Karlsruhe Institute of Technology

Enumerating Alternative Paths

Bachelor's Thesis of

Tim Domnick

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	TT-Prof. Dr. Thomas Bläsius
Second reviewer:	Dr. rer. nat. Torsten Ueckerdt
Advisors:	Michael Zündorf
	Adrian Feilhauer

13 May 2024 – 13 September 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 13 September 2024

(Tim Domnick)

Abstract

Modern navigation systems often provide users with the option of choosing from a selection of routes, in addition to the shortest available route. Even if the suggested routes are not optimal, they should nevertheless appear reasonable to the user. These *alternative paths* are therefore required not to contain any unnecessary local detours and not to be considerably longer than the shortest path. In practice, heuristic algorithms are usually employed to efficiently generate a small set of choices.

In this thesis, we study the theoretical question of how all such alternative paths can be enumerated exhaustively. We present three algorithmic approaches that are able to solve this problem. They are based on the enumeration of shortest paths, backtracking and shortest-path trees, respectively. For all three approaches we demonstrate that they have super-exponential runtime in the worst case. We further discuss different variations of the problem, namely via paths, which are composed of several shortest paths between so-called via vertices, unweighted graphs and alternative paths that contain cycles. Under certain conditions regarding the input parameters, we can show that the number of via vertices in via paths and the number of occurrences of a vertex in alternative paths with cycles can be bounded. For unweighted graphs, we provide an input transformation that allows alternative paths to be computed more efficiently in certain cases.

Zusammenfassung

Moderne Navigationssysteme bieten ihren Nutzern oft die Möglichkeit, neben der kürzesten verfügbaren Route auch aus weiteren Routen zu wählen. Selbst wenn die vorgeschlagenen Routen nicht optimal sind, sollten sie dem Nutzer dennoch sinnvoll erscheinen. Diese *Alternativpfade* dürfen daher keine unnötigen lokalen Umwege enthalten und nicht wesentlich länger als der kürzeste Pfad sein. In der Praxis werden in der Regel heuristische Algorithmen eingesetzt, um effizient eine kleine Menge von Alternativen zu generieren.

In dieser Arbeit untersuchen wir die theoretische Frage, wie alle derartigen Alternativpfade erschöpfend aufgezählt werden können. Wir stellen drei algorithmische Ansätze vor, mit denen sich dieses Problem lösen lässt. Sie basieren auf der Aufzählung kürzester Pfade, Backtracking bzw. Kürzeste-Wege-Bäumen. Für alle drei Ansätze zeigen wir, dass sie im schlimmsten Fall eine superexponentielle Laufzeit haben. Darüber hinaus diskutieren wir verschiedene Varianten des Problems, nämlich Via-Pfade, die aus mehreren kürzesten Pfaden zwischen sogenannten Via-Knoten zusammengesetzt werden, ungewichtete Graphen sowie Alternativpfade, die Zyklen enthalten. Unter bestimmten Bedingungen hinsichtlich der Eingabeparameter können wir zeigen, dass die Anzahl der Via-Knoten in Via-Pfaden und die Anzahl der Vorkommen eines Knotens in Alternativpfaden mit Zyklen beschränkt werden kann. Für ungewichtete Graphen stellen wir eine Transformation der Eingabe vor, mithilfe derer Alternativpfade in bestimmten Fällen effizienter berechnet werden können.

Contents

1 Introduction

When traveling from one place to another, especially by car, we often turn to navigation systems for help. Upon request, these systems suggest routes to the desired destination. Many commercial navigation systems provide the user with a small set of possibly suboptimal options. The user can then choose the option that best suits their individual preferences. These preferences can be diverse and are often based on the user's knowledge and therefore hard to quantify. A user may, for example, have a preference for or against particular route sections or road types, such as highways. In addition, a user may prefer routes that offer more opportunities for refueling, rest stops or similar amenities. All these reasons motivate the search for such route options, which we call *alternative paths*. We adapt the definition by Abraham, Delling, Goldberg, and Werneck and require that alternative paths must not contain any short detours nor be overly longer than the shortest path [ADGW13]. These criteria are referred to as *local optimality* and *maximum stretch*, respectively. Navigation systems generally use heuristics to efficiently produce a small number of options. In this thesis, we are be concerned with finding all such alternative paths. To the best of our knowledge, it is an open question whether the set of all alternative paths can be computed efficiently.

1.1 Related Work

The problem of finding alternative paths is closely related to the problem of finding the shortest path. This is a well-established problem in computer science that has been studied extensively for a long time. Today, there are a variety of algorithms that solve different variations of the problem on different kinds of graphs. Arguably the best known of these algorithms is Dijkstra's algorithm [Dij59], which we use in Section 4.1. Additionally, Yen's algorithm can be used for enumerating shortest paths in order [Yen71].

Our definition of alternative paths is based on the work of Abraham, Delling, Goldberg, and Werneck [ADGW13]. Their definition involves three criteria for alternative paths, namely limited sharing, local optimality and bounded stretch. We focus on the latter two, because with limited sharing, the order in which we find alternative paths would affect whether subsequent paths also qualify as alternative paths. They also introduce the concept of *single-via paths* which are obtained by concatenating two shortest paths. This is a restriction of alternative paths that is easier to deal with in practice. We discuss this in more detail in Section 5.1. Moreover, they present and analyze several algorithms that employ heuristics to find alternative paths.

Other work also uses the criterion of local optimality, which is intended to eliminate short, seemingly unnecessary detours. Döbler and Scheuermann quantify this notion in terms of a *locality optimality ratio* [DS16]. Their work includes an algorithm for finding the most locally-optimal paths while also restricting themselves to single-via paths. Fischer seeks to identify almost all locally optimal single-via paths between many pairs of vertices, whereby the number of missed paths can be arbitrarily reduced at the expense of runtime [Fis20]. The algorithm presented in their work is based on one of the heuristic algorithms by Abraham, Delling, Goldberg, and Werneck [ADGW13].

Bader, Dees, Geisberger, and Sanders propose *alternative graphs* as a compact representation of multiple alternative paths [BDGS11]. For their definition of alternative paths, they provide several potential criteria that can be used to evaluate the quality of alternative graphs. They show that the optimization of two such criteria is already $\mathcal{NP}$-hard and therefore also provide an overview on heuristic algorithms, namely Pareto, Plateau and Penalty methods. They experimentally compare these methods and evaluate the results through a user study.

1.2 Outline

We begin in Chapter 2 by establishing the fundamental concepts and notation in the field of graph theory and enumeration problems that we use throughout this thesis. After that, we formally introduce the notion of alternative paths along with the associated enumeration problem in Chapter 3. Here, we also briefly address the computational complexity of the problem by showing that it belongs to the class EnumP. In Chapter 4, we present three algorithms for solving this problem. The first approach, described in Section 4.1, employs algorithms for enumerating shortest paths. The algorithm in Section 4.2 applies backtracking. Lastly, in Section 4.3, we use shortest-path trees to generate alternative paths. In Chapter 5, we continue with variations of the alternative paths problem. We first discuss via paths in Section 5.1 and then consider unweighted graphs and cyclic alternative paths in Sections 5.2 to 5.3. Finally, we draw a conclusion in Chapter 6 and provide an outlook on open questions.

2 Preliminaries

In this thesis, we focus on enumerating alternative paths in a given graph. We therefore provide fundamental definitions and notation of both graph theory and enumeration problems in the following.

2.1 Graph Theory

Graphs A *(directed) graph* $G = (V, E)$ is a pair consisting of a finite, non-empty set of *vertices* V and a set of *edges* $E \subseteq V \times V$. We denote the cardinalities of these sets by $n := |V|$ and $m := |E|$. A vertex $v \in V$ is said to be *adjacent* to another vertex $u \in V$ if $(v, u) \in E$. The function $w : V \times V \to \mathbb{R}_{>0} \cup \{\infty\}$ assigns a positive *weight* to every pair of vertices, where $w(e) = \infty \iff e \notin E$.

Paths A *path* in G is a sequence $p = (v_1, \ldots, v_k)$ of vertices where $k \geq 1$ and $(v_i, v_{i+1}) \in E$ for all $i \in \{1 \,..\, k-1\}$. The number of edges in p is called its *length*, denoted by $|p| := k - 1$. The path's *(total) weight* $w(p)$ is the sum of its individual edge weights: $w(p) := \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. A path is *simple* if all its vertices are pairwise distinct; otherwise, it is called *cyclic*. A graph is *acyclic* if all its paths are simple, and *cyclic* if it contains a cyclic path. Directed, acyclic graphs are abbreviated as *DAGs*.

Subpaths The i-th vertex in a path $p = (v_1, \ldots, v_k)$ is denoted by $p[i] := v_i$ for $i \in \{1 \,..\, k\}$. A *subpath* of p is a subsequence of vertices $p[i \,..\, j] := (p[i], \ldots, p[j])$ where $1 \leq i \leq j \leq k$. A subpath of p is called a *prefix* of p if $i = 1$ and a *suffix* of p if $j = |p| + 1$. Two paths $p_1 = (v_1, \ldots, v_{k-1}, x)$ and $p_2 = (x, u_2, \ldots, u_{k'})$, which share a vertex $x \in V$, can be concatenated to a path $p_1 + p_2 := (v_1, \ldots v_{k-1}, x, u_2, \ldots u_{k'})$ of length $|p_1 + p_2| = |p_1| + |p_2|$.

Shortest Paths Given two vertices $s, t \in V$, a *shortest path* $\hat{p}$ is an s-t path, i.e. $\hat{p} = (s, \ldots, t)$, with minimal weight among all s-t paths. Shortest paths are said to be *optimal*. The *distance* from s to t, denoted by $d(s, t)$, is defined as $d(s, t) := w(\hat{p})$, or $d(s, t) := \infty$ if no s-t path exists. Many shortest path algorithms find either the shortest paths from a given source vertex to all other vertices, or the shortest paths between all pairs of source and target vertices. We refer to these as *SSSP* (single-source shortest paths) or *APSP* (all-pair shortest paths) algorithms, respectively.

