyclic*.

Remark 2.3.2. Because loops are not relevant for us we do not call a loop a cycle and therefore every cycle has to consist of at least two elements. Loops can always be removed from our dependency relation since they do not change the model.

Lemma 2.3.7. *Let $R \subseteq A \times A$ be an acyclic binary relation and $B \subseteq A$ a nonempty finite subset. Then we can find an R-minimal element of B, that means we find an element $b \in B$ such that for all $c \in B \setminus \{b\}$ it holds $c \not R b$.*

Proof. We prove this by contradiction. Let us assume there exists no R-minimal element in B. We will construct now a cycle. Since B is nonempty there exists a $b_0 \in B$. Let us now assume we already constructed a sequence $(b_0, ..., b_k)$ such that $b_{i+1}Rb_i$. Since b_k is not B-minimal we can find a $b_{k+1} \in B \setminus \{b_k\}$ such that $b_{k+1}Rb_k$. If there exists an $i < k$ such that $b_i = b_{k+1}$ we have found a cycle $(b_i, ..., b_k)$. Since B is finite this has to happen at some point and therefore we always get a cycle. □

With this lemma we can apply Theorem 2.3.6 iteratively.

Theorem 2.3.8. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3 with a transitive, acyclic dependency relation D such that for all $(a_1, a_2) \in D$ it holds $s(a_1) = s(a_2)$, $t(a_1) = t(a_2)$ and $c(a_1) = c(a_2)$.*

Then we can construct in polynomial time an instance

$$(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p', c, P', \emptyset)$$

with the same solutions as

$$(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D).$$

Proof. We prove this by induction on the size $n = |D|$. If $|D| = 1$ we can use Theorem 2.3.6 and get a new instance with an empty dependency relation with the same solutions.

Let now be $|D| = n + 1$ and $(a, b) \in D$. We apply Lemma 2.3.7 to the subset

$$B = \{c \in A | aDc\}.$$

Because $b \in B$ we know that B is nonempty and B is finite since A is finite. Therefore, we get a minimal element $b_0 \in B$. We apply now Theorem 2.3.6 to the pair $(a, b_0) \in D$. And we get an instance

$$(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p', c, P', D \setminus \{(a, b_0)\})$$

with the same solutions as the original instance. We prove now that $D \setminus \{(a, b_0)\}$ is transitive. Let $a_1, a_2, a_3 \in A$ such that $(a_1, a_2) \in D \setminus \{(a, b_0)\}$ and $(a_2, a_3) \in D \setminus \{(a, b_0)\}$. We get a_1Da_2 and a_2Da_3 and therefore a_1Da_3 since D is transitive. If $a_1 \neq a$ or $a_3 \neq b_0$ we get $(a_1, a_3) \in D \setminus \{(a, b_0)\}$.

Let us assume $a_1 = a$ and $a_3 = b_0$, then we get $a_2 \neq b_0$ since $(a, a_2) \in D \setminus \{(a, b_0)\}$. But b_0 was minimal in B and this is a contradiction to a_2Db_0. Therefore, this case cannot arise and we proved transitivity of $D \setminus \{(a, b_0)\}$. With that we can use the induction hypothesis and get an instance

$$(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p'', c, P'', \emptyset)$$

which has the same solutions than the original instance. □

One problem of Theorem 2.3.8 is that you cannot give upper bounds for the profit and penalty values occurring in the modified p' and P'. Since the value P_{diff} from Theorem 2.3.6 for some dependency aDb strongly depends on the penalty values of a or b with all other arcs. Therefore, if we remove a dependency aDb we modify the penalty of (a, b) and if we remove afterwards the dependency bDc the modified penalty (a, b) will be used in the sum of P_{diff} and therefore it can happen a sum up of all such values P_{diff} and at the end P_{diff} can be very large. To avoid this we need a more restricted version of these theorems which still applies to instances which we get when we use a transformation like in Theorem 2.3.4.

Theorem 2.3.9. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3 with a dependency relation D, which has no chains and no forks and such that for all $(a_1, a_2) \in D$ it holds $s(a_1) = s(a_2)$, $t(a_1) = t(a_2)$ and $c(a_1) = c(a_2)$. D has no chains means that there are no $a_1, a_2, a_3 \in A$ such that $a_1 D a_2 D a_3$. D has no forks means that there are no $a_1, a_2, a_3 \in A$ such that*

$$(a_1 D a_2 \wedge a_1 D a_3) \vee (a_1 D a_3 \wedge a_2 D a_3).$$

Let further be $P_{\text{diff}}(a_1, a_2) \in \mathbb{R}$ for $a_1, a_2 \in A$ such that

$$P_{\text{diff}}(a_1, a_2) > p(a_1) - p(a_2) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a, a_1) - P(a, a_2) < 0}} (P(a, a_1) - P(a, a_2))$$

and

$$p'(a) = p(a) - P_{\text{diff}}(a, b) \quad \forall a \in A, b \in B \, aDb$$

$$p'(a) = p(a) \quad \forall a \in \{a \in A : \forall b \in B \, a \not D b\}$$

$$P'(a_1, a_2) = P'(a_2, a_1) = P(a_1, a_2) - P_{\text{diff}}(a_1, a_2) \quad \forall (a_1, a_2) \in D$$

$$P'(a_1, a_2) = P(a_1, a_2) \forall (a_1, a_2) : a_1 \not D a_2 \wedge a_2 \not D a_1$$

Then the set of solutions S of the instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ is exactly the same as the set of solutions S' of the instance

$$(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p', c, P', \emptyset).$$

Proof. First of all we mention that D and every subset E of D is transitive since D and therefore also every subset E of D does not contain any chains.

We construct now iteratively p_k, P_k and D_k by applying Theorem 2.3.6 in every step such that at the end $p_n = p'$, $P_n = P'$ and $D_n = \emptyset$. Let first be $p_0 = p$ and $P_0 = P$. Let us assume we already constructed p_k, P_k and D_k. We want to apply Theorem 2.3.6 for the instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p_k, c, P_k, D_k)$. Since in each step we applied

2.3.6 we know $D_k \subseteq D$ and therefore that D_k is transitive. If D_k is empty we are done. Otherwise let $(a_1, a_2) \in D_k$. We know

$$\begin{aligned} P_{\text{diff}}(a_1, a_2) &> p(a_1) - p(a_2) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a,a_1) - P(a,a_2) < 0}} (P(a, a_1) - P(a, a_2)) \\ &= p_k(a_1) - p_k(a_2) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P_k(a,a_1) - P_k(a,a_2) < 0}} (P_k(a, a_1) - P_k(a, a_2)) \end{aligned}$$

The last inequality holds since D has no chains and no forks and therefore there is no $a \in A \setminus \{a_1, a_2\}$ such that aDa_1 or a_1Da or aDa_2 or a_2Da and therefore the profit values and penalty values occurring in $P_{\text{diff}}(a_1, a_2)$ did not change until now. Therefore, we can again apply Theorem 2.3.6 for the pair (a_1, a_2) with $P_{\text{diff}} = P_{\text{diff}}(a_1, a_2)$ and get new values p_{k+1}, P_{k+1} and D_{k+1}.

At some point this procedure stops and $D_n = \emptyset$ and it is easy to see that then $p_n = p'$ and $P_n = P'$ must hold. Since in every step the new instance has the same solutions than the old instance we get at the end that the instance with p', P' and $D' = \emptyset$ also has the same solutions than the original instance. □

Until now we always assumed that there are only dependencies between arcs with the same start and endpoints and the same costs. In the next theorem we will prove how to remove dependencies which do not satisfy this condition. But for this we need again the large value $P_{\max}$ from definition 2.3.2.

Theorem 2.3.10. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3 with $G = (V, A, s, t)$.*

Let further be

$$p'(a) = p(a) - P_{\max}(p, P) \cdot |\{a' \in A : aDa'\}| \quad \forall a \in A$$

$$P'(a_1, a_2) = P'(a_2, a_1) = P(a_1, a_2) - P_{\max}(p, P) \cdot |\{(a_1, a_2), (a_2, a_1)\} \cap D'| \quad \forall a_1, a_2 \in A$$

If the set of solutions S of the instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ is non-empty, then it is equal to the set S' of solutions of the instance

$$(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p', c, P', \emptyset).$$

If S is nonempty, all solutions w in $S = S'$ satisfy $p(w) = p'(w) > -P_{\max}(p, P)/2$ and, if S is empty, S' only contains solutions w with $p'(w) < -P_{\max}(p, P)/2$.

Proof. First we calculate the new profit $p'(w)$ for an arbitrary walk w.

$$\begin{aligned}
p'(w) &= \sum_{a\in A} u^w(a)p'(a) - \sum_{\substack{\{a_1,a_2\}\subseteq A \\ a_1\neq a_2 \\ u^w(a_1)\geq 1 \wedge u^w(a_2)\geq 1}} P'(a_1,a_2) \\
&= \sum_{a\in A} u^w(a)p(a) - P_{\max}(p,P) \sum_{\substack{a\in A \\ u^w(a)=1}} |\{a'\in A : aDa'\}| \\
&\quad - \sum_{\substack{\{a_1,a_2\}\subseteq A \\ a_1\neq a_2 \\ u^w(a_1)\geq 1 \wedge u^w(a_2)\geq 1}} P(a_1,a_2) - P_{\max}(p,P)|\{(a_1,a_2),(a_2,a_1)\}\cap D| \\
&= \sum_{a\in A} u^w(a)p(a) - P_{\max}(p,P)|\{(a_1,a_2)\in D|u^w(a_1)=1\}| \\
&\quad - \sum_{\substack{\{a_1,a_2\}\subseteq A \\ a_1\neq a_2 \\ u^w(a_1)\geq 1 \wedge u^w(a_2)\geq 1}} P(a_1,a_2) \\
&\quad + P_{\max}(p,P)|\{(a_1,a_2)\in D|u^w(a_1)=1\wedge u^w(a_2)=1\}| \\
&= p(w) - P_{\max}(p,P)|\{(a_1,a_2)\in D|u^w(a_1)=1\wedge u^w(a_2)=0\}|
\end{aligned}$$

From this calculation we directly get with Lemma 2.3.2 that if a walk w contains a_1 but not a_2 for some $(a_1,a_2)\in D$ it holds

$$p'(w) \leq p(w) - P_{\max}(p,P) < \frac{P_{\max}(p,P)}{2} - P_{\max}(p,P) = -\frac{P_{\max}(p,P)}{2}.$$

If a walk w satisfies (2.6) for D we get also by the above calculation that $p(w)=p'(w)$ and by Lemma 2.3.2 it holds

$$p'(w) = p(w) > -\frac{P_{\max}(p,P)}{2}$$

Therefore we know that, if there exists a solution $w\in S$ it has a higher profit than all walks which are not satisfying (2.6) for D and therefore is also optimal in the new instance, that means $w\in S'$. On the other side if a walk w is in S' and $p(w) > -\frac{P_{\max}(p,P)}{2}$ we know that it satisfies (2.6) for D and therefore is also a valid walk for the original instance. But for all valid walks w of the original instance it holds $p(w)=p'(w)$ and therefore the walk w has also to be optimal under p, that means $w\in S$. We proved now that, if $S\neq\emptyset$, we get $S=S'$ and, if $S=\emptyset$, either $S'=\emptyset$ or every solution w in S' does not satisfy (2.6) for D and therefore $p'(w) < -\frac{P_{\max}(p,P)}{2}$.

□

In the theorems 2.3.8, 2.3.9 and 2.3.10 we saw three possibilities to remove the dependency relation from the problem formulation. The former two can only be used for a special dependency relation structure but the last one can be used for all dependency relations. Therefore, the last one is a general transformation from RTPP3 to the following problem formulation.

Problem (RTPP4)**.** Let G, $C_{\min}$, $C_{\max}$, p, c, v_{start}, v_{end} and P be as in the problem definition of RTPP3. We now have no dependency relation D.

The problem is now to find a walk $w = (w_1, ..., w_k)$ in G with maximal total profit $p(w)$ satisfying the conditions (2.3) - (2.5).

We call the tuple $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ an instance of RTPP3.

Until now we have seen transformations from RTPP1 to RTPP2, from RTPP2 to RTPP1, from RTPP2 to RTPP3 and from RTPP3 to RTPP4. To close the circle we will now find a transformation from RTPP4 to RTPP2.

Theorem 2.3.11. *Let $I = (G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ be an instance of* (RTPP4) *with $G = (V, A, s, t)$. Let P' be defined by*

$$P'(a, b) = P'(b, a) = P(a, b) \quad \forall a, b \in A : a \neq b$$

$$P'(a, a) = P_{\max}(p, P) \quad \forall a \in A$$

Then $I' := (G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P')$ is an instance of (RTPP2) *such that the following relation between the solution set S_1 of I and the solution set S_2 of I' hold.*

- *If S_2 is empty or the profit of the solutions in S_2 is lower $-P_{\max}(p, P)/2$, then the solution set S_1 is empty*
- *If S_2 contains solutions with profit higher or equal than $-P_{\max}(p, P)/2$ we get $S_1 = S_2$.*

Proof. Since in RTPP2 every arc can be used twice and in RTPP4 every arc can only be used once it is clear that every valid walk in RTPP4 is also a valid walk in RTPP2. By definition of the new instance we get for a valid walk w in RTPP4

$$\begin{aligned} p'(w) &= \sum_{a \in A} u^w(a) \cdot p(a) - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^w(a_1) \geq 1 \text{ and } u^w(a_2) \geq 1}} P'(a_1, a_2) \\ &\quad - \sum_{\substack{a \in A \\ u^w(a) \geq 2}} P'(a, a) \\ &= \sum_{a \in A} u^w(a) \cdot p(a) - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^w(a_1) \geq 1 \text{ and } u^w(a_2) \geq 1}} P(a_1, a_2) = p(w) \end{aligned}$$

since $u^w(a) \leq 1$ for all $a \in A$. In the next step we prove that for a walk w with $u^w(a_0) = 2$ for some $a_0 \in A$ it holds $p'(w) \leq -P_{\max}(p, P)$. Since the diagonal of P was irrelevant in the formulation of problem RTPP4 we can assume that $P(a, a) = 0$ for all $a \in A$. Then we get

$$\begin{aligned} p'(w) &= \sum_{a \in A} u^w(a) \cdot p(a) - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^w(a_1) \geq 1 \text{ and } u^w(a_2) \geq 1}} P'(a_1, a_2) \\ &\quad - \sum_{\substack{a \in A \\ u^w(a) \geq 2}} P'(a, a) \\ &\leq -P_{\max}(p, P) + \sum_{a \in A} u^w(a) \cdot p(a) - \sum_{\substack{\{a_1, a_2\} \subseteq A \\ a_1 \neq a_2 \\ u^w(a_1) \geq 1 \text{ and } u^w(a_2) \geq 1}} P(a_1, a_2) \\ &= -P_{\max}(p, P) + p(w) < -P_{\max}(p, P) + \frac{P_{\max}(p, P)}{2} = -\frac{P_{\max}(p, P)}{2}. \end{aligned}$$

Therefore we have two cases

- S_2 is empty or the solutions have profit lower than $-P_{\max}(p, P)/2$. We already know that every solution $w \in S_1$ is also a valid walk in RTPP2 and therefore we get since $p'(w) = p(w) > -P_{\max}(p, P)/2$ that such a w cannot exist and therefore that $S_1 = \emptyset$.
- S_2 is not empty and the solutions have profit higher or equal than $-P_{\max}(p, P)/2$. Therefore, we get that $u^w(a) \leq 1$ for all $w \in S_2$ and $a \in A$ since otherwise we would get $p'(w) < -P_{\max}(p, P)/2$. But this means that all solutions $w \in S_2$ are also valid walks in RTPP4. Since $p(w) = p'(w)$ for all valid walks in RTPP4 we get $S_1 = S_2$.

□

In Figure 2.7 we can see now all transformations we have formulated until now from this we get the following corollary.

Corollary 2.3.12. *Let P_1 and P_2 be two distinct problems of the problems RTPP1, RTPP2, RTPP3 or RTPP4 and let I be an instance of P_1. Then we can construct in linear time an instance I' of P_2 such that the solution set of I' can be transformed into the solution set of I in linear time.*

2.4 Further Problem Transformations

The goal of this section is to transform our problem formulation with some restrictions into the arc orienteering problem defined in [22]. Although the problem is defined for a

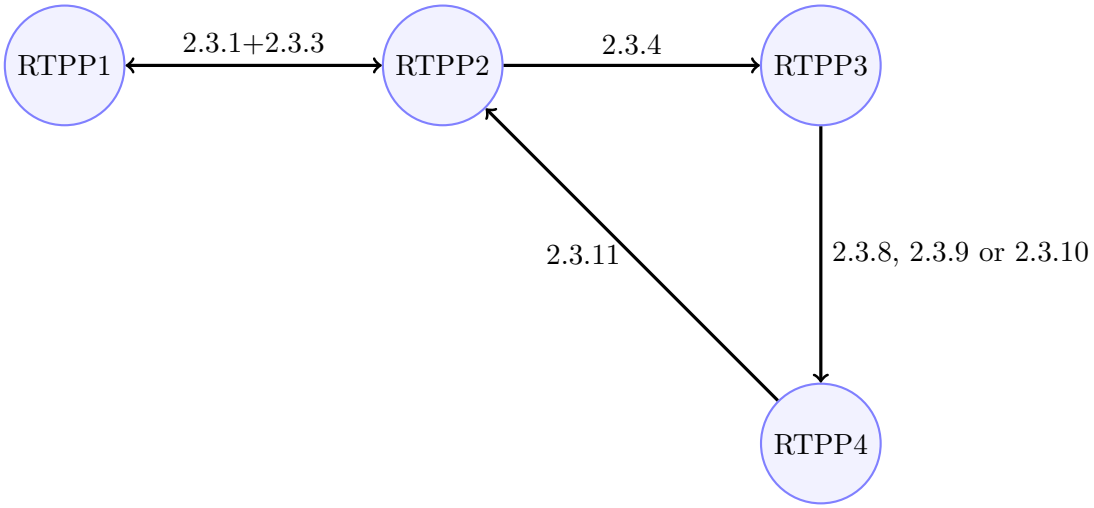

Figure 2.7: Problem transformations.

simple directed graph we will keep the notation of a directed multigraph to stay consistent with our notation.

Definition 2.4.1 (Path)**.** Let $G = (V, A, s, t)$ be a directed multigraph. A walk w in G where every node is visited at most once is called a *path*. That means a walk $w = (w_1, ..., w_k)$ is a path if and only if $t(w_i) \neq t(w_j)$ for all $i \neq j$ and $s(w_1) \neq t(w_k)$.

A path cannot be closed by definition, but if we drop the condition $s(w_1) \neq t(w_k)$ and close the walk, that means add the condition $s(w_1) = t(w_k)$ we call the walk w a *cycle*.

Problem (Arc Orienteering Problem)**.** Let $G = (V, A, s, t)$ be a simple directed graph, $c : A \to [0, \infty)$ a cost function and $p : A \to [0, \infty)$ a profit function. Let further be $C_{\max} \in \mathbb{R}$, $v_{\text{start}} \in V$ a start node and $v_{\text{end}} \in V$ an end node. The problem is to find a path w from v_{start} to v_{end} which has costs $c(w)$ smaller or equal $C_{\max}$ and maximizes the profit $p(w)$. We call this problem Arc Orienteering Problem (AOP).

Be aware that profit values are restricted to be non-negative in contrary to our problem formulations and that walks we search in AOP are not allowed to use one node twice and therefore a simple directed graph is enough. We also do not have penalties. In the first step we will transform problem RTPP4 into a problem where we search an optimal path instead of an optimal walk.

Problem (RTPP5)**.** Let $I = (G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ be an instance of RTPP4.

The problem is now to find instead of a walk a path $w = (w_1, ...w_k)$ in G with maximal total profit $p(w)$ satisfying the conditions (2.3) and (2.4).

In this interpretation we call the tuple $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ an instance of RTPP5.

Theorem 2.4.1. *Let $I = (G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ be an instance of RTPP4. We can construct an in instance $I' = (G', v'_{\text{start}}, v'_{\text{end}}, C'_{\min}, C'_{\max}, p', c', P')$ of the problem RTPP5 such that solutions of the one instance can easily be transformed into solutions of the other instance. The size $|I'|$ of I' is in $\mathcal{O}(k|I|)$ where k is the maximum of the indegrees and outdegrees of all nodes in G.*

Proof. Let $G = (V, A, s, t)$, since every node in the new graph can only be visited once we have to duplicate a node v for each ingoing and outgoing arc. Then we have to connect all nodes standing for ingoing arcs a to all nodes standing for outgoing arcs b with new arcs $x_{v,a,b}$. Additionally we have to add a new start and end node. The result is the following

$$V' := \{v_a^i : v \in V, a \in A, t(a) = v\} \cup \{v_a^o : v \in V, a \in A, s(a) = v\} \cup \{v'_{\text{start}}, v'_{\text{end}}\}$$

$$\begin{aligned} A' \quad := \quad & A \cup \{x_{v,a,b} : v \in V, a, b \in A, t(a) = v, s(b) = v\} \\ & \cup \{x_{v'_{\text{start}},a} : a \in A, s(a) = v_{\text{start}}\} \cup \{x_{v'_{\text{end}},a} : a \in A, t(a) = v_{\text{end}}\} \end{aligned}$$

$$s'(a) = s(a)_a^o \quad \forall a \in A$$

$$s'(x_{v,a,b}) = v_a^i \quad \forall v \in V, a, b \in A, t(a) = v, s(b) = v$$

$$s'(x_{v'_{\text{start}},a}) = v'_{\text{start}} \ \forall a \in A, s(a) = v_{\text{start}} \quad s'(x_{v'_{\text{end}},a}) = v^i_{end,a} \ \forall a \in A, t(a) = v_{\text{end}}$$

$$t'(a) = t(a)_a^i \quad \forall a \in A$$

$$t'(x_{v,a,b}) = v_b^o \quad \forall v \in V, a, b \in A, t(a) = v, s(b) = v$$

$$t'(x_{v'_{\text{start}},a}) = v^o_{start,a} \ \forall a \in A, s(a) = v_{\text{start}} \quad s'(x_{v'_{\text{end}},a}) = v'_{\text{end}} \ \forall a \in A, t(a) = v_{\text{end}}$$

We define $G' = (V', A', s', t')$. Figure 2.8 illustrates an example transformation for a small graph. The costs, the profits and the penalties for the new arcs are defined as follows:

$$c'(a) = c(a),\ p'(a) = p(a) \quad \forall a \in A$$

$$c'(x) = p'(x) = 0 \quad \forall x \in A' \setminus A$$

$$P'(a, b) = P(a, b) \quad \forall a, b \in A$$

$$P'(x, a') = P'(a', x) = 0 \quad \forall x \in A' \setminus A, a' \in A'$$

Let now be $w = (w_1, ..., w_k)$ a valid walk for the instance I. We define walk w' in G' by

$$w' = (x_{v'_{\text{start}},w_1}, w_1, x_{t(w_1),w_1,w_2}, w_2, x_{t(w_2),w_2,w_3}, ..., w_{k-1}, x_{t(w_{k-1}),w_{k-1},w_k}, w_k, x_{v'_{\text{end}},w_k}).$$

It is easy to check that every walk in G' which uses an arc at most once is a path since every node v' in G' has either $deg^+(v') = 1$ or $deg^-(v') = 1$. Therefore, w' is a

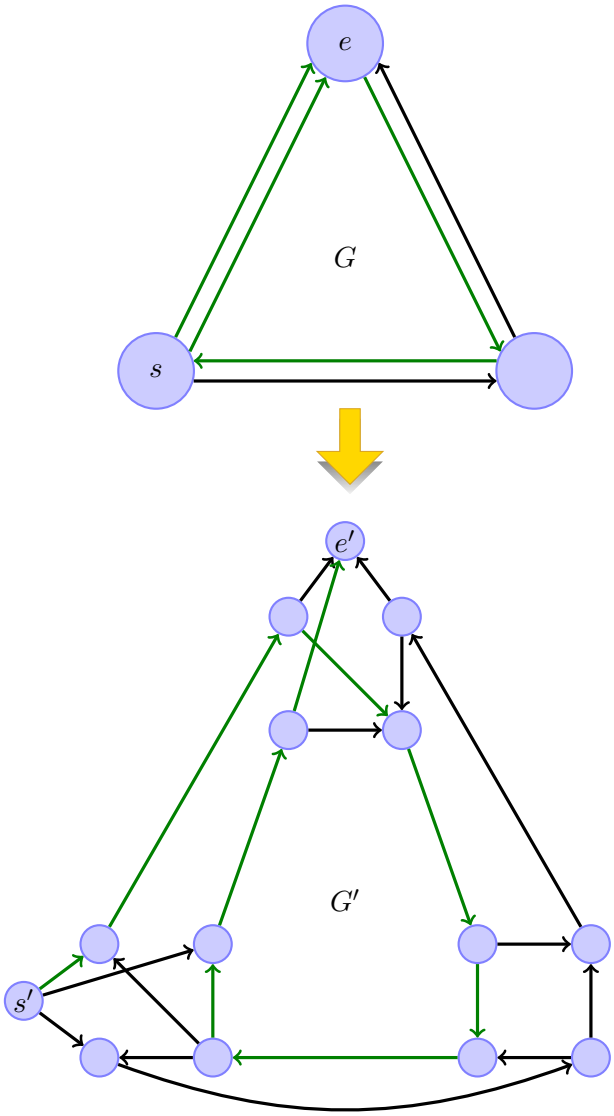

Figure 2.8: Transformation of Theorem 2.4.1 for a small graph G to G'. Two corresponding walks from s to e respectively s' to e' are colored green.

path. Since $c'(w') = c(w)$ we know that w' is a valid path for instance I'. We also get $p'(w') = c(w)$.

