FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Complexity of a TSP related problems in sparse networks
Student:	Vitalii Shakhmatov
Supervisor:	doc. RNDr. Dušan Knop, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

TSP is a prominent problem in both theoretical and practical computer science. Therefore, it is not surprising that there are many variants of this well-known problem arising from various applications. Select one particular variant of TSP, discuss its importance, and design an efficient algorithm for a selected class of sparse graphs (e.g., low feedback edge number).

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 17 February 2023 in Prague.

Bachelor's thesis

COMPLEXITY OF A TSP RELATED PROBLEMS IN SPARSE NETWORKS

Vitalii Shakhmatov

Faculty of Information Technology
Katedra teoretické informatiky
Supervisor: doc. RNDr. Dušan Knop, Ph.D.
May 10, 2023

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Vitalii Shakhmatov. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Shakhmatov Vitalii. *Complexity of a TSP related problems in sparse networks.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

I would like to express my deepest gratitude to my supervisor, doc. RNDr. Dušan Knop, Ph.D., for his invaluable guidance, support, and encouragement throughout the development of this bachelor thesis. His expertise and insights have been crucial to the completion of this work.
I would also like to extend my heartfelt appreciation to my family, who have always been there for me, providing emotional support, and believing in me throughout my academic journey. Their love and care have been a constant source of strength.
Last but not least, I wish to express my sincere thanks to my girlfriend, who has been a pillar of support during the challenging moments of working on this thesis. Her understanding, patience, and unwavering love have meant the world to me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 10, 2023 ....

Abstract

In this bachelor thesis, we introduce and examine the MULTI-GRAPH MULTI-CONSTRAINED OPTIMAL PATH (MGMCOP) problem. This problem is related to the TRAVELING SALESMAN PROBLEM (TSP), which is known to be an NP-complete problem. The MGMCOP problem has wide-ranging applications in diverse real-life scenarios, particularly in sparse networks where computational efficiency is very important. We present a dynamic programming algorithm that yields pseudo-polynomial time complexity for solving this problem. However, the main part of our work is about imposing specific constraints on the input data. These constraints, carefully designed and implemented, reduce the time complexity of our algorithm to be polynomial. This complexity reduction is achieved without sacrificing solutions accuracy and practical utility in sparse networks.

Keywords traveling salesman problem, multi-graph multi-constrained optimal path problem, algorithms for NP-hard problems, dynamic programming, sparse networks, constraint optimization

Abstrakt

V této bakalářské práci představujeme a zkoumáme problém MULTI-GRAPH MULTI-CONSTRAINED OPTIMAL PATH (MGMCOP). Tento problém souvisí s problémem OBCHODNÍHO CESTUJÍCÍHO (TSP), který je známý jako NP-úplný problém. Problém MGMCOP má široké uplatnění v různých reálných scénářích, zejména v řídkých sítích, kde je výpočetní efektivita velmi důležitá. Představujeme algoritmus dynamického programování, který pro řešení tohoto problému poskytuje pseudo-polynomiální časovou složitost. Nicméně, hlavní část naší práci se týká uplatnění konkrétních omezení na vstupní data. Tato omezení, pečlivě navržená a implementovaná, snižují časovou složitost našeho algoritmu na polynomiální. Toto snížení složitosti je dosaženo bez obětování přesnosti řešení a praktické užitnosti v řídkých sítích.

Klíčová slova problém obchodního cestujícího, multi-graph multi-constrained optimal path problém, algoritmy pro NP-těžké problémy, dynamické programování, řídké sítě, optimalizace s omezeními

List of abbreviations

BFS	breadth-first search.
DES	the Data Encryption Standard.
DFS	depth-first search.
GTSP	Graphical Travelling Salesman Problem.
MCOP	Multi-constrained optimal path.
MGMCOP	Multi-graph Multi-constrained Optimal Path.
MK	Multi-choice Knapsack.
NDTM	Non-Deterministic Turing Machine.
NP	Non-deterministic Polynomial time.
OPMP	Optimal path in a multi-path.
QoS	Quality of Services.
RSP	Restricted Shortest Path.
SAT	Boolean satisfiability.
TM	Turing Machine.
TSP	Travelling Salesman Problem.

Chapter 1 Introduction

Often, when people hear *Computer science*, it is hard for them to imagine something that could be used in their daily life. They are more likely to think about computers, processors, motherboards, and other terms they do not understand, and that has nothing in common with computer science. In reality, though, there are lots of situations people are dealing with every day to which certain problems from computer science could be applied. And often, people are just not aware of it.

For example, take a look at the following situation that might happen to everyone. Bob needs to get to the university in time. He has only 30 minutes before a lecture starts. Bob does not want to be late for the lecture, so he is ready to pay more to get to the university faster. But on the other hand, he is still a student, which is why he has to be smart in terms of money spending. It means that he has to find a way to get to the university within 30 minutes and with minimal cost.

Suppose that there is only one type of transport in the previous case, for example, Bob's car. Bob is searching for a route with minimal gas consumption, the computer science problem called Restricted Shortest Path (RSP) can be applied. This problem will be discussed later. For sure, there could be more than one transport type. Bob can get a taxi, or he can ride halfway by bus and then rent a scooter. In this case, a new problem can be derived by modifying the original Restricted Shortest Path problem and an effective algorithm to solve the problem found, which will also help Bob in his situation.

Although computer science shares many similarities with real-life situations, there are notable differences, particularly in the approaches used to tackle problems. In computer science, problems are often defined as broadly as possible, such as finding a route in a graph with limitless size and an infinite number of cycles or packing a knapsack with infinite capacity and an endless choice of items. In contrast, real-life scenarios are more specific. In Bob's case, he is well aware of the routes in his city. He just needs to find the one that matches his requirements.

What will happen if both two approaches discussed above are applied? What if some constraints are added to the problem in computer science? Will it be possible to optimize an algorithm when the input data is somehow limited? The answer is yes. There are lots of techniques approaching to reduce algorithms' time complexity by sacrificing something. For example, approximation algorithms. This technique is designed to provide near-optimal solutions for complex optimization problems while achieving better performance than exact algorithms. An "exact" algorithm, in this context, refers to an algorithm that produces the optimal solution for the given problem. In contrast, an approximation algorithm trades off the guarantee of finding the absolute optimal solution in favor of improved time or resource efficiency. But do we always need an exact solution? No. Another way to achieve better performance is to impose some constraints on the input data. It means that we are no longer capable of solving all the instances of the problem. However, solutions for those instances that are solvable are calculated faster. This approach requires finding an optimal balance between imposed constraints and the integrity of the original problem. Because if there are too many constraints, or those constraints are too strict, the problem may become not applicable. For example, suppose that the algorithm solving Bob's problem is now limited to accepting just one route. In this case, the algorithm can not be utilized to solve the problem.

And that is what this bachelor thesis aims to cover.

Chapter 2

Objectives and structure

2.1 Objectives

This section describes the objectives of each chapter and the bachelor thesis generally in a more technical way than the previous chapter does.

Starting by defining the primary goal of the thesis, subgoals that are necessary for reaching the main one are derived after. Also, to avoid a high level of abstraction in the thesis, goals are separated into two distinct categories: *theoretical* and *practical*, where the latest one inherits from the first one, i.e., it defines which practical usage can be achieved by reaching the theoretical goals.

That being said, the main theoretical goal can be defined as follows: *given a Multi-graph Multi-constrained Optimal Path (MGMCOP) problem, analyze an algorithm solving the problem and impose constraints on input data so that polynomial time complexity of the algorithm can be reached.*

To reach the goal defined above, several things should be done; these can also be considered to be theoretical goals. First, some terms (e.g., NP-*hard* problem, graph, path, etc.) must be explained. Then, several existing NP-*hard* problems should be analyzed and the Multi-graph Multi-constrained Optimal Path problem must be proven to be NP-*hard*. Finally, input data constraints must be proposed based on the input data analysis. Please note that *input data constraint* does not mean just numeric limits (e.g., limit the maximum number of edges in the graph to be 10). The actual meaning is much more generic, i.e., constraints are any limitation in the most general sense (e.g., an input graph structure must be a tree or a forest, the graph may not contain any cycles, etc.)

To set the main practical objective of the thesis, Bob's problem introduced in the *Introduction* is referenced. By accomplishing the theoretical goals of the thesis, a solution to Bob's problem must be found. Then, the algorithm solving the MGMCOP problem should be introduced. The algorithm's correctness and time complexity should be then proven.

To summarize all the goals mentioned above, the following list can be composed:

- provide the reader with all needed theoretical definitions and terms,
- analyze existing NP-Hard problems,
- define an NP-Hard problem applicable in Bob's case (MGMCOP,
- prove that the MGMCOP problem is NP-*hard* or even NP-*complete*,
- find an algorithm solving the problem,
- analyze the algorithm, prove the algorithm's correctness, completeness, and time-complexity,
- analyze input data structure and impose constraints, allowing to reduce the algorithm's time complexity.

2.2 Structure of the thesis

In Chapter 3 existing well-known problems, such as Travelling Salesman Problem (TSP), Restricted Shortest Path (RSP), Multi-constrained optimal path (MCOP) etc., will be defined and analyzed. In Chapter 3, it is assumed that the reader is familiar with the basics of computer science and graph theory. Otherwise, the reader can find all the needed definitions in Chapter 4.

Chapter 4, as mentioned before, contains all the necessary definitions, such as NP-*hard* and NP-*complete* sets of problems, how to prove that problem is NP-*hard* or NP-*complete*, some definitions from graph theory, etc. A new NP-*hard* problem called Multi-graph Multi-constrained Optimal Path (MGMCOP) is introduced later in Chapter 4. This problem is well fit for Bob's problem discussed earlier. Chapter 4 also provides a proof that the MGMCOP problem is NP-*hard*.

In Chapter 5, an algorithm solving the MGMCOP problem with a time-complexity, completeness, and correctness proofs is introduced to the reader.

In Chapter 6, the results of the bachelor thesis will be discussed and summarized.

Chapter 3
Literature review

As was mentioned in Chapter 2, some NP*-complete and* NP*-hard problems are introduced in this chapter. However, it is important to understand that there are too many problems and we can not analyze them all. This is why we should choose problems to analyze based on defined criteria. Generally speaking, we do not need to analyze problems that are not related to Bob's case. Those criteria are defined below. Problems we seek should satisfy at least one of the following conditions:*

- *it is a graph-based problem,*
- *problem's goal is to find a path between two vertices,*
- *problem's goal is to find an optimal solution among a set of possible solutions.*

Speaking of graph-based problems, one of the most known problems is the TRAVELLING SALESMAN PROBLEM, TSP for short. The basic idea of it can be formulated as follows. A traveling salesman wants to visit each of a set of towns at least once, starting from and returning to his hometown. One of his problems is to find the shortest such trip [1].

The TSP problem has been widely used in various fields of science for over 7 decades now. It has model character in many branches of Mathematics, Computer Science, and Operations Research. Heuristics, linear programming, and branch and bound, which are still the main components of todays most successful approaches to hard combinatorial optimization problems, were first formulated for the TSP problem and used to solve practical instances already in 1954 by Dantzig, Fulkerson and Johnson. [1]

As Ball states ([1]) the traveling salesman problem is an NP-*hard* problem, as it was proved by Karp in 1972 ([2]), as NP-completeness theory developed. Furthermore, it is one of the earliest problems to be proved to be NP-*hard*. Obviously, being one of the earliest problems in computer science theory, the TSP problem has been used to derive a lot of different problems based on it. There are a lot of variants of TSP nowadays, the RURAL POSTMEN problem, for example. Some of the techniques and approaches used to solve the original TSP problem and its derivation may be useful in terms of this bachelor thesis.