Trees A graph $G = (V, E)$ is called a *tree* if a *root* vertex $r \in V$ exists such that for every $v \in V$ there is exactly one path from r to v. In a *shortest-path tree*, these paths represent the shortest paths from r to all other vertices in an underlying graph. Such a shortest-path tree can be obtained using an SSSP algorithm.

2.2 Enumeration Problems

Many theoretical problems in computer science are *decision problems* where we are interested in whether a solution exists to a given problem instance. Most decision problems can also be stated in the form of *search problems* or *counting problems*. With search problems, we want to find any solution, while counting problems ask for the number of existing solutions. *Enumeration problems* add to these classes of problems. In their case, we want to retrieve all existing solutions. Enumeration problems along with their complexity have been extensively studied by Strozecki [Str10]. We adapt the following definitions from their work.

Enumeration Problems We call a binary predicate Π a *problem*. Values for the first operand are referred to as *instances* of Π. A value y is called a *solution* to an instance x if $\Pi(x, y)$ holds. The corresponding enumeration problem EnumΠ is a function which maps instances to their set of solutions, i.e., $x \mapsto \{y \mid \Pi(x, y)\}$. We require that the solution set for every instance is finite. An algorithm $\mathcal{A}$ is said to solve the enumeration problem if, given an instance x, it outputs all solutions to x without redundancy. In other words, the output of $\mathcal{A}$ is a sequence $y_1, \ldots, y_n$ such that $\{y_1, \ldots, y_n\} = \{y \mid \Pi(x, y)\}$, and $y_i \neq y_j$ for all $i \neq j$.

Complexity Like decision problems, enumeration problems can be classified by their computational complexity. The well-known complexity class $\mathcal{NP}$ contains all decision problems for which solutions can be verified in polynomial time by a deterministic Turing machine. The corresponding complexity class for enumeration problems is EnumP. Enum$\Pi \in$ EnumP if and only if, for all instances x of Π, the time required to verify whether $\Pi(x, y)$ holds for a given y is polynomial in the size of x. The computation model used is a *random-access machine (RAM)*. A RAM uses registers to store and retrieve arbitrary values in constant time. Other basic operations, including arithmetic and outputting a solution, also require constant time.

In their work, Strozecki also introduce further complexity classes that may allow a more precise classification of these problem [Str10]. Those include the classes *DelayP (polynomial delay)*, where a polynomially bounded time between any two consecutive outputs can be achieved, *IncP (incremental polynomial time)*, where the delay between two outputs increases polynomially with each step, and *TotalP (polynomial total time)*, where the time required to output all solutions is polynomially bounded in the number of solutions. Algorithms that satisfy the latter property regarding their runtime are also called *output polynomial*. The inclusions DelayP $\subseteq$ IncP $\subsetneq$ TotalP $\subsetneq$ EnumP are known, assuming $\mathcal{P} \neq co\mathcal{NP} \cap \mathcal{NP}$.

Similarly to the notion of $\mathcal{NP}$-completeness, the hardest problems in EnumP are *EnumP-complete*. To show the EnumP-completeness of a problem, one usually proves the existence of a *parsimonious reduction* of a known EnumP-complete problem, such as the enumeration variant of SAT. These kinds of reductions provide a bijection between the solution sets of two enumeration problems. For EnumP-complete enumeration problems, no output polynomial algorithm can exist if $\mathcal{P} \neq \mathcal{NP}$.

In Chapter 4, we identify instances for the presented algorithms where the number of paths in a subgraph increases with the factorial of the number of vertices. To express this number precisely, we use the upper *incomplete gamma function*. The upper incomplete gamma function Γ is a generalization of the factorial function. For $a > 0$ and $b \geq 0$, it is defined as $\Gamma(a, b) = \int_b^\infty t^{a-1} \mathrm{e}^{-t} \, \mathrm{d}t$. If a is an integer, it can also be expressed as

$\Gamma(a,b) = (a-1)!e^{-b}\sum_{k=0}^{a-1}\frac{b^k}{k!}$. We use the case where a is an integer and $b = 1$, in which $\Gamma(a,1) = \frac{1}{e}\sum_{k=0}^{a-1}\frac{(a-1)!}{k!} = \frac{1}{\mathrm{e}}\cdot\sum_{k=0}^{a-1}\binom{a-1}{k}\cdot k!$ holds [PEB53]. For the purposes of our runtime analysis, the asymptotic growth can be described as $\Gamma(a,1) \in \Theta((a-1)!)$.

3 Alternative Paths Problem

Given a graph $G = (V, E)$ with vertices $s, t \in V$, our aim is to find other viable options for s-t paths in addition to the shortest path, which we call alternative paths. We adapt the definition of *admissible alternative paths* from Abraham, Delling, Goldberg, and Werneck [ADGW13] by simplifying the used criteria and omitting the *limited sharing* criterion entirely. The rationale behind this adaption is detailed below. Alternative paths may not be optimal. However, we require them to still be sensible by disallowing small detours. This intuition is captured by the notion of *local optimality*. A path p is called locally optimal with respect to some parameter T if every subpath $p' = (s', \dots, t')$ of p, where $w(p') \leq T$, is optimal, i.e., $w(p') = d(s', t')$. Moreover, alternative paths should not be substantially longer than the shortest path. Those two criteria are reflected in the following definition.

Definition 3.1 (Alternative Path)**:** *Given a graph $G = (V, E)$ with a source vertex $s \in V$ and a target vertex $t \in V$, a local optimality parameter $T \in [0, d(s, t)]$, and a stretch parameter $S \geq d(s, t)$, a simple s-t path p is called an* alternative path *if*

- *p is locally optimal with respect to T (Local Optimality)*
- *p's weight is bounded by $w(p) \leq S$ (Maximum Stretch)*

The shortest s-t path $\hat{p}$ is itself considered an alternative path, according to this definition.

Our definition of alternative paths differs slightly from the definition according to Abraham, Delling, Goldberg, and Werneck. Regarding local optimality, they introduce rounding for non-continuous graphs. For the sake of simplicity, we do not consider such rounding in this thesis, as it does not fundamentally change the problem. However, the routine for verifying local optimality, which we use in Chapter 4, can easily be adapted if desired. Hence, the presented algorithms are compatible with both definitions of local optimality. Moreover, instead of maximum stretch, they use the criterion of *uniformly bounded stretch*. In the latter case, the maximum stretch criterion is applied not only to the entire path but also to all subpaths. They give an example in which an alternative path that does not meet this criterion is deemed "unnatural". When considering the runtimes of our algorithms in Chapter 4, the maximum stretch parameter S can be chosen to be arbitrarily large. In the case of uniformly bounded stretch, the corresponding parameter could also be made sufficiently large so that the criterion is met. We would therefore make the same observations regarding the runtime. Those further suggest that the difficulty of the problem rather lies in the criterion of local optimality.

In addition, Abraham, Delling, Goldberg, and Werneck use a third criterion for alternative paths, namely limited sharing [ADGW13]. This ensures that alternative paths are sufficiently different from one another. If we intend to give a user of a navigation system a small selection of alternative paths, this is reasonable. However, we focus on an exhaustive search of all alternative paths, for which this restriction has no justification. Furthermore, the order in which alternative paths are enumerated would otherwise be relevant, as whether a path is an alternative path would depend on the previous outputs. This would have a fundamental impact on the difficulty of the problem. For these reasons, we omit this criterion entirely.

Usually, the parameters T and S are not immediate inputs to alternative path algorithms. Instead, the parameters $\alpha \in [0,1]$ and $\varepsilon \geq 0$ are used, from which $T := \alpha \cdot d(s,t)$ and $S := (1+\varepsilon) \cdot d(s,t)$ can then be derived using the shortest s-t path. This allows us to choose fixed parameters, eliminating the need to manually scale them with input size. We thus use this approach for the inputs to the algorithms in Chapter 4.

Having introduced the concept of alternative paths, we now formally define the associated problem.

Definition 3.2 (Alternative Paths Problem)**:** *An instance* $(G, s, t, \alpha, \varepsilon)$ *to the* Alternative Paths Problem *consists of a graph* $G = (V, E)$ *with a source vertex* $s \in V$ *and a target vertex* $t \in V$*, a local optimality parameter* $\alpha \in [0,1]$*, and a stretch parameter* $\varepsilon \geq 0$*. A solution to the problem is an alternative path from* s *to* t*, as defined in Definition 3.1.*

The corresponding *Alternative Paths Enumeration Problem* asks for the set of all alternative paths in a given instance. Our aim is to find an algorithm that solves this problem by outputting all alternative paths, one by one and non-redundantly.

Before we discuss algorithmic solutions to the problem, we briefly address the complexity of the Alternative Paths Enumeration Problem. According to the theorem below, it can be classified in EnumP. To the best of our knowledge, no relationships to other complexity classes have been identified. Potential candidates for a more precise classification of the complexity are mentioned in Chapter 6.

Theorem 3.3: *The Alternative Paths Enumeration Problem belongs to the complexity class EnumP.*

Proof. Let $(G = (V, E), s, t, \alpha, \varepsilon)$ be an instance of the Alternative Paths Enumeration Problem. We show that every solution, i.e., every alternative path, has polynomial size, and it can be decided whether a given path is an alternative path in polynomial time.

By definition, an alternative path is simple. Therefore, its size is bounded by the total number of vertices n.

Let p be a possible candidate for an alternative path in G. It is easy to verify that p is a simple s-t path in polynomial time. We begin at the first vertex of p which must be s. We proceed along p until we reach the last vertex in p which must be t. In doing so, we confirm that each vertex in p is indeed adjacent to the subsequent vertex. Along the way, we mark all vertices as "visited" and abort in case we encounter a previously visited vertex, i.e., we have detected a cycle.