Let now on the other hand be $w' = (w'_1, ...w'_k)$ a valid path for the instance I'. All outgoing arcs of a node v_a^i are of the form v_b^o for some arc $b \in A$ with the target node v_b^o. The only outgoing arc for a node v_a^o is the arc $a \in A$ which has as target a node $t'(a) = \hat{v}_a^i$ for another node $\hat{v}$. From the last two statements we get that the sequence of visited nodes through w' must have the form

$$(v'_{\text{start}}, v^o_{start,a_1}, v^i_{1,a_1}, v^o_{1,a_2}, v^i_{2,a_2}, v^o_{2,a_3}, ..., v^i_{end,a_{l-1}}, v^o_{end,a_l}, v'_{\text{end}})$$

for some arcs $a_j \in A$ and nodes $v_j \in V$. It is easy to check that $t(a_j) = s(a_{j+1})$ has to hold by definition. We can now define the walk $w = (a_1, ...a_{l+1})$ in A. We get again $c(w) = c'(w')$ and $p(w) = p'(w')$ and therefore that w is a valid walk for the instance I.

We just presented transformations for valid walks in I to valid paths in I' and vice versa such that they have the same profit. That means this transformations also transform walks with maximal profit in I into paths with maximal profits in I' and vice versa.

To measure the size of I' we have to measure the size of A'. We can calculate by using the definition of A'

$$\begin{aligned} |A'| &= |A| + \sum_{v \in V} deg^+(v) \cdot deg^-(v) + deg^-(v_{\text{start}}) + deg^+(v_{\text{end}}) \\ &\leq 3|A| + 4|V|k^2 \leq 3|A| + 8|A|k \leq 11|I|k \end{aligned}$$

Since we only have to save non zero values for the functions c', p' and P' their size does not change. We also get $|V'| \leq |V| + |A'|$ since we added at least as many arcs as we added nodes. All in all we get $|I'| \leq c|I|k$ for an appropriate constant c and therefore $|I'| = \mathcal{O}(k|I|)$.

□

In the next step we want to get rid of the penalties. First of all we do that for parallel arcs. We can do this before applying Theorem 2.4.1 and therefore for instances of the problem RTPP4, but we need additional restrictions. The idea is similar to the one of Theorem 2.3.6.

Theorem 2.4.2. *Let $I = (G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ be an instance of RTPP4 with $G = (V, A, s, t)$ and let $a_1, a_2 \in A$ be two parallel arcs, that means $s(a_1) = s(a_2)$ and $t(a_1) = t(a_2)$ with the same costs $c(a_1) = c(a_2)$. Let further be $P(a_1, a_2) > 0$ and*

$$p(a_1) - p(a_2) \geq \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a_1,a) - P(a_2,a) > 0}} P(a_1, a) - P(a_2, a) \tag{2.9}$$

or $P(a_1, a_2) < 0$ and

$$p(a_1) - (p(a_2) - P(a_1, a_2)) \geq \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a_1,a) - P(a_2,a) > 0}} P(a_1, a) - P(a_2, a). \tag{2.10}$$

We define a new instance $I' = (G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p', c, P')$ *of RTPP4 with*

$$p'(a) = p(a) \quad \forall a \in A \setminus \{a_2\}$$

$$p'(a_2) = p(a_2) - P(a_1, a_2)$$

$$P'(a, b) = P(a, b) \quad \forall (a, b) \in A \times A \setminus \{(a_1, a_2), (a_2, a_1)\}$$

$$P'(a_1, a_2) = P'(a_2, a_1) = 0.$$

Let S *be the solution set of* I *and* S' *be the solution set of* I'.

Then in the case $P(a_1, a_2) > 0$ *we get* $S \supseteq S'$ *and if* $S \neq \emptyset$ *also* $S' \neq \emptyset$. *In the other case if* $P(a_1, a_2) < 0$ *we get* $S \subseteq S'$ *and if* $S' \neq \emptyset$ *also* $S \neq \emptyset$.

If the inequality (2.9) *respectively* (2.10) *is strict we get* $S = S'$.

Proof. It is easy to see that for a walk w which does not use a_2 or uses a_1 and a_2 we get $p'(w) = p(w)$.

The only interesting case is therefore a walk $w = (w_1, ..., w_\ell, a_2, w_{\ell+2}, ..., w_k)$ which uses a_2 and does not use a_1. Let $w' := (w_1, ..., w_\ell, a_1, w_{\ell+2}, ..., w_k)$ the same walk but using a_1 instead of a_2. We get $c(w') = c(w)$ and therefore w' is also a valid walk in both instances. We have to distinct the following two cases

1. $P(a_1, a_2) > 0$. In this case we get $p'(w) = p(w) - P(a_1, a_2) < p(w)$. We can calculate

$$\begin{aligned} p(w') &= p(w) + p(a_1) - p(a_2) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ u^w(a) = 1}} P(a_1, a) - P(a_2, a) \\ &\geq p(w) + p(a_1) - p(a_2) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a_1, a) - P(a_2, a) > 0}} P(a_1, a) - P(a_2, a) \\ &\overset{(2.9)}{\geq} p(w). \end{aligned}$$

   Therefore if $w \in S$ we also have $w' \in S$ since we have $p'(w) < p(w) \leq p(w') = p'(w')$ we know that $w \notin S'$. Since w was an arbitrary walk which uses a_2 but not a_1 we get that S' does not contain any such walks. But from this and the fact that I and I' are the same for other walks we get

$$S' = S \setminus \{w : w \text{ is a valid walk which uses } a_2 \text{ and not } a_1\}.$$

   If S is nonempty it contains a walk w. If w uses a_2 and not a_1 we can define again w' like above and get $p(w') = p(w)$ and therefore $w' \in S$, that means S contains at least one walk which does not use a_2 or uses a_1 and a_2, but then this walk is also in S' and therefore S' is nonempty.

2. $P(a_1, a_2) < 0$. In this case we get $p'(w) = p(w) - P(a_1, a_2) > p(w)$. Now we calculate

$$\begin{aligned} p'(w') &= p'(w) + p(a_1) - (p(a_2) - P(a_1, a_2)) \\ &\quad - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ u^w(a) = 1}} P(a_1, a) - P(a_2, a) \\ &\geq p'(w) + p(a_1) - (p(a_2) - P(a_1, a_2)) \\ &\quad - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a_1, a) - P(a_2, a) > 0}} P(a_1, a) - P(a_2, a) \\ &\overset{(2.10)}{\geq} p'(w) \end{aligned}$$

We argue the same way as in case 1 with S and S' interchanged and get

$$S = S' \setminus \{w : w \text{ is a valid walk which uses } a_2 \text{ and not } a_1\}.$$

In an analogue way to case 1 we can again prove that if S' is nonempty also S is nonempty.

□

Remark 2.4.1. Let $I = (G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ with $G = (V, A, s, t)$ be an instance of $RTPP2$ and I_1 the result after applying the transformation described in Theorem 2.3.4. By definition of the transformation I_1 does not contain any penalties concerning copied arcs, only for original arcs. Let I_2 now be the result after applying the transformation described in Theorem 2.3.9 where we use $P_{\text{diff}}(a', a) = 0$ if possible for an arc $a \in A$ and its copy a' to remove their dependencies. Note that the only dependencies in I_1 are of this form. This is possible if and only if

$$P(a, a') = p(a) - p(a') > \sum_{\substack{x \in A \setminus \{a, a'\} \\ P(x, a) > 0}} P(x, a). \tag{2.11}$$

That means in words for the instance I that the penalty of using one arc twice is bigger than all other (positive) penalties summed up for this arc. This is a natural condition if we think of why we introduced penalties. We wanted to prevent using arcs twice if possible or prevent using opposite arcs if possible, but the attractiveness should decrease more if we use an arc twice in one direction as if we use an arc and its opposite arc.

If we would allow equality in (2.11) that would mean that if we use $P_{\text{diff}}(a', a) = 0$ in the transformation that we would maybe generate a new solution where a' is used and a not, but this does not matter since in this case another solution would be to exchange

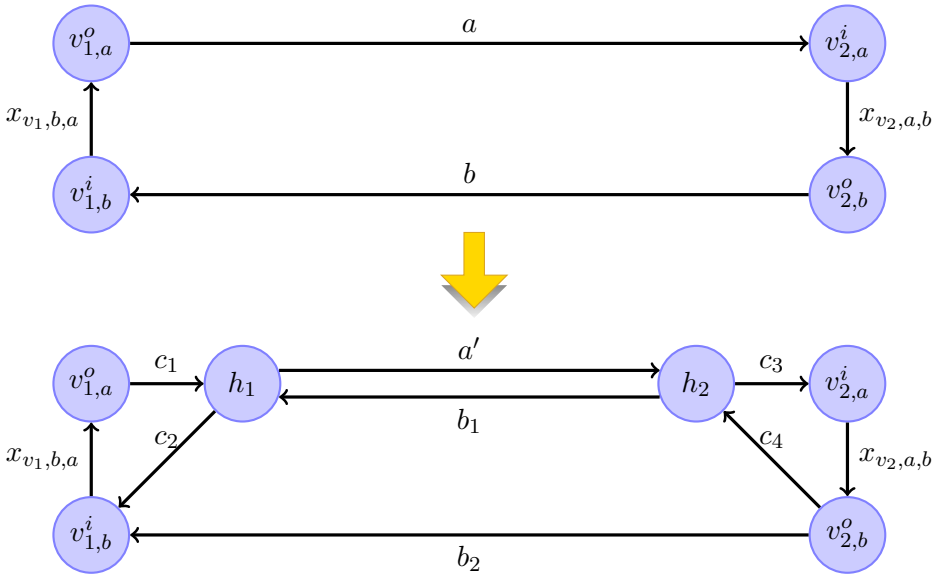

Figure 2.9: Transformation of Theorem 2.4.3 for a part of the graph G'.

a with a'. Therefore, we can also allow equality in (2.11). If (2.11) or equality holds for all $a \in A$ we can use $P_{\text{diff}}(a', a) = 0$ for all $a \in A$ and get that the dependencies got dropped without changing the penalties and therefore $P(a, a') = 0$ for all $a \in A$. That means we have no penalties between any copied arcs and therefore do not need to apply Theorem 2.4.2 to them.

The only situation where we would have to apply the theorem is if there are already in G parallel arcs with penalties. In this the conditions of Theorem 2.4.2 would have to be checked and therefore restrict the problem.

The next step is to remove penalties between opposite arcs.

Theorem 2.4.3. *Let $I = (G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P)$ with $G = (V, A, s, t)$ an instance of RTPP4 and $I' = (G', v'_{\text{start}}, v'_{\text{end}}, C'_{\min}, C'_{\max}, p', c', P')$ the instance of RTPP5 which we get after applying the transformation described in 2.4.1 to I. Let a, b be two opposite arcs in G, that means $s(a) = t(b)$ and $t(a) = s(b)$, with $P(a, b) > 0$ and no other penalties. That means $P(a, x) = P(b, y) = 0$ for all $x \neq b$ and $y \neq a$.*

Then we can transform I' into an instance I'' such that the number of nonnegative penalties in I'' is smaller than the number of nonnegative penalties in I'. Solutions from one instance can easily be transformed into solutions of the other.

Proof. Let $v_1 = s(a) = t(b)$ and $v_2 = t(a) = s(b)$, let further be G' the graph of I', then we consider the subgraph H' of G' consisting of the nodes $v^o_{1,a}$, $v^i_{1,b}$, $v^i_{2,a}$, $v^o_{2,a}$ and the arcs a, b, $x_{v_1,b,a}$, $x_{v_2,a,b}$. This part of the graph and the transformation we will be going to describe are illustrated in Figure 2.9.

We formally define G'' by $G'' = (V'', A'', s'', t'')$ where

$$V'' = V' \cup \{h_1, h_2\}$$

$$A'' = A' \setminus \{a, b\} \cup \{a', b_1, b_2, c_1, c_2, c_3, c_4\}$$

$$s''(x) = s'(x) \wedge t''(x) = t'(x) \quad \forall x \in A' \setminus \{a, b\}$$

$$s''(a') = h_1, s''(b_1) = h_2, s''(b_2) = v^o_{2,b}$$

$$s''(c_1) = v^o_{1,a}, s''(c_2) = h_1, s''(c_3) = h_2, s''(c_4) = v^o_{2,b}$$

$$t''(a') = h_2, t''(b_1) = h_1, t''(b_2) = v^i_{1,b}, t''(c_1) = h_1, t''(c_2) = v^i_{1,b}, t''(c_3) = v^i_{2,a}, t''(c_4) = h_2$$

We further define for some $\epsilon > 0$

$$c''(a') = c'(a), c''(b_1) = c''(b_2) = c'(b)$$

$$c''(c_i) = 0 \quad \forall 1 \leq i \leq 4 \quad \text{and} \quad c''(x) = c'(x) \quad \forall x \in A' \setminus \{a, b\}$$

$$p''(a') = p'(a) + 2\epsilon, p''(b_1) = p'(b) + 2\epsilon, p''(b_2) = p'(b) - P(a, b)$$

$$p''(c_i) = -\epsilon \quad \forall 1 \leq i \leq 4 \quad \text{and} \quad p''(x) = p'(x) \quad \forall x \in A' \setminus \{a, b\}$$

$$P''(x, y) = P'(x, y) \quad \forall x, y \in A' \setminus \{a, b\}$$

$$P''(x, y) = P''(y, x) = 0 \quad \forall x \in A'', y \in A'' \setminus (A' \setminus \{a, b\})$$

In the next step we will provide transformations of valid paths from the one instance to the other such that the costs stay the same and the profit stays the same or gets bigger. Let therefore be w a valid walk in I'. If w does not use any arcs in H' then w is also a valid walk in I'' and has the same costs and profit. If w uses *arcs* in H' it has to use either a or b since after $x_{v_1,b,a}$ only a can follow and after $x_{v_2,a,b}$ only b can follow. We have three cases.

1. w uses a but not b. We can now replace the arc a in w by the sequence c_1, a', c_3 and get a new path $w' \in I''$. It is easy to check that this path has the same costs and profit as w.

2. w uses b but not a. We can replace the arc b in w by the sequence c_4, b_1, c_2 and get a new path $w' \in I''$. Again, we get that $c''(w') = c'(w)$ and $p''(w') = c'(w)$.

3. w uses a and b. This is the interesting case since now the penalty takes effect. Now we define w' by replacing a through the sequence c_1, a', c_3 and b through b_2. It is again easy to see that $c''(w') = c'(w)$. For the total profit we note that the sequence c_1, a', c_3 together has the profit $c'(a) + 2\epsilon - 2\epsilon = c'(a)$ and the profit of b_2 equals the profit of b minus the penalty $P'(a, b)$. Therefore, we have again $p''(w') = p'(w)$.

Let no on the other hand be w' a valid path in I''. There are four cases to consider.

1. w' does not use a', b_1 or b_2. If w' uses one of the helping arcs c_i with $i \in \{1,..,4\}$ we can construct a new path with a higher profit. Assume for example w' uses c_1 or c_2 then the only possibility is that at some point w' has a subsequence $x_{v_1,\alpha,a}, c_1, c_2, x_{v_1,b,\beta}$ where $\alpha \in A$ is some arc with $t(\alpha) = v_1$ and $\beta \in A$ an arc with $s(\beta) = v_1$. We can replace this subsequence by $x_{v_1,\alpha,\beta}$ and get a new path w. Since all the changed arcs have costs 0 we do not change the costs and we get

$$\begin{aligned} p''(w') - p'(w) &= p''(x_{v_1,\alpha,a}) + p''(c_1) + p''(c_2) + p''(x_{v_1,b,\beta}) - p'(x_{v_1,\alpha,\beta}) \\ &= p''(c_1) + p''(c_2) = -2\epsilon. \end{aligned}$$

   Therefore $p'(w) > p''(w')$. This also means that w' was not optimal in I'' since w is a valid path in I' and in I'' with the same costs and profit.

2. w' uses a' but not b_2. Furthermore w' can also not use b_1 since it can visit the node h_2 only once. Therefore, w' must contain a subsequence c_1, a', c_3 which we can replace by a and get a new path w for the instance I'. It has again the same costs and also the same profit.

3. w' uses b_1 but not b_2. Furthermore w' can also not use a' since it is a path. Therefore, w' must contain a subsequence c_4, b_1, c_2 which we can replace by b and get a new path w for the instance I' which has the same costs and profit.

4. w' uses b_2.

   If w' does not use a' or b_1 we can do the same as in point 1 and get rid of usages of c_i for $i = 1, ..., 4$. After this we can replace b_2 by the sequence c_4, b_1, c_2. The new path has the same costs, but a better profit. Then we can apply point 2 and get a path in I' with a better profit than the original path.

   If on the other hand w' uses a' or b_1, we can first of all rule out the case that w' uses b_1 because this is impossible since it cannot use c_4 since w' is a path and therefore it cannot reach the node h_2 to use the arc b_1. Therefore, in this case w' uses a'. Then w' must have a subsequence c_1, a', c_3. We can replace now this subsequence by a and replace b_2 by b and get a new path w for the instance I'. By the same calculations as above, we get $c'(w) = c''(w')$ and $p'(w) = p''(w')$.

We provided now transformations to transform a valid path in I' into a valid path in I'' with the same costs and a greater or equal profit and a transformation to transform a valid path in I'' into a valid path in I' with the same costs and a greater or equal profit. This implies that if a path is optimal in I' this transformation constructs an optimal path in I'' and vice versa.

□

Remark 2.4.2. If we would use $\epsilon = 0$ instead of $\epsilon > 0$ in the proof of 2.4.3 we would generate additional solutions, because we allow routes of the form $..., c_1, c_2, ...$ This is

not intended since c_i are helper arcs to use the arcs a' or b_1, but it would not change the profit and therefore also the other optimal solutions are still optimal. That means we can choose $\epsilon = 0$ so that we do not have negative profit on the arcs c_i.

Theorem 2.4.4. *Let $I = (G, v_{\text{start}}, C_{\min}, C_{\max}, p, c, P)$ be an instance of RTPP1 with $G = (V, A, s, t)$. Let I further have the following properties:*

1. *The following holds for all $a \in A$:*

$$p(a) \geq P(a,a) \geq \sum_{\substack{b \in A \setminus \{a\} \\ P(a,b) > 0}} P(a,b)$$

2. *$P(a,b) \neq 0$ only if $a = b$ or a and b are parallel arcs with the same costs or a and b are opposite arcs*

3. *Every arc a_1 has at most one opposite arc a_2 with nonzero penalty. If such an arc a_2 exists, it holds*

$$\max\left(p(a_1) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a,a_1) > 0}} P(a,a_1), p(a_2) - \sum_{\substack{a \in A \setminus \{a_1, a_2\} \\ P(a,a_2) > 0}} P(a,a_2)\right) \geq P(a_1,a_2) \geq 0.$$

4. *If b_1, b_2 are parallel arcs either for $a_1 = b_1, a_2 = b_2$ or for $a_1 = b_2, a_2 = b_1$ it must hold the following:*

   - *If $P(a_1,a_2) > 0$ (2.9) holds.*
   - *If $P(a_1,a_2) < 0$ (2.10) holds.*
   - $p(a_2) \geq P(a_1,a_2)$

Then we can transform I into an instance I' of AOP, such that every solution of the instance I' can easily be transformed into a solution of I. The size $|I'|$ of I' is in $\mathcal{O}(k|I|)$ where k is the maximum of the indegrees and outdegrees of all nodes in G.

Proof. We will apply a series of transformations. Let $I_0 = I$, I_1 be the result of Theorem 2.3.1 applied to I_0, I_2 be the result of Theorem 2.3.4 applied to I_1, I_3 be the result of Theorem 2.3.9 applied to I_2 with $P_{\text{diff}}(a,a') = 0$ for all arcs a and its copy a', I_4 be the result of Theorem 2.4.2 applied possible multiple times to I_3 for all parallel arcs which have nonzero penalties, I_5 be the result of Theorem 2.4.1 applied to I_4 and I_6 be the result of Theorem 2.4.3 applied possible multiple times to I_5 for all opposite arcs which have nonzero penalties by using $\epsilon = 0$ as described in Remark 2.4.2.

We have to prove now that all these transformations are applicable and that I_6 at the end is an instance of AOP.

First of all it is clear that Theorem 2.3.1 is applicable to $I_0 = I$ and that the properties 1-4 also hold for I_1. It is also clear that Theorem 2.3.4 is applicable to I_1. The properties 2-3 also hold for I_2. Property 4 also holds for I_2 since $P(a, a') = 0$ for all arcs a and its copy a'. From property 1 we get two new properties holding for I_2:

5. $p(a) \geq 0$ for all $a \in A$
6. It holds for all arcs a and its copy a':

$$p(a) - p(a') \geq \sum_{\substack{b \in A \setminus \{a, a'\} \\ P(a,b) > 0}} P(a, b)$$

We want now to apply Theorem 2.3.9 to I_2 with $P_{\text{diff}}(a_1, a_2) = 0$. This is possible since property 6 holds for I_2. Therefore, Theorem 2.3.9 only removes the dependencies and does not change anything else. That means properties 2-5 also hold for I_3.

Because of property 1 parallel arcs can only have nonzero penalties if they have the same costs and therefore because of property 4 we can apply Theorem 2.4.2 to all parallel arcs with nonzero penalties and the new profits are still nonnegative. Therefore, property 5 also holds for I_4. The properties 2 and 3 together get transformed into the following new property which holds for I_4.

7. For every arc a_1 there exist at most one other arc a_2 such that $P(a_1, a_2) \neq 0$. If such an a_2 exists, it is an opposite arc of a_1 and it holds $\max(p(a_1), p(a_2)) \geq P(a_1, a_2) \geq 0$.

We can apply now Theorem 2.4.1 to I_4 and it is clear that the property 5 also holds for I_5.

Now we apply Theorem 2.4.3 with $I = I_4$ and $I' = I_5$ for every pair of opposite arcs with nonzero penalties and with $\epsilon = 0$ as in Remark 2.4.2. This is possible since property 7 holds for I_4. Since we chose $\epsilon = 0$ and property 7 holds for I_4 and property 5 holds for I_5 we get that property 5 also holds for I_6 and that I_6 does not have any penalties. The only difference of I_6 and an instance of AOP is now that I_6 has a minimal tour cost of $C_{\min}$. But we can solve that easily since if the solution of the AOP gives us a solution with tour costs under $C_{\min}$ we know that I_6 has no valid solution.

□

Remark 2.4.3. Theorem 2.4.4 has four conditions which look very restricting but if we have a closer look, we see that most of them are only needed since AOP has only positive arcs. Without that we could ignore the restrictions 1. and 3. of Theorem 2.4.4. The restrictions 2. and 4. are very natural since we introduced penalties in Chapter 2 to reduce the attractiveness if we use an arc in the same or in the opposite direction twice. Therefore, 2. is not really a restriction. 4. is only a restriction for parallel arcs in the original graph with penalties, which was also not the basic idea of penalties.

CHAPTER 3

Complexity

In this chapter, we will discuss the theoretical complexity of the RTPP. Since all the transformations described in Section 2.3 are polynomial time reductions it does not matter which of the problem formulations RTPP1, RTPP2, RTPP3 or RTPP4 we analyze, since all of them will be in the same complexity class. We will therefore analyze our original problem formulation RTPP1.

Since RTPP1 is an optimization problem we will need complexity theory for optimization problems.

3.1 Complexity Theory for Optimization Problems

In this section we present some fundamental definitions and theorems. We will use the notation of the book [7] and we will also omit proofs of the stated theorems, the interested reader can read the proofs in [7].

First of all we will present a formal definition of an optimization problem.

Definition 3.1.1. An optimization problem $\mathcal{P}$ is characterized by the following quadruple of objects $(I_{\mathcal{P}}, \mathrm{SOL}_{\mathcal{P}}, m_{\mathcal{P}}, \mathrm{goal}_{\mathcal{P}})$, where:

1. $I_{\mathcal{P}}$ is the set of instances of $\mathcal{P}$;
2. $\mathrm{SOL}_{\mathcal{P}}$ is a function that associates to any input instance $x \in I_{\mathcal{P}}$ the set of feasible solutions of x;
3. $m_{\mathcal{P}}$ is the measure function, defined for pairs (x, y) such that $x \in I_{\mathcal{P}}$ and $y \in \mathrm{SOL}_{\mathcal{P}}(x)$. For every such pair (x, y), $m_{\mathcal{P}}(x, y)$ provides a positive integer which is the value of the feasible solution y;
4. $\mathrm{goal}_{\mathcal{P}} \in \{\mathrm{MIN}, \mathrm{MAX}\}$ specifies whether $\mathcal{P}$ is a maximization or a minimization problem.

Every optimization problem has an associated decision problem which is defined as follows.

Definition 3.1.2. Let $\mathcal{P}$ be an optimization problem. Then we define the decision problem $\mathcal{P}_D$ as follows. For an instance $x \in I_{\mathcal{P}}$ and a positive integer $K \in \mathbb{Z}^+$, decide whether there exists a feasible solution $y \in \mathrm{SOL}_{\mathcal{P}}(x)$ with $m_{\mathcal{P}}(x, y) \geq K$ in the case of $\mathrm{goal}_{\mathcal{P}} = \mathrm{MAX}$ or $m_{\mathcal{P}}(x, y) \leq K$ in the case of $\mathrm{goal}_{\mathcal{P}} = \mathrm{MIN}$.

In the next step we will define a class of optimization problems called NPO, which is strongly connected to the class NP.

Definition 3.1.3. NPO is the class of all optimization problems $\mathcal{P} = (I, S, m, goal)$ such that

1. $I \in \mathsf{P}$, i.e. we can decide in polynomial time whether a given x is a valid instance,
2. there is a polynomial p such that for all $x \in I$ and $y \in S(x)$, $|y| \leq p(|x|)$, and for all y with $|y| \leq p(|x|)$, we can decide $y \in S(x)$ in time polynomial in $|x|$.
3. m is computable in polynomial time.

The connection between the class NPO and the class NP can be seen in the following two theorems.