To understand the TSP problem even better, the formal definition of the problem is introduced to the reader.

► **Definition 3.1** ([3]). *A* ***Hamilton cycle*** *of graph G is a cycle that contains every vertex of G.*

► **Definition 3.2** ([1]). *Let $G = (V, E)$ be the complete undirected graph. Given an objective function $c\colon E \to \mathbb{R}$, that associates the 'length' $c(e)$ with every edge e of G, the* SYMMETRIC

Travelling Salesman Problem consists of finding a Hamilton cycle (a cycle visiting every node exactly once) such that its c-length (the sum of the lengths of its edges) is as small (large) as possible.

Although the definition above assumes that the graph is complete and undirected, later variants of the TSP problem also work with other graphs. The TSP problem can be applied in a more general way than suggested by its definition [1].

> [4] Two difficulties arise in stating the Travelling Salesman Problem as above. First, the graph G may not be *Hamiltonian* (i.e., G may not have a Hamilton cycle). Second, even when G is Hamiltonian, the shortest way to visit all the nodes of G may not be to follow a Hamilton cycle. Instead, it may be shorter to go through some nodes more than once and/or use some edges more than once.
>
> To overcome these difficulties, we can introduce a new version of the TSP problem, which we call the Graphical Travelling Salesman Problem (GTSP).

▶ **Definition 3.3** ([4])**.** *A* ***tour*** *of a connected graph G is a cycle going at least once through each node of G.*

▶ **Definition 3.4** ([4])**.** *The Graphical Travelling Salesman Problem (GTSP) consists in finding a tour of G whose length is minimum.*

> [4] Of course GTSP is NP-*hard*, since, given a graph G, the solution of GTSP with the length function $\mathrm{l}(e) = 1$ for all $e \in E$, would show whether G is Hamiltonian, a known NP-*complete* problem.

The TSP problem is well-studied and very generic in terms of its applications, i.e., it can be applied in a lot of different cases, not only in one implied from the definition, and some of the studies can be applied in our case. Still, the main objective of the TSP problem is to find a cycle, whereas Bob needs to find a path, so we can take a look at another TSP related problems, that can be utilized in our case.

Speaking of problems, their main goal is to find an optimal path, another NP-*hard* problem, called Restricted Shortest Path, RSP for short, can be taken into consideration.

The decision version of the RSP problem is an NP-*complete* problem. This problem can be described informally as follows: *Given some cities connected by roads (there might be no direct road between two cities). Each road has two parameters, the length and the duration of passing it. Intuitively, when more roads are passed, the overall length is equal to the sum of the lengths of each road, and the overall duration of passing equals the sum of the durations of each road. The route between two cities has to be found such that it takes no more than desired time to get to the destination city, and the length is minimal.*

Formally it is defined as follows.

▶ **Definition 3.5** ([5])**.** *Given a directed graph $G = (V, E)$, two objective functions $c\colon E \to \mathbb{Z}^+$ and $t\colon E \to \mathbb{Z}^+$, that associate with every edge a 'length' and 'transition time', respectively. The length $c(S)$ and transition time $t(S)$ of a set of edges S is defined as the sum of the length and transition times, respectively, of the edges in this set. A directed path from vertex i to vertex j (an i-j path) is called a T-path if its transition time is less than or equal to T. The Restricted Shortest Path (RSP) problem is to compute, for a given value T, a shortest 1-n T-path.*

There is also another NP-*hard* problem, called Multi-constrained optimal path, MCOP for short, which is a generalized version of the RSP problem, discussed above. Instead of time and one additional parameter as it is in RSP problem, each edge of MCOP has time and K additional parameters. Same as in RSP, the objective is to find a path satisfying K constraints and maximizing (minimizing) the remaining parameter. For K equals 1, the MCOP is equivalent to the RSP problem [6].

MCOP is defined as follows.

▶ **Definition 3.6** ([6]). *Consider a network that is represented by a directed graph $G = (V, E)$, where V is the set of vertices and E is associated with a primary cost parameter $c(i, j)$ and K additive parameters $w_k(i, j)$, $k = 1, 2, \ldots, K$; all parameters are non-negative. Given K constraints q_k, $k = 1, 2, \ldots, K$, the* Multi-constrained optimal path *(MCOP) problem is to find a path P from a source node s to a destination node t such that:*

$$w_k(P) \overset{\text{def}}{\equiv} \sum_{(i,j)\in P} w_k(i,j) \leq q_k \textit{ for } k = \{1, 2, ..., K\}, \tag{3.1}$$

and

$$c(p) \overset{\text{def}}{\equiv} \sum_{(i,j)\in P} c(i,j) \textit{ is minimized over all feasible paths satisfying Equation (3.1).} \tag{3.2}$$

The desicion version of MCOP, the Multi-constrained optimal path (MCOP) problem, is NP-*complete*. It aims only at finding any feasible path w.r.t. multiple constraints (no path optimization is done) [6].

Almost in every article discussing the three problems mentioned above, i.e., RSP, MCP, MCOP, Quality of Services (QoS) networks are considered as a use case. Quality of Services (QoS) parameters are a set of metrics used to evaluate and ensure the desired level of performance and reliability in a network or system. These parameters help to quantify and manage the quality of the services provided, such as data transmission or communication between devices. This is correct in the general case because it helps us to understand the complexity of the problem. Although in Bob's case, it makes it harder to see how the input may be limited to reduce algorithm's time complexity, which is the main goal of the thesis. That is why this bachelor thesis is considering those three problems only in terms of application to Bob's problem. Chapter 4 will cover it in a more detailed way.

Before moving on to Chapter 4, there is one more problem that has to be discussed. This is the Knapsack problem, which at the first glance, does not suit Bob. It has a lot of practical applications and often appears as a subproblem in analysis and solving of more complicated problems [7]. As the reader will find out later, the dynamic programming algorithm for solving the Knapsack problem will be used to solve the Multi-graph Multi-constrained Optimal Path problem.

The 0-1 knapsack problem may be formulated informally in the following way: *Given a knapsack of limited capacity and some number of different items, each of those can be described by two parameters: weight and price. The knapsack problem is to find a combination of items such that the sum of weights is no more than the knapsack capacity, and the price is maximal (minimal) among all the other combinations.*

It is defined formally as follows.

▶ **Definition 3.7** ([7]). *We formulate* 0-1 knapsack problem *as*

$$v(K) = \max \sum_{j=1}^{n} a_j x_j,$$

subject to

$$\sum_{j=1}^{n} a_j x_j \leq b,$$

$$a_j \in \mathbb{N}$$

$$x_j \in \{0, 1\}, \quad j \in N = \{1, \ldots, n\}$$

where $v(K)$ is the value of the objective function in an optimal point denoted by x^*.

Without loss of generality we may assume that all data are positive integers, i.e.,

$$0 < a_j \leq b \quad \mathit{for}\ j \in N \quad \mathit{and} \quad \sum_{j=1}^{n} a_j > b.$$

In the definition above, the capacity of the knapsack is denoted by b, the weight of the object j is denoted by a_j, and x_j is set to one if the object should be placed to the knapsack and zero otherwise. That being said, the sum of the weights of objects in a knapsack $\sum_{j=1}^{n} a_j x_j$ must not exceed the capacity b of the knapsack.

In the same spirit as in the TSP problem, being one of the most studied NP-*complete* problems, the KNAPSACK problem has a variety of different modifications and variants, such as the MULTI-CHOICE KNAPSACK and the NESTED KNAPSACK problems.

If the set of variables N is partitioned into m classes N_k, $k \in \{1, ..., m\}$ and if we require that exactly one variable has to be chosen from each class, then we get the MULTI-CHOICE KNAPSACK (MK) problem ([7]).

▶ **Definition 3.8** ([7]). *If the set of variables N is partitioned into m classes N_k, $k = \{1, \ldots, m\}$ and if exactly one variable is required to be chosen from each class, then the MULTI-CHOICE KNAPSACK (MK) problem, which may be formulated as*

$$v(MK) = \max \sum_{k=1}^{m} \sum_{j \in N_k} c_j x_j,$$

subject to

$$\sum_{k=1}^{m} \sum_{j \in H_k} a_j x_j \leq b,$$

$$\sum_{j \in N_k} x_j = 1, k \in M = \{1, \ldots, m\},$$

$$x_j = \{0, 1\}, j \in N = \bigcup_{k=1}^{m} N_k = \{1, \ldots, n\}.$$

This version of the KNAPSACK problem will be used later as a subproblem in Bob's case.

That being said, there are a lot of various problems that could be utilized in Bob's case. In Chapter 4 we will discuss these problems and their application to Bob's case in detail.

Chapter 4
Theoretical basis

The present chapter endeavors to provide the theoretical foundations necessary for the formal definition, NP-*hardness* proof, algorithmic solution, and other goals of this thesis. This chapter employs a top-down approach. The first part of the chapter aims to define the problem in a very intuitive manner, so the reader can better understand the problem itself. This will help the reader to answer *why* certain definitions and problems are being discussed. Understanding how these definitions are applied in terms of the bachelor thesis, the reader can easier understand the definitions themselves. Later in this chapter, when the reader already has a basic understanding of what needs to be done, the required terms are formally defined.

4.1 Intuitive definition of the problem

The base of the problem is fairly simple: given a graph, each edge in the graph has two parameters – the cost of passing through the edge and the duration of travel. The goal is to find a path such that the total duration is less than the given maximum and the cost is minimized. As was already mentioned, there could be a few different types of transport, which means that in some cases one can get from one vertex to an adjacent vertex by different means of transport, and thus it should be possible to have more than one edge between two vertices in the input graph.

That being said, the goal is to find a path starting at the given point s and finishing at the given point t such that the overall duration is no more than the given maximum and the price is minimal among all the different solutions.

4.2 Complexity classes

In the field of computer science, understanding the complexity of algorithms and the problems they solve is crucial. Two major complexity classes, P and NP, play a central role in this area. Intuitively, P represents problems that can be efficiently solved by a computer, while NP consists of problems for which a potential solution can be efficiently verified. Although we have not yet provided formal definitions, this basic understanding of P and NP will help us explore their significance and the implications of their relationship.

In the following section, we will discuss the importance of the P and NP complexity classes and the consequences that would arise if it were proven that P equals NP. This discussion will provide a foundation for our deeper exploration of these concepts, which will be followed by a formal introduction and definition of P and NP.

4.2.1 P versus NP

The P versus NP problem is to determine whether every language accepted by some nondeterministic Turing machine in polynomial time is also accepted by some deterministic Turing machine in polynomial time ([8]).

> [8] If $\mathsf{P} = \mathsf{NP}$ is proved by exhibiting a truly feasible algorithm for an NP-*complete* problem such as BOOLEAN SATISFIABILITY (deciding whether a collection of propositional clauses has a satisfying assignment), the practical consequences would be stunning. First, most of the hundreds of problems shown to be NP-*complete* can be efficiently reduced to BOOLEAN SATISFIABILITY, so many of the optimization problems important to industry could be solved. Second, mathematics would be transformed, because computers could find a formal proof of any theorem which has a proof of reasonable length. This is because formal proofs (say in Zermelo-Fraenkel set theory) are easily recognized by efficient algorithms, and hence bounded proof existence is in NP. Although the formal proofs may not be intelligible to humans, the problem of finding intelligible proofs would be reduced to that of finding a good recognition algorithm for formal proofs. Similar remarks apply to the fundamental problems of artificial intelligence: planning natural language understanding, vision, and even creative endeavors such as composing music and writing novels. In each case success would depend on finding good algorithms for recognizing good results, and this fundamental problem itself would be aided by the SAT solver by allowing easy testing of recognition theories.