For deciding whether p is locally optimal, we need to calculate distances between vertices. This can be done in polynomial time using a well-known shortest path algorithm, such as Dijkstra's algorithm [Dij59]. There are $\binom{|p|}{2} \in \Theta(|p|^2) \subseteq \mathcal{O}(n^2)$ pairs of vertices along p. For every vertex pair, we sum the edge weights along the subpath p' connecting those two vertices to receive its total weight $w(p')$. This requires time linear in the length of the subpath. If $w(p') \leq T = \alpha \cdot d(s,t)$, we need to check whether $w(p')$ is optimal. We can compare $w(p')$ to the weight of the shortest path between those vertices. In total, checking the local optimality of p requires polynomial time.

To check whether p adheres to the maximum stretch, we sum the weights along p to obtain $w(p)$. We then confirm that $w(p) \leq S = (1+\varepsilon) \cdot d(s,t)$ where $d(s,t)$ has already been calculated in the previous step. This can be done in $\Theta(|p|)$ time.

Altogether, deciding whether p is an alternative path is possible in polynomial time. This implies that the Alternative Paths Enumeration Problem is in EnumP. ∎

4 Algorithmic Approaches

In this chapter, we discuss three approaches to solving the Alternative Paths Enumeration Problem algorithmically. The first approach uses existing algorithms to enumerate shortest paths and then checks them against the criteria for alternative paths. In the second approach, backtracking is used to systematically search the space of all paths. In the third approach, we iteratively perform shortest-path algorithms and concatenate the resulting paths. We demonstrate that all three approaches encounter a similar issue. If there are many paths that must first be explored almost to their end until it can be determined that they are not alternative paths, then enumerating the alternative paths is no longer efficiently possible.

All three approaches involve finding and verifying candidates for alternative paths. The primary task in deciding whether a given simple path is an alternative path is verifying its local optimality. This requires a comparison of the weights of several subpaths to the distances between corresponding vertex pairs. To this end, before enumerating the alternative paths, we perform a preprocessing step. This step involves an APSP algorithm, such as the Floyd-Warshall algorithm [Flo62], to calculate the distances between all pairs of vertices. The calculated values can thereafter be accessed in constant time. Verifying local optimality of a simple path is then possible in time linear in the path's length. We use the following fact.

Lemma 4.1: *A path p is optimal if and only if every subpath of p is optimal.*

Proof. "⇒": We prove the contraposition of this statement. Let $p' := (s', \dots, t')$ be a subpath of a path $p = (s, \dots, s') + p' + (t', \dots, t)$ that is not optimal, i.e., $w(p') > d(s', t')$. Then, there exists a shortest s'-t' path $\hat{p}'$ where $w(\hat{p}') = d(s', t') < w(p')$. However, $\hat{p} := (s, \dots, s') + \hat{p}' + (t', \dots, t)$ is also an s-t path and $w(\hat{p}) = w(p) - (w(p') - w(\hat{p}')) < w(p)$. Therefore, p is not optimal.

"⇐": This implication is trivial since p is a subpath of p. ∎

To determine the local optimality of a simple path, we proceed as follows. Let p be a simple path for which local optimality should be checked with respect to T. At a high level, we iterate over the subpaths of p and assess their optimality where necessary. We start by placing two markers ℓ and r on the first vertex in p, i.e., $\ell = r = 1$. These markings denote the first and last vertex of the subpath $p' := p[\ell \,..\, r]$ currently under consideration. Both markings will be gradually moved forward along p until all necessary subpaths have been considered. For each p', we compare its weight $w(p')$ to the distance between its first and last vertex $d(p[\ell], p[r])$. We use Lemma 4.1 to minimize the number of comparisons performed. It suffices to check only those subpaths $p' = p[\ell..r]$ where $w(p') \leq T$ and $w(p[\ell..(r+1)]) > T$. Therefore, we gradually move the marking r along p until one more movement would raise $w(p')$ above T. We then check whether p' is optimal, i.e., $w(p') = d(p[\ell], p[r])$. If this is not the case, p is not locally optimal, and we abort. Otherwise, we first move the marking r by one vertex. Then, we move the marking ℓ until $w(p') \leq T$ and repeat the process. We terminate when the marking r has reached the last vertex of p and the according subpath has been checked for optimality. Since both markings are only moved by one vertex at a time, we can easily keep track of $w(p')$ on the fly by adding or subtracting the according edge's weight. Furthermore, both markings are moved by at least one vertex during every iteration. Thus, verifying whether p is locally optimal takes $\Theta(|p|)$ time.

All other tasks, namely calculating T and S from the parameters and verifying a path's adherence to the maximum stretch, are computationally straightforward. The former is a simple evaluation of the defining formula for T and S, utilizing the previously computed distance from s to t. The latter can be achieved by summing the weights along the path in linear time and comparing the total weight to the value of S. Altogether, the runtime for verifying whether a given simple path is an alternative path scales linearly with the path's length.

4.1 Shortest Paths

A natural approach to enumerating alternative paths involves enumerating shortest paths and filtering out non-alternative ones. For instance, we can use Yen's algorithm for enumerating shortest simple paths [Yen71]. An advantage of this approach is that alternative paths are enumerated in ascending order of weight. This allows for a simple termination condition when the weight reaches the maximum stretch.

However, this approach is unsuitable if many short non-alternative paths precede an alternative path. All these paths have to be checked before the alternative path can be output. Consider the example shown in Figure 4.1 which scales with ℓ. Here, we choose $\delta > 0$, $T \geq 2+\delta$ and $S \geq \ell(2+\delta)+\delta$. For now, disregard the edge (s,t). As we go from s to t, ℓ decisions are made regarding whether to take the top or bottom edge. Thus, there are exponentially many, specifically 2^ℓ, s-t paths. One of these paths is the shortest path $\hat{p}$ which is obtained by always choosing the top edge and has weight $w(\hat{p}) = \ell \cdot 2$. The remaining $2^\ell - 1$ paths are not alternative paths since choosing one of the bottom edges introduces a local detour of weight $2+\delta \leq T$. This detour exceeds the optimal subpath's weight of 2, thereby violating local optimality. Choosing the bottom edge at each step results in the longest path, with a weight of $\ell(2+\delta)$. The path $p = (s,t)$, with weight $\ell(2+\delta)+\delta$, is an alternative path, but it would be the last to be output by a shortest paths algorithm.

In Figure 4.1, there are many paths that are not locally optimal due to the same subpath. For instance, there are 2^{l-1} paths that include the bottom edge at s, none of which are locally optimal. However, all of them are output by a k shortest paths algorithm and therefore have to be verified. In order to avoid such cases, we interlink the enumeration of the shortest paths with their verification.

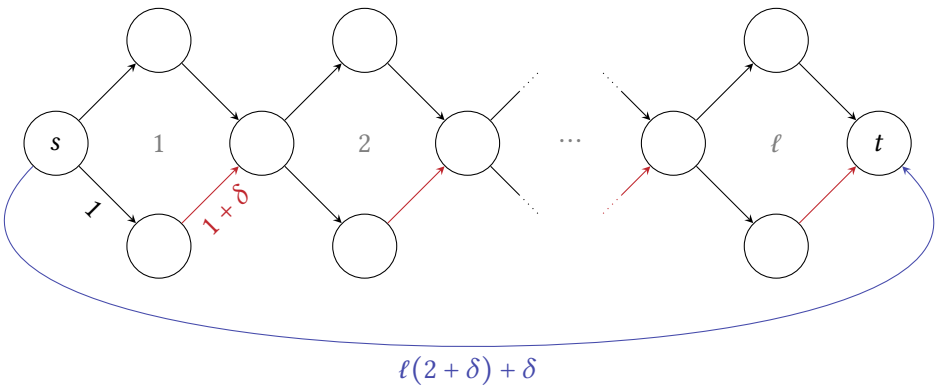


Figure 4.1: A problem instance with $T \geq 2+\delta$ and $S \geq \ell(2+\delta)+\delta$ which contains an exponential number of non-alternative paths. The alternative path (s,t) is longer than those paths and will therefore be found last by a shortest paths algorithm.

In the following, we modify Dijkstra's algorithm to achieve this [Dij59]. Similar adaptations of other k shortest path algorithms are also conceivable. See Algorithm 4.1 for a reference in pseudocode. We utilize a priority queue that initially contains the path (s). Unlike Dijkstra's original algorithm, which stores only individual vertices, we store entire paths. This is necessary in order to enumerate shortest paths as opposed to just finding the shortest one. At any point, the queue contains only simple paths that are locally optimal and adhere to the maximum stretch. In each iteration, we remove the shortest path $p_{\min}$ from the queue. If the last vertex of $p_{\min}$ is t, we know that $p_{\min}$ fulfills the criteria of an alternative path, so we output $p_{\min}$. Otherwise, for all vertices v' that are adjacent to the last vertex v in $p_{\min}$, we consider the path $p' := p_{\min} + (v, v')$ obtained by appending v' to $p_{\min}$. Because we are only interested in simple paths, we disregard any neighbors that are already part of $p_{\min}$. If this path is locally optimal and adheres to the maximum stretch, we add it to the queue. For verifying the local optimality of p' we can utilize the fact that $p_{\min}$ is already locally optimal. If p' is not locally optimal the violating subpath must contain the newly added edge (v, v'). Therefore, it suffices to only check the subpath $p'[\ell \mathinner{..} |p'| + 1]$ where $w(p'[\ell \mathinner{..} |p'| + 1]) \leq T$ and $w(p'[(\ell - 1) \mathinner{..} |p'| + 1]) > T$. This subpath can be found similarly to the local optimality check described in the beginning of Chapter 4 by initially placing both markings on the last vertex of p' and moving the marking ℓ backward. The algorithm terminates when the queue is empty.

Algorithm 4.1: A Dijkstra-based algorithm for enumerating alternative paths in ascending order of weight.