Theorem 3.1.1. *Let $\mathcal{P} \in \mathsf{NPO}$, then $\mathcal{P}_D \in \mathsf{NP}$.*

Theorem 3.1.2. *Let $\mathcal{P} \in \mathsf{NPO}$ such that $\mathcal{P}_D$ is NP-complete. Then $\mathcal{P} \leq_P^T \mathcal{P}_D$.*

3.2 Complexity of RTPP1

Our goal is to apply Theorem 3.1.1 and 3.1.2. For that we have to formalize the problem in terms of Definition 3.1.1. It is important to mention that since we speak of our problem instances as finite objects we can assume that all numeric values appearing in it are in $\mathbb{Q}$.

We define I as the set of all instances $(G, v_{\mathrm{start}}, C_{\mathrm{min}}, C_{\mathrm{max}}, p, c, P)$ of RTPP1 with a finite graph G, $C_{\mathrm{min}} \in \mathbb{Q}$, $C_{\mathrm{max}} \in \mathbb{Q}$, $p(a) \in \mathbb{Q}$, $c(a) \in \mathbb{Q} \cap [0, \infty)$, $P(a, b) \in \mathbb{Q}$ for all $a, b \in A$ where A is the set of arcs of the graph G.

We further define $\mathrm{SOL}(x)$ for some instance $x = (G, v_{\mathrm{start}}, C_{\mathrm{min}}, C_{\mathrm{max}}, p, c, P)$ as the set of all walks in G satisfying (2.4) - (2.3).

Since we defined the profit of a walk y depending on the instance x we will write $p^x(y)$ instead of $p(y)$ to notate the dependency to the instance x. The next step is to define $m(x, y)$ for some instance x and valid walk y, but we have to be careful since $m(x, y)$ has to be a positive integer and we therefore cannot use $p^x(y)$. Let $x = (G, v_{\mathrm{start}}, C_{\mathrm{min}}, C_{\mathrm{max}}, p, c, P)$ with $G = (V, A, s, t)$. We can use that the property of being maximal is invariant under translation and multiplication with positive factors. Since G is finite also A is finite and therefore also the number of profit values and penalties is finite. From this we get that there has to exist a positive integer $N^x \in \mathbb{N} \setminus \{0\}$ such

that $N^x \cdot p(a) \in \mathbb{Z}$ and $N^x \cdot P(a,b) \in \mathbb{Z}$ for all $a, b \in A$. If we assume that all numeric values in our problem have a finite binary representation, we can define $N^x = 2^{n_x}$ where n_x is the largest number of decimal places of all binary representations of $p(a)$ and $P(a,b)$ for $a, b \in A$. Then we get $p(a) \cdot 2^{n_x} \in \mathbb{Z}$ and $P(a,b) \cdot 2^{n_x} \in \mathbb{Z}$ for all $a, b \in A$.

We define now $m(x,y) = (p^x(y) + P_{\max}(p,P)) \cdot 2^{n_x}$ for an arbitrary instance $x = (G, v_{\text{start}}, C_{\min}, C_{\max}, p, c, P)$ and a walk $y \in \text{SOL}(x)$. We know that $p^x(y) > -P_{\max}(p,P)$ and therefore we get $m(x,y) > 0$. Since we have a maximization problem we define goal = MAX.

Using the definitions from above we can define the optimization problem $\mathcal{Q} = (I, \text{SOL}, m, \text{goal})$. In the next step we want to prove that this optimization problem is in NPO.

Theorem 3.2.1. $\mathcal{Q} \in \text{NPO}$.

Proof. We have to prove the three conditions for being in NPO.

1. Checking if $x = (G, v_{\text{start}}, C_{\min}, C_{\max}, p, c, P)$ is an instance can be done in polynomial time. We only have to check if G is a directed multigraph, v_{start} is a node of G, $C_{\min} \in \mathbb{Q}$, $C_{\max} \in \mathbb{Q}$, p is a function from A to $\mathbb{Q}$ where A is the set of arcs of G, c is a function from A to $\mathbb{Q} \cap [0, \infty)$, P is a symmetric function from $A \times A$ to $\mathbb{Q}$. All these steps can be done in linear time on the size of x.

2. Every valid walk $y \in S(x)$ contains at most $2|A|$ arcs where A is the set of arcs of the graph G of the instance x. Therefore, we get $|y| \leq c|A| \leq c|x|$ where c is a constant depending on how we store a walk and how we measure the size of it. Therefore, we can use the polynomial $p(n) = c \cdot n$. For a $|y|$ with $|y| \leq c \cdot |x|$ it is also easy to verify if it is a valid walk which satisfies the needed conditions for being in $S(x)$ in time polynomial in $|x|$.

3. For a fixed x and y we can compute n_x and $P_{\max}(p,P)$ in time polynomial in $|x|$ and therefore we can compute $m(x,y) = p^x(y) \cdot 2^{n_x} + P_{\max}(p,P)$ in time polynomial in $|x|$ and $|y|$.

□

From the above, we get by using Theorem 3.1.1 that $\mathcal{Q}_D \in \text{NP}$. It is clear that $\mathcal{Q}_D$ has the same complexity than the new problem when we replace the measure $m(x,y) = (p^x(y) + P_{\max}(p,P)) \cdot 2^{n_x}$ by $m(x,y) = p^x(y)$. Since the measure of each walk y only gets transformed and the transformation is fixed for a fixed x and is only translation and multiplication with a positive factor, the set of walks with the maximal measures is in both cases the same. Therefore, we also get that the decision problem $\mathcal{Q}'_D$ which we get by replacing the measure in $\mathcal{Q}_D$ through $p^x(y)$ is in NP. The next step will be to prove that $\mathcal{Q}'_D$ is NP-complete. Since we already proved that it is in NP it is sufficient to prove that it is NP-hard. For this we use the fact that the decision problem of the famous traveling salesman problem is NP-hard. We first formulate the traveling salesman problem as described in [18].

Definition 3.2.1 (Hamiltonian Tour)**.** Let G be an undirected graph with $n = |V|$ nodes. A *Hamiltonian tour* in G is a closed walk $w = (v_0, ..., v_n)$ which visits each node exactly once and comes back to the start node, that means $v_0 = v_n$ and $v_i \neq v_j$ for all $0 < i \leq j$.

Definition 3.2.2 (K_n)**.** For a natural number n the complex undirected graph K_n is defined as an undirected graph (V, E) with $|V| = n$ and all possible edges $E = \{uv | u, v \in V, u \neq v\}$.

Problem (TSP_D)**.** Given the complete undirected graph K_n with edge weights c_{uv} and a number b decide if there exists a Hamiltonian tour in K_n with length at most b.

Theorem 3.2.2. *The problem TSP_D is* NP*-complete.*

The next step will be to find a polynomial time reduction from TSP_D to $\mathcal{Q}'_D$ which will then imply that $\mathcal{Q}'_D$ is NP-hard.

Theorem 3.2.3. *There is a polynomial time reduction from TSP_D to $\mathcal{Q}'_D$.*

Proof. Let $n \in \mathbb{N}$ and c_{uv} the weights of the edges of the complex undirected graph $K_n = (V, E)$ and b a number. We will use the undirected version of K_n as our graph G that is $G = (V, A, s, t)$ where

$$A = \{(u, v) | u, v \in V, u \neq v\}$$

$$s((u, v)) = u \quad t((u, v)) = v \quad \forall (u, v) \in A$$

Important is here the difference of E and A. In E we used the notation uv for an undirected edge which is a short form for $\{u, v\}$ and therefore is the same as $vu = \{v, u\}$. But $(u, v) \neq (v, u)$. Therefore, A has twice as many arcs as E has edges. Furthermore, we define $c((u, v))) = 1$ for all arcs $(u, v) \in A$ and $C_{\min} = C_{\max} = n$. That means we only search walks, containing n arcs. Since TSP_D is a decision problem corresponding to a minimization problem and $\mathcal{Q}_D$ is a decision problem corresponding to a maximization problem we have to inverse the profit and therefore we define

$$p((u, v)) = -c_{uv} \quad \forall (u, v) \in A$$

Let P_0 be the penalty function with no penalties, that means $P_0((u, v), (u', v')) = 0$ for all $(u, v), (u', v') \in A$. We want to forbid that one node is used more than once and therefore we use the large value $C := b + P_{\max}(p, P_0)$. We define now

$$P((u, v), (u, v')) = C \quad \forall u, v, v' \in V$$

$$P((u, v), (u', v')) = 0 \quad \forall u, v, u', v' \in V, u \neq u'$$

In the first case we can also have $v = v'$ and therefore we have $P((u, v), (u, v)) = C$ this will be used that no arc is used twice. Since a Hamiltonian path always uses every

node of the graph we can set any node as start and end node, let therefore be $v_{\text{start}} \in V$ an arbitrary node in V.

We have now generated an instance $x = (G, v_{\text{start}}, C_{\min}, C_{\max}, p, c, P)$ together with the value $-b$ for the problem $\mathcal{Q}'_D$.

We have two cases, let first be the answer of this instance yes, that means that there exists a valid walk y in G such that $p^x(y) \geq -b$. Assume y would visit a node $v \in V$ twice, that means this walk has to leave this node also twice and therefore we have a w_1 and a w_2 such that either $w_1 \neq w_2$ and $u^y((v, w_1)) \geq 1$ and $u^y((v, w_2)) \geq 1$ or $w_1 = w_2$ and $u^y((v, w_1)) = 2$. We prove now that in both cases $p^x(y) < -b$ which is a contradiction.

In the first case let u be the usage vector of y without (v, w_1), that means $u((v', w')) = u^y((v', w'))$ for all $(v', w') \in A \setminus \{(v, w_1)\}$ and $u((v, w_1)) = 0$. Then we get

$$p^x(y) = p(u^y) \leq p(u) + p(v, w_1) - C < \frac{P_{\max}(p, P_0)}{2} + p(v, w_1) - b - P_{\max}(p, P_0) \leq -b$$

In the second case let u be the usage vector of y which is using (v, w_1) only once, that means $u((v', w')) = u^y((v', w'))$ for all $(v', w') \in A \setminus \{(v, w_1)\}$ and $u((v, w_1)) = 1$. Then we get again exactly the same calculation as above and therefore again $p^x(y) < -b$.

Therefore we know that every valid walk in G with $p^x(y) \geq -b$ visits each node at most once. But since $c(y) = n$ we get that the walk has length n and therefore visits exactly n nodes. Therefore, we can use $y = ((u_0, v_0), ..., (u_n, v_n))$ to define a Hamiltonian tour y' in the original graph K_n by $y' = (u_0 v_0, ..., u_n, v_n)$. And since no penalties apply in this case we get by definition that $p(y)$ equals exactly the -1 times the length of y' and therefore we get that the length of y' is lower or equal than b and therefore that the answer of the original instance is also yes, there exists a Hamiltonian path in K_n with length lower or equal to b.

Let now be the answer of the generated instance x together with $-b$ of the problem $\mathcal{Q}'_D$ no, that means that there exists no valid walk y with $p^x(y) \geq -b$. We want to prove that then also no Hamiltonian tour exists in K_n with length lower or equal to b. Assume there would exist a Hamiltonian tour y in K_n with length lower or equal b, then we can transform this tour in a walk y' in G which is a valid walk. But since there are again no penalties for this walk we get that $p(y')$ is -1 times the length of y and therefore $p(y') \geq -b$ which is a contradiction. Therefore, also the answer of the original instance is no.

We just provided a polynomial transformation of an instance of TSP_D to an instance of $\mathcal{Q}'_D$ such that the two instances have the same solutions, that means in this case either the answer of both instances is yes or the answer of both instances is no. Therefore, we just provided polynomial-time many-one reduction from TSP_D to $\mathcal{Q}'_D$.

□

Corollary 3.2.4. *$\mathcal{Q}'_D$ is NP-complete and therefore also $\mathcal{Q}_D$ is NP-complete.*

Now we can apply 3.1.2 and get the following corollary.

Corollary 3.2.5. *$\mathcal{Q} \leq_P^T \mathcal{Q}_D$ and therefore also $RTPP1 \leq_P^T \mathcal{Q}_D$.*

Proof. The first statement follows from 3.1.2 and the second statement follows from the fact, that $\mathcal{Q}$ is almost the same as RTPP1 with the only difference, that the profit function is transformed, which does not change anything for the optimization problem. □

We now define the complexity class NP-equivalent, which also contains function problems and not only decision problems as introduced in [14].

Definition 3.2.3 (NP-hard)**.** A problem $\mathcal{P}$ is called NP-*hard* if there exist an NP-complete problem $\mathcal{P}'$ such that $\mathcal{P}' \leq_P^T \mathcal{P}$.

Definition 3.2.4 (NP-easy)**.** A problem $\mathcal{P}$ is called NP-*easy* if there exists a problem $\mathcal{P}' \in \mathsf{NP}$ such that $\mathcal{P} \leq_P^T \mathcal{P}'$.

Definition 3.2.5 (NP-equivalent)**.** A problem $\mathcal{P}$ is called NP-*equivalent* if it is NP-hard and NP-easy.

Corollary 3.2.6. *The problem RTPP1 is* NP-*equivalent.*

Proof. We already proved that $\mathcal{Q}'_D$ is NP-complete and that $\mathcal{Q}'_D \leq_P^T RTPP1 \leq_P^T \mathcal{Q}'_D$, where the first reduction is trivial. Therefore, we get by definition that RTPP1 is NP-hard and NP-easy. □

CHAPTER 4

Related Work

In this chapter, we will discuss related problems to RTPP. We will see that there are many optimization problems similar to ours, but none of them has a concept of penalties like we use it.

Since our problem is to find a route we will first start with one of the best known optimization problems, the traveling salesman problem, which we already introduced in Chapter 3.

4.1 The Traveling Salesman Problem

The traveling salesman problem is maybe the most popular optimization problem and therefore a lot of researchers investigated it. Its goal is to find an optimal route visiting all given cities. For a formal definition of the problem see problem formulation TSP_D in Chapter 3, which is the corresponding decision problem of the original optimization problem.

As stated in [18] its history starts in the 19th century and since then the approaches to solve it were improved constantly. In 1954 they solved a problem with 48 cities and in 1991 they already solved a problem with 4461 cities. In [3] a modern branch-and-cut algorithm for solving the traveling salesman problem is described. The algorithm solved a problem with 85,900 cities. For much larger problems the interesting question is how good the solution can be approximated. In [17] different approximation algorithms are compared and the best of them could solve an instance with 3 million cities within an optimality range of 2%.

4.2 Traveling Salesman Problems with Profits

In [10] a class of problems called traveling salesman problems with profits is defined. All problems in this class search a route, where each node is visited at most once, which is

optimal concerning profit of nodes and costs of the route. The main difference of these problems to our problem is that every node is only visited at most once.

The solution approaches for problems in this class are often adapted solution approaches from the traveling salesman problem. On the side of exact methods there are different branch-and-bound and branch-and-cut solution procedures (see for example [11]). In [10] four main operations, which are used by heuristic approaches to transform routes and improve their quality, are listed:

- Adding a vertex to the route to improve the profit.
- Deleting a vertex from the route to improve the costs.
- Resequencing the route to improve the costs while having the same profit. This only works if the profits are associated with nodes and not with arcs.
- Replacing a vertex as a mix of the first two transformations.

There were also many meta heuristics applied to traveling salesman problems with profits.

4.2.1 The Orienteering Problem

One more specific problem in the class of traveling salesman problems with profits is the orienteering problem. It was introduced in [23] where it was called score orienteering event (S.O.E.) instead of the more commonly used name, the orienteering problem (OP). Its goal is to find a path from a fixed start point to a fixed end point with maximal score such that the traveling time is smaller than some upper bound $T_{\max}$ where the score of a path is the sum of the scores of the visited nodes.

The main differences to our problem are that scores are assigned to nodes and not to arcs and that it only allows paths, that means walks, where every node is only visited once. The scores in OP are required to be nonnegative, but there are many extensions of OP and some of them allow also negative scores.

The OP and many of its extensions are analyzed in [24]. As already mentioned in Section 4.2 there are a wide variety of solution approaches for the OP.

Some approaches to solve the OP exactly as mentioned in [24] are branch-and-bound approaches and a branch-and-bound algorithm using a cutting plane method to improve upper bounds. In [11] a branch-and-cut algorithm is presented which we will use for one of our solution approaches described in Chapter 5.

There are many different heuristic approaches for the OP. As [24] gives a rather good list of heuristic approaches, we refer the interested reader to that survey.

One interesting variant of the orienteering problem is the arc orienteering problem where the profits are on the arcs instead of the nodes, which is already very similar to our problem, is described in Section 4.3.3.

4.3 Arc routing problems with profits

In contrast to the traveling salesman problems with profits arc routing problems with profits assign profits to arcs and not to nodes. There are many different arc routing problems with profits and [4] gives a thorough overview of them.

Four interesting problems in this class are the maximum benefit chinese postman problem, the prize collecting rural postman problem, the arc orienteering problem and the cycle trip planning problem.

4.3.1 Maximum Benefit Chinese Postman Problem

The maximum benefit chinese postman problem, or short MBCPP, has many similarities with our problem. It was introduced in [13] and is defined on a directed graph. Every arc can be used multiple times and for every time the profit and the costs can be different. There is also a limit for every arc from which on the arc does not give any profit. In contrast to our problem the goal of the MBCPP is to maximize the difference between profit and costs of the tour. Another difference is that the MBCPP has no maximal tour length and minimal tour length. This makes it possible to solve the problem with a minimum cost flow algorithm with subtour elimination as proposed in [13]. As another exact algorithm a branch-and-cut algorithm was proposed in [9], which can solve instances up to 1000 vertices and 3000 edges within one hour. There were also heuristics and approximation algorithms proposed as in [15] and [16].

4.3.2 The Prize Collecting Rural Postman Problem

The prize collecting rural postman problem, PCRPP, is also called privatized rural postman problem and was introduced in [6]. It is a special case of the MBCPP and is very similar to our problem. The PCRPP is defined on a undirected graph and is restricted such that every edge only gives profit once. This implies that an optimal route uses each edge at most twice. In [5] a branch-and-cut algorithm is presented to solve the PCRPP which is based on a formulation introduced in [6].

4.3.3 The Arc Orienteering Problem

The arc orienteering problem, short AOP, was introduced in [22] and we already defined it in Chapter 2. The main difference to our formulation is that every node can be visited only once and therefore we have no penalties or something similar. Another difference is that the profits are restricted to be nonnegative. In Theorem 2.4.4 we proved that under some restrictions an instance of our problem can be transformed into an instance of the AOP.

The problem was formulated in [22] also in the context of cycle trip planning. The goal was to design a web-based tool to plan cycle trip tours in the region of East Flanders. They designed a greedy randomized adaptive search procedure (GRASP) to solve the problem. Their procedure is based on routes which can consist of multiple subtours

and the interpretation of such a route is the path after connecting the unconnected parts of the tour in a cheapest possible way. The neighborhood used for the GRASP is constructed by inserting a new arc into one of the subtours of a route.

The numerical experiments show that the GRASP algorithm compared to a simple MIP model solved by CPLEX constructs the best solution for small instances much faster and constructs a good approximations for big instances, where the CPLEX method fails to solve the instance. The algorithm could solve all but one instance with a tour length of 20 kilometers and some instances with a tour length of 40 kilometers optimally. For the other instances up to a tour length of 100 kilometers the algorithm found good approximations with gaps around 0.04%.

We will apply our algorithm to the benchmark instance used for testing their GRASP algorithm and compare the results in Chapter 7.

4.3.4 The Cycle Trip Planning Problem

The CTPP was introduced in [25]. The CTPP can be formalized as a variant of the AOP with the following differences:

- A node can be visited more than once.
- Every arc can only be used once.
- If an arc has an opposite arc, it is not allowed to use both of them.
- There is not only one start node, but rather a set of start nodes from which at least one has to be visited. The vertex which is used as start vertex cannot be used during the route, that means it is only allowed to visit the start vertex once.

If we ignore the situation with more than one start node and the fact that the start node can only be used once we have almost the same situation as in our problem, but with the restriction that we can only use every arc once and that we have no penalties. Therefore, this problem is much closer to our problem formulation than the AOP, but it is still easier to transform our problem into an instance of the AOP as we did in Theorem 2.4.4 than to transform it into an instance of CTPP. The reason for this is that the fact that every node can only be visited once can be used to simulate penalties as described in Figure 2.9.

In [25] they presented a branch-and-cut procedure to exactly solve the problem. For big instances, the branch-and-cut procedure does not terminate in an acceptable time and therefore they described an iterated local search (ILS) procedure. The ILS procedure is based on an insert move sub procedure which inserts between two points a path such that the gained profit is maximal with the restriction that the new tour still has to be valid. This sub procedure itself is again a smaller variant of the CTPP.

The computational results presented in [25] show for the benchmark instances of the AOP from [22] that the ILS finds the best tour in most situations. In their tests the ILS found the best tour whenever the exact solution was known, that means whenever the

branch-and-cut procedure terminated. They also show that the ILS always finds better or the same solution than the GRASP algorithm presented in [22]. Since the ILS is a solver for CTPP and not AOP it can happen that it finds a tour where a node is visited more than once, but this happened only in a few cases.

There are also new benchmark instances created for the CTPP in [25]. We will apply our algorithm to these benchmark instances and to the original AOP benchmark instances and compare the results in Chapter 7.

CHAPTER 5

Mixed Integer Programming Approach for RTPP3

In this chapter we will present an exact approach based on mixed integer programming to solve our problem RTPP3, i.e., the variant where arcs can only be used once and the start node and the end node are different. Our approach is based on a mixed integer formulation and to eliminate subtours we will present three different possibilities. Before defining the MIP models we propose a preprocess phase that may reduce instances considerably.

5.1 Preprocessing

We already saw in Chapter 3 that our problem is NP-hard and therefore the running time of every algorithm which exactly solves the problem will heavily depend on the size of the input graph. Therefore, we try in a preprocessing phase to shrink the graph as much as possible. That means we want to remove all nodes and arcs where we know that they will never be included in a valid tour.

Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3 and let $v \neq v_{\text{start}}$ be a node which is included in a tour $w = (w_1, ...w_k)$ that means there exists a $j \in \{1, ..., k\}$ such that $t(w_j) = v$. Then the walk $w^1 := (w_1, ..., w_j)$ is a walk from v_{start} to v and the walk $w^2 := (w_{j+1}, ..., w_k)$ is a walk from v to w_{end} and we have $c(w) = c(w^1) + c(w^2)$. Let $c_1(v)$ be the costs of the shortest path from v_{start} to v and $c_2(v)$ the costs of the shortest path from v to v_{end}. Then we get $c(w) = c(w^1) + c(w^2) \geq c_1(v) + c_2(v)$. That means we can delete all nodes where $c_1(v) + c_2(v) > C_{\max}$. We cannot remove more nodes since then we would remove the valid tour consisting of the shortest path from the start node to the node and the shortest path from the node to the end node.

Obviously, we have to remove all arcs whose source node or target node got deleted. But we can remove even more arcs. Consider a walk $w = (w_1, ..., w_k)$ containing an arc w_j

for some $1 \leq j \leq k$. Then we can similarly to above define the walk $w^1 := (w_1, ..., w_{j-1}$ and the walk $w^2 := (w_{j+1}, ..., w_k)$. We get

$$C_{\max} \geq c(w) = c(w^1) + c(w_j) + c(w^2) \leq c_1(s(w_j)) + c(w_j) + c_2(t(w_j))$$

and therefore we can additionally remove all arcs where the costs of the arc plus the cost of the shortest path from the start node to the source node of the arc plus the cost of the shortest path from the target node of the arc to the end node is bigger than $C_{\max}$. Again we can not do better since otherwise we would remove the tour consisting of the two shortest paths and the given arc.

Another task for the preprocessing is the following. As already mentioned in Chapter 1 and in Section 2.1, one benefit of well planned bicycle routes for people with special needs is that they can specify that the route should not lead too far away from the start point, so that it is possible to get back quickly at any time. Therefore a maximal distance will be specified, which was not part of our problem formulation since it only concerns the preprocessing. The preprocessing therefore has to remove all nodes from the graph, from where the shortest path back to the start node is longer than the given maximal distance.

Algorithm 5.1 is a pseudo code realizing ideas discussed above. As we can easily check with the observations from above it removes only nodes and arcs which will never be part of a valid route and on the other hand, for every remaining node or arc there exists a valid route which contains it. The running time of Algorithm 5.1 is in $\mathcal{O}(|V|^2)$, since the running time of the Dijkstra algorithm is in $\mathcal{O}(|V|^2)$.

5.2 From a Usage Vector to a Walk

Since the mixed integer formulation is based on usage functions and not on paths we will first analyze how to restrict usage functions such that they correspond to a valid walk and present an algorithm to generate the walk from the usage vector.

First of all, we introduce a new notation which will help us to formulate the statements easier.