Although there are plenty of positive consequences of proving that $\mathsf{P} = \mathsf{NP}$, there are also some negative ones. For example, that complexity-based cryptography would become impossible. The security of the Internet, including most financial transactions, depends on assumptions that computational problems such as large integer factoring or breaking the Data Encryption Standard (DES) cannot be solved feasibly. All of these problems are efficiently reducible to BOOLEAN SATISFIABILITY. On the other hand, quantum cryptography would survive a proof of $\mathsf{P} = \mathsf{NP}$, and might ensure Internet security ([8]).

Another cryptography-related example that will not survive a proof of $\mathsf{P} = \mathsf{NP}$ is crypto-currencies. Such currencies as Bitcoin and Ethereum, for example, use Elliptic Curve Cryptography, which is also based on the assumption of $\mathsf{P} \neq \mathsf{NP}$. If that assumption is wrong, it means that one can easily calculate a private key from the corresponding public key and, hence, gain access to the assets related to that address on the blockchain. Please note that no one will benefit from it. Even a thief that steals all the crypto-currencies from every account in the world. Because as soon as any currency becomes cryptographically not secure, it immediately becomes insolvent. And it is obviously not secure if $\mathsf{P} = \mathsf{NP}$ is proven.

Later on, it is assumed that $\mathsf{P} \neq \mathsf{NP}$, since at the time when this bachelor thesis is being written, it has not been proven otherwise.

4.2.2 Definitions

Now we can move on to definitions of P and NP classes. However, to define the P class, we will require one additional definition of something called DTIME.

Please note that the following definitions use such terms as Turing Machine (TM) and Non-Deterministic Turing Machine (NDTM). For readers who may not be familiar with these specific terms, it is recommended to consult the original source material for more precise definitions ([9]). This will help to ensure a clearer understanding of the concepts presented within the text.

▶ **Definition 4.1** ([9]). *Let $T\colon \mathbb{N} \to \mathbb{N}$ be some function. We let $\mathbf{DTIME}(\boldsymbol{T(n)})$ be the set of all Boolean (one bit output) functions that are computable in $c \cdot T(n)$-time for some constant $c > 0$.*

The following class will serve as our rough approximation for the class of decision problems that are efficiently solvable [9].

▶ **Definition 4.2** ([9]). $\mathsf{P} = \bigcup_{c \geq 1} \mathsf{DTIME}(n^c)$

> [9] The class P is felt to capture the notion of decision problems with "feasible' decision procedures. Of course, one may argue whether $\mathsf{DTIME}(n^{100})$ really represents "feasible" computation in the real world. However, in practice, whenever we show that a problem is in P, we usually find an n^3 or n^5 time algorithm (with reasonable constants), and not an n^{100} algorithm. (It has also happened a few times that the first polynomial-time algorithm for a problem had high complexity, say n^{20}, but soon somebody simplified it to say an n^5 algorithm.)

The next important class to be defined is the NP class. This complexity class will serve as our formal model for the class of problems having efficiently verifiable solutions: a decision problem/language is in NP if given an input x, we can easily verify that x is a YES instance of the problem (or equivalently, x is in the language) if we are given the polynomial-size *solution* for x, that *certifies* this fact [9]. The NP class is formally defined as follows:

▶ **Definition 4.3** ([9]). *A language $L \subseteq \{0,1\}^*$ is in* $\mathbf{NP}$ *if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M such that every $x \in \{0,1\}^*$,*

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \ s.t. \ M(x,u) = 1$$

If $x \in L$ and $u \in \{0,1\}^{p(|x|)}$ satisfy $M(x,u) = 1$, then we call u a ***certificate*** *for x (with respect to the language L and machine M).*

> [9] The class NP can also be defined using a variant of Turing Machine called Non-Deterministic Turing Machine (abbreviated NDTM). In fact, this was the original definition and the reason for the name NP, which stands for Non-deterministic Polynomial time. The only difference between an NDTM and a standard TM is that an NDTM has *two* transition functions δ_0 and δ_1. In addition the NDTM has a special state we denote by q_{accept}. When an NDTM M computes a function, we envision that at each computational step M makes an arbitrary choice as to which of its two transition functions to apply. We say that M outputs 1 on a given input x if there is *some* sequence of these choices (which we call the *non-deterministic choices* of M) that would make M reach q_{accept} on input x. Otherwise — if *every* sequence of choices makes M halt without reaching q_{accept} — then we say that $M(x) = 0$. We say that M runs in $T(n)$ time if for every input $x \in \{0,1\}^*$ and every sequence of non-deterministic choices, M reaches either the halting state or q_{accept} within $T(|x|)$ steps.

▶ **Definition 4.4** ([9]). *For every function $T\colon \mathbb{N} \to \mathbb{N}$ and $L \subset \{0,1\}^*$, we say that $L \in$* $\mathbf{NTIME}(\boldsymbol{T(n)})$ *if there is a constant $c > 0$ and a $c \cdot T(n)$-time NDTM M such that for every $x \in \{0,1\}^*$, $x \in L \Leftrightarrow M(x) = 1$.*

The next theorem gives an alternative definition of NP, the one that appears in most texts [9].

▶ **Theorem 4.5** ([9]). $\mathsf{NP} = \bigcup_{c \in \mathbb{N}} \mathsf{NTIME}(n^c)$

The proof of the theorem above will not be provided in terms of this bachelor thesis. However, the reader can find the formal proof in the original source ([9]).

There are also other complexity classes that have to be covered in terms of this bachelor thesis; those are NP-*hard* and NP-*complete* classes. However, there is one more important thing that must be defined before moving to these complexity classes. That is the so-called Karp reduction, which enables reducing a problem to another one. To say that *one problem can* *be reduced to another* means that there is a deterministic, polynomial-time reduction algorithm that takes an instance of the first problem as input and returns the corresponding instance of the second problem. Given two problems A and B, if an algorithm converting every instance of the problem A to the corresponding instance of the problem B in polynomial time exists, then A is *reducible* to B. The formal definition is provided below.

▶ **Definition 4.6** ([9]). *We say that a language $A \subseteq \{0,1\}^*$ is **polynomial-time Karp reducible** to a language $B \subseteq \{0,1\}^*$ (sometimes shortened to just "polynomial-time reducible") denoted by $A \leq_p B$ if there is a polynomial-time computable function $f\colon \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in A$ if and only if $f(x) \in B$.*

▶ **Definition 4.7** ([9]). *We say that B is **NP-hard** if $A \leq_p B$ for every $A \in$ NP.*

▶ **Definition 4.8** ([9]). *We say that B is **NP-complete** if B is NP-hard and $B \in$ NP.*

That being said, to prove that a problem is NP-*hard*, it is enough to reduce a known NP-*hard* problem to it. This is because an NP-*hard* problem is one that is at least as hard as any NP problem, and if the known NP-*hard* problem can be reduced to the new problem, then the new problem is at least as hard as the known NP-*hard* problem, thus, is at least as hard as any other NP problem, which implies that the problem is NP-*hard*. Additionally, to prove that the new problem is NP-*complete*, it should be proven that the problem is in NP, i.e., the problem has a polynomial certificate.

4.3 Graph theory

The first thing that has to be noticed when speaking of the problem of finding an optimal way from one point to another is that there may be more than one connection between two given adjacent points; as an example, consider two public transport stops connected by both bus and tram routes. Constructing a graph for this example results in a situation in which there are two edges between two adjacent vertices. Going even further, we might say that these two stops are not that far away from each other, and one can walk or drive by car, taxi, or scooter; all these options obviously have different prices and take a different amount of time, which is why for each of the options a separate edge must be added to the graph. That being said, the graph may contain an unlimited number of edges between every two adjacent vertices. The graph with the described properties is called *multi-graph.* However, we will not require an additional definition for multi-graph because the following definition of the graph covers the multi-graph case either.

▶ **Definition 4.9** ([3]). *A **graph** G is an ordered pair $(V(G), E(G))$ consisting of a set $V(G)$ of vertices and a set $E(G)$, disjoint from $V(G)$, of edges, together with an incidence function ψ_G that associates with each edge of G an unordered pair of (not necessarily distinct) vertices of G.*

▶ **Definition 4.10** ([3]). *Given a graph G we denote the number of vertices in G by $\mathrm{v}(G)$ and call it the **order** of G.*

▶ **Definition 4.11** ([3]). *Given a graph G we denote the number of edges in G by $\mathrm{e}(G)$ and call it the **size** of G.*

Later on, the term *graph* will be used to refer to both non-multi-graphs and multi-graphs because the definition of the graph above (4.9), as it was already mentioned, covers both cases.

Speaking of searching the path between two vertices in the given multi-graph, the term *path* must be formally defined. However, it requires some additional definitions.

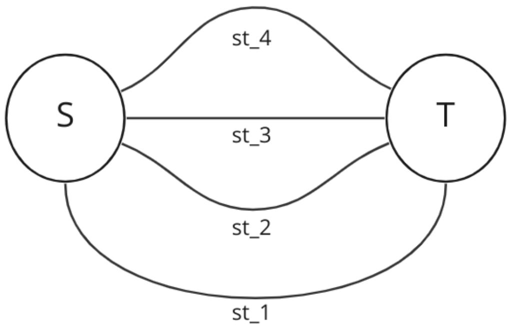
▶ **Definition 4.12** ([3]). *A **walk** in a graph G is a sequence $W := v_0 e_1 v_1 \dots v_{\ell-1} e_\ell v_\ell$, whose terms are alternately vertices and edges of G (not necessarily distinct), such that v_{i-1} and v_i, are the ends of e_i, $1 \leq i \leq \ell$.*

[3] In a simple graph, a walk $v_0e_1v_1\dots v_{l-1}e_lv_l$ is determined, and is commonly specified, by the sequence $v_0v_1\dots v_l$ of its vertices. Indeed, even if a graph is not simple, we frequently refer to a sequence of vertices in which consecutive terms are adjacent vertices as a 'walk'. In such cases, it should be understood that the discussion is valid for any walk with that vertex sequence. This convention is especially useful in discussing paths, which may be viewed as walks whose vertices (and edges) are distinct.

▶ **Definition 4.13.** *A walk having all the edges distinct is called* ***trail****.*

▶ **Definition 4.14.** *If a trail defines a sequence of vertices in which all vertices are distinct, we call such a trail a* ***path****.*

It is important to recognize that the definition of a path provided above(4.18) also applies to multi-graphs. This is because the definition specifies the sequence of edges that make up the path, and even when there are multiple edges between two vertices, only one of these edges is included in the resulting sequence. However, we do require a definition of a path that will cover every possible path between the source and target vertices. The reason why this definition is required is that it will help us to define the MGMCOP in a more understandable way. We will call such a path **multi-path**, which can be defined as follows.