Input: A graph $G = (V, E)$ with weights w.
A source vertex $s \in V$ and a target vertex $t \in V$.
The local optimality parameter $\alpha \in [0, 1]$.
The maximum stretch parameter $\varepsilon \geq 0$.
Output: All alternative paths from s to t in ascending order of weight.

```
d ← compute distances using APSP algorithm
T ← α · d(s, t)
S ← (1 + ε) · d(s, t)
Q ← initialize empty min priority queue
insert path (s) with priority w((s)) = 0 into Q
while Q is not empty do
    p_min ← remove shortest path from Q
    v ← last vertex in p_min
    if v = t then output p_min
    else
        for v' ← adjacent vertices of v that are not in p_min do
            p' ← p_min + (v, v')
            // the value of w(p_min) is known from the priority queue
            w(p') ← w(p_min) + w(v, v')
            if p' is locally optimal wrt. T and w(p') ≤ S then
                insert p' with priority w(p') into Q
```

Theorem 4.2: *Algorithm 4.1 is correct.*

Proof. For Algorithm 4.1 to be correct, it must output all and only alternative paths in the input instance.

First, we see that the queue Q contains only paths that are locally optimal and adhere to the maximum stretch. This is true for both the initial path (s) in Line 5 and every other path that is inserted into Q in Lines 14 to 15. Every path in the queue Q is further simple. This is again true for the initial path (s). Since p' is formed by appending a vertex to a simple path $p_{\min}$ that is not already part of $p_{\min}$ in Lines 11 to 12, p' is also simple. This implies that every path output in Line 9 is an alternative path.

Next, let p be a simple path starting at s that is both locally optimal and adheres to the maximum stretch. We show inductively that p will eventually be inserted into Q if p is a subpath of at least one alternative path. In the base case $|p| = 1$, meaning $p = (s, v')$ for some $v' \in V \setminus \{s\}$, p is inserted into Q during the first iteration. This is true since $p_{\min} = (s)$ in Line 7 and s is adjacent to v'. Therefore, $p' = p$ in Line 12 at some point. For the inductive step, let $|p| > 1$ and let Q contain $p[1 \,..\, |p|]$. Consider the iteration where $p_{\min} = p[1 \,..\, |p|]$ in Line 7. If $v = t$ in Line 9, then p cannot be extended to an alternative path since it already contains t. Otherwise, $p' = p$ in Line 12 since $p[|p|]$ is adjacent to $p[|p| + 1]$. Therefore, p' is inserted into Q in Lines 14 to 15. We conclude that every alternative path will eventually be inserted into Q. Since $p_{\min}$ is unambiguous for every p', no path is inserted into Q more than once. This implies that the algorithm terminates, because there is only a finite number of simple paths which Q can contain. ∎

Algorithm 4.1 handles the instance in Figure 4.1 well. Path prefixes that are not locally optimal are disregarded in Line 14. Therefore, the algorithm does not consider every single path with the same violating prefix. However, a similar problem arises when there are many paths where local optimality is violated near their ends. Given $n \in \mathbb{N}$, we construct such an instance with $G = (V, E)$ as follows. Refer to Figure 4.2 for an example where $n = 6$.

$$
\begin{aligned}
V :=& \{v_1, \ldots, v_n\} \\
E :=& \{v_1, \ldots, v_{n-3}\} \times \{v_1, \ldots, v_{n-2}\} \\
&\cup \{(v_1, v_{n-1}), (v_{n-2}, v_n), (v_{n-1}, v_n)\}
\end{aligned}
$$

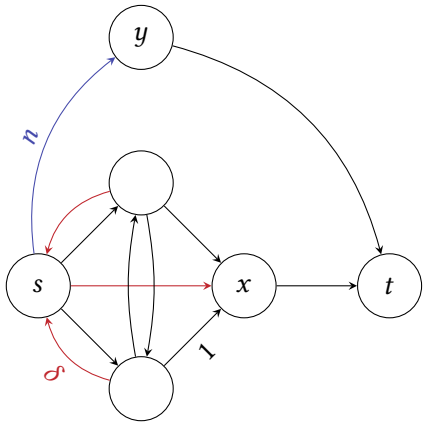


Figure 4.2: An instance for Algorithm 4.1 where $n = 6$. We choose $0 < \delta < \frac{1}{2}$, $T = 1$, and $S \geq 7$. The shortest path is (s, x, t). The only other alternative path is (s, y, t). There are $e\Gamma(n-3, 1) - 1 = 4$ paths from s to x that violate local optimality in their last edge.

We specify $s := v_1$ and $t := v_n$. Additionally, we denote v_{n-2} by x and v_{n-1} by y. The edge weights are defined by the following weight function:

$$\forall (v,u) \in E: \quad w(v,u) := \begin{cases} \delta & (v,u) = (s,x) \text{ or } u = s \\ n & (v,u) = (s,y) \\ 1 & \text{otherwise} \end{cases}$$

In this definition, δ is a small value, such that $0 < \delta < \frac{1}{2}$. The shortest s-t path is given by $\hat{p} = (s,x,t)$ with $w(\hat{p}) = 1 + \delta$. We choose $\alpha := \frac{1}{1+\delta}$ and $\varepsilon \geq \frac{n-\delta}{1+\delta}$. Then, $T = \alpha \cdot w(\hat{p}) = 1$ and $S = (1+\varepsilon) \cdot w(\hat{p}) \geq n+1$. Observe that there are super-exponentially many, specifically $\sum_{k=0}^{n-4} \binom{n-4}{k} \cdot k! = e\Gamma(n-3,1) \in \Theta((n-4)!)$, simple s-x paths. Here, k is the number of vertices between s and x. Each such path has a suffix (v,x) consisting solely of the last edge. Apart from the case where $v = s$, this suffix violates local optimality due to the shorter v-x path (v,s,x), where $w((v,s,x)) = 2\delta < 1 = w(v,x)$. Note that we cannot actually use this shorter subpath in an alternative path, as it would introduce a cycle. The algorithm considers all $e\Gamma(n-3,1) - 1$ of these paths and discards them only after x is appended. Besides $\hat{p}$, the only alternative path is $p := (s,y,t)$. This is also the longest path and will therefore be output last. The algorithm thus requires $\Omega((n-4)!)$ time in total before outputting p.

4.2 Backtracking

Instead of enumerating shortest paths, we can also perform a tree search to explore all s-t paths. However, since there are usually too many s-t paths for an exhaustive search to be feasible, we prune non-alternative paths at the earliest opportunity.

In this section, we present an algorithm that conducts a depth-first tree search on the input graph. Paths are pruned as soon as they violate either of the local optimality or maximum stretch criteria. See Algorithm 4.2 for a reference in pseudocode. The currently considered path is denoted by p. The search starts in s and thus p is initially (s). We constantly keep track of the weight of p. This enables verification of the maximum stretch criterion without any additional runtime overhead. Additionally, we mark all vertices on p as visited in order to prevent cycles. For the local optimality checks, we apply the same routine as in Section 4.1. Because we prune early, the prefix of p up to the second last vertex is always locally optimal. Thus, verifying local optimality near the last edge of p is sufficient. Local optimality and adherence to the maximum stretch are verified each time the tree search is about to proceed to the next vertex.

Similarly to the algorithm described in Section 4.1, non-alternative paths are discarded late if they violate the criteria close to their ends. Consider the instance from Section 4.1 which is shown in Figure 4.2. In Section 4.1, y is traversed last among all vertices adjacent to s. This is guaranteed by the large edge weight $w(s,y)$. Here, the order in which adjacent vertices are selected by the tree search is arbitrary and not influenced by edge weights. We therefore let $w(s,y) = 1$ and explicitly require y to be traversed last. See Figure 4.3 for an example where $n = 6$. The tree search traverses all $e\Gamma(n-3,1)$ paths from s to x before discovering the alternative path (s,y,t). It backtracks only after the local optimality violation at x is found. Consequently, the algorithm requires $\Omega((n-4)!)$ time before outputting (s,y,t), too.

Algorithm 4.2: A backtracking algorithm for enumerating all alternative paths.

Input: A graph $G = (V, E)$ with weights w.
A source vertex $s \in V$ and a target vertex $t \in V$.
The local optimality parameter $\alpha \in [0, 1]$.
The maximum stretch parameter $\varepsilon \geq 0$.

Output: All alternative paths from s to t.

```
1  d ← compute distances using APSP algorithm
2  T ← α · d(s, t)
3  S ← (1 + ε) · d(s, t)
4  p ← (s)
5  w(p) ← 0
6  dfs_recursive(p, w(p))
7  Procedure dfs_recursive(p, w(p))
8      v ← last vertex in p
9      mark v as visited
10     if p is locally optimal wrt. T and w(p) ≤ S then
11         if v = t then output p
12         else
13             for v' ← adjacent vertices of v that are not visited do
14                 p' ← p + (v, v')
15                 w(p') ← w(p) + w(v, v')
16                 dfs_recursive(p', w(p'))
17     unmark v as visited
```

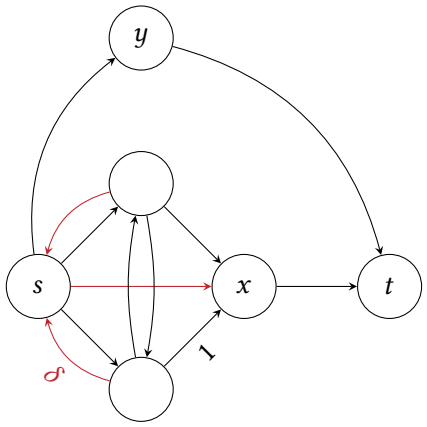


Figure 4.3: An instance for Algorithm 4.2 where $n = 6$, adapted from Figure 4.2. The algorithm traverses all $e\Gamma(n-3, 1) = 5$ paths from s to x, of which only (s, x) is a prefix of an alternative path $\hat{p} = (s, x, t)$. The alternative path (s, y, t) is found last if y is traversed after all other vertices adjacent to s.

4.3 Iterative Shortest-Path Trees

In both previous approaches, we generate a potentially large number of candidate paths and then verify which ones qualify as alternative paths. We apply pruning to reduce the number of candidates at an early stage. In the following, we approach the problem from a different angle. We try to generate the candidates in such a way that they are already locally optimal. To do this, we combine shortest paths which, in themselves, are trivially locally optimal. A possible violation of local optimality must then be checked around the vertices where two shortest paths are concatenated. For this purpose, we consider subgraphs with a radius T around a center vertex, i.e., all vertices in the subgraph have a maximum distance of T from this center vertex.