Definition 5.2.1. Let $G = (V, A, s, t)$ be a directed multigraph as in Definition 2.2.1. Let $S \subseteq V$ be a set of nodes then we define the arc sets

$$\delta^{in}(S) := \{a \in A : s(a) \notin S, t(a) \in S\},$$

$$E(S) := \{a \in A : s(a) \in S, t(a) \in S\}.$$

Let further be $v \in V$ then we define the arc sets

$$A^{in}(v) = \{a \in A : t(a) = v\},$$

$$A^{out}(v) = \{a \in A : s(a) = v\}.$$

Algorithm 5.1: General Preprocessing

INPUT: instance $(G = (V, A, s, t), v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ of RTPP3 and optionally a maximal distance $d_{\max} \in [0, \infty]$.
OUTPUT: A subgraph $G' = (V', A', s', t')$ of G.
if $d_{\max}$ is not set **then**
  Set $d_{\max} := \infty$
end if
Apply Dijkstra with the source node v_{start} to get the costs $c_1(v)$ of the shortest path from v_{start} to v for all $v \in V$.
Apply Dijkstra on the reversed arcs with the target node v_{end} to get the costs $c_2(v)$ of the shortest path from v to v_{end} for all $v \in V$.
Set $V' := \emptyset$, $A' := \emptyset$
for all $v \in V$ **do**
  if $c_1(v) + c_2(v) \leq C_{\max}$ **and** $c_2(v) \leq d_{\max}$ **then**
    Add v to V'
  end if
end for
for all $a \in A$ **do**
  if $s(a) \in V'$, $t(a) \in V'$ and $c_1(s(a)) + c(a) + c_2(t(a)) \leq C_{\max}$ **then**
    Add a to A', set $s'(a) := s(a)$, $t'(a) := t(a)$
  end if
end for
return $G' = (V', A', s', t')$

Theorem 5.2.1. *Let $G = (V, A, s, t)$ be a directed multigraph and $w = (w_1, ..., w_k)$ a walk in G as in Definition 2.2.2 with $s(w_1) \neq t(w_k)$. Let $s = s(w_1)$ be the start node of the walk and $t = t(w_k)$ the end node of the walk. Then, we get the following identities for the usage function u^w.*

$$\sum_{a \in A^{in}(v)} u^w(a) = \sum_{a \in A^{out}(v)} u^w(a), \quad \forall v \in V \setminus \{s, t\} \tag{5.1}$$

$$\sum_{a \in A^{out}(s)} u^w(a) = 1 + \sum_{a \in A^{in}(s)} u^w(a) \tag{5.2}$$

$$\sum_{a \in A^{in}(t)} u^w(a) = 1 + \sum_{a \in A^{out}(t)} u^w(a) \tag{5.3}$$

$$\sum_{a \in \delta^{in}(S)} u^w(a) \geq \min(1, u^w(b)) \quad \forall S \subseteq V \setminus \{s\},\ b \in E(S) \tag{5.4}$$

Proof. By definition of a walk we have $t(w_i) = s(w_{i+1})$ for all $1 \leq i \leq k-1$. Therefore, we get for all $v \in V \setminus \{t\}$

$$\sum_{a \in A^{in}(v)} u^w(a) = |\{i \in \{1, ..., k\} : t(w_i) = v\}| = |\{i \in \{1, ..., k-1\} : s(w_{i+1}) = v\}|.$$

For $v \in V \setminus \{s, t\}$ we further have $s(w_1) \neq v$ and therefore

$$\sum_{a \in A^{in}(v)} u^w(a) = \{i \in \{1, ..., k\} : s(w_i) = v\}| = \sum_{a \in A^{out}(v)} u^w(a).$$

For $v = s$ we get $s(w_1) = v = s$ and therefore

$$\sum_{a \in A^{out}(s)} u^w(a) = 1 + |\{i \in \{1, ..., k-1\} : s(w_{i+1}) = v\}| = 1 + \sum_{a \in A^{in}(v)} u^w(a).$$

The situation for the node t is similar. Since $s \neq t$ we get $s(w_1) \neq t$ and therefore we have

$$\begin{aligned}\sum_{a \in A^{in}(t)} u^w(a) =& |\{i \in \{1, ..., k\} : t(w_i) = t\}| = 1 + |\{i \in \{1, ..., k-1\} : s(w_{i+1}) = t\}| \\ =& 1 + |\{i \in \{1, ..., k\} : s(w_i) = t\}| = 1 + \sum_{a \in A^{out}(t)} u^w(a).\end{aligned}$$

Let $S \subseteq V \setminus \{s\}$ be an arbitrary set of nodes without the start node and let $b \in E(S)$ be an arc with $s(b) \in S$ and $t(b) \in S$.

If $u^w(b) = 0$ that means w does not use b the inequality (5.4) is trivially satisfied.

Let $u^w(b) \geq 1$. That means there exists at least one $i_0 \in \{1, ..., k\}$ such that $w_{i_0} = b$. Let furthermore be

$$i_1 := \min\{i \in \{1, ..., k\} : t(w_i) \in S\}.$$

This minimum exists since $t(w_{i_0}) = t(b) \in S$ and therefore we get $i_1 \leq i_0$.

If $i_1 > 1$ we have $s(w_{i_1}) = t(w_{i_1-1}) \notin S$. If $i_1 = 1$ we have $s(w_{i_1}) = s(w_1) = s \notin S$. That means we always have $s(w_{i_1}) \notin S$. But that implies $w_{i_1} \in \delta^{in}(S)$ and since $u^w(w_{i_1}) \geq 1$ we get

$$\sum_{a \in \delta^{in}(S)} u^w(a) \geq u^w(w_{i_1}) \geq 1 \geq \min(1, u^w(b)).$$

□

We will see that the conditions (5.1) - (5.4) are enough to characterize that a usage function corresponds to one or more walks. We will provide an algorithm which constructs a walk given a usage function with these three properties, such that the usage function of the walk equals the given usage function. The algorithm is based on a subprocedure which extracts a valid walk from some point v_0 to some point v_1 and returns the walk and a modified usage function. Algorithm 5.2 is a pseudo code for this subprocedure.

Algorithm 5.3 constructs a walk given a usage function and a start and an end node such that the conditions (5.1) - (5.4) are satisfied. Its running time is in $\mathcal{O}(|A|^2)$.

Algorithm 5.2: Extract(G,u,v_0,v_1)

INPUT: graph $G = (V, A, s, t)$, usage function u, start node v_0, target node v_1
REQUIRES: The usage function u satisfies the conditions (5.1) - (5.4).
OUTPUT: A walk w and the modified usage function u_{mod}
Set $w := ()$ to the empty walk
Set $v := v_0$
Set $u_{mod} := u$
repeat
    Set a to any arc in $A^{out}(v)$ with $u_{mod}(a) \geq 1$
    Add a to the end of w, set $u_{mod}(a) := u_{mod}(a) - 1$, set $v := t(a)$
until $v = v_1$
return w and u_{mod}

Algorithm 5.3: Construct a Walk

INPUT: graph $G = (V, A, s, t)$, usage function u, start node s, target node t
OUTPUT: The to u corresponding walk w
Set $rem := u$
Call Extract(G,rem,s,t). Set w to the resulting walk and rem to the resulting u_{mod}.
while there exists an arc $w_i \in w$ and an arc $a \in A^{out}(t(w_i))$ with $rem(a) \geq 1$ or an arc $a \in A^{out}(s)$ with $rem(a) \geq 1$ **do**
    if $a \in A^{out}(s)$ **then**
        Call Extract(G,rem,s,s). Insert the resulting walk at the beginning of w and set rem to the resulting u_{mod}.
    else
        Call Extract(G,rem,$t(w_i)$,$t(w_i)$). Insert the resulting walk into w after the arc w_i and set rem to the resulting u_{mod}.
    end if
end while
return w

Lemma 5.2.2. *Let $G = (V, A, s, t)$ be a directed multigraph, u a usage function and $v_0, v_1 \in V$ such that one of the two condition holds:*

1. *$v_0 = v_1$ and $\sum_{a \in A^{in}(v)} u(a) = \sum_{a \in A^{out}(v)} u(a)$ for all $v \in V$ and $\sum_{a \in A^{out}(v_0)} u^w(a) \geq 1$.*

2. *$v_0 \neq v_1$ and (5.1), (5.2) and (5.3) hold for $s = v_0$, $t = v_1$, $u^w = u$.*

Then Algorithm 5.2 applied to G, u, v_0 and v_1 terminates and returns a nonempty walk w from v_0 to v_1 such that $u^w(a) \leq u(a)$ for all $a \in A$. It further holds $u^w(a) + u_{mod}(a) = u(a)$ for all $a \in A$, where u_{mod} is the returned usage function.

Proof. Since v gets only set in the repeat loop whenever a new arc a is added to w in line 9, it is clear that it always holds $v = t(a)$ for a the last arc in w.

When we add a new arc a to w we know that $a \in A^{out}(v)$ with $v = t(a_{end})$ where a_{end} is the last arc in w before we add a or $v = v_0$ if w is empty. Therefore, we get that if w is not empty we have $s(a) = t(a_{end})$ and therefore w is always a valid walk.

Regardless if condition 1 or condition 2 of this lemma holds we always have $\sum_{a \in A^{out}(v)} u^w(a) \geq 1$. Thus we know that in the first iteration, we will find an arc a in $A^{out}(v) = A^{out}(v_0)$ with $u_{mod}(a) \geq 1$ on line 8.

We will prove now that we always find an arc $a \in A^{out}(v)$ with $u_{mod}(a) \geq 1$ on line 8, that means that the algorithm has no errors. We already know that the algorithm finds such an a if it is the first iteration. Therefore, we can assume that we are in an iteration, which is not the first and therefore $v \neq v_1$.

We prove by induction that at any iteration step after the first iteration

$$\sum_{a \in A^{in}(v')} u_{mod}(a) = \sum_{a \in A^{out}(v')} u_{mod}(a) \quad \forall v' \in V \setminus \{v, v_1\} \tag{5.5}$$

and

$$\sum_{a \in A^{out}(v)} u_{mod}(a) = 1 + \sum_{a \in A^{in}(v)} u_{mod}(a). \tag{5.6}$$

hold. Directly after the first iteration step we have

$$\sum_{a \in A^{out}(v_0)} u_{mod}(a) = \left(\sum_{a \in A^{out}(v_0)} u(a) \right) - 1,$$

$$\sum_{a \in A^{in}(v)} u_{mod}(a) = \left(\sum_{a \in A^{in}(v)} u(a) \right) - 1,$$

$$\sum_{a \in A^{out}(v')} u_{mod}(a) = \sum_{a \in A^{out}(v')} u(a) \quad \forall v' \in V \setminus \{v_0\}$$

$$\sum_{a \in A^{in}(v')} u_{mod}(a) = \sum_{a \in A^{in}(v')} u(a) \quad \forall v' \in V \setminus \{v\}.$$

This implies that directly after the first iteration (5.5) and (5.6) hold, regardless which of the two conditions of the lemma are satisfied.

After adding an arc a' to w and changing v to $t(a')$, the value $u_{mod}(w)$ gets decreased by one. That implies that also the sums $\sum_{a \in A^{out}(s(a'))} u_{mod}(a)$ and $\sum_{a \in A^{in}(t(a'))} u_{mod}(a)$ get decreased by one. All other sums in (5.5) and (5.6) stay the same. Since $s(a')$ equaled v before we added a', we know (5.6) held for $s(a')$ before we added a' to w. The right side gets decreased by one after adding a' and therefore (5.5) holds for $s(a')$ after adding a' to w. With the same argumentation we know that before adding a' to w (5.5) held for $t(a')$ and therefore after adding a' to w (5.6) holds for the new $v = t(a')$. All in all we get again that (5.5) and (5.6) hold in any iteration after the first iteration.

From (5.6) it directly follows that after the first iteration, we always have

$$\sum_{a \in A^{out}(v)} u_{mod}(a) \geq 1$$

and therefore that there exists an $a \in A^{out}(v)$ with $u_{mod}(a) \geq 1$. All in all we get that in the first iteration and also in all iterations after the first we always find an $a \in A^{out}(v)$ with $u_{mod}(a) \geq 1$. That means our algorithm has no errors.

We also know that $u_{mod}(a) \geq 0$ for all $a \in A$ at any iteration since we only decrease $u_{mod}(a)$ by 1 if $u_{mod}(a) \geq 1$. In every iteration the nonnegative sum

$$\sum_{a \in A} u_{mod}(a) \geq 0$$

gets decreased by 1 and since this sum was finite at the beginning of the algorithm our algorithm has to terminate at some point.

Therefore, we know that the algorithm has no errors, always terminates and returns a valid walk. Since the repeat loop gets called at least once we get that the resulting $w = (w_1, ..., w_k)$ is nonempty, that means $k > 0$. In the first iteration, we have $v = v_0$ and therefore $s(w_1) = v = v_0$. Since the algorithm terminated, we know after the last iteration $v = v_1$ and therefore $v = t(w_k) = v_1$.

Since u_{rem} was equal to u at the beginning and afterwards the values of it only get smaller we get $u_{rem}(a) \leq u(a)$ for all $a \in A$ at the end. Since we always reduced $u_{rem}(a)$ if and only if a was added to the walk we get $u(a) - u_{rem}(a) = u^w(a)$ for all $a \in A$. From this it also follows $u^w(a) \leq u(a)$ for all $a \in A$ since $u_{rem}(a) \geq 0$ for all $a \in A$.

□

Theorem 5.2.3. *Let $G = (V, A, s, t)$ be a directed multigraph, $s, t \in V$ with $s \neq t$ and u a usage function such that the conditions* (5.1) - (5.4) *are satisfied. Then, Algorithm 5.3 applied to G, u and s and t always terminates and returns a walk w with $u^w = u$.*

Proof. First of all we will prove that all calls of Algorithm 5.2 terminate. For this we will prove that the conditions of Lemma 5.2.2 are always satisfied.

For the call in line 4 we have $v_0 = s$ and $v_1 = t$ and $v_0 = s \neq t = v_1$. Since $rem = u$ we have that the conditions (5.1) - (5.4) are also satisfied for rem with $s = v_0$ and $t = v_1$, therefore condition 2 of the lemma is satisfied. We therefore know that the returned walk

w is a valid walk from s to t. We also know $rem(a) = u(a) - u^w(a)$ for all $a \in A$. By Theorem 5.2.1 we get that u^w also satisfies the conditions (5.1) - (5.4). Therefore, we get that

$$\sum_{a \in A^{in}(v)} rem(a) = \sum_{a \in A^{out}(v)} rem(a) \tag{5.7}$$

holds for all $v \in V$. We will prove by induction that (5.7) holds during the whole while loop and that every call of Algorithm 5.2 on line 7 or line 9 satisfies the conditions of Lemma 5.2.2.

Let $v = s$ if we are on line 7 and $v = t(w_i)$ if we are on line 9. Then we have $v_0 = v_1 = v$. We know that there exists an arc in $A^{out}(v)$ with $rem(a) \geq 1$. Since by induction (5.7) holds we have that condition 1 of Lemma 5.2.2 is satisfied.

Therefore, we can apply Lemma 5.2.2 and get that the returned walk is a valid walk from v to v. Let us call rem the usage function before the call, w the walk before the call, u_{mod} the returned usage function and w' the returned walk. Then we get $rem = u_{mod} + u^{w'}$. Since w' is a walk from v to v we get that $u^{w'}$ also satisfies (5.7) with $u^{w'}$ instead of rem. But that implies that also u_{mod} satisfies (5.7) with u_{mod} instead of rem. Since after the call of the algorithm rem gets set to u_{mod} we just proved that the new rem still satisfies (5.7).

With that, we proved that every call of Algorithm 5.2 terminates correctly. That also implies that w is always a valid walk from s to t after every iteration since we only insert a walk from $t(w_i)$ to $t(w_i)$ after w_i or a walk from s to s at the beginning of w in every iteration.

We always have $rem(a) = u(a) - u^w(a)$ for all $a \in A$. That implies since $rem(a)$ is always nonnegative, that the algorithm has to terminate at some point. We proved now that the algorithm terminates and always returns a valid walk from s to t.

It remains to prove that $u^w(a) = u(a)$ for all $a \in A$ at the end, which is the same as saying $rem(a) = 0$ for all $a \in A$.

We know that at the end since the while loop exited we have $rem(a) = 0$ for all $a \in A^{out}(t(w_i))$ for all $w_i \in w$ and $rem(a) = 0$ for all $a \in A^{out}(s)$. But since rem satisfies (5.7) this implies $rem(a) = 0$ for all $a \in A^{in}(t(w_i))$ for all $w_i \in w$ and $rem(a) = 0$ for all $a \in A^{in}(s)$.

We define now the set

$$S = V \setminus (\{s\} \cup \{t(w_i) : w_i \in w\}) \subseteq V \setminus \{s\}.$$

Let $b \notin A^{out}(t(w_i)) \cup A^{in}(t(w_i))$ for all $w_i \in w$ and $b \notin A^{out}(s) \cup A^{in}(s)$, that means exactly $s(b) \in S$ and $t(b) \in S$ and that means exactly $b \in E(S)$. Since u satisfies 5.4 we get

$$\sum_{a \in \delta^{in}(S)} u(a) \geq \min(1, u(b)).$$

For $a \in \delta^{in}(S)$ we have $t(a) \notin S$ and therefore $a \in A^{in}(s)$ or $a \in A^{in}(t(w_i))$ for some $w_i \in w$. But we already proved that then $rem(a) = 0$ which means $u(a) = u^w(a)$. But

$u^w(a) = 0$ since $s(a) \in S$ and all in all we get $u(a) = 0$. Therefore, we get

$$\min(1, u(b)) \leq \sum_{a \in \delta^{in}(S)} u(a) = 0.$$

This implies $u(b) = 0$. Since $rem(b) \leq u(b)$ we also have $rem(b) = 0$.

All in all we proved now $rem(a) = 0$ for all $a \in A$ at the end of the algorithm. But this implies $u(a) = u^w(a)$ for all $a \in A$.

□

Putting Theorem 5.2.1 and Theorem 5.2.3 together we get the following corollary.

Corollary 5.2.4. *Let G be a directed multigraph, u a usage function and s and t two nodes in G with $s \neq t$. Then there exists a walk w in G from s to t with $u^w = u$ if and only if the conditions* (5.1) - (5.4) *are satisfied.*

5.3 Mixed Integer Linear Program

In this section we will provide three mixed integer linear programming formulations to solve RTPP3. Using Corollary 5.2.4 we can reformulate RTPP3 such that instead of searching a walk with optimal total profit we search a usage vector, satisfying (5.1) - (5.4), with optimal total profit. Then we solve the MIP and apply Algorithm 5.3 to the result to get the final walk.

Since a usage function u can have more than only one walk w with $u^w = u$ we have to mention that Algorithm 5.3 only constructs one optimal walk and therefore does only return one and not all solutions of the problem. We could modify the algorithm such that it returns all optimal solutions, although the number of optimal solutions corresponding to a usage function can be exponential in the size of the walks and since it is enough for our purposes to find one solution we stay with the algorithm as we defined it.

5.3.1 Variables

Let in this section and the following sections be $G = (V, A, s, t)$ the given directed multigraph with an enumerated arc set $A = \{a_1, ..., a_m\}$. We will use in our mixed integer linear programs the boolean variable vector $(x_i)_{i=1}^m$ which will represent the usage function, that means $x_i = u(a_i)$. Furthermore, we will need helper variables to define penalties. We therefore introduce for every pair of arcs (a_i, a_j) with $P(a_i, a_j) \neq 0$ a boolean helper variable u_{ij} which is 1 if a_i and a_j are both used and 0 otherwise.

5.3.2 Integer Linear Programming Formulation

Definition (MIP1)**.** Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3. We call the following integer linear program *MIP1.*

$$\max_{x_i, u_{ij}} \sum_{i=1}^{m} p(a_i)x_i - \sum_{\substack{1 \le i < j \le m \\ P(a_i, a_j) \neq 0}} P(a_i, a_j)u_{ij} \tag{5.8}$$

$$s.t. \sum_{a_i \in A^{in}(v)} x_i = \sum_{a_i \in A^{out}(v)} x_i \quad \forall v \in V \setminus \{v_{\text{start}}, v_{\text{end}}\} \tag{5.9}$$

$$\sum_{a_i \in A^{out}(v_{\text{start}})} x_i = 1 + \sum_{a_i \in A^{in}(v_{\text{start}})} x_i \tag{5.10}$$

$$\sum_{a_i \in A^{in}(v_{\text{end}})} x_i = 1 + \sum_{a_i \in A^{out}(v_{\text{end}})} x_i \tag{5.11}$$

$$\sum_{a_i \in \delta^{in}(S)} x_i \ge x_j \quad \forall S \subseteq V \setminus \{v_{\text{start}}\}, a_j \in E(S) \tag{5.12}$$

$$C_{\min} \le \sum_{i=1}^{m} c(a_i)x_i \tag{5.13}$$

$$C_{\max} \ge \sum_{i=1}^{m} c(a_i)x_i \tag{5.14}$$

$$x_i \le x_j \quad \forall (a_i, a_j) \in D \tag{5.15}$$

$$u_{ij} \le x_i, \quad u_{ij} \le x_j, \quad u_{ij} \ge x_i + x_j - 1, \quad u_{ij} \ge 0 \tag{5.16}$$

$$x_i \le 1 \quad \forall i = 1, ..., m \tag{5.17}$$

$$x_i \ge 0 \quad \forall i = 1, ..., m \tag{5.18}$$

$$x_i, u_{jk} \in \mathbb{Z} \quad \forall i = 1, ..., m, 1 \le j < k \le m, P(a_j, a_k) \neq 0 \tag{5.19}$$

The objective function (5.8) is the total profit of the usage function u corresponding to x with $u(a_i) = x_i$. The constraints (5.9) - (5.11) together with the subtour elimination constraints (5.12) ensure that u corresponds to a walk (see Theorem 5.2.1 and Lemma 5.2.2, since we always have $u^w(b) \le 1$ for all $b \in A$ it holds $\min(1, u^w(b)) = u^w(b)$).

The constraints (5.13) and (5.14) ensure that the total cost of u is in the interval $[C_{\min}, C_{\max}]$ and the constraints (5.15) ensure the dependencies between arcs. It is easy to check that, since x_i are boolean variables and can only have the values 0 or 1, the constraints (5.16) ensure that $u_{ij} = 1$ if and only if $x_i = 1$ and $x_j = 1$. Furthermore the constraints (5.17) - (5.19) ensure that all occuring variables x_i and u_{ij} are boolean variables, that means they can only have the values 0 or 1.

Since the number of subtour elimination constraints (5.12) is exponential in the size of V it is inefficient to solve the program as defined above with standard procedures like branch-and-bound or the cutting-plane method. We therefore propose a branch-and-cut procedure which starts with no subtour elimination constraints and adds them if the solution of the LP-relaxation violates them.

Algorithm 5.4 describes the procedure of finding violated subtour elimination constraints in pseudo code. It uses a parameter $\tau \in [0, 1)$, which regulates how much a constraint has to be violated in the relaxation to get added to the model. The running time of the algorithm 5.4 depends on which algorithm is used to compute the maximum flow between two nodes. The highest label push-relabel algorithm, which we will also use in our implementation (see Chapter 6), has a running time of $\mathcal{O}(|V|^2|A|^{1/2})$ (see [8]). Therefore we get that the running time of Algorithm 5.4 is in $O(|V|^3|A|^{1/2})$.

Algorithm 5.4: Find Violated Subtour Elimination Constraints

1: **INPUT:** instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ and a solution vector x of the LP-relaxation at the current branch-and-bound node.
2: Set $G' = (V', A', s', t')$ to the graph G reduced to all arcs $a_i \in A'$ with $x_i > 0$ and with the node set $V' = \{s(a_i) : a_i \in A'\} \cup \{t(a_i) : a_i \in A'\}$.
3: Set $V_0 = \{v_{\text{start}}\}$.
4: Sort the arcs $a_i \in A'$ by x_i in decreasing order.
5: **while** $\exists a_i \in A' : s(a_i) \notin V_0 \wedge t(a_i) \notin V_0$ **do**
6: Choose an a_j satisfying that the source and the target node are not in V_0 with maximal x_j.
7: Recursively apply breadth-first-search to add all nodes to V_0 which are accessible by nodes from V_0 with a total flow bigger or equal than $x_j - \tau$.
8: **if** Source and target node of a_j are still not in V_0 **then**
9: Compute the maximal flow from v_{start} to $s(a_j)$ with the arc values x_j
10: **if** maximal flow is smaller than $x_j - \tau$ **then**
11: Add a new constraint like in (5.12) with S is the set of nodes on the sink side such that the weight of the cut $(S, V' \setminus S)$ equals the maximal flow between v_{start} and $s(a_j)$ and with a_j as we defined above.
12: **end if**
13: Add $s(a_i)$ to V_0
14: **end if**
15: **end while**

Theorem 5.3.1. *Let $\tau \in [0, 1)$ be a parameter, $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ an instance of RTPP3 and x a solution of an LP-relaxation at some branch-and-bound node.*

Then, Algorithm 5.4 terminates and all newly added constraints are violated by x. If x is integral and Algorithm 5.4 does not add any cuts, then all subtour elimination constraints (5.12) *are satisfied by x.*

Proof. First of all the algorithm always terminates since in every iteration of the while loop at least one arc, $s(a_i)$ or $t(a_i)$, gets added to V_0. Since V is finite this means that the while loop has at most $|V|$ iterations and therefore the algorithm terminates.

Let $\sum_{a_i \in \delta^{in}(S)} x_i \geq x_j$ be a newly added constraint with $S \subseteq V \setminus \{v_{\text{start}}\}$ and a_j.

First of all we prove that $a_j \in E(S)$ really holds. It holds $s(a_j) \in S$ since S is a cut including the sink node $s(a_j)$. Since the weight of the cut $(S, V' \setminus S)$ is smaller than

x_j we also know that $t(a_j) \in S$ since otherwise the weight would be at least x_j. That implies $a_j \in E(S)$.

The weight of the cut $(S, V' \setminus S)$ is exactly $\sum_{a_i \in \delta^{in}(S)} x_i$ and is smaller than x_j and therefore the newly added constraint is violated by x.

Let us now assume that x is integral, that means $x_i \in \{0, 1\}$ for all $1 \leq i \leq m$ and that the algorithm did not add any new constraints. We want to prove now that x satisfies all constraints (5.12) and therefore is an optimal solution of the integer program MIP1.