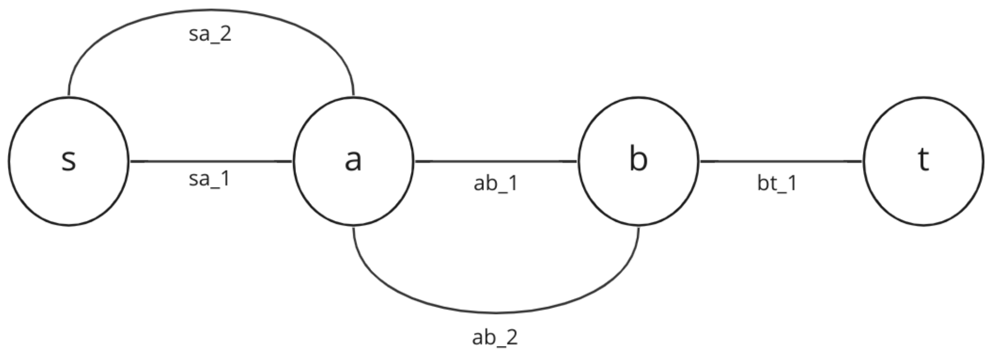
■ **Figure 4.1** Multi-edge with 4 edges

▶ **Definition 4.15.** *Given a graph $G = (V, E)$ and two adjacent vertices s and t. Let the $ME(s,t)$ be the set $\{e \in E\colon \psi_G(e) = \{s,t\}\}$ (set of all edges between vertices s and t). We call an $ME(s,t)$ the* ***multi-edge*** *from s to t. In certain cases, we may refer to it as just ME.*

▶ **Definition 4.16.** *A* ***multi-walk*** *in a graph G is a sequence $MW := v_0E_1v_1\dots v_{l-1}E_lv_l$, whose terms are alternately vertices and multi-edges of G (not necessarily distinct), such that E_i is $ME(v_{i-1}, v_i)$, $1 \le i \le l$.*

▶ **Definition 4.17.** *A walk having all the multi-edges distinct is called* ***multi-trail****.*

▶ **Definition 4.18.** *If a multi-trail defines a sequence of vertices in which all vertices are distinct, we call such a trail a* ***multi-path****.*



■ **Figure 4.2** Multi-path with 4 vertices containing 4 different paths

In the same way, as we define an operator determining how many edges there are in the graph in Definition 4.10, we can define an operator determining the number of multi-edges in a graph.

▶ **Definition 4.19.** *Given a graph G we denote the number of multi-edges in G by* $\mathrm{me}(G)$*.*

At this stage, there is one important thing to mention. Without loss of generality, all the graphs discussed later on are considered to be connected, i.e., there is at least one path between any two vertices. This is because there is no possible path between two vertices that belongs to disconnected components. This is why every connected component of the disconnected graph may be treated as a separate connected graph itself. Saying that in more simple words, one can not get from the city A to the city B if there are no roads between those two cities, no matter what was the exact location inside the city A and what is the target location inside the city B.

As was already mentioned, graphs occurring in real-life cases (for example, metro maps) are not as general as a graph by its definition. They are often trees or *almost* trees (for example, consider, for example, the map of Prague metro, which contains just one cycle and by removing just one edge a tree may be obtained). Graphs with a low number of edges that must be removed to obtain a tree are certainly *almost* trees. Such graphs have one important feature, the maximum number of paths between two vertices is significantly lower than in a general case. This is because it depends exponentially on the number of edges to be removed, discussed above, rather than the number of all edges in the graph as it is in the general case. Before formally defining what almost tree is, the term tree itself must be discussed.

▶ **Definition 4.20** ([3])**.** *A graph is **acyclic** if it does not contain a cycle.*

▶ **Definition 4.21** ([3])**.** *A **tree** is an acyclic connected graph.*

Although this definition could also be applied to multi-graphs in theory, the graph still has to be simple to be a tree. This is because the graph having at least two vertices with at least two edges between them has a cycle by definition and thus can not be a tree. However, the same as multi-path generalize the definition of the simple path, we need to define the term *multi-tree*, which will provide us with a higher level of abstraction when speaking of trees. The following statement, which is equivalent to the tree definition (4.21), will help us to define the multi-tree later.

▶ **Statement 4.22.** *A graph G is a* ***tree*** *if for any two vertices s, t there is exactly one path from s to t.*

Although this statement is equivalent to the definition of the tree, it still does not provide us with a required level of abstraction. However, by making a few simple modifications to that statement, we can define a *multi-tree* as follows.

▶ **Definition 4.23.** *A graph G is a* ***multi-tree*** *if for any two vertices s, t there is exactly one multi-path from s to t.*

Please note that this definition is applicable for both graphs and multi-graphs because the *multi-path* in the non-multi-graph is basically the same as *path*, despite the fact that the multi-path is a sequence of sets of edges whenever the path is a sequence of edges. Still, in a graph that is not multi-graph, it is possible to derive a path from the multi-path, which is why the multi-path in the non-multi-graph corresponds to exactly one path and vice versa.

Now, let us define one more important property of every tree.

▶ **Theorem 4.24** ([3])**.** *Given a graph G, if G is a tree, then* $\mathrm{e}(G) = \mathrm{v}(G) - 1$.

A proof of this statement is not provided in terms of this thesis, because it is equivalent to the proof of the next statement defining the number of multi-edges in the multi-graph. However, the reader can find a proof in the original source ([3]).

▶ **Theorem 4.25.** *Given a graph G, if G is a multi-tree, then* $\mathrm{me}(G) = \mathrm{v}(G) - 1$.

Proof. By induction on $\mathrm{v}(G)$. When $\mathrm{v}(G) = 1$, then, obviously, $\mathrm{me}(G) = 0 = \mathrm{v}(G) - 1$.

Suppose the Theorem 4.25 true for all multi-trees on fewer than v vertices, and let G be a multi-tree on $v \geq 2$ vertices. Let $ME(s, t)$ is a multi-edge in G. Then $G - ME(s, t)$ contains no multi-path from s to t, since $ME(s, t)$ is the unique multi-path from s to t in G by the definition of the multi-tree (Definition 4.23). Thus $G - ME(s, t)$ is disconnected. The components of $G - ME(s, t)$ – G_1 and G_2 – being acyclic, are multi-trees. Moreover, each has fewer than v vertices. Therefore, by the induction hypothesis

$$\mathrm{me}(G_i) = \mathrm{v}(G_i) - 1 \text{ for } i = \{1, 2\}$$

Thus,

$$\mathrm{me}(G) = \mathrm{me}(G_1) + \mathrm{me}(G_2) + 1 = \mathrm{v}(G_1) - 1 + \mathrm{v}(G_2) - 1 + 1 = \mathrm{v}(G) - 1$$

◀

Now, when all the required definitions are provided, the problem can be formally defined.

▶ **Definition 4.26.** *Given a multi-graph $G = (V, E)$, two vertices $s, t \in V$, and a non-negative integer value $d_{\max}$, which stands for the maximal duration. Given two functions $d\colon E \to \mathbb{N}$ and $p\colon E \to \mathbb{N}$, assigning the duration and price values to each edge of the graph, i.e., $\forall e \in E\colon d(e)$ – the duration of passing over the edge e, $p(e)$ – the price of passing the edge e. The duration of a walk W is defined as a sum of the durations of edges in the walk: $d(W) := \sum\limits_{e \in W} d(e)$. The price of a walk W is defined as a sum of the prices of edges in the walk; $p(W) := \sum\limits_{e \in W} p(e)$. The* Multi-graph Multi-constrained Optimal Path *(MGMCOP) is to find a path P from s to t such that $d(P) \leq d_{\max}$ and $p(P)$ is minimal.*

possible

There is an important thing that must be understood at this moment. Every multi-path from point A to point B may or may not contain an overall optimal solution, and that can not be determined before processing the multi-path itself. Actually, finding the optimal path in a given multi-path is also an NP-*hard* problem, which will be proven later. For now, just assuming that it is NP-*hard*, we can claim that determining which of many multi-paths contains an overall optimal path is NP-*hard* as well. This will be proven later formally. This is because to find an optimal path among multiple multi-paths, we need to find an optimal path in each multi-path and then compare found paths with each other to find an overall optimal path. This will also be proven later. This implies that as the number of multi-paths increases, the algorithm's time complexity also increases. It means that the time complexity of the problem can not be better than exponential in cases when we have an exponential number of multi-paths in the graph. Hence, we have to impose constraints on the number of multi-paths in the graph so it is not exponential. We may use the so-called "feedback multi-edge set number" defined below as an indicator of the maximum possible number of multi-paths in the graph.

▶ **Definition 4.27.** *The **feedback edge set** of a graph G is a set S of edges such that $G \setminus S$ is acyclic.*

▶ **Definition 4.28.** *The **feedback edge set number** of a graph G is a size of a minimal possible feedback edge set of G. We will denote the feedback edge set number of the graph as* $\operatorname{fesn}(G)$.

Again, to gain the desired level of abstraction of the definition, we need to modify it slightly to suit the multi-tree definition.

▶ **Definition 4.29.** *The **feedback multi-edge set** of a connected graph $G = (V, E)$ is a set S of multi-edges such that graph $(V, E \setminus (\bigcup_{ME \in S} ME))$ is multi-tree.*

Please note that the feedback multi-edge set is defined for connected graphs. However, this definition does not require that restriction. It can also be defined for general graphs, but to do that, we have to define what is a multi-acyclic graph (in order to make an equivalent definition as a feedback edge set). Since we do not have this definition, and as was mentioned above, we are considering all the graphs to be connected, we do not require the general definition.

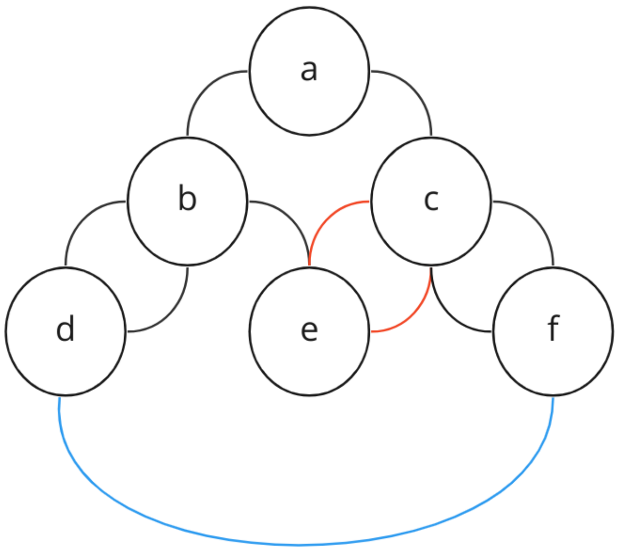
▶ **Definition 4.30.** *The **feedback multi-edge set number** of a connected graph G is a size of a minimal possible feedback multi-edge set of G. We will denote the feedback multi-edge set number of the graph as* $\operatorname{fmesn}(G)$.

▶ **Theorem 4.31.** *Given a graph $G = (V, E)$, a number of multi-edges in G is $|V| - 1 + \operatorname{fmesn}(G)$.*

Proof. By the Definition 4.30 of the feedback multi-edge set number, we know that to obtain a multi-tree from G we have to remove $\operatorname{fmesn}(G)$ edges. Also, we know that every multi-tree has $\operatorname{v}(G) - 1$ edges (Theorem 4.25). It implies that $\operatorname{me}(G) = \operatorname{v}(G) - 1 + \operatorname{fmesn}(G)$. ◀

The graph shown in the Figure 4.3 has the feedback multi-edge set number equal to 2. It means that only two multi-edges must be removed to obtain a multi-tree. Those two multi-edges are highlighted with red and blue colors. Please note that we may have removed the edge between vertices b and e, which would have resulted in a multi-tree as well. In this case, we have to remove fewer edges to obtain a multi-tree than shown in the figure. However, by the definition, those two solutions are equal. This is because we are removing exactly two multi-paths in both cases.