In this section, we iteratively compute shortest-path trees. We use an SSSP algorithm, such as Dijkstra's algorithm [Dij59], for this purpose. In the following, we assume that all shortest paths are unique. This guarantees that the shortest paths beginning at a source vertex indeed form a tree. Otherwise, we would generally only receive a DAG of shortest paths. The algorithm also works in this case. We merely have to consider several shortest paths to the same vertex independently of each other.

We begin the first iteration at the source vertex s and form a shortest-path tree with radius T, i.e., we only include vertices v for which $d(s,v) \leq T$. Every vertex v now represents a shortest path from s to v. In the second step, we consider all edges (v,u) that raise the according s-u path's weight above T, i.e., $d(s,v) \leq T$, but $d(s,v) + w(v,u) > T$. Here, we ignore vertices u that are already contained in the s-v path in order to prevent cycles.

The latter step is necessary for two reasons so that no alternative paths are missed. Firstly, individual edges can have weights $> T$. If we were to limit ourselves to paths of weight $\leq T$ in each step, those edges would never be considered. Therefore, we explicitly include such edges. Secondly, the resulting paths can subsequently be used as the basis for the next iteration of the algorithm. This is because, on the one hand, we ensure that we do not miss any relevant paths. Obviously, this step considers paths that exceed the radius T, i.e., $d(s,u) > T$. Yet paths that remain within the radius T, i.e., $d(s,u) \leq T$, are also taken into account. Those are either optimal and therefore already covered by the shortest-path tree. Or they are not optimal, but must then be longer than T in order not to violate local optimality. In this case, they are included in the second step. On the other hand, we ensure that no paths are considered more than once. If, in the next iteration, we were to start generating a shortest-path tree at a path that is shorter than T, we would likely recreate paths that are already included in the shortest-path tree of the current iteration.

Note that there may be multiple of those edges leading to the same vertex u. In this case, we handle the according s-u paths independently of each other. All resulting s-u paths are checked for local optimality and adherence to the maximum stretch. Paths that violate one of these criteria are discarded. We repeat the whole process of computing a shortest-path tree at u followed by the second step for every remaining s-u path. To prevent cycles, those trees must not include vertices that are already part of the corresponding s-u path. The resulting paths in each iteration are concatenated. Paths are continued until they reach t. This can happen both while forming the shortest-path trees or when appending the additional edges in the second step. If those paths meet the criteria for alternative paths, they are output. The algorithm terminates once there are no more s-u paths remaining to form a shortest-path tree.

To perform the second step, we need to find edges that raise the according path's weight above T and verify whether it is locally optimal and adheres to the maximum stretch. This can be done on the fly while forming the shortest-path trees. Edges that exceed weight

T are considered anyway when searching for the shortest paths. They can then be stored immediately for further processing in the next step. If the shortest paths are found edge by edge, we can easily keep track of the paths' total weights and abort if S is exceeded. We can further apply the routine from Section 4.1 to verify local optimality.

Algorithm 4.3: An algorithm for enumerating all alternative paths using shortest-path trees (or DAGs).

Input: A graph $G = (V, E)$ with weights w.
A source vertex $s \in V$ and a target vertex $t \in V$.
The local optimality parameter $\alpha \in [0, 1]$.
The maximum stretch parameter $\varepsilon \geq 0$.
Output: All alternative paths from s to t.

```
1  d ← compute distances using APSP algorithm
2  T ← α · d(s, t)
3  S ← (1 + ε) · d(s, t)
4  Q ← initialize empty priority queue
5  insert (s) with priority w((s)) = 0 into Q
6  while Q is not empty do
7      p ← remove shortest path from Q
       // We can skip paths that go beyond t.
8      for q ← enumerate SSSP from the last vertex in p in ascending order do
           // The following can be done on the fly during SSSP enumeration.
9          if w(q) > T then break
10         if p + q is simple, locally optimal wrt. T and w(p + q) ≤ S then
11             v ← last vertex in q
12             if v = t then output p + q
13             else
14                 insert p + q with priority w(p′) into Q
15                 for u ← adjacent vertices of v that are not part of p + q do
16                     q′ ← q + (v, u)
17                     p′ ← p + q′
18                     if w(q′) > T, p′ is locally optimal wrt. T and w(p′) ≤ S then
19                         v′ ← last vertex in p′
20                         if v′ = t then output p′
21                         else
22                             insert p′ with priority w(p′) into Q
```

As with the two previous algorithms, problems arise when there are many paths with a late violation of local optimality. We take the instance from Section 4.1, shown in Figure 4.2, and tailor it to this algorithm. Refer to the example in Figure 4.4 where $n = 7$. Given $n \in \mathbb{N}$, the graph $G = (V, E)$ is defined as follows.

$$
\begin{aligned}
V :=& \{v_1, \ldots, v_n\} \\
E :=& \{v_1, \ldots, v_{n-3}\} \times \{v_1, \ldots, v_{n-3}\} \\
&\cup \{(v_1, v_{n-2}), (v_1, v_{n-1}), (v_{n-3}, v_{n-2}), (v_{n-2}, v_{n-1}), (v_{n-1}, v_n)\}
\end{aligned}
$$

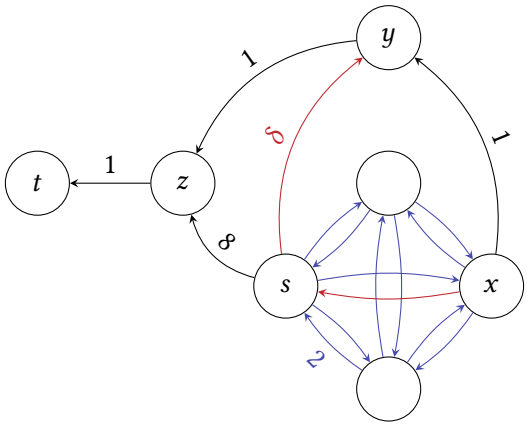


Figure 4.4: An instance for Algorithm 4.3 where $n = 7$. We choose $0 < \delta < \frac{1}{2}$ and $T = 1$. There are $e\Gamma(3,1) = 5$ simple paths from s to x, none of which is a prefix of an alternative path. The algorithm extends paths in the clique subgraph by one vertex per iteration. The alternative path (s,z,t) is found only after all of those s-x paths have been considered.

We specify $s := v_1$ and $t := v_n$, and we denote v_{n-3} by x, v_{n-2} by y, and v_{n-1} by z. The edge weights are defined by the following weight function:

$$\forall (v,u) \in E: \quad w(v,u) := \begin{cases} \delta & (v,u) = (s,y) \text{ or } (v,u) = (x,s) \\ 2 & (v,u) \in \{v_1,\dots,v_{n-3}\} \times \{v_1,\dots,v_{n-3}\} \setminus \{(x,s)\} \\ 2(n-3) & (v,u) = (s,z) \\ 1 & \text{otherwise} \end{cases}$$

Again, δ is a small value, such that $0 < \delta < \frac{1}{2}$. The shortest path is $\hat{p} = (s,y,z,t)$ with $w(\hat{p}) = 2 + \delta$. We choose $\alpha := \frac{1}{2+\delta}$ and $\varepsilon \geq \frac{2n-7-\delta}{2+\delta}$ so that $T = \alpha \cdot w(\hat{p}) = 1$ and $S = (1+\varepsilon) \cdot w(\hat{p}) \geq 2(n-3)+1$. Apart from (x,s), the edge weights in the subgraph consisting of the vertices $s,\dots,x$ are equal to $2 > T$. As a consequence, each shortest-path tree formed by the algorithm within this subgraph consist only of the root itself since other vertices lie outside the T radius. Therefore, the paths progress by only one vertex in each iteration. Because there are $\sum_{k=0}^{n-5} \binom{n-5}{k} \cdot k! = e\Gamma(n-4,1) \in \Theta((n-5)!)$ simple s-x paths, the algorithm performs $\Theta((n-5)!)$ iterations on that subgraph. As before, none of these paths can be extended to a locally optimal path since $w(x,y) = 1 > 2\delta = w((x,s,y))$. The only alternative path besides $\hat{p}$ is $p := (s,z,t)$. The algorithm inserts the path (s,z) into the queue during the first iteration. However, all s-x paths are shorter than (s,z), so it is only removed again after all s-x paths have been considered. Therefore, p is being output after $\Omega((n-5)!)$ iterations.

5 Problem Variations

In this chapter, we explore some restrictions and variations of the Alternative Paths Enumeration Problem. In Section 5.1, we examine so-called via paths, which are a subset of alternative paths. Afterward, in Section 5.2 we focus on unweighted input graphs. The motivation is to develop more efficient approaches than those discussed in Chapter 4 when the output or input is restricted to a specific case. Lastly, we generalize the problem by briefly addressing cyclic alternative paths in Section 5.3.

5.1 Via Paths

Abraham, Delling, Goldberg, and Werneck argue that, due to the possibly large number of alternative paths, finding the best one may be impractical [ADGW13]. They therefore study so-called *single-via paths*, which are formed by concatenating two shortest paths. We define *k-via paths* as the natural generalization of such single-via paths in order to extend the solution set. Single-via paths are the special case for $k = 1$.

Definition 5.1 (k-via Path)**:** *Let $k \in \mathbb{N}$. A path p is a k-via path if it is a concatenation of $k+1$ shortest paths. Specifically, $p = (s, \ldots, v^{(1)}, \ldots, v^{(2)}, \ldots, v^{(k)}, \ldots, t)$ where $(v^{(i)}, \ldots, v^{(i+1)})$ is the shortest path from $v^{(i)}$ to $v^{(i+1)}$ for $1 \leq i < k$. A k-via path is induced by a sequence of* via vertices $v^{(1)}, \ldots, v^{(k)}$.