Suppose that there exists a constraint $\sum_{a_i \in \delta^{in}(S)} x_i \geq x_j$ with $S \subseteq V \setminus \{v_{\text{start}}\}$ and $a_j \in E(S)$ which is violated by x. Since x is integral that means that $\sum_{a_i \in \delta^{in}(S)} x_i = 0$ and $x_j = 1$. Therefore, the total flow between v_{start} and any point of S is 0. Since all arcs in G' have weight bigger than 0 this implies that there is no path in G' from v_{start} to any point of S and therefore we never add a point of S to V_0, since points get only added through a breadth-first-search algorithm starting from v_{start} or after adding a new constraint which we assumed never happens. At some iteration of the while loop the arc a_j will get chosen since $s(a_j) \in S$ and $t(a_j) \in S$ will not be added to V_0 during the whole algorithm. The maximal flow between v_{start} and $s(a_j)$ is 0 and is therefore smaller than $x_j - \tau = 1 - \tau > 0$. But that means that the corresponding constraint to S and a_j would get added which is a contradiction to our assumption. □

Algorithm 5.5 describes now the complete algorithm to solve RTPP3 in pseudo code.

Algorithm 5.5: Solving RTPP3 with Branch-and-cut

```
1: INPUT: instance (G, v_start, v_end, C_min, C_max, p, c, P, D) of RTPP3
2: OUTPUT: optimal walk w if it exists
3: Solve MIP1 with a branch-and-cut procedure using Algorithm 5.4 to find
   violated constraints
4: if there exists a valid solution of MIP1 then
5:    Apply Algorithm 5.3 to the usage function u(a_i) = x_i where x is the solution
      vector of the MIP1
6:    return the resulting walk
7: else
8:    return INFEASIBLE
9: end if
```

Theorem 5.3.2. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3.*

Then Algorithm 5.5 applied to this instance terminates and returns a valid solution if one exists and INFEASIBLE otherwise.

Proof. By Theorem 5.3.1 the branch-and-cut procedure using Algorithm 5.4 to find violated constraints is correct and returns an optimal solution if there exists one. If there exists no optimal solution the branch-and-cut procedure will return no solution and therefore the algorithm will return INFEASIBLE. If it returns an optimal solution x Algorithm 5.3

will return a walk w such that $u^w(a_i) = x_i$. By construction of MIP1 we know that this optimal walk is an optimal solution of the instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ of RTPP3. □

Remark 5.3.1. Since Algorithm 5.4 is correct for all parameter $\tau \in [0, 1)$ we can choose the parameter in a way that the algorithm is fastest. We will consider this parameter tuning in Chapter 7.

5.4 Flow Approach for Eliminating Subtours

In this section we will present a modification of MIP1 by replacing the subtour elimination constraints (5.12). The idea of the new subtour elimination constraints is that a walk is a sequence of arcs and therefore every arc has a fixed time when it is used, if we interpret the costs as time. For example, if we have a walk $w = (w_1, ..., w_k)$ then the arc w_1 is used at time 0 and the arc w_2 is used at time $c(w_1)$ and so on.

Since we also want to prevent subtours where each arc has cost equal to 0 we have to add a very small value $\varphi > 0$ to the costs of every arc.

To formalize this idea we use a flow formulation, where each used arc a_i can be seen as a sink and produces $c(a_i) + \varphi$ which gets added to the flow. At the end node v_{end} the amount of flow coming in is between $C_{\min}$ and $C_{\max} + m\varphi$.

Definition (MIP2)**.** Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3. We define the linear program *MIP2* by the following modification of MIP1.

Let furthermore be $\varphi > 0$.

We add a new variable vector $(f_i)_{i=1}^m$ which has not to be integral. Then we replace constraints (5.12) by the following constraints:

$$0 \leq f_i \leq (C_{\max} + m\varphi) \cdot x_i \quad \forall 1 \leq i \leq m \tag{5.20}$$

$$\sum_{a_i \in A^{out}(v)} f_i = \sum_{a_i \in A^{in}(v)} (f_i + (c(a_i) + \varphi)x_i) \quad \forall v \in V \setminus \{v_{\text{end}}\} \tag{5.21}$$

$$C_{\min} + \varphi \sum_{i=1}^{m} x_i \leq \sum_{a_i \in A^{in}(v_{\text{end}})} (f_i + (c(a_i) + \varphi)x_i +) - \sum_{a_i \in A^{out}(v_{\text{end}})} f_i \leq C_{\max} + \varphi \sum_{i=1}^{m} x_i \tag{5.22}$$

Algorithm 5.6 is the modification of Algorithm 5.5 if we solve (MIP2) instead of (MIP1).

The following theorem shows that the two formulations MIP1 and MIP2 are equivalent and as a corollary we get that Algorithm 5.6 is correct.

Theorem 5.4.1. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3 and $\varphi > 0$ a parameter. Then, for every feasible solution $((x_i)_{i=1}^m, (u_{ij}))$ of MIP1 we can find a vector $(f_i)_{i=1}^m$ such that $((x_i)_{i=1}^m, (u_{ij}), (f_i)_{i=1}^m)$ is a feasible solution of MIP2. On the other hand, if $((x_i)_{i=1}^m, (u_{ij}), (f_i)_{i=1}^m)$ is a feasible solution of MIP2 then $((x_i)_{i=1}^m, (u_{ij}))$ is a feasible solution of MIP1.*

Algorithm 5.6: Solving RTPP3 with Branch-and-bound using a Flow Formulation

1: **INPUT:** instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ of RTPP3
2: **OUTPUT:** optimal walk w if it exists
3: Solve MIP2 with a branch-and-bound procedure
4: **if** there exists a valid solution of MIP2 **then**
5: Apply Algorithm 5.3 to the usage function $u(a_i) = x_i$ where x is the solution vector of the MIP2
6: **return** the resulting walk
7: **else**
8: **return** INFEASIBLE
9: **end if**

Proof. Let $((x_i)_{i=1}^m, (u_{ij}))$ be a feasible solution of MIP1. By Theorem 5.2.3 we can apply Algorithm 5.3 and get a walk $w = (w_1, ..., w_k)$ with $u^w(a_i) = x_i$ for all $i = 1, ..., m$. Since $x_i \leq 1$ we know that $w_j \neq w_\ell$ if $j \neq \ell$. Let i_j be the index of the arc w_j, that means $w_j = a_{i_j}$. Then we can define:

$$f_{i_j} = \sum_{\ell=1}^{j-1} (c(w_\ell) + \varphi) \quad \forall j = 1, ..., k$$

$$f_i = 0 \quad \forall i \notin \{i_j : j = 1, ..., k\}$$

We have to prove now that $((x_i)_{i=1}^m, (u_{ij}), (f_i)_{i=1}^m)$ is a feasible solution of MIP2. It is enough to show that the constraints (5.20)-(5.22) are satisfied.

By definition $c(a) \geq 0$ for all $a \in A$ and therefore we get $f_i \geq 0$ for all $i = 1, ..., m$. On the other hand, we have if $x_i = 0$ then $f_i = 0$ and otherwise for $i = i_k$

$$f_i = f_{i_k} = \sum_{\ell=1}^{k-1} (c(w_\ell) + \varphi) \leq \sum_{i=1}^{m} x_i (c(a_i) + \varphi) = c(w) + k\varphi \leq C_{\max} + m\varphi$$

and therefore the constraints (5.20) are satisfied. We furthermore calculate for $v \in V \setminus \{v_{\text{start}}, v_{\text{end}}\}$:

$$\sum_{a_i \in A^{out}(v)} f_i = \sum_{w_j \in A^{out}(v)} f_{i_j} = \sum_{w_j \in A^{out}(v)} (f_{i_{j-1}} + c(w_{j-1}) + \varphi)$$

First of all $w_1 \in A^{out}(v_{\text{start}})$ and therefore $w_1 \notin A^{out}(v)$ since $v \neq v_{\text{start}}$, that implies that $f_{i_{j-1}}$ and w_{j-1} are well defined and therefore the calculation above is correct. The condition $w_j \in A^{out}(v)$ is the same as $s(w_j) = v$, but we know $s(w_j) = t(w_{j-1})$ by definition of a walk. That implies that the condition $w_j \in A^{out}(v)$ is equivalent to the condition $w_{j-1} \in A^{in}(v)$ and therefore we get

$$\begin{aligned} \sum_{a_i \in A^{out}(v)} f_i &= \sum_{w_j \in A^{out}(v)} (f_{i_{j-1}} + c(w_{j-1}) + \varphi) = \sum_{w_{j-1} \in A^{in}(v)} (f_{i_{j-1}} + c(w_{j-1}) + \varphi) \\ &= \sum_{w_\ell \in A^{in}(v)} (f_{i_\ell} + c(w_\ell) + \varphi) = \sum_{a_i \in A^{in}(v)} (f_i + (c(a_i) + \varphi) x_i). \end{aligned}$$

For $v = v_{\text{start}}$ we get

$$\sum_{a_i \in A^{out}(v)} f_i = \sum_{a_i \in A^{out}(v) \setminus \{w_1\}} f_i = \sum_{w_j \in A^{out}(v), j>1} f_{i_j}.$$

Therefore, we have again $j \neq 1$ in the last sum and we can use the same calculation as above and get again

$$\sum_{a_i \in A^{out}(v)} f_i = \sum_{a_i \in A^{in}(v)} (f_i + (c(a_i) + \varphi)x_i)$$

Thus (5.21) holds.

For v_{end} we can use the same calculation as above to get

$$\sum_{a_i \in A^{out}(v_{\text{end}})} f_i = \sum_{w_{j-1} \in A^{in}(v_{\text{end}}), 2 \leq j \leq k} (f_{i_{j-1}} + c(w_{j-1}) + \varphi)$$

but now we have to consider that $w_k \in A^{in}(v_{\text{end}})$ is not used in the above sum and therefore we get

$$\begin{aligned}
\sum_{a_i \in A^{out}(v_{\text{end}})} f_i &= \sum_{w_j \in A^{in}(v_{\text{end}}), 1 \leq j \leq k} (f_{i_{j-1}} + c(w_{j-1}) + \varphi) - f_{i_k} - c(w_k) - \varphi \\
&= \sum_{w_j \in A^{in}(v_{\text{end}}), 1 \leq j \leq k} (f_{i_{j-1}} + c(w_{j-1}) + \varphi) - \sum_{j=1}^{k} (c(w_j) + \varphi) \\
&= \sum_{a_i \in A^{in}(v_{\text{end}})} (f_i + (c(a_i) + \varphi)x_i) - c(w) - k\varphi
\end{aligned}$$

And therefore we get

$$\sum_{a_i \in A^{in}(v_{\text{end}})} (f_i + c(a_i)x_i) - \sum_{a_i \in A^{out}(v_{\text{end}})} f_i = c(w) + k\varphi$$

and the constraint (5.22) follows from the constraint $C_{\min} \leq c(w) \leq C_{\max}$ and the fact $k = \sum_{i=1}^{m} x_i$.

Let now on the other hand be $((x_i)_{i=1}^m, (u_{ij}), (f_i)_{i=1}^m)$ a feasible solution of MIP2. We want to prove that then $((x_i)_{i=1}^m, (u_{ij})$ is a feasible solution of MIP1. It is enough to show that the constraints (5.12) are satisfied.

For the next step we will use the notation $\delta^{out}(S)$ which is defined analogous to $\delta^{in}(S)$ in Definition 5.2.1. That means we define

$$\delta^{out}(S) := \{a \in A : s(a) \in S, t(a) \notin S\}$$

for a set of nodes $S \subseteq V$.

Suppose now that one of the constraints (5.12) is not satisfied by $(x_i)_{i=1}^m$ and let $S \subseteq V \setminus \{v_{\text{start}}\}$ be the corresponding set of nodes and $a_j \in E(S)$. That means we have

$$\sum_{a_i \in \delta^{in}(S)} x_i < x_j.$$

Since all values x_i can only be 0 or 1 for all $i = 1, ..., m$, we get that $x_j = 1$ and $x_i = 0$ for all $a_i \in \delta^{in}(S)$.

We know by the constraints (5.9) and (5.11) that

$$\sum_{a_i \in A^{in}(v)} x_i \geq \sum_{a_i \in A^{out}(v)} x_i \quad \forall v \in V \setminus \{v_{\text{start}}\}$$

and therefore also for all $v \in S$.

We calculate:

$$\begin{aligned}\sum_{a_i \in \delta^{in}(S)} x_i + \sum_{a_i \in E(S)} x_i = \sum_{a_i : t(a_i) \in S} x_i = \sum_{v \in S} \sum_{a_i \in A^{in}(S)} x_i \geq \sum_{v \in S} \sum_{a_i \in A^{out}(S)} x_i \\ = \sum_{a_i : s(a_i) \in S} = \sum_{a_i \in \delta^{out}(S)} x_i + \sum_{a_i \in E(S)} x_i\end{aligned}$$

From this we get

$$\sum_{a_i \in \delta^{out}(S)} x_i \leq \sum_{a_i \in \delta^{in}(S)} x_i = 0$$

and therefore also $\sum_{a_i \in \delta^{out}(S)} x_i = 0$.

Either S or $V \setminus S$ do not contain v_{end}. Let V_0 be S if S does not contain v_{end} and otherwise $V_0 = V \setminus S$. Since $\delta^{out}(V \setminus S) = \delta^{in}(S)$ and $\delta^{in}(V \setminus S) = \delta^{out}(S)$ we always have

$$\sum_{a_i \in \delta^{out}(V_0)} x_i = 0 \text{ and } \sum_{a_i \in \delta^{in}(V_0)} x_i = 0.$$

From this we get

$$\sum_{a_i : s(a_i) \in V_0} f_i - \sum_{a_i : t(a_i) \in V_0} f_i = \sum_{a_i \in \delta^{out}(V_0)} f_i + \sum_{a_i \in E(S)} f_i - \sum_{a_i \in \delta^{in}(V_0)} f_i - \sum_{a_i \in E(S)} f_i = 0$$

But on the other hand we get by the constraints (5.21) and the fact that $V_0 \subseteq V \setminus \{v_{\text{end}}\}$:

$$\begin{aligned}\sum_{a_i : s(a_i) \in V_0} f_i - \sum_{a_i : t(a_i) \in V_0} f_i = \sum_{v \in V_0} \left(\sum_{a_i \in A^{out}(v)} f_i - \sum_{a_i \in A^{in}(v)} f_i \right) \\ = \sum_{v \in V_0} (c(a_i) + \varphi) x_i \geq (c(a_j) + \varphi) x_j \geq \varphi > 0\end{aligned}$$

Therefore, we get a contradiction which implies that the constraints (5.12) are all satisfied by the instance $((x_i)_{i=1}^m, (u_{ij})$ and therefore the instance is a feasible solution of MIP1. □

Corollary 5.4.2. *Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3 and $\varphi > 0$ a parameter, then Algorithm 5.6 applied to this instance terminates and returns a walk w which is a solution of the given instance of RTPP3.*

Remark 5.4.1. The strength of the constraints (5.20)-(5.22) depends on the parameter φ. Since in most applications the size k of the solution walk will be much smaller than the number m of arcs in the graph we want that φ is as small as possible. That would have the effect that the difference between $m\varphi$ and $k\varphi$ is small. As we do not know in previous how big the cost values of our instances are it seems reasonable to let the parameter φ depend on m and on $C_{\max}$. Therefore, we define

$$\varphi := \delta \frac{C_{\max}}{m}$$

where $m = |A|$ is the number of arcs in the given graph and $\delta > 0$ a new parameter. In concrete applications, the value of δ should be chosen as small as possible, but not too small, such that no numerical issues arise. We will decide how to choose δ in Chapter 7.

In many similar problems the subtour elimination with cuts is stronger than the subtour elimination with single commodity flows, i.e. the set of feasible solutions in the relaxation of the formulation with cuts is a subset of the projection of the set of feasible solutions in the relaxation of the single commodity flow formulation. But in our case we get the following theorem.

Theorem 5.4.3. *The relaxations of the formulation* (MIP1) *and the formulation* (MIP2) *are not comparable, i.e. there exist examples in which the set of feasible solutions of the relaxation of the first formulation is no subset of the projection of the set of feasible solutions in the relaxation of the second formulation and vice versa.*

Proof. Example 5.4.1 describes an example situation for both directions.

□

Example 5.4.1. Let $G = (V, A, s, t)$ be the directed multigraph visualized in Figure 5.1. Let furthermore be c the cost functions according to the values written in parentheses in Figure 5.1. Let p be any profit function for the graph G and we define

$$C_{\min} := 0 \text{ and } C_{\max} := 100.$$

The start node v_{start} and the end node v_{end} are the nodes with the corresponding labels. There are no penalties, that means $P(a, b) = 0$ for all $a, b \in A$, and no dependencies, that means $D := \emptyset$.

Then we consider the instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ of RTPP3. We will prove that the set of feasible solutions of the relaxation of MIP1 according to the given instance is not comparable to the set of feasible solutions of the relaxation of MIP2, regardless how we choose the parameter φ. We consider the relaxations at the root node, that means MIP1 when removing the integrability conditions and the same with MIP2.

We need a small φ, so that the relaxation of MIP2 is enough tight. Therefore, we will use the notation of Remark 5.4.1 and take $0 < \delta < 1$, that means in our case

$$\varphi = \delta \frac{C_{\max}}{m} = \delta \frac{100}{10} = 10\delta < 10.$$

It is easy to check, that the solution vector $\mathbf{x}^1$ with

$$x_i^1 := \begin{cases} \frac{1}{2} & i = 1,2,3 \\ 1 & i = 8 \\ 0 & otherwise \end{cases}$$

together with the empty vector $\mathbf{u}$ is a feasible solution of the relaxation of MIP1. The vector u is empty since there are no penalties. But we will prove that there exists no flow vector $\mathbf{f}^1$ such that $(\mathbf{x}^1, \mathbf{u}, \mathbf{f}^1)$ satisfies (5.20)-(5.22).

To prove this we assume that there is such a flow vector $\mathbf{f}^1$. By (5.20) we get that $f_9^1 = 0$ and $f_5^1 = 0$ and therefore we can conclude using (5.21) that

$$f_2^1 = f_2^1 + f_9^1 = f_1^1 + (c(a_1) + \varphi) \cdot x_1 + 0 = f_1^1 + (90 + \varphi) \cdot \frac{1}{2} = f_1^1 + 45 + \frac{\varphi}{2} \geq 45 + \frac{\varphi}{2}$$

$$f_3^1 = f_3^1 + f_5^1 \geq f_2^1 + (c(a_2) + \varphi) \cdot x_2 + 0 + 0 \geq 90 + \varphi$$

holds. But then we get by using (5.20)

$$90 + \varphi \leq f_3^1 \leq (C_{\max} + m\varphi)x_3 = \frac{100 + 10\varphi}{2} = 50 + 5\varphi$$

which implies $\varphi \geq 1$ which is a contradiction to our restriction $\varphi < 1$. Therefore, there cannot exist a flow vector $\mathbf{f}^1$ such that $(\mathbf{x}^1, \mathbf{u}, \mathbf{f}^1)$ is a feasible solution of the relaxation of MIP2.

Consider now on the other hand the solution vector $(\mathbf{x}^2, \mathbf{u}, \mathbf{f}^2)$ with

$$x_i^2 := \begin{cases} \frac{1}{2} & i = 4,6,7 \\ 1 & i = 5,8 \\ 0 & otherwise \end{cases} \qquad f_i^2 := \begin{cases} 2 + \varphi & i = 5 \\ 4 + 2\varphi & i = 7 \\ 5 + \frac{5}{2}\varphi & i = 8 \\ 0 & \text{otherwise} \end{cases}$$

and the empty vector $\mathbf{u}$. It is easy to check that this solution vector is a feasible solution of the relaxation of MIP2 regardless what we choose for φ, also if $\varphi = 0$ or $\varphi > 10$. But we will see that the solution vector $(\mathbf{x}^2, \mathbf{u})$ does not satisfy (5.12).

Consider the node set $S := \{v_2, v_3\}$ with the arc $a_5 \in E(S)$. Then we get

$$\sum_{a_i \in \delta^{in}(S)} x_i = x_2 + x_4 + x_8 = 0 + \frac{1}{2} + 0 = \frac{1}{2} < 1 = x_5.$$

Therefore, the solution vector $(\mathbf{x}^2, \mathbf{u})$ is not feasible in the relaxation of MIP1.

Remark 5.4.2. In both examples presented in 5.4.1 we did not use the arcs a_9 and a_{10} of the graph visualized in figure 5.1. But they are still very important. In Section 5.1 we presented a preprocessing algorithm which removes unnecessary nodes and arcs. If we would use the same graph without the arcs a_9 or a_10 some arcs would get removed from our

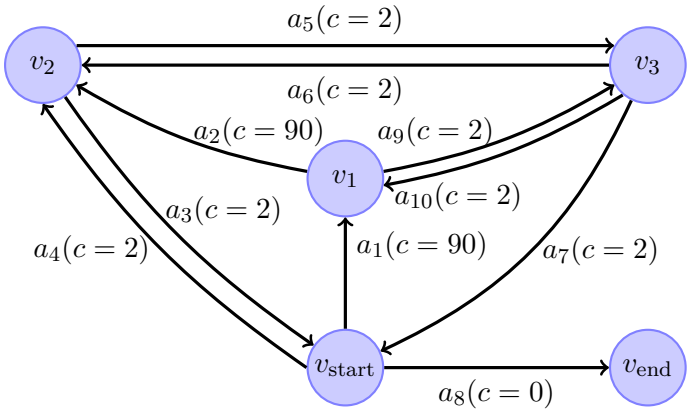

Figure 5.1: Graph where the set of feasible solutions of the relaxation of MIP1 and the set of feasible solutions of the relaxation of MIP2 is incomparable

graph through preprocessing. Therefore it would not be a good example since the linear programs MIP1 and MIP2 are only solved after applying the preprocessing. For example, without the arc a_9 the shortest path from v_1 to v_{end} would have 92 costs and therefore the arc a_1 would get removed during preprocessing since $90 + 92 = 182 > C_{\max} = 100$. Without the arc a_{10} the shortest path from v_{start} to v_1 would have 90 costs and therefore the arc a_2 would get removed during preprocessing since $90 + 90 + 2 = 182 > C_{\max} = 100$.

5.4.1 Mixed Approach for Eliminating Subtours

Theorem 5.4.3 states that the relaxation of MIP1 and the relaxation of MIP2 are not comparabale. Therefore it would be interesting to merge both formulations to get a combined formulation whose relaxation is stronger than the relaxations of the single formulations.

Definition (MIP3)**.** Let $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ be an instance of RTPP3. The linear program *MIP3* is the union of the programs MIP1 and MIP2, that means we add the constraints 5.12 to the formulation of MIP2.

Theorem 5.4.4. *The relaxation of MIP3 is tighter than the relaxations of MIP1 and MIP2, that means the set of feasible solutions of the relaxation of MIP3 is a subset of the set of feasible solutions of the relaxation of MIP1 or MIP2. There are cases where the relaxations are strictly tighter, that means the set of feasible solution is a proper subset.*

Proof. Since MIP3 is the union of all constraints from MIP1 and MIP2 it is clear that the relaxations are tighter than the relaxations of MIP1 or MIP2. The examples in 5.4.1 show that there are instances where the relaxation of MIP3 is strictly tighter than the relaxation of a single formulation.

□

Based on MIP3 we can now define Algorithm 5.7.

Algorithm 5.7: Solving RTPP3 with Branch-and-cut and a Flow Formulation

1: **INPUT:** instance $(G, v_{\text{start}}, v_{\text{end}}, C_{\min}, C_{\max}, p, c, P, D)$ of RTPP3
2: **OUTPUT:** optimal walk w if it exists
3: Solve MIP3 with a branch-and-cut procedure using Algorithm 5.4 to find violated constraints of the form (5.12).
4: **if** there exists a valid solution of MIP3 **then**
5: Apply Algorithm 5.3 to the usage function $u(a_i) = x_i$ where x is the solution vector of the MIP3
6: **return** the resulting walk
7: **else**
8: **return** INFEASIBLE
9: **end if**

CHAPTER 6

Implementation

In this chapter, we will shortly present our implementation of the algorithms 5.5, 5.6 and 5.7. The code itself is well documented and therefore we will present here only an overview.

6.1 General

The code is written in `C++` and uses the IBM ILOG CPLEX Optimization Studio 12.5 to solve the mixed integer problems MIP1, MIP2 or MIP3. It got compiled with `g++` 4.8.2.

As basic data structure for directed multigraphs we implemented a generic graph structure, which stores nodes and arcs such that they can be accessed in constant time. It collects all parallel arcs with the same start and end node together to one so called single arc. Therefore, the single arcs and the nodes form a directed simple graph with loops.

For our testing purposes, we implemented different running modes which can be changed by an input parameter for the main program:

1. `--NORMAL` solves the given instance.

2. `--TESTING` solves the given instance, or the given set of instances with different maximal costs $C_{\max}$ and maximal distances to the end node. For every solved instance, it generates a log file in a given directory.

3. `--WRITE_TESTING_DATA` reads all log files in a given directory and summarizes the results in a csv file.

4. `--PARAMETER_TUNING` trys different values for the parameter τ in the interval $(0, 1)$ to solve the given problem instances and tries to dynamically find a local minimum where the running times summed up are minimal.

5. `--PARAMETER_FINDING` splits the interval $[0, 1)$ into parts and tries for every part one value as τ parameter applied to the given instances.

6. `--FIND_START_POINTS` searches in the given graph a maximal set of starting points such that two points in this set have distance at least the given parameter maximal distance.

Additional to this modes the following parameter specify the input instances and some behavior:

1. `-ix` or `-i path` where `x` can be 0,1,2,3 or 4 for the five used test graphs described in Chapter 7 or `path` a path to the graph data file or directory.

2. `-px` where x can be 0 or 1. It specifies the used parser, 0 means it tries to parse the given file as an input file and 1 means it tries to parse the given directory as an AOP benchmark input directory.

3. `-0`, `-1` or `-2` specifies the used solver. `-0` is the implementation of Algorithm 5.5, `-1` is the implementation of Algorithm 5.6 and `-2` is the implementation of Algorithm 5.7.

4. `-s nodeID` specifies the id of the start node.

5. `-d value` specifies the maximal distance of a used node to the end node.

6. `-l value` specifies the minimal cost $C_{\min}$.