The feedback multi-edge set number can be used to impose constraints on the input graph. For example, if the algorithm accepts only graphs with feedback multi-edge set number zero, it means that only multi-trees are accepted. If the feedback multi-edge set number equals 1, then only one multi-edge must be removed to obtain a multi-tree from a given graph, and so on. What consequences does the limiting of the feedback multi-edge set number have? It can affect the maximum possible number of multi-paths between two vertices in the graph. In the case



■ **Figure 4.3** Feedback multi-edge set of the graph of size 2 (highlighted with red and blue colors)

of multi-tree (the feedback multi-edge set number equals zero), by definition, there is just one multi-path between every two distinct vertices. If the feedback multi-edge set number equals 1, the maximum number of multi-paths increases by 1 since there is just one additional multi-edge that can be used in a certain multi-path. To understand the effect of the constraint imposed, the general question must be answered: *What is the maximum number of multi-paths between two vertices in a graph having the feedback multi-edge set number equals to K, i.e., how the number of multi-paths in a graph depends on the feedback multi-edge set number of that graph?*

The following theorem gives an answer to this question.

▶ **Theorem 4.32.** *Given a connected graph G, the number of multi-paths between every two vertices of the graph is at most $2^{\text{fmesn}(G)}$.*

Proof. Without loss of generality, suppose we are searching for all multi-paths between the vertices s and t. Let the S be a minimal feedback multi-edge set of the graph G. We know that the size of S is $\text{fmesn}(G)$. For every subset of S, there is at most one unique multi-path between s and t that traverses all the edges in the subset. Let us prove it.

Suppose for a subset $S_i \in S$ there is a multi-path from s to t. We want to prove that there are no other multi-paths between these edges. Let us prove it by contradiction. Suppose there is another multi-path. This could possibly happen in two cases:

1. The second multi-path consists of the same set of multi-edges but the order they are included in the sequence is different.
2. The second multi-path consists of another sequence of multi-edges.

In the first case, suppose we have two multi-paths MP_1 and MP_2 that share the set of multi-edges but include it in a different order. This means that there is two multi-edges ME_a and ME_b such that in MP_1 ME_a precedes ME_b, and in MP_2, vice versa, ME_b precedes ME_a. This is true due to the different order of multi-edges in these multi-paths. This means that we can traverse all the multi-edges from s to ME_b in MP_1, then traverse all the multi-edges from ME_b to ME_a in MP_2, and finally, traverse all the edges from ME_a to t. This means that there is a cycle in the multi-path that is a contradiction by the definition of the multi-path.

In the second case, we can neither remove multi-edges from S_i nor add other multi-edges from S since it will break our initial assumption that we are traversing only multi-edges from S_i. The only option is adding or removing multi-edges that were initially in the multi-tree $(G - S)$. However, this would mean that there is a cycle in the initial multi-tree which is wrong by definition.

The fact that there is at most one multi-path from s to t traversing all the multi-edges in every subset of S implies that there is at most $2^{\text{fmesn}(G)}$. This is because S has exactly $2^{\text{fmesn}(G)}$ subsets. ◀

▶ **Theorem 4.33.** *Given a connected graph $G = (V, E)$ and two vertices $s, t \in V$, all multi-paths from s to t can be found in $\mathcal{O}((|V| + \text{fmesn}(G)) \cdot 2^{\text{fmesn}(G)})$.*

Proof. We have previously shown that the maximum number of multi-paths in G is $2^{\text{fmesn}(G)}$. To find all multi-paths between two given vertices in G, we can use a traversal algorithm, such as depth-first search (DFS) or breadth-first search (BFS).

In the worst-case scenario, we have $2^{\text{fmesn}(G)}$ multi-paths between the given vertices, and for each multi-path, we have to traverse the maximum number of vertices ($|V|$) and the maximum number of multi-edges ($|V| - 1 + \text{fmesn}(G)$). This is because the number of multi-edges in the graph depends on its feedback multi-edge set number, as shown in Theorem 4.25.

Therefore, the time complexity upper bound for the algorithm to find all multi-paths in G is given by:

$$\mathcal{O}((|V| - 1 + fmesn(G) + |V|) \cdot 2^{fmesn(G)}) \equiv \mathcal{O}((|V| + \text{fmesn}(G)) \cdot 2^{\text{fmesn}(G)})$$

◀

The actual algorithm searching all paths (multi-paths) between two given vertices is introduced later in the bachelor thesis (Section 5.5).

One important thing should be noted here. This upper bound is not the most accurate. For a general graph, it gives the time complexity of searching all paths between vertices $\mathcal{O}(|V| \cdot 2^{|V|^2}$. When all the paths between two vertices could be found in $\mathcal{O}(2^{|V|})$. However, it is just enough to show that the overall time complexity of the algorithm solving Multi-graph Multi-constrained Optimal Path is polynomial when certain constraints are imposed. This will be covered in detail later.

That being said, by limiting the feedback multi-edge set number of the graph by some constant integer K, the overall number of multi-paths in the graph is limited by 2^K, which is also a constant value, even though it is exponential. Choosing a relatively low value of K leads to a polynomial number of paths in the graph w.r.t. to the number of vertices in the graph. As was mentioned before, the overall number of multi-paths in the graph does have an effect on the time complexity. This implies that by imposing constraints on the number of multi-paths, we can achieve a better time complexity.

4.4 NP-Hardness proof

As was mentioned before, the problem of finding an optimal path in a given multi-path is also NP-*hard* considering that the number of edges between two adjacent vertices is not anyhow limited. If this problem is proven NP-*hard*, then the fact that MGMCOP is NP-*hard* becomes obvious. Let us formally define the problem of finding an optimal path in a given multi-path.

▶ **Definition 4.34.** *Given a multi-path MP and two functions $d\colon E \to \mathbb{N}$ and $p\colon E \to \mathbb{N}$, assigning the duration and price values to each edge of the graph, i.e., $\forall e \in E\colon d(e)$ – the duration of passing over the edge e, $p(e)$ – the price of passing e. We are also given a positive integer $d_{\max}$ representing the maximal duration. The Optimal path in a multi-path (OPMP) problem is to find a path P such that $d(P) < d_{\max}$ and the $p(P)$ is minimum possible.*

We can formulate the relation between the OPMP and the MGMCOP problems in the following theorem.

▶ **Theorem 4.35.** *The MGMCOP problem is* NP*-hard if the OPMP problem is* NP*-hard.*

Proof. Formally, to prove that the MULTI-GRAPH MULTI-CONSTRAINED OPTIMAL PATH problem is NP-*hard* given the fact that the OPTIMAL PATH IN A MULTI-PATH is NP-*hard*, we have to prove that OPMP is reducible to MGMCOP (Definition 4.6).

The fact that OPMP is reducible to MGMCOP is obvious since OPMP is a special case of MGMCOP (this will be presented below). This implies that we do not need to reduce the OPMP problem to the MGMCOP problem since every instance of OPMP is also by definition an instance of MGMCOP.

The OPMP problem is a special case of the MGMCOP problem in case if the input graph is a simple multi-path from s to t and we are searching for an optimal path between these two vertices. In this case, solving MGMCOP means solving OPMP, which implies that OPMP is a special case of MGMCOP.

◀

Theorem 4.35 implies that to prove that the MGMCOP problem is NP-*hard*, it is sufficient to prove that the OPMP problem is NP-*hard*.

As was mentioned before, to prove that the problem is NP-*hard* it is required to reduce the problem known to be NP-*hard* to it.

Let us refer to the definition of the OPMP (Definition 4.34) to understand the OPMP problem even better and determine which existing NP-*hard* problem can be reduced to it. Basically, what this definition is saying is that given N sequences, each of which contains N_i objects described by two parameters p and d, the objective is to choose exactly one object from each of the sequences such that the sum of d is less than some given integer and the p is minimal among other solutions. So the goal is to choose one object from each of the N groups. One can notice that this definition is fairly close to the definition of the KNAPSACK problem (Definition 3.7). Obviously, they are not quite the same. Nevertheless, there are some vital similarities. Objects in the KNAPSACK problem are defined in a similar way as edges in OPMP. In both cases, there are two parameters, and the final goal is also to find a combination that satisfies the given condition for one parameter while optimizing the value of another one. There is a variation of the KNAPSACK problem that is even more similar to the OPMP problem. This is the MULTI-CHOICE KNAPSACK problem defined before (Definition 3.8).

The MULTI-CHOICE KNAPSACK problem is exactly what is needed. Basically, it defines the OPMP problem using other words while keeping the same theoretical and mathematical meaning. What is even more important is that this problem is proven NP-*hard* problem [7]. This is because it is a special case of the original KNAPSACK problem. Given N different objects as in the original problem, they can be split into N different groups, each group contains one object from a given set, and one *empty* object that has both weight and value set to zero. Adding this *empty* object to the knapsack does not change the state of the knapsack, and hence it corresponds to the situation when the object was not chosen at all in the original problem. This simple modification of the problem makes the reduction of the NP-*hard* problem, i.e., MULTI-CHOICE KNAPSACK problem, to OPMP reasonably obvious.

▶ **Theorem 4.36.** *OPTIMAL PATH IN A MULTI-PATH problem is* NP*-hard.*

Proof. As proof of the NP-hardness of the OPTIMAL PATH IN A MULTI-PATH, the polynomial reduction algorithm from the MULTI-CHOICE KNAPSACK problem is provided. Given an instance $\mathcal{I}$ of the MULTI-CHOICE KNAPSACK problem with the number of groups K and a knapsack capacity *cap*, the instance of the OPMP problem can be constructed by implementing the following rules:

- Create a graph $G = (V, E)$ with $K + 1$ vertices $V = (v_0, v_1, \ldots, v_K)$ and yet empty set of edges $E = \emptyset$;
- For each group N_i in $\mathcal{I}$ for each element in the group $e_i \in N_i$ add an edge from v_{i-1} to v_i with price equals the element's value, and duration equals the element's weight;
- Assign the capacity of the knapsack to the maximal duration of the path $d_{\max} := cap$;
- Set the v_0 as the source vertex and v_k as the destination

This algorithm takes $\mathcal{O}(n)$, where n is the number of all elements among all groups, i.e., it is linear to the size of the input because it does not require any computations to be made since it just restructures the input structure to the needed form requiring constant time for each element.

An algorithm for OPMP will return a valid solution for the MK problem. Because it will find a combination of edges (objects in the initial problem) that make up a path (knapsack in the initial problem) that is no longer than the given duration (capacity) and minimal price (weights). We can find a path with a maximal price simply by changing the functor (max instead of min). There is another way around it. We can multiply the price of every edge by -1 and find a minimal price. Then, we can invert the solution price, which would be the maximum possible price.

It implies that OPMP is at least as hard as the MULTI-CHOICE KNAPSACK problem that makes the OPMP problem NP-*hard*. ◀

The OPMP problem has been proven to be NP-*hard*. This implies that the MGMCOP problem is also NP-*hard* (Theorem 4.35).