Note that the sequence of via vertices for a given k-via path is not necessarily unique. Additionally, the same path p can be expressed with varying numbers of via vertices. In particular, for a k-via path p, we can insert any vertex along p into the sequence of via vertices, thus obtaining p as a $(k+1)$-via path. We refer to p as *minimal* with respect to its sequence of via vertices if no via vertex can be omitted while still inducing the same path p. When the actual value of k is arbitrary at some point, we use the general term *via path*.

In this section, we focus on enumerating all via paths in a given instance of the Alternative Paths Enumeration Problem. Note however that not every alternative path is a via path. Consequently, this approach may identify only a strict subset of the solutions for certain instances. Refer to Figure 5.1 for an example of such an instance.

In order to enumerate all via alternative paths in a graph, a naive algorithm would iterate over all sequences of k via vertices where k ranges from 1 to $n-2$. For every sequence of via vertices it would have to determine whether the induced via path fulfills the criteria of being an alternative path. For a given k, there are $\frac{n!}{(n-k)!}$ possible sequences of k via vertices. However, we can establish an upper bound on the number of via vertices k which is necessary to cover all via alternative paths.

Theorem 5.2: *Given an instance $(G = (V, E), s, t, \alpha, \varepsilon)$ for the Alternative Paths Enumeration Problem, for every minimal k-via alternative path, the inequality $k \leq \left\lceil 2 \cdot \frac{1+\varepsilon}{\alpha} - 1 \right\rceil$ must hold.*

Proof. Let p be a minimal k-via alternative path with via vertices $v^{(1)}, \ldots, v^{(k)}$ and let $v^{(j)}$, $v^{(j+1)}$, $v^{(j+2)}$ for $1 \leq j \leq k-2$ be any three consecutive via vertices in p. Let further $p' := (v^{(j)}, \ldots, v^{(j+2)})$ denote the subpath of p from $v^{(j)}$ to $v^{(j+2)}$. If $w(p') \leq T$, then p'

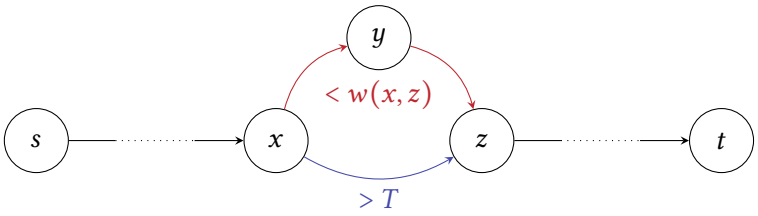


Figure 5.1: An example for an alternative path $(s, \ldots, x, z, \ldots, t)$ which is not a via path for any sequence of via vertices. Let $w(x, z) > T$ and let the edges $(x, y), (y, z)$ be short enough, such that $w((x, y, z)) < w(x, z)$. The alternative path $(s, \ldots, x, z, \ldots, t)$ is not a via path, because no via path can contain the edge (x, z) which is not a part of any shortest path between two vertices.

is a shortest path by the definition of alternative paths. Therefore, we can omit $v^{(j+1)}$ from the sequence of via vertices since the shortest path from $v^{(j)}$ to $v^{(j+2)}$ runs through $v^{(j+1)}$ anyway, assuming unique shortest paths. This contradicts the fact that p is minimal. It follows that $w((v^{(i)}, \ldots, v^{(i+2)})) > T$ for every $1 \leq i \leq k-2$.

Thus, the following relationship holds for the total weight of p. If k is odd, then

$$w(p) = w\left((s, \ldots, v^{(2)})\right) + \sum_{i=1}^{\frac{k-3}{2}} w\left((v^{(2i)}, \ldots, v^{(2i+2)})\right) + w\left((v^{(k-1)}, \ldots, t)\right) > \left\lceil \frac{k}{2} \right\rceil \cdot T.$$

If k is even, then

$$w(p) = w\left((s, \ldots, v^{(2)})\right) + \sum_{i=1}^{\frac{k-2}{2}} w\left((v^{(2i)}, \ldots, v^{(2i+2)})\right) + w\left((v^{(k)}, \ldots, t)\right) > \frac{k}{2} \cdot T = \left\lceil \frac{k}{2} \right\rceil \cdot T.$$

Note that in the latter case, $w((v^{(k)}, \ldots, t))$ can be arbitrarily small. $w(p)$ is further upper bounded by S since p is an alternative path. By combining these two bounds for $w(p)$, $\left\lceil \frac{k}{2} \right\rceil \cdot T < S$, or equivalently $\left\lceil \frac{k}{2} \right\rceil < \frac{S}{T} = \frac{1+\varepsilon}{\alpha}$, holds. It follows that $k < 2 \cdot \frac{1+\varepsilon}{\alpha}$, or $k \leq \left\lceil 2 \cdot \frac{1+\varepsilon}{\alpha} - 1 \right\rceil$. ∎

This relationship holds regardless of the size of the input graph. In many applications, the parameters α and ε often take on typical values, such as $\varepsilon = 0.25$ and $\alpha = 0.25$. Observe that if we consider these parameters as constants, then the number k of necessary via vertices to cover all via paths is upper bounded by some constant K, according to Theorem 5.2. For the above parameter choices, $k \leq 10 = K$. The naive algorithm described above would therefore only have to iterate over all sequences of k via vertices where k ranges from 1 to K. There exist $\sum_{k=1}^{K} \frac{n!}{(n-k)!} \leq \sum_{k=1}^{K} n^k \in \mathcal{O}\left(n^K\right)$ such sequences of via vertices. Given that verifying whether a given path is an alternative path can be done in polynomial time, this algorithm would output all via alternative paths in polynomial time.

However, in the general case, k cannot be bounded by a constant. In particular, this implies that k-via alternative paths comprise strictly more solutions than single-via alternative paths.

Lemma 5.3: *For every $k \in \mathbb{N}$, there exists an instance of the Alternative Paths Enumeration Problem with a minimal k-via alternative path.*

Proof. Given $k \in \mathbb{N}$, we define a graph $G = (V, E)$ as follows:

$$V := \{v_0, v_1, \ldots, v_k, v_{k+1}\}$$
$$E := \{(v_i, v_{i+1}) \mid 0 \leq i < k+1\} \cup \{(v_i, v_{i+2}) \mid 0 \leq i < k\}$$

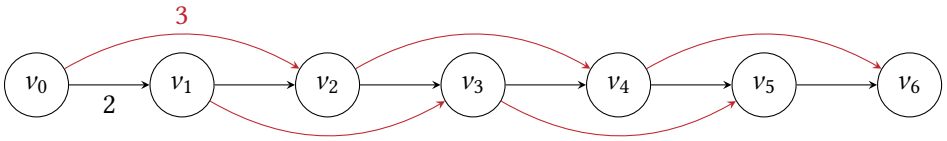


Figure 5.2: An example for an instance containing a minimal 5-via alternative path, which is induced by the via vertices v_1, v_2, v_3, v_4, v_5.

The edge weights are defined by the following weight function:

$$\forall (v_i, v_j) \in E: \quad w(v_i, v_j) := \begin{cases} 2 & j = i+1 \\ 3 & j = i+2 \end{cases}$$

We specify $s := v_0$ and $t := v_{k+1}$. See Figure 5.2 for an example where $k = 5$.

For two vertices v_i and v_{i+2}, the shortest path between those vertices is (v_i, v_{i+2}) with weight 3. There is also a second path (v_i, v_{i+1}, v_{i+2}) of weight 4. When k is odd, the shortest s-t path is $\hat{p} = (v_0, v_2, v_4, \ldots, v_{k+1})$ with $w(\hat{p}) = \frac{k+1}{2}$, or when k is even, $\hat{p} = (v_0, v_2, v_4, \ldots, v_{k-2}, v_k, v_{k+1})$ with $w(\hat{p}) = \frac{k}{2}+2$. Also, $p = (v_0, v_1, \ldots, v_{k+1})$ with $w(p) = 2(k+1)$, is a k-via path since each edge (v_i, v_{i+1}) is the only and therefore the shortest path between v_i and v_{i+1}. p is further minimal. We cannot omit any via vertex v_i since the shortest path between v_{i-1} and v_{i+1} does not run through v_i. It remains to show that p is an alternative path. If k is odd, we choose $\alpha < \frac{8}{k+1}$ and $\varepsilon \geq 3$. Then, $T = \alpha \cdot w(\hat{p}) < \frac{8}{k+1} \cdot \frac{k+1}{2} = 4$ and $S = (1+\varepsilon) \cdot w(\hat{p}) \geq 4 \cdot \frac{k+1}{2} = 2(k+1)$. If k is even, we choose $\alpha < \frac{8}{k+4}$ and $\varepsilon \geq \frac{3k}{k+4}$. Similarly, $T = \alpha \cdot w(\hat{p}) < \frac{8}{k+4} \cdot (\frac{k}{2}+2) = 4$ and $S = (1+\varepsilon) \cdot w(\hat{p}) \geq (1+\frac{3k}{k+4}) \cdot (\frac{k}{2}+2) = \frac{k}{2}+2+\frac{3k}{2} = 2(k+1)$. In both cases, p is locally optimal with respect to T, and satisifies $w(p) \leq S$. ∎

Many such instances that contain a minimal k-via alternative path where k is relatively large, also contain a k'-via alternative path where $k' < k$. In Figure 5.2 for instance, there is a single-via alternative path for the via vertex v_1. In this case, the naive algorithm described above would be able to find and output a via alternative path early on. The question arises as to whether there are also instances where a k-via alternative path only exists for large k.