7. `-u value` specifies the maximal cost $C_{\max}$.

8. `-t value` specifies the maximal computation time in seconds.

6.2 Parsing and Preprocessing

For our tests we used two different sets of instances, which will be described in detail in Chapter 7. We implemented for both types of instances a parser which parses the input files describing the street network and constructs the corresponding graph. Both parsers are based on an abstract parser which provides functionality like parsing a line or parsing only the next block until a given delimiter and manages a buffer for the input file.

Since the instances are mostly given in form of problem RTPP1 we implemented the transformations described in 2.3.1 and 2.3.4. There also no penalties given for the instances and therefore we define in a preprocessing step penalties for parallel and opposite arcs. We assume that an arc which has a profit value larger than its cost is considered as a more attractive arc compared to the average and an arc which has a profit value lower than its cost is considered not so attractive. If a graph does not satisfy this assumption, we would have to scale the profits appropriately. With this assumption it makes sense to use the costs of an arc for the penalty. Therefore the implementation

defines the penalty of two opposite arcs as the sum of their costs and the penalty of two parallel arcs as 1.5 times the sum of their costs. This functionality together with the preprocessing described in Algorithm 5.1 is defined in a preprocess namespace which is used before solving an instance.

6.3 Solving

Since the models MIP1, MIP2 and MIP3 only differ in the subtour elimination constraints, we use a base class for all three implementations, which initializes the model and all constraints except the subtour elimination constraint.

In the implementation of Algorithm 5.5 we also initialize a user cut callback and a lazy constraint callback (see [1]). The user cut callback gets called after solving a relaxation and is used to find unsatisfied constraints of the form (5.12). The user cut callback implements Algorithm 5.4. For finding a maximal flow from the start node to a given node as it is used in the algorithm on line 9 we used the max-flow min-cut algorithm of Shekhovtsov and Hlavac, see [19], [20] and [21].

If the solution of a relaxation satisfies all integral constraints the user cut callback will not be called and therefore we also need a lazy constraint callback which is called whenever an integral solution got found. Here we do not need a max-flow algorithm to identify unsatisfied constraints. We do simply breadth-first search on the graph of all used arcs to identify all weakly connected components. Then every component other than the one containing the start node is a cut.

In the implementation of Algorithm 5.6 we do not need any user cut callbacks or lazy constraint callbacks, we only have to add the constraints (5.20)-(5.22) to our model. And in the implementation of Algorithm 5.7 we add the constraints (5.20)-(5.22) to the model and additionally add the user cut callback to the model. We do not need the lazy constraint callback since the constraints (5.20)-(5.22) already ensure that the solution has no subtours.

CHAPTER 7

Evaluation

In this chapter we will evaluate our implementations of the Algorithms 5.5, 5.6 and 5.7. Throughout this chapter we will simply call the implementation of the Algorithm 5.5 as cut implementation, the implementation of Algorithm 5.6 as flow implementation and the implementation of Algorithm 5.7 as mixed implementation.

We will use different graphs and different start nodes to test the running times of all three algorithms. Since the algorithms presented in [22] and [25] are made for very similar problems, we will also try to compare our test results with the test results presented in those two papers.

All tests were run on a single core of an Intel Xeon X5650 processor with 2.6 GHz and 8 GB RAM available. The code was compiled with g++ 4.8.2. The general purpose solver IBM ILOG CPLEX Optimization Studio 12.5 was used.

7.1 Test Instances

There are two different types of test instances which we will use for our tests. The unit for lengths of arcs and for distances are for all test instances meters.

7.1.1 Instances Provided by the AIT Austrian Institute of Technology with Artificial Profits

The first type of test instances is based on four graphs provided by the AIT Austrian Institute of Technology.

Test Graph Instance

The first graph is an artificial test graph with 157 nodes and 417 arcs. The profits of the arcs are set equally to the artificial costs of the arcs. Since the graph is not very big we only use one fixed start node for our tests.

Because of the artificial nature of the graph we will see that the running times of solving this instance may behave differently to the running times of other instances although they have the same parameters and almost the same number of nodes and arcs.

Josefstadt Instance

The second graph provided by the AIT represents a street network of a part of Vienna around the district Josefstadt. Every node in this graph is associated with a longitude and a latitude representing a point on the map. The origin of the street data is the OpenStreetMap project (see [2]).

The whole graph has 2216 nodes and 5056 arcs. The costs of the arcs are the lengths of the streets in meters and the profits are artificial values directly proportional to the costs. Therefore the profits have nothing to do with the real world attractiveness of the streets.

This street network is very dense compared to other street networks examined in this thesis. Although the graph has many nodes and arcs the distances between the nodes are relatively small. Hence, we use this graph only with one fixed start point, since testing with multiple start points in such a small area would lead to similar results and correlated running times.

Josefstadt Instance with Profits Determined by Real Masures

The third graph provided by the AIT is similar to the graph of the Josefstadt instance described in the previous section as it represents the same street system around the district Josefstadt in Vienna. But the Josefstadt instance in the previous section has artificial arc profits. Therefore, a solution of this instance may not be attractive in reality at all. Hence, we want to find profits which somehow really measure the attractiveness of an arc to get a more realistic testing scenario.

To achieve this goal we use popularity data provided by the cycle to work campaign "Österreich radelt zur Arbeit" by the Austrian bicycle advocacy group "Radlobby Österreich". In this project they measured the popularity of an arc by how many participants are using this arc riding to their work place.

As a second attractiveness indicator we use a value measuring the suitability of a street for bicycles. That means cycleways will have a very high value and main roads will have lower values.

The profit is now a mix of this two measures such that each measure contributes one half and the resulting profit values are proportional to the corresponding costs. With those profits we hope to get an useful measure of attractiveness, but since attractiveness of a street is always subjective it is very hard to verify that.

The graph itself is a little different to the graph of the Josefstadt instance described in the previous section. It has 2767 nodes and 5833 arcs.

Kittsee Instances

The fourth graph provided by the AIT represents a street network in the area of Kittsee in Burgenland in Austria. As before every node is associated with a longitude and a latitude and the origin of the data is the OpenStreetMap project. The costs of an arc are again equal to the length of the corresponding street part and the profits of an arc are artificial values proportional to the costs.

The area around Kittsee was chosen by the AIT because a project partner, a rehabilitation center, is located in Kittsee. Therefore, it makes sense to test the algorithm applied to the street network around this rehabilitation center.

The graph has 6175 nodes and 15664 arcs. Hence, it is a very big graph compared to the others and also represents a street system covering a very wide area. Thus it makes sense trying to find more than one start point to gain many test instances. To find some start points we used a randomized algorithm which randomly iterates over all points in the graph and selects the points as long as their distance to all previously selected points is above some threshold. As distance we used the geometrical distance between the points, which we can calculate with the latitudes and the longitudes of the points.

With that implementation we found 10 possible start points such that every two of them are at least 8 kilometers apart from each other. Therefore we get 10 test instances from this graph and we will call them Kittsee 1 instance up to Kittsee 10 instance.

7.1.2 Benchmark Instances

The second type of test instances are basically benchmark instances for the problems AOP and CTPP (see Section 4.3.3). They are based on a graph which was used as AOP instance for testing purposes in [22]. The graph corresponds to the street network of East-Flanders in Belgium and contains 989 vertices and 2961 arcs, with a total track length of 3585 km. The costs of an arc is given again as the length of the street part in meters. In [22] they used the costs of an arc as profit of the arc.

In [25] they used the same graph as CTPP instance and added random arc profits. This has the effect that the solution is not always the one with maximal costs. However, they also tested their algorithm with the old profits which were equal to the costs of an arc. We will also use both variants, one time with profits equals costs and one time with this random arc profits to be able to compare the results with both tests.

The graph together with its random arc profits and testing results is available at `https://www.mech.kuleuven.be/en/cib/op/` (last downloaded 2015-04-14).

As we want to compare our implementation with the results in [22] and [25] we will use the same start points as they did. Therefore we get the ten instances with the start node ids 2, 6, 10, 14, 18, 22, 26, 30, 34 and 38. For every of this ten instances we have two versions, one where the profits equal the costs and one where the profits equal the random values constructed in [25].

Since CTPP has the restriction that every arc can only be used once, we skip the preprocessing step where we duplicate every arc for all these benchmark instances. Then we straight forwardly get an instance of RTPP3 with no duplicated arcs and thus every

original arc can only be used once. To simulate the restriction of CTPP, that opposite arcs cannot be used together, we will apply a fixed penalty for using opposite arcs. In all our tests we used as penalty 400000 which was much bigger than all possible profits of solution walks. Therefore, if our algorithm returns a solution with negative profit we know that the instance corresponds to an unsolvable instance in CTPP. On the other hand, if our implementation returns a solution with positive profit we know that no two opposite arcs are used together and therefore the solution is also a valid solution of CTPP.

Another difference of CTPP to RTPP3 is that CTPP can have multiple possible start nodes, but in [25] they also tested all instances with one fixed start node and therefore we can directly compare our results with those results.

7.2 Parameter Tuning

Before we can test our instances and compare the results we want to find good parameter values $\tau \in [0, 1)$ (see Algorithm 5.4) and $\delta > 0$ (see the definition of MIP2 and Remark 5.4.1).

Since we do not have any 0-cost cycles with positive profits in our test instances it would be possible to use the parameter $\delta = 0$. Although, we want to use a parameter $\delta > 0$ to also get meaningful test results for the case that there would be a 0-cost cycle with positive profits.

We do not really need a parameter tuning for δ. We only have to consider that we chose δ as small as possible such that there are still no numerical issues. Since all real values are represented with double precision we are numerically on the save side if we use $\delta = 10^{-8}$ which is small enough that it does not change the strength of the formulation very much. One numerical issue which we have still to consider is the CPLEX feasibility tolerance. We do not want that a 0-cost subtour is feasible because of the feasibility tolerance. Therefore we redefine φ to be 1.1 times the feasibility tolerance of CPLEX (which is by default 10^{-6}) if φ would be smaller than the feasibility tolerance.

What remains is to find a good value for the parameter $\tau \in [0, 1)$. We will do this by testing different τ values for our instances.

7.2.1 Parameter τ Test for Josefstadt, Kittsee and the Test Graph Instance

Table 7.1 lists the running times of the cut implementation with different parameters τ applied to the instances Josefstadt, Kittsee 1 – Kittsee 10 and test graph with $C_{\max} = 2000$, $C_{\max} = 3000$ and $C_{\max} = 4000$. In all tests $C_{\min}$ is set to $1/2 \cdot C_{\max}$.

We can see that for $C_{\max} = 2000$ the best value of the tested values for τ in terms of computation time would be 0.9, but if $C_{\max} = 3000$ this would be the worst choice. Also for $C_{\max} = 4000$ it is not the best choice. This shows us that none of the tested values minimizes the computation time for all given instances.

Another interesting case is the instance Kittsee 3 with $C_{\max} = 4000$. There we see that for τ smaller or equal to 0.4 the running time is more than an hour and with a value bigger or equal to 0.5 the running time is under 20 seconds. Thus small changes in the parameter can drastically change the running times. Another example of this is the Josefstadt instance with $C_{\max} = 4000$ where the running time with parameter 0.5 is 2.4 hours and the running time with parameter 0.6 is 16 hours.

What we can see is that at least for these twelve test instances the value 0.5 is a very good choice for τ. For $C_{\max} = 2000$ it has the second best running time of all tested values and for $C_{\max} = 3000$ and $C_{\max} = 4000$ it has the best running time of all tested values. But to be sure that this value is really a good choice we need more tests on other instances.

7.2.2 Parameter τ Test for the Benchmark Instances

In Table 7.2 the running times of the same test applied to the benchmark instances where the profits equal the costs are shown. Since almost all instances could get solved under a second for $C_{\max} = 20000$ we omitted these data since it would not be very interesting. The interesting cases are $C_{\max} = 30000$ and $C_{\max} = 40000$. Like before we used $C_{\min} = 1/2 \cdot C_{\max}$ in the tests. The second column of the table contains the start node IDs of the different start nodes. Again we can observe that 0.5 is among the best tested values for τ, but this time the bigger values from 0.6 up to 0.9 are slightly faster.

Table 7.3 shows the running times of the cut implementation applied to the benchmark instances with random profits. Compared to the instances where profits equal costs those instances are much faster. Therefore we omit the test results for $C_{\max} = 20000$ and $C_{\max} = 30000$. To get a better overview we add a new test with $C_{\max} = 50000$. Again $C_{\min}$ is always set to $1/2 \cdot C_{\max}$. For $C_{\max} = 40000$ we see that the value 0.5 is again a good choice but the values 0.6, 0.7 and 0.9 are slightly better. For $C_{\max} = 50000$ the value 0.5 is the best choice of all tested values in terms of the sum of the computation times.

7.2.3 Parameter τ Test for Josefstadt Instance with Realistic Profits

In this section we want to check if the value 0.5 is also a good choice for τ in a realistic scenario with realistic profits. Therefore we apply our cut implementation to the Josefstadt instance with realistic profits. Table 7.4 shows the running times for $C_{\max} = 1000$, $C_{\max} = 2000$ and $C_{\max} = 3000$. As we can see the value 0.5 is again a good choice. For $C_{\max} = 1000$ 0.8 and 0.9 are better but for $C_{\max} = 2000$ it has the best running time of all tested values. For $C_{\max} = 3000$ we see that 0.6 would have a much better running time and 0.2 would also have a better running time.

7.2.4 Conclusion

We tested all instances with the ten parameters 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 and 0.9. We saw that 0.5 is always among those values which have the best running

Table 7.1: Parameter τ testing for the cut implementation applied to the instances Josefstadt, Kittsee and test graph.

		Times (in sec. unless otherwise specified) for various values for τ									
$C_{\max}$	**Instance**	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
	Josefstadt	17	10	16	10	13	9	5	7	9	4
	Kittsee 1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 2	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 3	74	51	183	99	33	26	41	51	29	26
	Kittsee 4	1	1	1	1	1	1	1	1	1	1
	Kittsee 5	3	1	1	1	1	1	1	1	1	1
2000	Kittsee 6	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 7	1	1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 8	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 9	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 10	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	test graph	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Sum	**95**	**64**	**202**	**111**	**48**	**37**	**48**	**60**	**40**	**32**
	Josefstadt	426	290	165	265	389	104	184	660	383	1639
	Kittsee 1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 2	<1	<1	<1	<1	<1	1	<1	<1	<1	<1
	Kittsee 3	174	22	27	49	39	15	8	10	9	64
	Kittsee 4	23	15	13	10	9	3	3	5	5	6
	Kittsee 5	5	3	5	4	3	2	1	1	1	2
3000	Kittsee 6	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 7	26	19	110	19	17	38	30	15	14	22
	Kittsee 8	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 9	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 10	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	test graph	205	50	28	21	22	12	10	7	5	10
	Sum	**858**	**401**	**349**	**368**	**478**	**174**	**237**	**699**	**418**	**1745**
	Josefstadt	12h	4.8h	5.9h	2.9h	6.4h	2.4h	16h	8.7h	12h	6.9h
	Kittsee 1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 2	1h	2.5h	2.2h	1.6h	2.5h	19	7	16	15	12
	Kittsee 3	687	477	1039	612	312	33	95	128	99	90
	Kittsee 4	83	24	28	20	14	5	9	9	16	7
	Kittsee 5	17	10	11	8	9	3	3	4	4	2
4000	Kittsee 6	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 7	71	99	63	80	68	13	17	27	19	12
	Kittsee 8	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 9	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	Kittsee 10	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	test graph	258	403	126	116	75	78	31	48	48	59
	Sum	**13h**	**7.6h**	**8.5h**	**4.8h**	**9.1h**	**2.4h**	**16h**	**8.8h**	**12h**	**7h**

Table 7.2: Parameter τ testing for the cut implementation applied to the benchmark instances where the profits equal the costs.

		Times (in seconds unless otherwise specified) for various values for τ									
$C_{\max}$	**Start**	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
	2	13	7	7	6	6	5	3	2	3	6
	6	6	9	3	5	2	2	1	1	1	1
	10	11	8	5	9	9	1	1	1	1	1
	14	31	21	9	8	9	4	5	4	4	3
	18	3	2	2	2	1	1	1	1	1	1
30000	22	3	1	1	1	1	1	1	1	1	1
	26	11	5	2	2	2	1	1	1	1	1
	30	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
	34	371	66	34	19	16	10	8	10	10	11
	38	432	55	34	20	16	11	8	9	10	10
	Sum	**883**	**174**	**97**	**72**	**62**	**35**	**28**	**30**	**33**	**35**
	2	>10h	4.3h	389	2992	773	3912	1306	17	1758	1655
	6	>10h	>10h	411	218	388	753	221	105	98	752
	10	>10h	2502	2564	2015	750	518	1342	667	640	1662
	14	>10h	>10h	4.7h	1.6h	1.9h	1.2h	3465	1.1h	1.1h	2186
	18	>10h	2618	1692	1401	419	515	342	251	227	234
40000	22	>10h	>10h	1.4h	2782	1429	1254	759	706	458	397
	26	>10h	4.8h	481	2830	1564	1919	32	250	247	1101
	30	46	10	25	11	7	7	5	8	6	5
	34	7h	102	6	247	86	688	14	158	12	5
	38	>10h	>10h	5h	2.5h	1.8h	1.1h	1.7h	2148	3024	3192
	Sum	**>87h**	**>50h**	**13h**	**7.5h**	**5.2h**	**5h**	**3.8h**	**2.3h**	**3h**	**3.1h**

times. There are instances where the higher parameters like 0.6 or even higher have better running times but there are also instances where the opposite is true.

As the mix implementation is very similar to the cut implementation we use the same parameter value.

7.3 Comparison of the Three Implementations

In this section, we will compare our three implementations, the flow implementation, the cut implementation and the mixed implementation. As pointed out in Section 7.2 we will use the value 0.5 for the parameter τ in the cut implementation and the mixed implementation. Also as described in Table 7.2 we will use the value 10^{-8} for the parameter δ in the flow and the mixed implementation, but if the resulting φ is smaller than 10^{-6} we redefine φ to be $1.1 \cdot 10^{-6}$.

For all tests in this section the computation time for one instance is always limited to one hour. If an implementation needs longer than one hour its current solution after one

Table 7.3: Parameter τ testing for the cut implementation applied to the benchmark instances with random profits.

		Times (in seconds) for various values for τ									
$C_{\max}$	**Start**	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
	2	96	51	71	64	39	17	17	16	19	14
	6	653	46	27	19	15	11	7	11	15	11
	10	4	3	3	1	3	2	2	1	1	1
	14	3	1	2	1	1	1	1	1	1	1
	18	65	18	7	5	4	3	2	3	2	2
40000	22	55	29	22	17	13	4	6	5	6	5
	26	238	21	27	11	6	3	3	3	10	4
	30	6	5	4	2	2	1	1	1	1	1
	34	22	6	3	2	3	2	2	2	2	1
	38	149	3	10	7	3	3	2	2	3	2
	Sum	**1290**	**185**	**177**	**130**	**90**	**46**	**43**	**45**	**62**	**39**
	2	2368	1342	842	649	322	190	227	266	520	126
	6	1980	247	102	248	95	86	265	283	240	128
	10	6	2	2	1	1	1	1	1	1	1
	14	553	48	40	33	26	20	21	26	50	78
	18	10	3	2	3	3	2	2	6	5	6
50000	22	3139	292	359	183	84	136	71	92	105	258
	26	1075	104	48	53	28	15	42	71	53	49
	30	43	30	15	9	7	7	5	4	4	5
	34	285	313	115	86	93	89	152	68	46	55
	38	5190	788	327	240	118	123	104	62	202	124
	Sum	**14649**	**3169**	**1852**	**1504**	**777**	**668**	**890**	**878**	**1227**	**830**

Table 7.4: Parameter τ testing for the cut implementation applied to the Josefstadt instances with realistic profits.

	Times (in seconds) for various values for τ									
$C_{\max}$	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
1000	12	7	4	3	5	3	4	3	2	2
2000	314	55	98	81	75	36	72	109	81	108
3000	1557	1369	965	2663	2312	1033	202	1215	23468	3652

hour is taken which may not be optimal. In this cases the solution value and the CPU time are written in italic. Please note, that in some cases it was even not possible to find a feasible solution within the given time interval.

7.3.1 Comparison Test for Josefstadt, Kittsee and the Test Graph Instance

Our first comparison will be for the instances Josefstadt, Kittsee 1 – Kittsee 10 and test graph. In Table 7.5 the running times of three test runs are documented. The first column $d_{\max}$ of the table is the maximal distance a node, used in the solution walk, can have to the start/end node. This value mainly restricts the set of possible nodes and therefore also the set of possible arcs to use. The second column contains the model value $C_{\max}$ and the third column the name of the instance. The fourth column contains the number of possible arcs, that means the cardinality of the set of possible arcs including the artificial arc from the start node to the artificial end node counting duplicated arcs only once. As already mentioned this value mainly depends on $d_{\max}$ and is relevant for the size of the corresponding model. The fifth, seventh and ninth column, labeled with sol. contain the solution value returned by the corresponding implementation and the sixth, seventh and tenth column, labeled by time(s), contain the CPU time in seconds used by the corresponding implementation.

If one of the implementation needs less than 3600 seconds to solve an instance it clearly returns an optimal solution, but if the computation got aborted after one hour it can be the case that the returned solution is not optimal. For those cases the solution value and the CPU time are written in italic. If an implementation does not find a solution within one hour the solution column contains a hyphen. If the problem is infeasible the solution column contains inf.

To cover different situations the table contains three very different test runs. The first test run has relatively small $d_{\max} = 1200$ and $C_{\max} = 2400$ and therefore all instances can get solved within a few seconds. The second test has $d_{\max} = 2700$ and $C_{\max} = 10800$. This means the second test has $C_{\max} = 4d_{\max}$ in contrast to the other two tests which have $C_{\max} = 2d_{\max}$. That means the searched tour has at most 10800 meter length and is only allowed to use nodes which have a distance to the start node with at most 2700 meters. In this test run some instances can still be solved very fast but some instances already need more than one hour. The third test has very large values $d_{\max} = 9300$ and $C_{\max} = 18600$ and solving the instances needs for most of the instances longer than one hour. This has the effect that some of the solutions may not be optimal and we can compare which implementation found the better solution within one hour.

For the first test run with $d_{\max} = 1200$ and $C_{\max} = 2400$ the flow implementation can solve every instance in about 1 second and some of them even much faster. We can see that the cut implementation is slower compared to the flow implementation as it needs in 5 instances more than 1 second and for the Josefstadt instance it even needs half a minute. The running times of the mixed implementation are almost equal to the running times of the flow implementation.

Table 7.5: Implementation comparison with the Josefstadt, Kittsee and test graph instances.

				Flow		Cut		Mix	
$d_{\max}$	$C_{\max}$	Instance	#arcs	sol.	time(s)	sol.	time(s)	sol.	time(s)
1200	2400	Josefstadt	423	3034	1	3034	30	3034	2
		Kittsee 1	14	0	<1	0	<1	0	<1
		Kittsee 2	36	1664	<1	1664	<1	1664	<1
		Kittsee 3	233	2766	1	2766	10	2766	1
		Kittsee 4	184	2628	1	2628	5	2628	1
		Kittsee 5	191	2760	1	2760	4	2760	1
		Kittsee 6	30	1321	<1	1321	<1	1321	<1
		Kittsee 7	148	975	<1	975	4	975	<1
		Kittsee 8	24	0	<1	0	<1	0	<1
		Kittsee 9	19	0	<1	0	<1	0	<1
		Kittsee 10	1	inf	<1	inf	<1	inf	<1
		test graph	178	3068	<1	3068	<1	3068	<1
		Sum			**4**		**55**		**7**
2700	10800	Josefstadt	2288	14632	71	*12340*	*3600*	14632	44
		Kittsee 1	36	1029	<1	1029	<1	1029	<1
		Kittsee 2	196	10819	4	10819	262	10819	18
		Kittsee 3	674	13111	32	*8036*	*3600*	13111	29
		Kittsee 4	355	13564	1	13564	27	13564	1
		Kittsee 5	358	13738	8	13738	931	13738	10
		Kittsee 6	158	9721	1	9721	8	9721	2
		Kittsee 7	240	8075	<1	8075	<1	8075	<1
		Kittsee 8	36	0	<1	0	<1	0	<1
		Kittsee 9	207	8942	3	8942	56	8942	3
		Kittsee 10	22	2878	<1	2878	<1	2878	<1
		test graph	418	*12147*	*3600*	*12147*	*3600*	*12059*	*3600*
		Sum			**3719**		**11154**		**3706**
9300	18600	Josefstadt	5057	27013	414	-	*3600*	27013	52
		Kittsee 1	1916	14972	449	-	*3600*	-	*3600*
		Kittsee 2	4524	*21313*	*3600*	*9654*	*3600*	*14314*	*3600*
		Kittsee 3	4777	*23892*	*3600*	*15420*	*3600*	*17432*	*3600*
		Kittsee 4	3002	24930	2397	*16689*	*3600*	*23485*	*3600*
		Kittsee 5	2652	*24149*	*3600*	*16312*	*3600*	*22255*	*3600*
		Kittsee 6	2358	23547	703	*17656*	*3600*	*23547*	*3600*
		Kittsee 7	2614	*20104*	*3600*	*0*	*3600*	-	*3600*
		Kittsee 8	737	19113	111	*13485*	*3600*	19113	1776
		Kittsee 9	2872	*21647*	*3600*	*14123*	*3600*	*21647*	*3600*
		Kittsee 10	880	17404	465	*10135*	*3600*	17404	1879
		test graph	418	17862	90	17862	50	17862	135
		Sum			**22629**		**39650**		**32642**

The test run with $d_{\max} = 2700$ and $C_{\max} = 10800$ is already more interesting. The test graph instance cannot be solved by any implementation within one hour. The Flow and the cut implementation both find a solution with value 12147, but the mixed implementation does only find a solution with value 12059. For the other instances the flow implementation is clearly the fastest. Only for instance Kittsee 3 the mixed implementation needs 3 seconds less than the flow implementation which is a minimal difference. The cut implementation cannot solve the instances Josefstadt and Kittsee 3 and finds in both cases not the optimal solution which was computed by the other two implementations.