▶ **Theorem 4.37.** *Given a graph G, assuming the time complexity of solving the OPMP problem is known, and it is $\mathcal{O}(T)$, the time complexity of solving the MGMCOP problem with a given graph is $\mathcal{O}(T \cdot 2^{\text{fmesn}(G)})$*

Proof. Solving the MGMCOP problem for graph G means solving OPMP problem for each multi-path from the source vertex to the destination. The maximal number of multi-paths between any two nodes for a graph G is limited by its feedback multi-edge set number (Theorem 4.32) and equals $2^{\text{fmesn}(G)}$, which implies that OPMP problem must be solved $2^{\text{fmesn}(G)}$ times. The time complexity of one run of OPMP solver is $\mathcal{O}(T)$ multiplied by $2^{\text{fmesn}(G)}$ runs of the algorithm results in $\mathcal{O}(T \cdot 2^{\text{fmesn}(G)})$. ◀

This is the optimization on the theoretical level; no matter which algorithm is used to solve OPMP, limiting the feedback multi-edge set number of the graph will reduce overall time complexity. However, it does not answer the question of how an algorithm solving OPMP can be optimized; thus, the overall time complexity is still non-polynomial. The next chapter aims to solve that issue on the practical level, i.e., it provides the reader with an algorithm that can be used to solve both OPMP and MGMCOP problems, analyzes the algorithm, and, based on its properties, sets some limitations on the input instance of the problem so the algorithm can work faster. This is possible because many algorithms work better for one input data and worse for another; for example, consider an algorithm solving the GRAPHICAL TRAVELLING SALESMAN PROBLEM (Definition 3.4. There is no polynomial time algorithm for GTSP found yet. However, if it is known that the input graph is always one simple cycle and the lengths of edges are equal, the time complexity reduces to $\mathcal{O}(N)$ because the optimal solution, obviously, is to go through the entire cycle. Although this limitation allows optimizing the algorithm, it makes the problem useless since searching for a cycle in the cycle is the situation when the input graph is basically the solution. This is just an example; the goal is to strike a harmonious balance between imposing constraints on the input data and preserving the integrity of the problem itself. This will be covered in the next chapter.

Chapter 5

Analysis and implementation

5.1 Structure of the chapter

This chapter covers the algorithm solving the MGMCOP problem. The algorithm is inspired by the algorithm for the MK problem that will be discussed in Section 5.2.

Later in this chapter, the pseudo-code of this algorithm is introduced to the reader, along with proof of its correctness and time complexity.

Finally, reduced time complexity after imposing constraints is covered in this chapter.

5.2 The Multi-choice Knapsack problem

We have proven the MGMCOP problem to be NP-*hard* by introducing the polynomial Karp reduction algorithm for the Multi-choice Knapsack problem. Since the reduction was quite straightforward, we may reuse its algorithm to solve our problem. In order to review the main concern, let us briefly go over the problem once again. In the Multi-choice Knapsack problem we are given objects split into several different classes referred to by N_i (i is the index of the class). Each object has its weight and price parameters denoted by w_{ij} and p_{ij} respectively (i stands for the index of the class, j is the index of the object in the class). The objective is to choose exactly one object from each class such that the total weight is less than the defined maximum capacity and the price is the maximum possible.

The algorithm utilizes the dynamic programming principle. Generally speaking, dynamic programming is a problem-solving method that helps one find the best solution by breaking down a complex problem into smaller, simpler parts and storing their solutions in some data structure, which is often a table. Imagine we are trying to solve a big puzzle. Instead of trying to put all the pieces together at once, we start by solving smaller sections of the puzzle and then combine those solutions to complete the entire puzzle.

In dynamic programming, we also store the solutions of the smaller problems so that we do not have to solve them again later. This helps save time and makes the process more efficient. It is like remembering the steps we took to solve a math problem, so we do not have to start from scratch every time we encounter a similar problem.

To do that, we have to develop a recursive equation describing how the solution of the harder instance of the problem depends on the simpler one.

The dynamic programming algorithm shown below is introduced by Dudzinski and Walukiewicz [10]. The reader can refer to the original source for more details of the algorithm.

In the case of Multi-choice Knapsack we will store a two dimension table of solutions (called *dp*) defined as follows. Columns indices of the *dp* will correspond to the number of classes used. The range of values will be from 1 to the total number of classes M. It means that the column i will store solutions using only objects from the first i classes. Rows will be indexed by integer values from 1 to the defined maximal capacity of the knapsack C. It means that the dp at the column i and the row j will store the highest price possible with objects from the first i classes and a total weight of no more than j. Finally, the recursive equation describing how the entries of the dp depend on each other can be defined as follows.

$$dp[i][j] = \max_{1 \leq u \leq |N_i|, w_{iu} \leq j} (dp[i-1][j - w_{iu}] + p_{iu}) \tag{5.1}$$

When calculating the optimal value for the class i, we refer to the best solutions using the first $i-1$ classes, iterating over the objects in the N_i whose weight is lower than the current maximal capacity and choosing the highest solution. That means that dp contains the overall best solution in the right bottom field. This is because it stores the best solution using all the classes of objects with a total weight lower than the given capacity.

The only remaining thing is to fill the table with trivial instances so we can derive the other instances from them. In our case, we can fill the first column of the table using the following equation.

$$dp[1][j] = \max_{1 \leq u \leq |N_0|, w_{0u} \leq j} (p_{0u}) \tag{5.2}$$

The first column contains the prices of the objects just from the first class. It means that we can simply fill the table with the highest prices of the objects whose weight is lower than the current capacity j.

However, filling the table in dynamic programming does not provide us with the whole solution of the problem. In our case, we store the best price possible, not the objects that should have been chosen to reach that price. Nevertheless, we can restore the solution from the table. Generally, this process can be defined as follows.

In dynamic programming, once we have filled the table with solutions to smaller subproblems, we can rebuild the final solution by tracing back through the table. This process, called *reconstructing the solution*, allows us to figure out the steps or components that make up the optimal solution. Here is a general description of the process:

1. Start at the table entry that corresponds to the optimal solution of the entire problem. This is usually found in the last row or column of the table, depending on the problem we are solving.

2. Look at the table entry and identify the smaller subproblems it relies on. This often involves checking the neighboring cells in the table or, in some cases, specific patterns based on the problem we are solving.

3. Move to the table entry or entries associated with the smaller subproblems we identified in the previous step.

4. Repeat steps 2 and 3, moving through the table and identifying the smaller subproblems that contribute to the overall solution. Continue this process until we reach the base case or the starting point of the table.

5. As we trace back through the table, we have to note the choices or components that contribute to the optimal solution. These could be specific items, paths, or any other elements relevant to your problem.

6. Once we have reached the starting point, we should have a sequence of choices or components that make up the optimal solution. We have to arrange these components in the correct order to construct the final solution.

By following these general steps, we can reconstruct the optimal solution for a problem using dynamic programming, leveraging the table of smaller subproblem solutions we have already computed. We have to keep in mind that the specific process for reconstructing the solution will depend on the problem we are working on, so we have to adapt these steps accordingly.

In our case, starting with the right bottom field of the table, we are iterating over all the objects in the corresponding class, calculating the price and weight difference with the fields from the previous column. If the price and weight have matched, we have found the object that should have been chosen. We store that object and move on to the found field of the table. As soon as we reached the first column, we found all the required objects and, hence, the solutions to the original problem.

Formally, the price and the weight match if the following condition is satisfied.

$$\exists u : dp[i][j] = dp[i-1][j - w_{iu}] + p_{iu} \tag{5.3}$$

If the condition is satisfied for some u, we can store the object by its indices i (index of the class) and u (index of the object in the class). Finally, when we iterated over the entire table, we found indices of objects in the corresponding classes. Those objects are the solution of the MULTI-CHOICE KNAPSACK.

Please note that there is an alternative way of filling the trivial fields in the *dp*. If we add an additional column and row, representing the instance when we are not using any classes and the capacity is 0, respectively. Then, we can fill every field in those column and row with 0, because a knapsack without any object in it has a zero price, and similarly, we can not put any object to a knapsack with 0 capacity, thus, we have a 0 price. This way is even simpler to implement, because we do not have to use a separate equation for the first column of the table as was explained before.

Algorithm 1 Filling the dp table for the multiple choice knapsack problem

```
1: procedure FILLDPTABLEMK($N, C, p, w$)
2:     Initialize a table $dp$ with dimensions $(N+1) \times (C+1)$
3:     for $i \leftarrow 0$ to $N$ do
4:         $dp[i][0] \leftarrow +\infty$    ▷ we can not put any object to the knapsack if its capacity is zero
5:     end for
6:     for $j \leftarrow 0$ to $C$ do
7:         $dp[0][j] \leftarrow 0$  ▷ we can not put any object to the knapsack without using any class of objects
8:     end for
9:     for $i \leftarrow 1$ to $N$ do
10:        for $j \leftarrow 1$ to $C$ do
11:            $dp[i][j] \leftarrow dp[i][j-1]$
12:            for $u$ in $N_i$ do
13:                if $w_{iu} \leq j$ then
14:                    $dp[i][j] \leftarrow \max(dp[i][j], dp[i-1][j-w_{iu}] + p_{iu})$
15:                end if
16:            end for
17:        end for
18:    end for
19:    return $dp$
20: end procedure
```

This algorithm fills the table with values of optimal solutions of the corresponding subproblems. The overall best solution is stored in the $dp[N][C]$ field.

The following algorithm rebuilds the exact solution from the calculated table.

Algorithm 2 Rebuilding the solution from the DP table

```
procedure ReconstructSolutionMK($dp, N, C, p, w$)
    Initialize an empty set $solution$
    $i \leftarrow N$
    $j \leftarrow C$
    while $i > 0$ and $j > 0$ do
        $chosen \leftarrow None$
        for $u$ in $N_i$ do
            if $w_{iu} \le j$ and $dp[i][j] = dp[i-1][j - w_{iu}] + p_{iu}$ then
                $chosen \leftarrow u$
                break
            end if
        end for
        if $chosen \ne None$ then
            Add the chosen item $(i, chosen)$ to the $solution$
            $j \leftarrow j - w_{i,chosen}$
        else
            Error: No matching objects found
        end if
        $i \leftarrow i - 1$
    end while
    return $solution$
end procedure
```

Please notice that although this pseudo code contains error conditions, those will never happen if the table is valid, i.e., if the table has been successfully calculated and contains a valid solution at $dp[N][C]$, it is guaranteed that the solution exists and it is valid.

5.3 The Optimal path in a multi-path problem

As we have already reduced the Multi-choice Knapsack problem to the Optimal path in a multi-path problem and have shown that the algorithms for these two problems are basically the same, we can map the OPMP problem to the algorithm presented in the previous section (Algorithm 1, Algorithm 2).

First, we should define a recursive equation for the OPMP problem. It is almost the same as the equation for the MK problem, which was introduced above (Equation (5.1)). The only difference is that we use min instead of max since we seek a path with the lowest possible price. Also, we are using different notations for functions.

$$dp[i][j] = \min_{e \in MP_i, d(e) \le j} (dp[i-1][j - d(e)] + p(e)) \tag{5.4}$$

$$dp[1][j] = \max_{e \in N_0, d(e) \le j} (p(e)) \tag{5.5}$$

$$\exists e : dp[i][j] = dp[i-1][j - d(e)] + p(e) \tag{5.6}$$

The algorithm for the OPMP problem is provided below.