Theorem 5.4: *For every $k \in \mathbb{N}$, there exists an instance of the Alternative Paths Enumeration Problem which contains a minimal k-via alternative path, but no k'-via alternative path where $k' < k$.*

Proof. Given the number of via vertices $k \in \mathbb{N}$, we construct a graph $G = (V, E)$ as follows:

$$V := \{s, t\} \cup \{v_i \mid 0 \leq i \leq k+1\}$$
$$E := \{(v_0, v_{k+1})\} \cup \{(v_i, v_j) \mid 0 \leq i < j \leq k+1\}$$

The edge weights are defined by the following weight function:

$$w(s, v_0) := 2$$
$$w(v_{k+1}, t) := 2$$
$$\forall (v_i, v_j) \in E: \quad w(v_i, v_j) := \begin{cases} 2 & j - i = 1 \\ 1 & j - i > 1 \end{cases}$$

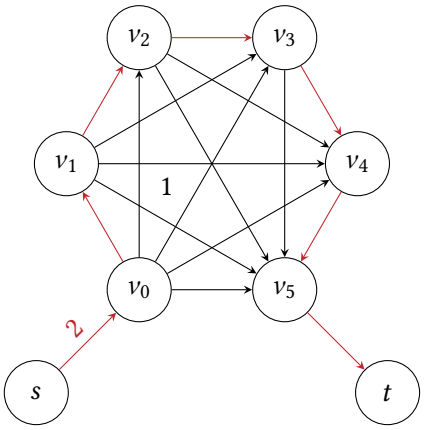


Figure 5.3: An example instance for Theorem 5.4 where $k = 4$. $\hat{p} = (s, v_0, v_5, t)$ is the shortest path. $p = (s, v_0, v_1, v_2, v_3, v_4, v_5, t)$ is a 4-via alternative path and there are no other k'-via alternative paths where $k' < 4$.

See Figure 5.3 for an example with $k = 4$ via vertices.

For the problem instance, we specify the source vertex $s := v_0$, the target vertex $t := v_k$ and the parameters $\alpha := \frac{3}{5}$ and $\varepsilon := \frac{(k+3)\cdot 2}{5} - 1$. We see that $\hat{p} = (s, v_0, v_{k+1}, t)$ is the shortest path and has weight $w(\hat{p}) = 2+1+2 = 5$. Therefore, $T = \alpha \cdot 5 = 3$ and $S = (1+\varepsilon)\cdot 5 = (k+3)\cdot 2$. Choosing $v_1, \ldots, v_k$ as via vertices, we receive the simple k-via path $p = (s, v_0, v_1, \ldots, v_k, v_{k+1}, t)$. We show that p is minimal and the only via alternative path. Hereafter, we refer to an edge $e \in E$ as an *outer* edge if $w(e) = 2$, and as an *inner* edge if $w(e) = 1$.

For two vertices v_i and v_j, where $j - i > 1$, the shortest path from v_i to v_j is always (v_i, v_j) with $w((v_i, v_j)) = 1$. In particular, it does not pass through any v_ℓ for $i < \ell < j$. Thus, omitting v_ℓ from the sequence of via vertices would also remove it from p. Therefore, v_ℓ cannot be omitted for $0 < \ell < k + 1$, implying that p is minimal.

p is also an alternative path. Consider a subpath p' of p. Since all edges along p have the same weight, $w(p') = 2 \cdot |p'|$. We need to confirm the optimality of p' if $w(p') \leq T = 3$. This holds only for $|p'| = 1$, i.e., if p' contains only a single (outer) edge. Such p' is trivially optimal since there are no other paths from $p'[1]$ to $p'[2]$. It follows that p is locally optimal. p further adheres to the maximum stretch since $w(p) = |p| \cdot 2 = (k + 3) \cdot 2 \leq S$. Therefore, p is an alternative path.

There is no other alternative path besides $\hat{p}$ and p. Consider an s-t path $p' \neq \hat{p}$ that contains a subpath $p'' := (v_{i_1}, v_{i_2}, v_{i_3})$ with two edges $e_1 := (v_{i_1}, v_{i_2})$ and $e_2 := (v_{i_2}, v_{i_3})$. Note that, by construction of G, $i_1 < i_2 < i_3$ holds. If both e_1 and e_2 are inner edges, then $w(p'') = 1 + 1 = 2$. If exactly one of the two edges is an inner edge, then $w(p'') = 2 + 1 = 3$. In both cases, $w(p'') \leq 3 = T$. However, there is a path (v_{i_1}, v_{i_3}) with $w((v_{i_1}, v_{i_3})) = 1 < w(p'')$. Therefore, p' is not locally optimal. Thus, for p' to be an alternative path, all edges along p' must be outer edges. Since there is exactly one s-t path p consisting of only outer edges, this completes the proof. ∎

5.2 Unweighted Graphs

In this section, we consider *unweighted graphs*. A graph $G = (V, E)$ is called unweighted if $w(u, v) = 1$ for all edges $(u, v) \in E$. Therefore, the weight of a path is equal to its length. In particular, the length of subpaths, which must be locally optimal, is bounded for a given T.

Let the local optimality parameter $T \geq 2$ be a constant. Without loss of generality, we can assume that $T \in \mathbb{N}$. Otherwise, T could simply be rounded down, as each path has an integer weight. We apply the following transformation on the unweighted input graph which enables us to easily identify alternative paths. Given $G = (V, E)$, we construct a new graph $G' = (V', E')$. The set of vertices V' contains all paths in G that have length T and are shortest paths. When determining the local optimality of an s-t path in G, the subpaths that we have to consider are exactly those of length T, according to Lemma 4.1. Consequently, an s-t path in G is locally optimal if and only if all its subpaths of length T lie in V'. We therefore connect these subpaths in such a way that the paths extracted from the transformed graph G' can only contain subpaths of length T that are in V'. To this end, a path $p \in V'$ is adjacent to another path $q \in V'$ if the last T vertices of p are equal to the first T vertices of q. We further add two auxiliary vertices s and t, along with edges connecting them to paths that begin at s or end at t, respectively. For an example of this transformation, see Figure 5.4.

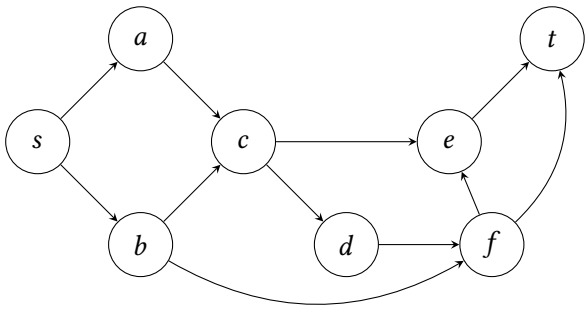
$$
\begin{aligned}
V' :=& \{p \mid p := (v_1, \ldots, v_{T+1}) \text{ is a path in } G \text{ and } d(v_1, v_{T+1}) = T\} \\
&\cup \{s, t\} \\
E' :=& \{(p, q) \mid p, q \in V' \setminus \{s, t\}, p[2 \,..\, T+1] = q[1 \,..\, T]\} \\
&\cup \{(s, p) \mid p \in V' \setminus \{s, t\}, p[1] = s\} \\
&\cup \{(p, t) \mid p \in V' \setminus \{s, t\}, p[T+1] = t\}
\end{aligned}
$$

Paths in the transformed graph can be mapped back to paths in the original graph. This is done by concatenating the paths represented by the vertices in the path of the transformed graph. The matching subpath of two subsequent paths must only occur once in the resulting path. Formally, a path $p' = (p_1, \ldots, p_k)$ in the transformed graph corresponds to the path $p = (p_1[1], p_2[1], \ldots, p_{k-1}[1], p_k[1], p_k[2], \ldots, p_k[T+1])$ in the original graph. The result of the transformation now allows us to extract alternative paths in the original graph. To this end, we first establish the following two statements.

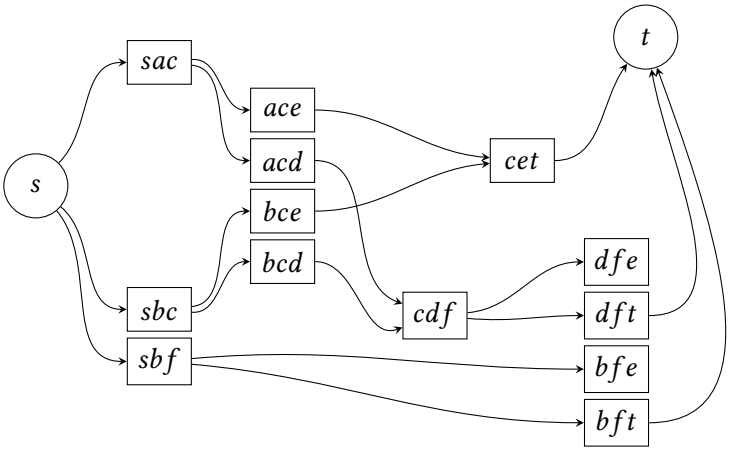
Lemma 5.5: *Every alternative path p in the input graph $G = (V, E)$ corresponds to an s-t path p' in the transformed graph.*

Proof. Let $p = (v_1, \ldots, v_k)$ be an alternative path in G where $s = v_1$ and $t = v_k$ with respect to parameters T and S. We know that $|p| = k - 1 \geq T$ since $T = \alpha \cdot d(s, t) \leq d(s, t) \leq |p|$. For every $1 \leq i \leq k - T$ the subpath $p_i := (v_i, \ldots, v_{i+T})$ has weight $w(p_i) = |p_i| = T$. By the local optimality criterion, it follows that p_i is optimal. Therefore, there exists a corresponding vertex for p_i in the transformed graph. For every $1 \leq i < k - T$, there is also an edge (p_i, p_{i+1}) in the transformed graph, since $p_i[2..T+1] = p_{i+1}[1..T]$. With $p_1[1] = v_1 = s$ and $p_{k-T}[T+1] = v_k = t$, the edges (s, p_1) and (p_{k-T}, t) exist. Thus, the path $(s, p_1, \ldots, p_{k-T}, t)$ exists in the transformed graph, and $p = (s) + p_1 + \cdots + p_{k-T} + (t)$. ∎

Lemma 5.6: *Every s-t path p' in the transformed graph corresponds to an s-t path p in the input graph that is locally optimal with respect to T.*



(a) The unweighted input graph.



(b) The transformed graph. Vertices that represent optimal paths are rectangular. Note that there is no vertex for the path (f, e, t) in the transformed graph since this path is not optimal.