In the third test run with $d_{\max} = 9300$ and $C_{\max} = 18600$ we see again, that the flow implementation is clearly the fastest and if it cannot solve an instance within 1 hour it computes the best feasible solution compared to the other two implementations. Only for the instance Josefstadt the mixed implementation is faster than the flow implementation and for the instance test graph the Cut formulation is faster than the flow formulation. Interesting is also that the test graph instance could not get solved within one hour in the second test run with smaller $d_{\max}$ and $C_{\max}$ and in this test run it can get solved by all three implementations within a few minutes.

In summary, we can say that the flow implementation is clearly the fastest of the three implementation for these test instances. The mixed implementation is in general faster than the cut implementation for these instances.

7.3.2 Implementation Comparison with the Benchmark Instances

We will compare the three implementations on the one hand with the benchmark instances with random profits and on the other hand with the benchmark instances where the profits equal the costs.

Table 7.6 contains the test results for the benchmark instances with random profits. The table structure is the same as for 7.5, which was described in detail in 7.3.1 with the difference that there is no $d_{\max}$ column anymore since we fix for this test $d_{\max} = C_{\max}/2$.

We can see that the flow implementation can solve the instances with $C_{\max} = 40000$ within a few seconds and is slightly faster than the mixed implementation. The cut implementation is for some instances, for example start node 34 and 38, faster than the other two, but in total it is the slowest since it needs almost half a minute for the instance with start node 2.

Also for $C_{\max} = 60000$ the flow implementation is in total the fastest, but for half the instances the mixed implementation is slightly faster than the flow implementation. The cut implementation was not able to solve each instance within one hour and also for most of the other instances it was slower than the flow implementation and the mixed implementation. For the two instances which it could not solve in one hour it did not find the optimal solution.

Table 7.7 contains the test results for the benchmark instance where the profits equals the costs and it has the same structure as table 7.6. We can see that compared to the tests in table 7.6 most instances need more CPU time in the case where the profits equal the costs.

Table 7.6: Implementation comparison with the benchmark instances with random profits.

			Flow		Cut		Mix	
$C_{\max}$	**Start**	**#arcs**	**sol.**	**time(s)**	**sol.**	**time(s)**	**sol.**	**time(s)**
	2	345	146	3	146	28	146	4
	6	381	139	6	139	8	139	10
	10	297	188	1	188	1	188	1
	14	359	151	4	151	2	151	4
	18	345	110	1	110	1	110	3
40000	22	343	118	2	118	8	118	4
	26	428	139	3	139	6	139	8
	30	197	94	<1	94	2	94	1
	34	289	151	3	151	1	151	2
	38	389	138	2	138	1	138	4
	Sum			**27**		**56**		**41**
	2	703	238	34	238	1316	238	38
	6	1053	246	152	*238*	*3600*	246	138
	10	609	289	21	289	20	289	7
	14	794	230	118	230	3577	230	175
	18	784	186	62	186	122	186	30
60000	22	735	186	55	186	809	186	47
	26	1089	246	77	*218*	*3600*	246	69
	30	483	163	12	163	145	163	21
	34	561	230	50	230	308	230	78
	38	811	213	21	213	39	213	49
	Sum			**602**		**13536**		**652**

Also in 7.7 we see that the flow implementation is in most situations the fastest. For $C_{\max} = 40000$ it is for all instances the fastest implementation except the instance with start node 6, where the mixed implementation is faster and the instance with start node 30, where the cut implementation is slightly faster. For $C_{\max} = 60000$ the mixed implementation is faster for the instances with the start nodes 2, 26, 30 and 38, but in total the flow implementation is again the fastest.

For some instances in 7.7 we can observe that two implementations return different solution values although both terminated within one hour and should return an optimal value. This behavior is caused by the relative gap tolerance of CPLEX. With this tolerance the MIP solver returns a feasible solution if the relative difference between its value and the global upper bound during branch-and-bound is smaller than the relative gap tolerance parameter. The default value for this parameter is 10^{-4}. In our case that means that the algorithm could return a solution which is not optimal but the relative difference to the optimal value is smaller than 10^{-4}. If we consider $C_{\max} = 40000$ the optimal solution will be close to 40000 and therefore every feasible solution gets accepted as optimal if it has a value bigger than $o - 40000 \cdot 10^{-4} = o - 4$, where o is the optimal

Table 7.7: Implementation comparison with the benchmark instances with profits equal the costs.

			Flow		Cut		Mix	
$C_{\max}$	**Start**	**#arcs**	**sol.**	**time(s)**	**sol.**	**time(s)**	**sol.**	**time(s)**
40000	2	345	39998	51	39998	1752	39998	388
	6	381	39996	39	39996	1223	39996	13
	10	297	39976	51	39976	665	39976	180
	14	359	39994	201	39994	2791	39994	420
	18	345	39993	45	39993	248	39993	61
	22	343	39983	114	39983	1452	39983	273
	26	428	39996	126	39996	1040	39996	170
	30	197	39992	4	39992	3	39992	5
	34	289	39997	51	39998	264	39997	227
	38	389	39995	213	*39995*	*3600*	39995	282
	Sum			**893**		**13038**		**2019**
60000	2	703	60000	192	*59983*	*3600*	60000	105
	6	1053	59999	260	*59974*	*3600*	59994	2602
	10	609	59999	64	*59970*	*3600*	59998	509
	14	794	59997	95	*59933*	*3600*	59998	394
	18	784	59998	300	59995	1945	59998	3411
	22	735	59994	3	*59994*	*3600*	59995	2844
	26	1089	59997	448	*59779*	*3600*	59999	375
	30	483	59999	106	59999	229	59998	54
	34	561	59994	140	59998	1172	60000	146
	38	811	59997	612	*59979*	*3600*	59995	45
	Sum			**2220**		**28546**		**10485**

solution value. We get that the absolute difference between a solution value and the real optimal solution value could be up to 4. We see this in the test instance with Start node 34, where the Flow and the mixed implementation return a solution value of 39997 and the cut implementation returns a solution value of 39998. For $C_{\max} = 60000$ the difference can even be 6 and we can observe that for all instances with $C_{\max} = 60000$ the solution values between at least two implementations differ, although both terminated in under one hour.

For the real world applications this issue will not be so important, since two routes where the total profit only differs by less than 0.01% can be considered to have the same attractiveness since attractiveness values can never be so exact. If we still want to avoid such problems we can lower the relative gap tolerance but then the algorithms will need much more computation time. We will consider this situation in section 7.4.1.

Summarizing we can say that also for the benchmark instances the flow implementation is the fastest and the mixed implementation is the second fastest.

Table 7.8: Implementation comparison with the Josefstadt instance with real profits

			Flow		**Cut**		**Mix**	
$d_{\max}$	$C_{\max}$	#arcs	**sol.**	**time(s)**	**sol.**	**time(s)**	**sol.**	**time(s)**
200	400	72	319	<1	319	<1	319	<1
200	800	113	586	<1	586	<1	586	<1
300	600	119	533	<1	533	1	533	<1
300	1200	144	846	<1	846	<1	846	<1
500	1000	216	783	<1	783	3	783	<1
500	2000	237	1217	<1	1217	1	1217	1
800	1600	424	1406	1	1406	13	1406	1
800	3200	459	2382	3	2382	21	2382	4
1200	2400	665	2159	1	2159	54	2159	6
1200	4800	716	3449	26	3449	2508	3449	78
1800	3600	1408	3331	27	*-1707*	*3600*	3331	86
1800	7200	1505	6199	109	-	*3600*	6199	28
2700	5400	3033	5213	1184	*2547*	*3600*	*4041*	*3600*
2700	10800	3167	*9766*	*3600*	*1642*	*3600*	*9638*	*3600*
4100	8200	5535	8105	2389	*-23*	*3600*	8105	1247
4100	16400	5575	-	*3600*	-	*3600*	-	*3600*
6200	12400	5834	*11315*	*3600*	*2428*	*3600*	*11301*	*3600*
6200	24800	5834	-	*3600*	-	*3600*	-	*3600*
Sum				**18141**		**31402**		**19452**

7.3.3 Implementation Comparison with the Josefstadt Instance with Real Profits

To compare the three implementations also for a realistic instance, we apply all three implementations to the Josefstadt instance with real profits for different values $d_{\max}$ and $C_{\max}$. Table 7.8 contains the resulting solutions and CPU times. It has the same structure as Table 7.5 which was described in detail in section 7.3.1 except that we only deal with the Josefstadt instance with real profits now.

As we can see in total the flow implementation is again the fastest. But there are two situations where the mixed implementation is much faster, the situation $d_{\max} = 1800$ and $C_{\max} = 720$ and the situation $d_{\max} = 4100$ and $C_{\max} = 8200$. As we can see for $d_{\max} = 6200$ the graph contains all 5834 arcs, that is the complete graph of the instance Josefstadt. We see that for $d_{\max} = 4100$ and $C_{\max} = 16400$ and for $d_{\max} = 6200$ and $C_{\max} = 24800$ all three implementations do not find any feasible solution within one hour. In the cases where the three implementation do not terminate within one hour we can see that the flow implementation always finds a good solution which is at most as good as the solutions the other two implementations find.

7.3.4 Conclusion

In all tests presented in the last three sections we saw that in total the flow implementation is the fastest and also returns the best feasible solutions if it is not able to solve it in one hour. Because of that we will further investigate the flow implementation in the following sections.

7.4 AOP and CTPP Benchmarks

In this section we will compare our flow implementation with the implementations presented in [22] and [25].

As we already saw in Section 7.3.2 it can happen that the flow implementation does not find the best solution if its value is very close to the value of the best solution. This is only a problem for the benchmark instances where the profits equal the costs since these instances have profits which are not integral and relatively big compared to the instances with random profits. To get better results for this instances we would have to set the relative gap tolerance parameter to a smaller value. Therefore we will use for the tests two kinds of flow implementations and call them Flow1 and Flow2. Flow1 is the flow implementation with the default relative gap tolerance of 10^{-4} and Flow2 is the flow implementation when we set the default relative gap tolerance to 10^{-5}.

7.4.1 Benchmark Instances where the Profits equal the Costs

Table 7.9 contains the test results of the benchmark instances where the profits equal the costs for $C_{\max} = 20000$, $C_{\max} = 40000$ and $C_{\max} = 60000$. It compares our implementations Flow1 and Flow2 with the CPLEX implementation of [22] which is called CPLEX1 in the table, the GRASP implementation of [22], the CPLEX implementation of [25] which is called CPLEX2 in the table and the ILS implementation of [25] (see Section 4.3.3 and Section 4.3.4). For the results it is important to know that CPLEX1 has also an one hour time limit like our implementations and CPLEX2 has no time limit.

Since CPLEX1 and GRASP are implementations for the AOP and not for the CTPP it can happen, that our implementations, CPLEX2 or ILS find solutions which are not feasible in AOP because they use a vertex more than once. Thus the solution values of CPLEX1 and GRASP may be smaller than the solution values of the other implementations although they are optimal. There are some cases where CPLEX1 returns different optimal solutions than the other implementations. The authors of [25] supposed that this may be because the implementations of [22] assume a symmetric graph structure and therefore work on a slightly different graph.

Since GRASP and ILS both only need about one second for each instance we do not list their CPU times in 7.9. As always the CPU times in 7.9 are in seconds.

We can observe that the running times of our implementations Flow1 and Flow2 are comparable with the running times of CPLEX1 and much faster than the running times of CPLEX2. CPLEX2 was not even able to solve the test instances with $C_{\max} = 60000$. For $C_{\max} = 40000$ and $C_{\max} = 60000$ we see that Flow1 returns sometimes a non-optimal

Table 7.9: Benchmark tests with the benchmark instances where the profits equal the costs for $C_{\max}2 = 0000$, $C_{\max} = 40000$ and $C_{\max} = 60000$. The CPU times are given in seconds if not other specified.

		Flow1		**Flow2**		**CPLEX1**		**GRASP**	**CPLEX2**		**ILS**
$C_{\max}$	**start**	**sol.**	**CPU**	**sol.**	**CPU**	**sol.**	**CPU**	**sol.**	**sol.**	**CPU**	**sol.**
20000	2	19497	<1	19497	<1	19495	1	19495	19497	22	19497
	6	18778	<1	18778	<1	17405	2	15874	18778	109	18778
	10	19711	<1	19711	<1	19712	1	19712	19711	127	19711
	14	19918	<1	19918	<1	19918	1	19918	19918	178	19918
	18	19602	<1	19602	<1	19602	1	19602	19602	190	19602
	22	19564	<1	19564	<1	19565	1	19565	19564	221	19564
	26	19919	<1	19919	<1	19871	1	19871	19919	251	19919
	30	inf	<1	inf	<1	inf	1	inf	inf	-	inf
	34	19944	<1	19944	<1	19943	1	19943	19944	326	19944
	38	19132	<1	19132	<1	19131	28	19131	19132	374	19132
40000	2	39998	51	39998	355	40000	100	39948	39998	420	39998
	6	39996	39	39996	79	39997	205	39930	39996	537	39996
	10	39976	51	39976	57	39976	43	39941	39976	4.2h	39976
	14	39994	201	39994	212	39987	210	39970	39994	21h	39994
	18	39993	45	39993	50	39706	29	39706	39993	26h	39993
	22	39983	114	39983	121	39982	100	39978	39982	37h	39982
	26	39996	126	39996	215	39997	226	39997	39996	37h	39996
	30	39992	4	39992	5	39992	13	39571	39992	38h	39992
	34	39997	51	39999	248	39998	65	39932	39999	38h	39999
	38	39995	213	39995	237	39994	2725	39967	39995	48h	39995
60000	2	60000	192	60000	209	*59997*	*3600*	59980	-	-	60000
	6	59999	260	*59999*	*3600*	*59984*	*3600*	59982	-	-	60000
	10	59999	64	*59999*	*3600*	*59999*	*3600*	59989	-	-	60000
	14	59997	95	*59999*	*3600*	*59997*	*3600*	59997	-	-	60000
	18	59998	300	60000	2782	*59992*	*3600*	59973	-	-	60000
	22	59994	3	60000	130	*59999*	*3600*	59913	-	-	60000
	26	59997	448	*59997*	*3600*	*59990*	*3600*	59988	-	-	60000
	30	59999	106	59999	1515	*59991*	*3600*	59799	-	-	59999
	34	59994	140	60000	1670	60000	215	59993	-	-	60000
	38	59997	612	*59999*	*3600*	*59962*	*3600*	59992	-	-	60000

Table 7.10: Benchmark tests with the benchmark instances where the profits equal the costs for $C_{\max} = 80000$ and $C_{\max} = 100000$. The CPU times are given in seconds if not other specified.

		Flow1		**Flow2**		**CPLEX1**		**GRASP**	**ILS**
$C_{\max}$	**start**	**sol.**	**CPU**	**sol.**	**CPU**	**sol.**	**CPU**	**sol.**	**sol.**
80000	2	79993	37	79999	1257	*79998*	*3600*	79974	80000
	6	79998	275	*79998*	*3600*	*79650*	*3600*	79977	80000
	10	79994	45	79999	1719	*79992*	*3600*	79997	80000
	14	80000	306	80000	330	*79804*	*3600*	79989	80000
	18	79996	437	*79999*	*3600*	*79654*	*3600*	79943	80000
	22	79996	127	80000	2927	*79991*	*3600*	79969	80000
	26	79998	1332	*79998*	*3600*	*79908*	*3600*	79983	80000
	30	79993	108	80000	2428	*79984*	*3600*	79977	80000
	34	79999	17	*79999*	*3600*	*79998*	*3600*	79977	80000
	38	79992	1096	*79995*	*3600*	*79481*	*3600*	79882	80000
100000	2	99996	68	100000	210	*99893*	*3600*	99992	100000
	6	99994	527	99999	1401	*99872*	*3600*	99989	100000
	10	99994	202	100000	1539	*99891*	*3600*	99952	100000
	14	99994	333	100000	2108	*99766*	*3600*	99965	100000
	18	99998	50	99999	546	*99798*	*3600*	99998	100000
	22	99991	265	100000	913	*99908*	*3600*	99997	100000
	26	99996	712	*99999*	*3600*	*11355*	*3600*	99917	100000
	30	99992	692	*99998*	*3600*	*99968*	*3600*	99884	100000
	34	99995	105	*99995*	*3600*	*74576*	*3600*	99998	100000
	38	99995	1081	*99998*	*3600*	*58225*	*3600*	99912	100000

solution and Flow2 does find a better solution but needs more time. This is caused again by the fact, that the relative difference betweeen the optimal solution and the solution of Flow1 is smaller than the default relative gap tolerance of CPLEX, which is 10^{-4}.

Compared to the solutions of GRASP we see that our implementations already find better solutions for $C_{\max} = 40000$ and $C_{\max} = 60000$. ILS always finds the optimal solution for almost all values of $C_{\max}$ and therefore it returns slightly better solutions for $C_{\max} = 60000$ than our implementations.

Table 7.10 contains the same tests as table 7.9 for $C_{\max} = 80000$ and $C_{\max} = 100000$. Since CPLEX2 did not solve any of these instances we omit this columns.

We can see in 7.10 that Flow1 is much faster than Flow2 and CPLEX1 but returns not always optimal solutions. Flow2 is for some cases faster than CPLEX1 which is not able to solve any instance under one hour. Also all returned solutions of Flow2 are always better than the solutions of CPLEX1. ILS can solve all instances optimally and therefore for the most cases better than our implementations.

For $C_{\max} = 80000$ and start node 2 or start node 10 we see that Flow2 terminates in under one hour and returns solutions with values about 79999, which are not optimal.

This is because 10^{-5} relative gap tolerance can lead to absolute differences of almost 1. The same happens for $C_{\max} = 100000$ and start node 6 or start node 18. To avoid this we would have to use an even smaller relative gap tolerance which would increase the running times again.

7.4.2 Benchmark Instances with Random Profits

Table 7.11 contains the test results for the benchmark instances with random profits. Since the profits are here integral and relatively small we do not need a lower relative gap tolerance and therefore we omit the results for Flow2. Since these instances with random profits were introduced in [25] we also have no test results with the implementations of [22]. The table has new columns gap for Flow1 and CPLEX2, which contain the gaps calculated by CPLEX and a new column diff which contains the relative difference between the solution of ILS and the solution computed by our implementation. Since the results of $C_{\max} = 20000$ are not very interesting (our implementation solves all instances in under one second) we omit them in this table.

The CPLEX implementation of [22] could not solve the instances for $C_{\max} = 80000$ and $C_{\max} = 100000$. For these instances we can see that our flow implementation is very fast. It could solve all instances except one instance to optimality in under one hour. The instance with $C_{\max} = 100000$ and start node 26 could not be solved in one hour but a solution was found which is better than the computed solution by ILS and has a worst case gap of 5%. If we apply our implementation with no time limit, then it solves the instance with $C_{\max} = 100000$ and start node 26 in two hours with an optimal value of 414. Therefore the real gap between the found solution and the optimal solution is only 1%.

Especially for $C_{\max} = 100000$ our flow implementation could find much better solutions than the ILS implementation with up to 25% difference. Also the running times for $C_{\max} = 40000$ and $C_{\max} = 60000$ of our flow implementation is much better than the running times of CPLEX2.

7.4.3 Conclusion

It is difficult to really compare our implementations with the implementations from [22] and [25] since we used different hardware for our tests. But besides that we saw that our flow implementation can easily hold up with the CPLEX implementations provided by [22] and [25].

For most instances our implementation computed also better solutions than the GRASP implementation from [22], although we have to mention that the GRASP implementation used only around 1 second per instance and our implementation used up to 1 hour which is a factor of 3600.

If we compare our implementation with the ILS we see that in the case where the profits equal the costs the ILS is very efficient and computes in a second almost an at least as good solution as our implementation computed in one hour. In the case where the profits are random the situation changes. The ILS is not able anymore to compute

Table 7.11: Benchmark tests with benchmark instances with random profits. The CPU times are given in seconds if not other specified.

		Flow1			**CPLEX2**			**ILS**	
C_{max}	**start**	**sol.**	**gap**	**CPU**	**sol.**	**gap**	**CPU**	**sol.**	**diff**
40000	2	146	0%	3	146	0%	231	146	0%
	6	139	0%	6	139	0%	1.2h	139	0%
	10	188	0%	1	188	0%	1.2h	188	0%
	14	151	0%	4	151	0%	1.4h	151	0%
	18	110	0%	1	110	0%	2362	110	0%
	22	118	0%	2	118	0%	1.1h	118	0%
	26	139	0%	3	139	0%	1.2h	139	0%
	30	94	0%	<1	94	0%	1.4h	94	0%
	34	151	0%	3	151	0%	1.7h	151	0%
	38	138	0%	2	138	0%	1.9h	138	0%
60000	2	238	0%	34	238	0%	577	238	0%
	6	246	0%	152	246	0%	1.4h	246	0%
	10	289	0%	21	289	0%	1.5h	289	0%
	14	230	0%	118	230	0%	3.2h	229	0.4%
	18	186	0%	62	186	0%	61h	186	0%
	22	186	0%	55	186	0%	103h	184	1.1%
	26	246	0%	77	246	0%	103h	228	7.3%
	30	163	0%	12	163	0%	116h	163	0%
	34	230	0%	50	230	0%	124h	230	0%
	38	213	0%	21	213	0%	170h	213	0%
80000	2	337	0%	102	-	-	-	305	9.5%
	6	329	0%	1413	-	-	-	307	6.7%
	10	396	0%	41	-	-	-	331	16.4%
	14	341	0%	150	-	-	-	275	19.4%
	18	257	0%	349	-	-	-	238	7.4%
	22	271	0%	65	-	-	-	252	7%
	26	329	0%	536	-	-	-	308	6.4%
	30	241	0%	132	-	-	-	241	0%
	34	339	0%	62	-	-	-	324	4.4%
	38	310	0%	230	-	-	-	310	0%
100000	2	437	0%	178	-	-	-	347	20.6%
	6	430	0%	1232	-	-	-	368	14.4%
	10	480	0%	201	-	-	-	393	18.1%
	14	439	0%	597	-	-	-	328	25.3%
	18	342	0%	1419	-	-	-	312	8.8%
	22	346	0%	192	-	-	-	326	5.8%
	26	*410*	*5%*	*3600*	-	-	-	379	7.6%
	30	353	0%	104	-	-	-	338	4.2%
	34	445	0%	138	-	-	-	364	18.2%
	38	397	0%	2814	-	-	-	385	3.0%

the optimal solutions and our implementation can compute the optimal solution almost for all instances. Although we have to mention again that ILS only uses one second per instance and our implementation uses up to one hour.

7.5 Real World Applicability

In this chapter we want to evaluate if our flow implementation is applicable in realistic situations as we motivated them in Chapter 1. That means we want to check if it is possible to compute useful recreational bicycle tours. By visualizing the tours we get a feeling how a solution looks like and if it is really an attractive tour.

As we already saw in the previous sections the maximal distance of the nodes in the route to the end node and the maximal tour length are crucial for the calculation speed and, if we set a time limit, also for the solution quality after this time limit.

Another factor for the computation time is the area we are searching for the tour. If we are on the countryside like for example in Kittsee it is clear that there are less streets than in a city and therefore we will be able to compute, in the same time, larger routes than in a city.

Clearly the length, a route should have, strongly depends on the person who wants to use it. People with special needs may not need as long tours as a professional racing cyclist would do. For some people routes around 5 km to 10 km will be enough for a recreational tour, on the other hand a racing cyclist would need tours up to 100 km or more.

7.5.1 Tour Planning in Josefstadt with Realistic Profits

Since we have realistic attractiveness values for the map around Josefstadt, it makes sense to further investigate the routes for this instance.

We already saw in table 7.8 the computation times of the flow implementation for different tour lengths. As we can see tours up to 10 km were computed exactly within one hour. For tours larger than 10 km it was not able to solve them exactly in one hour, but in some cases an approximation value got computed. For the instance where $d_{\max} = 2700$ and $C_{\max} = 10800$ the solution gap calculated by CPLEX was 1.67% and for the instance where $d_{\max} = 6200$ and $C_{\max} = 12400$ the solution gap calculated by CPLEX was 4.68%. That means in both cases the implementation computed a solution within 5% of the optimal solution. To summarize we can say that for routes up to 13 km the implementation returns a good solution within one hour and therefore is suitable for smaller routes up to 13 km, which could be enough for people with special needs, but definitely not for racing cyclists.

In Figure 7.1 we can see a route of 8 km length, starting and ending at the town hall of Vienna. We can see that most of the time the streets are only used once. Only three very small street parts got used twice. In the left upper corner we see the part of the route, which is marked by a green rectangle, with a higher degree of enlargement. There we can see a very small street part, colored red, which is used twice. Since this and also

the other two twice used parts are only parts of an intersection this does not really lower the attractiveness of the tour. We can also see on the bottom that a part of the Ring Road ("Ringstraße") is used in both directions. This is because in the card material this two parts were not opposite arcs and therefore the usage of both directions got not penalized. But since this are both cycleways on the left and on the right side, this does not really lower the attractiveness of the tour. If we would want to avoid something like this we would have to increase the penalties between the corresponding arcs.

To check if this tour is now really attractive in the sense of our given attractiveness-measurement Figure 7.2 shows the same tour together with all streets of the instance Josefstadt. The color of every street represents its attractiveness value and the thick lines represent the tour. As we can see the tour itself only uses green roads and avoids red roads.

7.5.2 Tour Planning in Kittsee

In this section we want to test the applicability of our implementation to route planning for patients of a Rehabilitation Center in Kittsee. Therefore we use as starting and ending point always the Rehabilitation Center.