Algorithm 3 Filling the dp table for the Optimal path in a multi-path problem

```
procedure FillDPTableOPMP($MP, d_{\max}$)
    Initialize a table $dp$ with dimensions $(|MP|) \times (d_{\max} + 1)$
    for $j \leftarrow 0$ to $d_{\max}$ do
        $dp[1][j] \leftarrow 0$        ▷ we need zero price to get from the source vertex to itself
    end for
    for $i \leftarrow 2$ to $|MP|$ do
        $dp[i][0] \leftarrow \infty$   ▷ we can not get to any vertex (except the source) with zero duration
    end for
    for $i \leftarrow 1$ to $|MP|$ do                    ▷ Iterate through multi-edges
        for $j \leftarrow 1$ to $d_{\max}$ do            ▷ Iterate through possible duration
            $dp[i][j] \leftarrow dp[i][j-1]$
            for each edge $e$ in $MP_i$ do             ▷ Iterate through edges in the multi-edge
                if $d(e) \leq j$ then
                    $dp[i][j] \leftarrow \min(dp[i][j], dp[i-1][j-d(e)] + p(e))$
                end if
            end for
        end for
    end for
    return $dp$
end procedure
```

We also need to modify the algorithm that reconstructs the actual solution from the dynamic programming table.

Algorithm 4 Rebuilding the solution from the DP table

```
procedure ReconstructSolutionOPMP($dp, MP, d_{\max}$)
    Initialize an empty sequence $solution$
    $i \leftarrow |MP|$
    $j \leftarrow d_{\max}$
    while $i > 0$ and $j > 0$ do
        $chosen \leftarrow None$
        for each edge $e$ in $MP_i$ do
            if $d(e) \leq j$ and $dp[i][j] = dp[i-1][j-d(e)] + p(e)$ then
                $chosen \leftarrow e$
                break
            end if
        end for
        if $chosen \neq None$ then
            Add the chosen item to the $solution$
            $j \leftarrow j - d(chosen)$
        else
            Error: No matching edges found
        end if
        $i \leftarrow i - 1$
    end while
    return $solution$
end procedure
```

Now, we can combine these two algorithms into one, which will take the multi-path and the duration limit and return the optimal path. To do that we should simply run two previous algorithms (Algorithm 3, Algorithm 4) one after the other, sending the output of the first one to the input of the second. The whole algorithm solving the OPTIMAL PATH IN A MULTI-PATH is presented below.

Algorithm 5 Optimal path in a multi-path

```
1: procedure OPMP($MP, d_{\max}$)
2:     $dp \leftarrow$ FILLDPTABLEOPMP($MP, d_{\max}$)
3:     return RECONSTRUCTSOLUTIONOPMP($dp, MP, d_{\max}$)
4: end procedure
```

5.3.1 Correctness

This section provides the reader with proof of the correctness of the OPMP algorithm.

We can prove the correctness of the OPMP algorithm by first proving the recursive equation introduced above (Equation (5.4)). Then we should prove that Algorithm 5 correctly implements the equation.

Proof. [Recursive equation correctness] Equation (5.4) is correct if it contains the lowest possible price for the path in the given multi-path at $dp[L][d_{\max}]$, where L is the length of the given multi-path and $d_{\max}$ is the maximal duration.

Prove by induction.

Base case: The base instance of the problem is introduced in Equation (5.5). It is the OPMP problem applied to the multi-path of length 1. It means that the multi-path contains only one multi-edge. Obviously, the lowest possible price of the multi-path is the lowest price of an edge among the edges in the multi-edge. This is what is defined in Equation (5.5). This equation is obviously correct. It means that for $i = 1$ and $j \in \{1, \ldots, d_{\max}\}, dp[i][j]$ contains the minimal price for a given configuration.

Inductive step: Suppose that for $j \in \{1, \ldots, d_{\max}\}, dp[k][j]$ contains the minimal price. Let us prove that for $k + 1$.

Since for each $j \in \{1, \ldots, d_{\max}\}$ we are iterating through all the edges in the multi-edge ME_{k+1} and choosing the minimal sum of the edge's price and the value of the previous state (k), we obtain the minimal possible price for $k + 1$. It means that $dp[k + 1][j]$ also contains the minimal possible price.

◀

This proves that $dp[L][d_{\max}]$ contains the minimal possible price of the path in the given multi-path.

Now we should prove that Algorithm 5 correctly implements Equation (5.4) and sets correct values of base instances as defined in Equation (5.5).

Proof. [Algorithm correctness] In line 2 of Algorithm 5 we call the FILLDPTABLEOPMP procedure. In line 4 of Algorithm 3 we set the additional column values to 0. This column represents the minimal price of the path from s to s. The path from vertex to itself is empty, thus, the price of the path is 0.

Then, in line 9 we iterate over all the multi-edges in the given multi-path. In line 10 we iterate through all the possible values of duration (from 1 to $d_{\max}$). Finally, in line 12 we iterate over all the edges in the multi-edge.

When we fill the values for the first multi-edge, we set the values as defined in Equation (5.5). This is because the previous column contains zero values, as it was set in line 4. It means that the base instances of the problem are set correctly.

For all the other multi-edges, we calculate the minimal sum of the edge price and the value of the optimal solution at the previous step. Lines 13-15 implement Equation (5.4). As we can see, Equation (5.4) is replicated in line 14.

Finally, the whole *dp* table is returned.

Now, we should prove that the algorithm reconstructing the solution from the calculated table is correct (Algorithm 4). In lines 3-4, we set an initial position in the table. This is the $dp[|MP|][d_{\max}]$ entry, where MP is the given multi-path and $d_{\max}$ is the maximal duration. This field contains the minimal price for the given multi-path. In line 5, we iterate until the $dp[1][1]$ field is reached, i.e., the initial state. In line 7, we iterate through all the edges in the multi-edge ME_i and validate the condition (Equation (5.6)). If the condition is matched, we add the found edge to the solution in line 14, decrease the current duration value in line 15, and decrease the current column number in line 19.

When we reach the initial state, i.e., $dp[1][1]$, we found the sequence of edges that makes up the solution. ◀

5.3.2 Time complexity

To determine the time complexity of the Algorithm 5, we should cover the entire algorithm line by line.

First, we have to fill the dynamic programming table. We do this by calling the FillDPTableOPMP procedure in line 2 of algorithm 5.

In the FillDPTableOPMP procedure we have the table with dimensions of sizes $d_{\max} + 1$ and $|MP| + 1$. The latest depends on the length of the input multi-path. We may impose the upper bound on the length of the multi-path in the graph as $|V|$, which corresponds to the multi-path traversing all the vertices in the graph. Obviously, by the path definition, there could not possibly be a longer multi-path in the graph. This is because having a longer multi-path means that some vertex is presented in the multi-path more than once. It immediately conflicts with the multi-path definition.

We are calculating the value of each field only once, iterating through all the edges in each multi-edge. So we have nested for loop (Algorithm 3, Line 9-18): $|MP| \times d_{\max} \times e_{max}$, where e_{max} is the maximal number of edges between two vertices in the graph.

We require almost the same number of operations to reconstruct the solution, except that we are iterating just through $|MP| \times e_{max}$ fields because we start in the last field and always go to the one in the previous column.

We obtain the following time complexity if we replace the $|MP|$ with its upper bound $|V|$.

▶ **Statement 5.1** (The OPMP time complexity). *The OPMP algorithm time complexity is:* $OPMP\_COMPLEXITY := \mathcal{O}(|V| \cdot d_{\max} \cdot e_{\max} + |V| \cdot e_{\max})$,

where V is the set of vertices of the input graph, e_{max} is the maximal number of edges between two vertices in the graph, $d_{\max}$ is the duration limit.

This statement gives us another hint of which constraints may be imposed to reduce the overall time complexity of the algorithm since it shows us which variables the algorithm's time complexity depends on. This will be covered in detail later in Section 5.7.

5.4 The Multi-graph Multi-constrained Optimal Path problem

In the previous section, we have introduced the algorithm for the Optimal path in a multi-path problem (algorithm 5). As was explained before, the OPMP is a sub-problem of the Multi-graph Multi-constrained Optimal Path problem, so we can now implement the algorithm for the latest problem that will apply the OPMP algorithm as a sub-procedure.

First, the algorithm should find all the multi-paths from the source vertex to the target. Then, it should apply the OPMP algorithm for each of those multi-paths. Finally, it should compare all the solutions and find the optimal one.

For now, let us consider that we have an algorithm to find all the multi-paths for a given multi-graph so we can use it inside the MGMCOP algorithm. The actual algorithm will be defined later.

Algorithm 6 Multi-graph Multi-constrained Optimal Path

```
 1: procedure MGMCOP($G, s, t, d_{\max}$)    ▷ $G$: input graph, $s$: source vertex, $t$: target vertex, $d_{\max}$: duration limit
 2:     solution ← None
 3:     for each multi-path $MP$ in findAllMultiPaths($G, s, t$) do    ▷ Iterate through multi-paths
 4:         tmp_solution ← $OPMP(MP, d_{\max})$
 5:         if tmp_solutions better than solution then
 6:             solution ← tmp_solution
 7:         end if
 8:     end for
 9:     return solution
10: end procedure
```

5.4.1 Correctness

In this section, we will prove the correctness of the MGMCOP algorithm, i.e., the algorithm actually returns the expected solution for all inputs.

This proof relies on the fact that the OPMP algorithm is correct, as it has been proven above (subsection 5.3.1).

Proof. The MGMCOP algorithm returns a correct solution because it iterates through all the possible multi-paths and calculates the optimal path using OPMP algorithm, which is proven to return a valid solution as well. It implies that the MGMCOP algorithm works correctly.

◀

5.4.2 Time complexity

This section contains the time complexity analysis of the MGMCOP algorithm.

Let us go through the algorithm step by step.

In line 3, it calls the findAllMultiPaths. Although this procedure has not yet been defined, the upper bound of the algorithm's time complexity has already been established (Theorem 4.33) – $\mathcal{O}((|V| + \text{fmesn}(G)) \cdot 2^{\text{fmesn}(G)})$.

Then, at the same line, we are iterating through all the found multi-paths. The maximal number of multi-paths in the graph G is bound by $2^{\text{fmesn}(G)}$.

Later, in line 4, we are applying the OPMP algorithm, which, as was proven before, works in $\mathcal{O}(|V| \cdot d_{\max} \cdot e_{\max} + |V| \cdot e_{\max})$.

In line 5, we are comparing two solutions, which may take linear time since given a sequence of edges, we have to sum all the prices of both paths and then compare the obtained values. However, we may consider the time complexity of the comparison to be constant – $\mathcal{O}(1)$. This is because we may store the path price along the sequence of edges, so we do not need to recalculate the price each time we compare two paths.

The overall time complexity of the MGMCOP algorithm, considering the OPMP time complexity equal to $OPMP\_COMPLEXITY$ is $\mathcal{O}((|V| + \text{fmesn}(G)) \cdot 2^{\text{fmesn}(G)} + 2^{\text{fmesn}(G)} \cdot$

OPMP_COMPLEXITY). Since we have already established the OPMP time complexity (Statement 5.1) we can derive the overall MGMCOP time complexity as follows.