Figure 5.4: An example for the transformation of an unweighted graph into the graph of optimal paths of length $T := 2$.

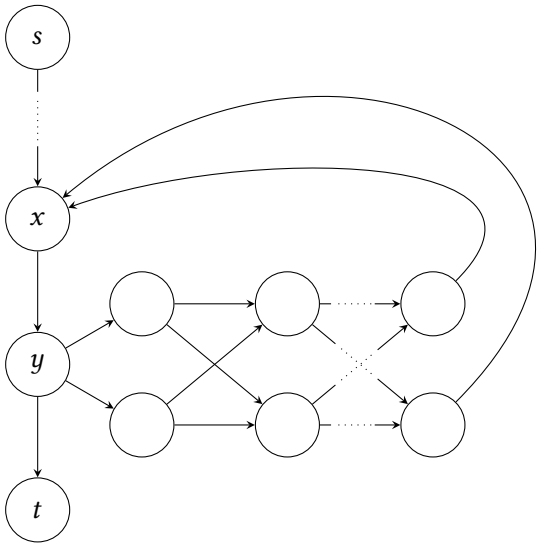


Figure 5.5: An example for a graph with an exponential number of cyclic, but locally optimal paths. The cycle $(x, y, \ldots, x)$ must have length $> T$ since shorter cycles can never be (locally) optimal.

Proof. Let $p' = (s, p_1, \ldots, p_k, t)$ be an s-t path in the transformed graph with $|p'| = k + 1$. Then $p = (p_1[1], p_2[1], \ldots, p_{k-1}[1], p_k[1], p_k[2], \ldots, p_k[T + 1])$ is an s-t path in the original graph. It is obtained by reconstructing p from p', as described above, after dropping the auxiliary vertices s and t. Every subpath of p with length $|p| = T$ is a vertex in the transformed graph and therefore optimal. It follows that p is locally optimal with respect to T. ∎

Although we can now easily find locally optimal paths in the original graph using s-t paths in the transformed graph, not every such path is also an alternative path. A locally optimal path might not be an alternative path for two reasons. Firstly, a path might exceed the maximum stretch S. This can be addressed straightforwardly, as the transformation preserves the relative lengths of paths. We can therefore simply enumerate s-t paths in the transformed graph ordered by their length and terminate as soon as the corresponding paths in the original graph exceed S. Secondly, a path might contain cycles. There might even be an exponential number of cyclic, but locally optimal, paths in the input graph. Figure 5.5 shows an example of such a graph. Therefore, filtering out cyclic paths while enumerating all paths in the transformed graph may not be feasible for cyclic graphs. However, depending on the choice of the local optimality and maximum stretch parameters α and ε, we can even rule out this case.

Lemma 5.7: *If $\alpha > \varepsilon$, then there exist no cyclic s-t paths that are both locally optimal with respect to T and of weight $\leq S$.*

Proof. Let $p = (s, \ldots, x, \ldots, x, \ldots, t)$ be an s-t path in G with a cycle subpath $(x, \ldots, x)$. If $|(x, \ldots, x)| \leq T$, then this subpath is not optimal due to the cycle and therefore p is not locally optimal with respect to T. Otherwise, if $|(x, \ldots, x)| > T$, then $w(p) > w((s, \ldots, x, \ldots, t)) + T \geq d(s, t) + T = (1 + \alpha) \cdot d(s, t) > (1 + \varepsilon) \cdot d(s, t) = S$, if $\alpha > \varepsilon$. ∎

Combining Lemmas 5.5 to 5.7, we obtain an algorithm for enumerating alternative paths when the input graph fulfills certain restrictions. If the unweighted input graph is a DAG or $\alpha > \varepsilon$ holds, then we can apply the transformation described above. Since there exist $\binom{n}{T} \in \Theta(n^T)$ paths of length T, the transformation requires polynomial time if T is a constant. This demonstrates that the problem tends to be simpler for smaller values of T, provided that S is also sufficiently small. On the other hand, individual edges in unweighted graphs are always shortest paths, so any alternative path can be represented as a via path. As seen in Section 5.1, the problem becomes simpler for large T too, by bounding the number of via vertices.

Following the transformation, we enumerate shortest s-t paths in the transformed graph, following the same idea as described in Section 4.1. This can also be accomplished in time polynomial in the number of paths [Yen71]. These paths can be converted back into paths in the original graph according to Lemma 5.6. Enumeration terminates when the paths exceed length S. All resulting paths are indeed alternative paths as per Lemma 5.7. As a result of Lemma 5.5, all alternative paths in the original graph have then been enumerated. Thus, under the specified conditions, it is possible to enumerate all alternative paths in time polynomial in the number of alternative paths.

5.3 Cyclic Paths

Having considered two constrained variations of the problem, we now turn to a generalization of the concept of alternative paths. In this section, we briefly discuss cyclic alternative paths by lifting the restriction that alternative paths need to be simple. Note that this consideration is only applicable with our adapted definition of alternative paths, since we do not require a uniformly bounded stretch. In the original definition by Abraham, Delling, Goldberg, and Werneck [ADGW13], cycles are already entirely eliminated by this criterion, since cycles trivially exceed any stretch.

However, for our definition, we can show the following statement about the maximum number of times the same vertex can occur in a path, depending on the local optimality and maximum stretch parameters α and ε.

Lemma 5.8: *For a cyclic alternative path p and a vertex x, the number of occurrences of x in p is bounded by $\left\lfloor \frac{1+\varepsilon}{\alpha} \right\rfloor + 1$.*

Proof. Let p be a cyclic alternative path and let x be a vertex that occurs multiple times in p. Consider any subpath $p' = (x, \ldots, x)$ of p that is a cycle starting and ending at x. By the local optimality criterion, we know that $w(p') > T$ must hold, since p' cannot be optimal. Therefore, there can be at most $\left\lfloor \frac{S}{T} \right\rfloor = \left\lfloor \frac{1+\varepsilon}{\alpha} \right\rfloor$ of such cycles in p. It follows that x can occur at most $\left\lfloor \frac{1+\varepsilon}{\alpha} \right\rfloor + 1$ times in p. ∎

6 Conclusion

In this thesis, we have addressed the theoretical problem of enumerating alternative paths in a graph. In contrast to practical navigation systems, our goal throughout has been to find the entire solution set. We presented three algorithms that are capable of enumerating all alternative paths in Chapter 4. In all three cases, we were able to construct an input instance for which the respective algorithm must consider a super-exponential number of paths that are not alternative paths. It remains an open question whether there is an algorithm that solves the Alternative Paths Enumeration Problem efficiently. Closely related is the question of the problem's computation complexity. In Chapter 3, we proved that the problem lies in EnumP. However, we do not know whether it can be further classified using either of the complexity classes DelayP, IncP or TotalP, as defined by Strozecki [Str10]. In other words, we are interested in an algorithm whose runtime grows at most polynomially with the size of the output, i.e., the number of alternative paths. Future research could explore the existence of algorithms with this property.

On the other hand, it may be difficult or even impossible to find such an algorithm. This is the case, for example, if the enumeration problem proves to be EnumP-complete. Our attempts to find a parsimonious reduction of 3-SAT were fruitless, both for our definition of alternative paths and for the original definition by Abraham, Delling, Goldberg, and Werneck [ADGW13]. The same applies to a possible polynomial-time many-one reduction to the decision problem of whether there exists another alternative path besides the shortest path. Therefore, the question remains whether there is a reduction that demonstrates the difficulty of the problem. This could also be a starting point for future research.

Apart from complexity, all sorts of variations of the problem could be studied in order to find more efficient approaches for constrained cases. In particular, restrictions of the concept of alternative paths and the input graphs could be considered, as we did in Sections 5.1 to 5.2. For example, there may be algorithms for which the original definition of alternative paths by Abraham, Delling, Goldberg, and Werneck [ADGW13] makes a significant difference. The criterion of limited sharing, where the order of the enumeration is relevant, could also be explored. Moreover, there is a variety of graph classes for which one could design tailored enumeration algorithms, such as undirected graphs, planar graphs, graphs with restricted weights, or parameterized graphs.

Bibliography

[ADGW13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "Alternative routes in road networks". In: *ACM J. Exp. Algorithmics* Volume 18 (Apr. 2013). ISSN: 1084-6654. DOI: *10.1145/2444016.2444019*.

[BDGS11] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. "Alternative Route Graphs in Road Networks". In: *Theory and Practice of Algorithms in (Computer) Systems*. Edited by Alberto Marchetti-Spaccamela and Michael Segal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 21–32. ISBN: 978-3-642-19754-3.

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* Volume 1 (Dec. 1959), pp. 269–271. ISSN: 0945-3245. DOI: *10.1007/BF01386390*.

[DS16] Holger Döbler and Björn Scheuermann. "On Computation and Application of k Most Locally-Optimal Paths in Road Networks". In: *Fachgespräch Inter-Vehicle Communication 2016*. 2016, pp. 32–35. DOI: *http://dx.doi.org/10.18452/1440*.

[Fis20] Samuel M. Fischer. "Locally optimal routes for route choice sets". In: *Transportation Research Part B: Methodological* Volume 141 (2020), pp. 240–266. ISSN: 0191-2615. DOI: *https://doi.org/10.1016/j.trb.2020.09.007*.

[Flo62] Robert W. Floyd. "Algorithm 97: Shortest path". In: *Commun. ACM* Volume 5 (June 1962), p. 345. ISSN: 0001-0782. DOI: *10.1145/367766.368168*.

[PEB53] Bateman Manuscript Project, A. Erdélyi, and H. Bateman. *Higher Transcendental Functions*. Vol. 2. McGraw-Hill Book Company, 1953.

[Str10] Yann Strozecki. "Enumeration complexity and matroid decomposition". Université Paris Diderot - Paris 7, 2010.

[Yen71] Jin Y. Yen. "Finding the K Shortest Loopless Paths in a Network". In: *Management Science* Volume 17 (1971), pp. 712–716. ISSN: 00251909, 15265501.