Table 7.12 contains the computation times of the Kittsee instance with the Rehabilitation Center as start and end node for various tour lengths and maximal distances. As we can see all tours up to a length of 25 km could get solved optimally within one hour. Longer tours were either solved within a 5% gap or no solution was found at all. But at least for all instances up to 60 km tour length the implementation found a solution with at most 5% gap.

If we assume that almost no patient in a Rehabilitation Center needs a bicycle tour with length longer than 60 km we can precalculate routes for every patients needs within a short time. If we want to get the optimal solutions for the longer tours around 40 km, we would need more than one hour calculation time.

In figure 7.3 a tour with 20 km length is shown. We can see that there are nodes which are used twice, but no arcs. Since the profits of the arcs in the Kittsee instance are artificial and have nothing to do with the reality, this route is only for visualization and may not be attractive in reality at all.

As discussed in chapter 1 there may be patients who want a route such that they can stop the route at any point and return to the start point within a short time. To test the applicability of our implementation to this situation we searched a route starting and ending at the Rehabilitation Center with a length of again 20 km, but now every node in the route should have a maximal distance of 2 km to the Rehabilitation Center. Compared to the tour in 7.3 this tour does not look anymore like a round trip. It uses some nodes twice but it still does not use arcs twice, except some very small arcs.

7.5.3 Conclusion

We saw in the last two sections that applicability in terms of computation time strongly depends on the map. In a city like Vienna the implementation may be only applicable for

Table 7.12: Route calculations starting and ending in a Rehabilitation Center in Kittsee.

$d_{\max}$	$C_{\max}$	#arcs	solution	time(s)	gap
200	400	37	420	<1	0%
200	800	37	569	<1	0%
300	600	59	533	<1	0%
300	1200	63	1189	<1	0%
500	1000	135	1065	<1	0%
500	2000	135	2141	2	0%
800	1600	251	1739	1	0%
800	3200	253	3497	2	0%
1200	2400	507	2718	5	0%
1200	4800	511	5390	17	0%
1800	3600	625	4232	12	0%
1800	7200	625	8185	60	0%
2700	5400	761	6640	10	0%
2700	10800	771	12979	13	0%
4100	8200	1188	10162	184	0%
4100	16400	1190	20385	76	0%
6200	12400	1892	17306	99	0%
6200	24800	1899	33595	446	0%
9300	18600	3026	25521	1017	0%
9300	37200	3030	48895	1406	0%
14000	28000	5186	37363	3600	3.53%
14000	56000	5192	74145	3600	1.71%
21000	42000	8939	59692	3600	4.29%
21000	84000	8951	118481	3600	2.58%
31500	63000	15516	-	3600	-
31500	126000	15518	-	3600	-
47300	94600	15665	135131	3600	2.99%

routes up to 10 km or 15 km and on the countryside like in Kittsee it may be applicable for routes up to 60 km or even more. For the purpose of a Rehabilitation Center this could be enough, but for a professional racing cyclist it may not be applicable.

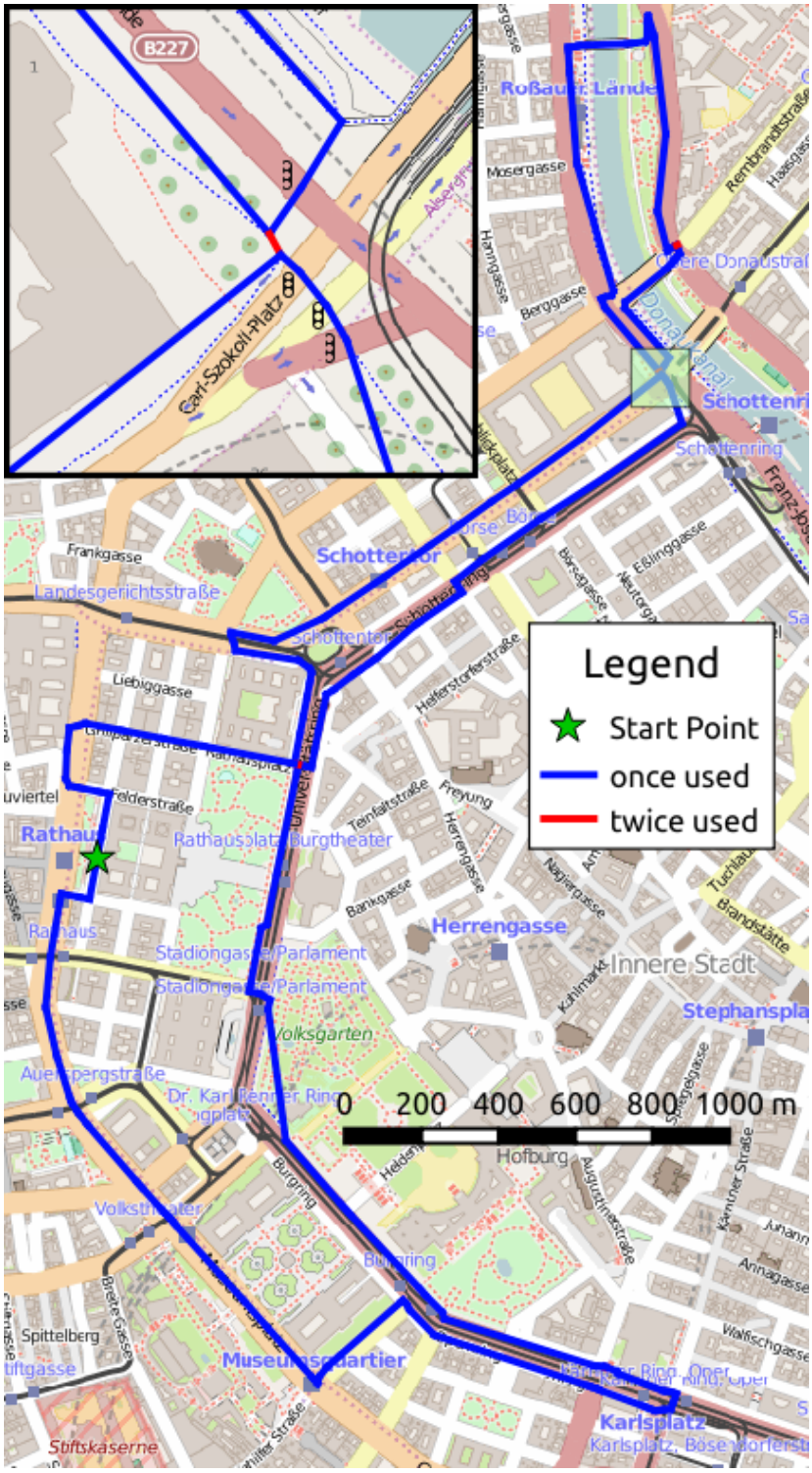

Figure 7.1: A tour of 8 km length, starting and ending at the town hall of Vienna. (©OpenStreetMap contributors)

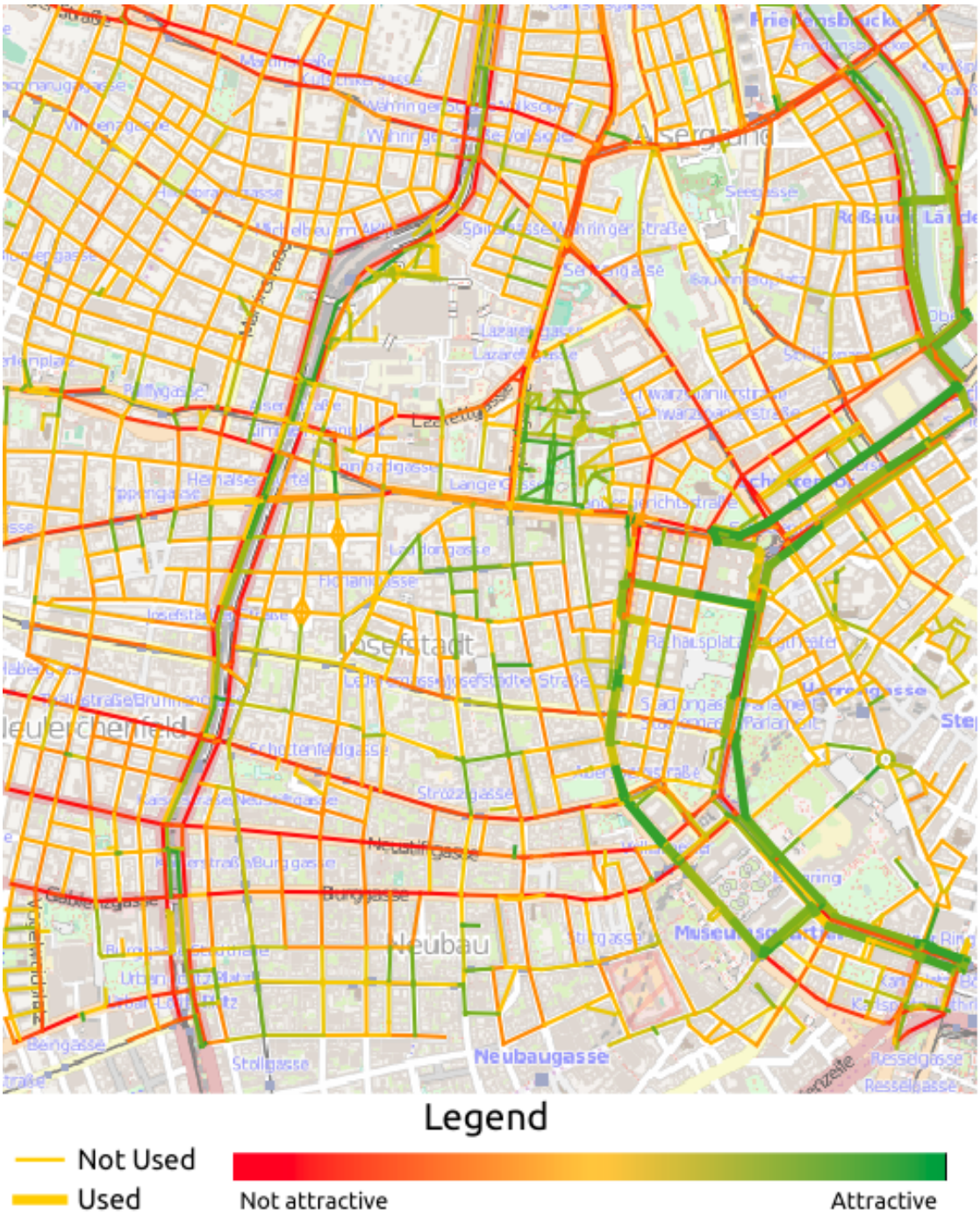

Figure 7.2: All streets of the instance Josefstadt colored according to their attractiveness values. The thick lines represent the 8 km tour from Figure 7.1. (©OpenStreetMap contributors)

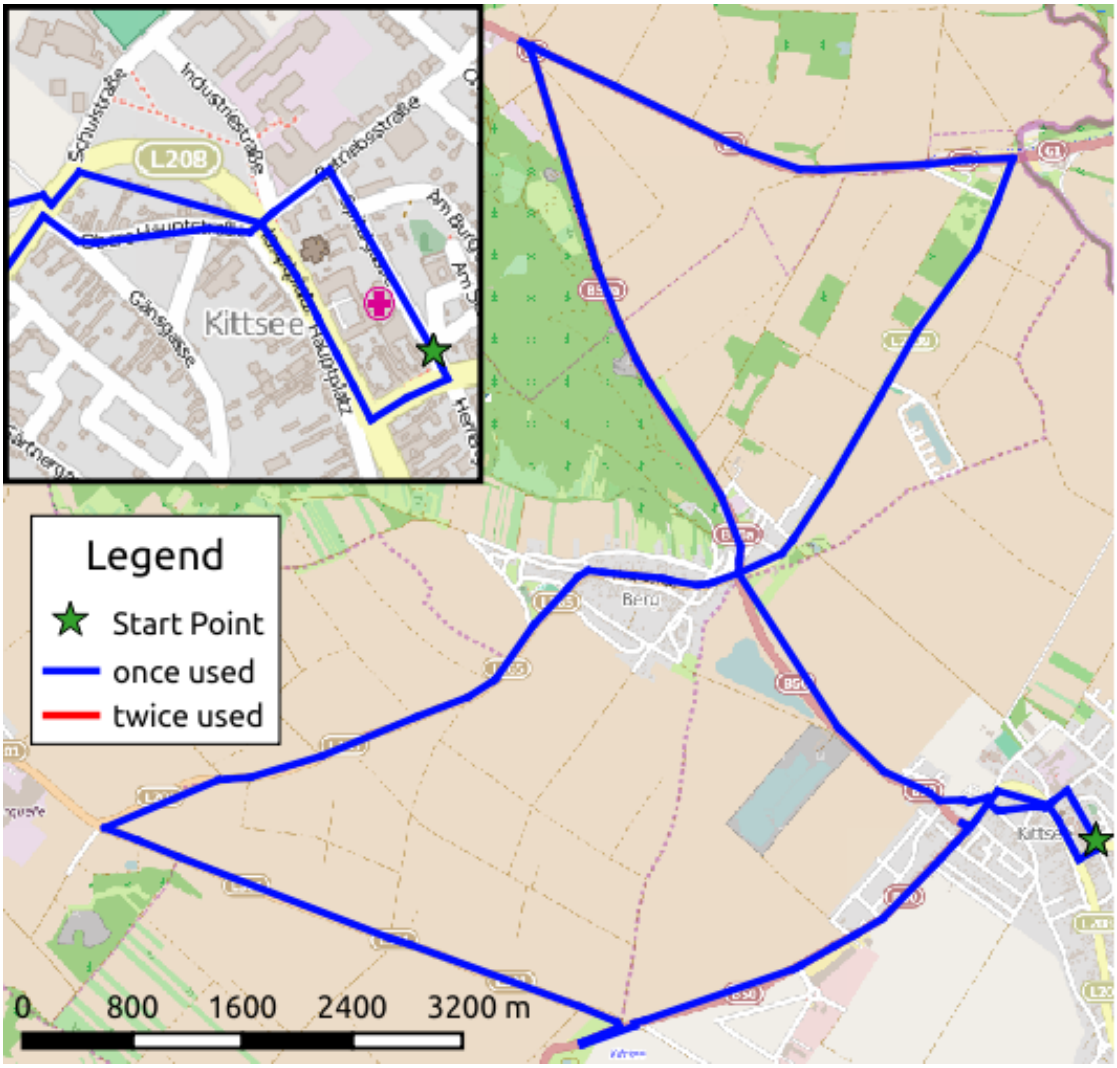

Figure 7.3: Tour starting and ending at a Rehabilitation Center in Kittsee with 20 km length. (©OpenStreetMap contributors)

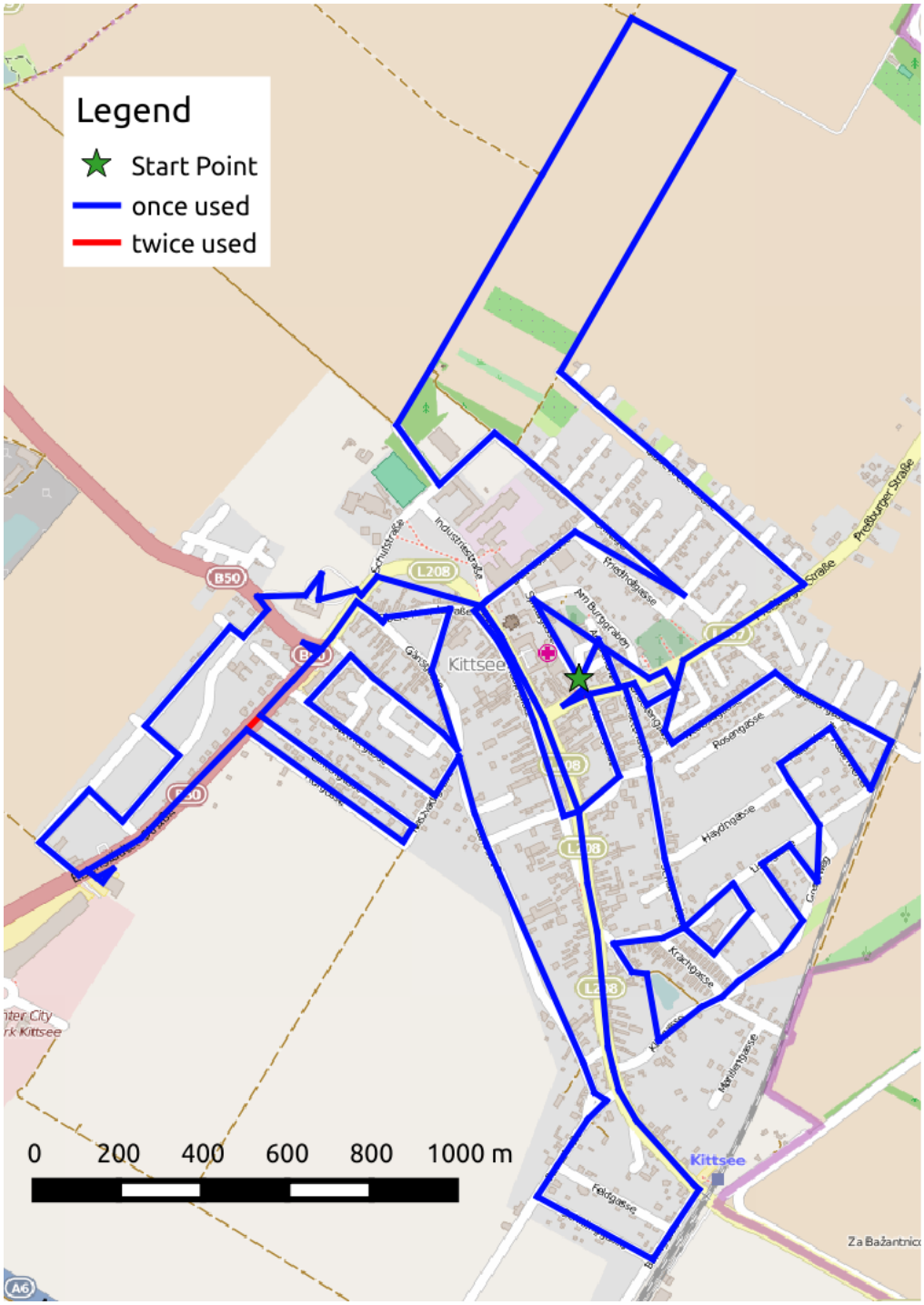

Figure 7.4: Tour starting and ending at a Rehabilitation Center in Kittsee with 20 km length only consisting of nodes with distance smaller or equal 2 km to the Rehabilitation Center. (©OpenStreetMap contributors)

CHAPTER 8

Conclusion

8.1 Summary

At the beginning of the thesis we formulated a mathematical problem for the task to find an attractive cycle route starting and ending at some fixed point. The formulation is a maximization problem on a directed multigraph, where the attractiveness gets represented by profits of the arcs and the length gets represented by the costs of the arcs. The goal is to maximize the attractiveness under the restriction to not exceed the maximal tour length. We allow to use streets in the same direction twice but penalize it by decreasing the profit.

Through transformations we can bring our problem into an easier to handle problem, where every arc can only be used once and the start and the end node are different. To simplify the transformations we introduce a new dependency relation which can be used to state that one arc can only be used in a tour if another arc is also used, that means one arc depends on another arc. All various versions of the problem formulation are NP-equivalent, or in other words, their decision problems are NP-complete.

To exactly solve the problem, we propose three different mixed integer linear programming formulations. The three programs only differ in their sub tour elimination constraints. The first one is based on a classical cut formulation, which can then be solved with branch-and-cut. The second one is based on a flow formulation and the third one uses both sub tour eliminations together and can be solved with branch-and-cut. The relaxations of the first and the second formulation are not comparable and therefore the relaxation of the third formulation is stronger than the relaxations of the first and the second formulations.

To test the algorithms we implemented the three solution approaches with C++, using the CPLEX technology. We used four different graphs as testing instances. The first graph is just a test graph with no real meaning, the second graph represents an area around Josefstadt in Vienna, the third graph an area around Kittsee in Austria and the

fourth graph an area around East-Flanders in Belgium. The latter was already used by other papers to test algorithms for similar problems.

After testing some parameters for the mixed integer program formulations, we compare the three programs by their running times on all four test graphs. The result is that in most cases the flow formulation is the fastest and therefore we use this formulation for further testing. By applying our implementation to a similar problem, we compare it with algorithms from other papers. At least for the tested instances our implementation is faster than their exact approaches and in some situations it also produces within a short time better solutions than their approximation approaches. We also test the applicability of our implementation for real world situations. The applicability strongly depends on where we search a route and how long the route should be. The test results show that on the countryside our algorithm is applicable for routes up to 60 km and in a big city it is applicable for routes up to 13 km.

8.2 Limitations

Since the proposed algorithms in this thesis are all exact algorithms of an NP-hard problem, one main limitation is, that the computation time will exponentially blow up for big graphs. We see that in our test results when we try to compute larger routes. Then the implementations do not find routes at all or only find approximations within a reasonable time.

8.3 Further Work

To use the algorithms proposed in this thesis in praxis, we need map material with realistic attractiveness values. Since this work does not cover the preprocessing of map material this could be done in a future work. Furthermore, not only attractiveness values should be provided, but also realistic penalties, such that the resulting solutions really correspond to the most attractive routes. Since attractiveness is always subjective, this may be hard to verify and will need statistical analysis tools.

To further improve the solving speed and solution quality one could combine the proposed exact algorithms from this thesis with heuristics. The heuristics proposed in [22] and [25] could be a good starting point for that.

Bibliography

[1] IBM Knowledge Center - What are user cuts and lazy constraints? http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.5.1/ilog.odms.cplex.help/CPLEX/UsrMan/topics/progr_adv/usr_cut_lazy_constr/02_defn.html. Accessed: 2015-04-21.

[2] OpenStreetMap. https://www.openstreetmap.org/about. Accessed: 2015-04-21.

[3] David L. Applegate. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[4] C. Archetti and M. G. Speranza. Arc routing problems with profits. Technical report, Working paper, Department of Economics and Management, University of Brescia, Italy, 2013.

[5] Julián Aráoz, Elena Fernández, and Oscar Meza. Solving the prize-collecting rural postman problem. *European Journal of Operational Research*, 196(3):886–896, 2009.

[6] Julián Aráoz, Elena Fernández, and Cristina Zoltan. Privatized rural postman problems. *Computers & Operations Research*, 33(12):3432–3449, 2006.

[7] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 2013.

[8] Boris V. Cherkassky and Andrew V. Goldberg. On implementing push-relabel method for the maximum flow problem. In Egon Balas and Jens Clausen, editors, *Integer Programming and Combinatorial Optimization*, number 920 in Lecture Notes in Computer Science, pages 157–171. Springer Berlin Heidelberg, 1995.

[9] Ángel Corberán, Isaac Plana, Antonio M. Rodríguez-Chía, and José M. Sanchis. A branch-and-cut algorithm for the maximum benefit Chinese postman problem. *Mathematical Programming*, 141(1-2):21–48, 2013.

[10] Dominique Feillet, Pierre Dejax, and Michel Gendreau. Traveling Salesman Problems with Profits. *Transportation Science*, 39(2):188–205, May 2005.

[11] Matteo Fischetti, Juan Jose Salazar Gonzalez, and Paolo Toth. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 10(2):133–148, 1998.

[12] Jonathan L. Gross, Jay Yellen, and Ping Zhang, editors. *Handbook of Graph Theory, Second Edition*. Chapman and Hall/CRC, Boca Raton, 2 edition edition, 2013.

[13] Chryssi Malandraki and Mark S. Daskin. The maximum benefit Chinese postman problem and the maximum benefit traveling salesman problem. *European Journal of Operational Research*, 65(2):218–234, 1993.

[14] R. Garey Michael and S. Johnson David. Computers and intractability: a guide to the theory of NP-completeness. *WH Freeman & Co., San Francisco*, 1979.

[15] W. L. Pearn and K. H. Wang. On the maximum benefit Chinese postman problem. *Omega*, 31(4):269–273, 2003.

[16] Wen-Lea Pearn and W. C. Chiu. Approximate solutions for the maximum benefit Chinese postman problem. *International journal of systems science*, 36(13):815–822, 2005.

[17] César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research*, 211(3):427–441, June 2011.

[18] Gerhard Reinelt. *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag, 1994.

[19] Alexander Shekhovtsov and Vaclav Hlavac. A Distributed Mincut/Maxflow Algorithm Combining Path Augmentation and Push-Relabel. In *Proceedings of the 8th International Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition (EMMCVPR)*, Lecture Notes in Computer Science. Springer, 2011.

[20] Alexander Shekhovtsov and Vaclav Hlavac. A Distributed Mincut/Maxflow Algorithm Combining Path Augmentation and Push-Relabel. Research Report K333–43/11, CTU–CMP–2011–03, Department of Cybernetics, Faculty of Electrical Engineering Czech Technical University, Prague, Czech Republic, June 2011.

[21] Alexander Shekhovtsov and Vaclav Hlavac. A Distributed Mincut/Maxflow Algorithm Combining Path Augmentation and Push-Relabel. *International Journal of Computer Vision*, 104(3):315–342, 2013.

[22] Wouter Souffriau, Pieter Vansteenwegen, Greet Vanden Berghe, and Dirk Van Oudheusden. The planning of cycle trips in the province of East Flanders. *Omega*, 39(2):209–213, April 2011.

[23] T. Tsiligirides. Heuristic Methods Applied to Orienteering. *The Journal of the Operational Research Society*, 35(9):797, September 1984.

[24] Pieter Vansteenwegen, Wouter Souffriau, and Dirk Van Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 209(1):1–10, 2011.

[25] C. Verbeeck, P. Vansteenwegen, and E. H. Aghezzaf. An extension of the arc orienteering problem and its application to cycle trip planning. *Transportation Research Part E: Logistics and Transportation Review*, 68:64–78, 2014.