▶ **Statement 5.2** (The MGMCOP algorithm time complexity)**.** *The MGMCOP algorithm time complexity is* $\mathcal{O}((|V| + \mathrm{fmesn}(G)) \cdot 2^{\mathrm{fmesn}(G)} + 2^{\mathrm{fmesn}(G)} \cdot (|V| \cdot d_{\max} \cdot e_{\max} + |V| \cdot e_{\max}))$

5.5 All multi-paths between vertices in the graph

To find all the multi-paths between two vertices, it is enough and sufficient to run the DFS algorithm without stopping when the first path is found. In that case, the one important thing that must be done is that the vertices states should be updated after DFS leaves the vertex. This is because a vertex could be traversed several times in terms of one DFS run because each vertex could be a part of multiple paths. Nevertheless, each vertex should be visited at most once in one recursion call stack. This is needed to detect cycles.

Here is the implementation in the pseudo-code of the algorithm.

Algorithm 7 Find all multi-paths between given two vertices

```
1: procedure findAllMultiPaths($G, s, t, currentMultiPath$)   ▷ $G$: input graph, $s$: source
   vertex, $t$: target vertex, $currentMultiPath$ is optional that is by default empty
2:     $solution \leftarrow \emptyset$
3:     if $s = t$ then                              ▷ The termination condition of the recursion
4:         Add $currentMultiPath$ to the $solution$
5:         return $solution$
6:     end if
7:     $visited(s) \leftarrow 1$
8:     for each adjacent vertex $v$ of $s$ do
9:         if not $visited(v)$ then
10:            $e \leftarrow sv$
11:            $sub\_result \leftarrow$ findAllMultiPaths($G, v, t,$ append($currentMultiPath, e$))
12:            $solution \leftarrow$ merge($solution, sub\_result$)
13:        end if
14:    end for
15:    $visited(s) \leftarrow 0$
16:    return $solution$
17: end procedure
```

The join function in Algorithm 7 should append the multi-edge e to the multi-path $currentMultiPath$ in an immutable manner. It means that it should not modify $currentMultiPath$. Instead, it should return a new multi-path that is a copy of $currentMultiPath$ with appended e. The merge function should merge two sets of multi-paths.

5.6 C++ implementation

This section introduces the C++ implementation of the MGMCOP algorithm to the reader. Please note that the implementation does not use the knowledge about the constraints imposed.

The whole idea is to implement the algorithm which will solve the general case of the Multigraph Multi-constrained Optimal Path and test that the algorithm with constrained input data will work in an efficient time.

To test the algorithm's time efficiency, there is a test suite along the C++ implementation. It tests the implementation in the following way. Given the constrained input data test the implementation with another constrained input data which are polynomially bigger than the original one and ensure that the algorithm runs polynomially longer.

A template class called `AdjacentMatrix` is the main class containing all the required algorithms. We use price and duration to describe the edges in the Multi-graph Multi-constrained Optimal Path. However, the templatization of the class enables easily changing the parameters used to describe the edges. This class takes one template argument – traits of the edge parameters used. This class must implement the defined interface. Otherwise, there will be a compilation error. This is ensured by the `std::enable_if` construct. This class provides an interface for edge parameters. It enables dividing the parameters implementation and `AdjacentMatrix` implementation. This is because the `AdjacentMatrix` does not know anything about the internal structure of the parameters. It knows neither the number of parameters nor its value ranges. For example, some parameters may be implemented to have even string values as long as the valid parameters' traits are implemented. For more details, please, refer to the C++ implementation itself. The code is well-documented and relatively easy to understand.

5.7 Constraints

Let us bring the time complexity of the MGMCOP one more time to be able to discuss the constraints that could affect the time complexity.

The MGMCOP works in $\mathcal{O}((|V|+\text{fmesn}(G))\cdot 2^{\text{fmesn}(G)}+2^{\text{fmesn}(G)}\cdot(|V|\cdot d_{\max}\cdot e_{\max}+|V|\cdot e_{\max}))$.

We may iterate through every term in the time complexity formula and explain why or why not we impose particular constraints.

As for both $|E|$ and $|V|$, it seems impractical to impose constraints on the size of the edge or vertex set. Because it would mean that the simplest possible graph, for example, a path with more vertices than the set maximum, will not be accepted by the algorithm, even though the calculation is simple. This is the reason why the size of the graph will not be affected by the imposed constraints.

The next term in the formula is $2^{\text{fmesn}(G)}$, which will be constrained. As it was discussed in the previous chapter, we want to limit the feedback multi-edge set number of the graph to reduce the number of possible multi-paths between two vertices in the graph. This is the first constraint that we are going to impose.

Then, the next variable the algorithm's time complexity depends on is $d_{\max}$, which also should be limited to be polynomially bounded by the number of vertices in the graph $|V|$.

The same rules should be applied to the maximal number of edges between two adjacent vertices. It should also be polynomially bounded by the number of vertices in the graph.

The following list summarizes all the constraints being imposed on the input graph.

- The feedback multi-edge set number constantly bounded – at most 10;
- The maximal duration of the path is polynomially bounded by the size of the input graph – $d_{\max} \sim |V|^d$;
- The maximal number of edges between two adjacent vertices in the graph is polynomially bounded by the size of the input graph – $e_{max} \sim |V|^e$,

where d and e are some chosen constants.

By applying all those three constraints, we get the following time complexity.

▶ **Statement 5.3** (The constrained complexity of the MGMCOP algorithm)**.** *After imposing chosen constraints, the MGMCOP algorithm has the following time complexity –* $\mathcal{O}((|V|+2^{10})\cdot 2^{10}+2^{10}\cdot(|V|\cdot|V|^d\cdot|V|^e+|V|\cdot|V|^e))$*, which is equivalent to* $\mathcal{O}(2^{10}\cdot(|V|^{d+e+1}+|V|^{e+1}+|V|))$*, which can be simplified even more to* $\mathcal{O}(2^{10}\cdot|V|^{d+e+1})$*, where* $|V|^d$ *– polynomial bound of the maximal duration,* $|V|^e$ *– polynomial bound of the maximal number of edges between adjacent vertices,* 2^{10} *– constant bound of the feedback multi-edge set number (10).*

For example, suppose the duration of each edge is measured in minutes. We may impose the following constraint: let the duration limit be at most $|V|^2$. Also, let the number of edges between two adjacent vertices be at most $|V|$. Finally, let the feedback edge-set number of the graph will be at most 10. In this case, the overall time complexity of the algorithm solving MGMCOP is $\mathcal{O}(2^{10} \cdot |V|^4)$

Limitations imposed in the example above are fairly reasonable in real-life examples. Suppose we search for the optimal route from point A to point B in some city. Suppose this city has 100 different public transport stops. It means that the duration limit should be no longer than 10000 minutes, which in most cases will be more than enough. Also, the number of connections between two adjacent stops is no more than 100. This is also true in almost all cases, even assuming that getting to an adjacent stop by a scooter, taxi, walking, bicycle, etc. are different means of transport and thus different connections. It is still hard to create more than 100 different connections. The feedback multi-edge set number is the most tricky constraint in that case. But if we look at some real-world transport maps (e.g., the Prague metro map, German motorways), they actually have the feedback multi-edge set number quite low (just 1 in the case of the Prague metro map). So this limitation is also satisfied in most scenarios. It means that by imposing reasonable limitations, we can achieve the polynomial time complexity of the algorithm.

As we can see, after imposing three chosen constraints, we obtain the polynomial time complexity of the algorithm. And since the main objective of the bachelor thesis is achieved by now, that leads us to the next chapter – *Conclusion*

Chapter 6
Conclusion

This chapter summarizes the results of the bachelor thesis.

In Chapter 3, we have discussed various NP-*complete* and NP-*hard* problems.

Then, in Chapter 4, we have introduced two new problems – the Optimal path in a multi-path and the Multi-graph Multi-constrained Optimal Path problems. We have proven them to be NP-*hard* by providing the polynomial Karp reduction algorithm from the Multi-choice Knapsack problem to the Optimal path in a multi-path problem and proving that OPMP is a subproblem of MGMCOP.

Later, we discussed the constraints of the input graph. By imposing those constraints, we can achieve better time complexity for the algorithm for solving the Multi-graph Multi-constrained Optimal Path problem.

In Chapter 5, we have introduced algorithms for both OPMP and MGMCOP. By analyzing those algorithms, we determined which constraints could be imposed on the input graph to make the algorithm more time efficient. Finally, we have discovered the overall time complexity of the MGMCOP algorithm for constrained input data.

Bibliography

1. BALL, M.O.; MAGNANTI, T.L.; MONMA, C.L.; NEMHAUSER, G.L. *Handbooks in Operations Research and Management Science*. Vol. 7. First. Elsevier Science, 1995. ISBN 978-0444892928. Available also from: `https://www.sciencedirect.com/handbook/handbooks-in-operations-research-and-management-science/vol/7/suppl/C`.
2. KARP, Richard M. *Reducibility among Combinatorial Problems", bookTitle="Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by MILLER, Raymond E.; THATCHER, James W.; BOHLINGER, Jean D. Boston, MA: Springer US, 1972. ISBN 978-1-4684-2001-2. Available from DOI: `10.1007/978-1-4684-2001-2_9`.
3. BONDY, J.A.; MURTY, U.S.R. *Graduate Texts in Mathematics*. Graph Theory. 2008. ISBN 978-1-84628-969-9. Available from DOI: `https://doi.org/10.1007/978-1-84628-970-5`.
4. CORNUÉJOLS, Gérard; FONLUPT, Jean; NADDEF, Denis. The traveling salesman problem on a graph and some related integer polyhedra. *Mathematical Programming*. 1985, vol. 33, no. 1, pp. 1–27. ISSN 1436-4646. Available from DOI: `10.1007/BF01582008`.
5. HASSIN, Refael. Approximation Schemes for the Restricted Shortest Path Problem. 1992, vol. 17, no. 1, pp. 36–42. Available from DOI: `10.1287/moor.17.1.36`.
6. KORKMAZ, T.; KRUNZ, M. Multi-constrained optimal path selection. 2001, vol. 2, 834–843 vol.2. Available from DOI: `10.1109/INFCOM.2001.916274`.
7. DUDZIŃSKI, Krzysztof; WALUKIEWICZ, Stanisław. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*. 1987, vol. 28, no. 1, pp. 3–21. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/0377-2217(87)90165-2`.
8. COOK, Stephen. The importance of the P versus NP Question. 2003, vol. 50, no. 1, pp. 27–29.
9. ARORA, Sanjeev; BARAK, Boaz. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. Available from DOI: `10.1017/CBO9780511804090`.
10. DUDZINSKI, Krzysztof; WALUKIEWICZ, Stanislaw. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*. 1987, vol. 28, no. 1, pp. 3–21. Available also from: `https://EconPapers.repec.org/RePEc:eee:ejores:v:28:y:1987:i:1:p:3-21`.

Content of medium

```
 README.txt ............................ media contents description and instructions
├─ Makefile .......................... Makefile containing build automation instructions
├─ scripts ................................................................ Helper scripts
│  └─ test.sh ................................ A script containing automation test logic
├─ tests ............................................... A directory containing test files
├─ src ...................................................................... Source code
│  ├─ adjacent_matrix.h ......................... Template functions and classes definitions
│  ├─ adjacent_matrix_impl.h ............... Template function and classes implementation
│  ├─ main.cpp ................................................................ main file
│  └─ tests.h ................................... Actual template instantiation and tests
└─ text
   └─ thesis.pdf ........................... the text of the bachelor thesis in PDF format
```