Ghent University
Faculty of Sciences
Department of Applied Mathematics and Computer Science

Alternative routing algorithms for road networks

Dissertation submitted in fulfillment of the requirements for
the degree of Doctor of Computer Science

Stéphanie Vanhove

19 november 2012

Promotor: prof. dr. Veerle Fack

Dankwoord

Eerst en vooral wil ik mijn promotor Prof. Veerle Fack bedanken voor het aanbrengen van de onderwerpen, de vele bijeenkomsten en het steeds zeer snel nalezen van allerlei stukken tekst. Het FWO-Vlaanderen wil ik bedanken voor de kansen die ze mij geboden hebben door mij een beurs te geven. Ook de leden van de jury wil ik bedanken voor de suggesties om deze tekst verder te verbeteren.

Natuurlijk zijn er nog veel andere mensen die bewust of onbewust hebben bijgedragen tot dit werk. De mensen van onze onderzoeksgroep "Combinatorische Algoritmen en Algoritmische Grafentheorie" wil ik bedanken voor allerlei hulp doorheen de jaren, voor het ter beschikking stellen van verschillende servers, en voor de discussies over de meest uiteenlopende onderwerpen tijdens de wekelijkse "caagtlunch". Ook de mensen van de vakgroep Geografie wil ik bedanken, in het bijzonder Kristien, die steeds een pasklaar antwoord had voor allerlei GIS-kwesties. Bovendien waren "de geografen" steeds zeer aangenaam reisgezelschap op conferenties! Ik wil ook Wouter, Katia, Herbert, Ann en Hilde bedanken, die steeds klaar stonden om al onze administratieve, technische en praktische beslommeringen zo snel mogelijk op te lossen.

Ik wil natuurlijk ook al mijn andere collega's bedanken, en in het bijzonder mijn bureaugenoten. Meer dan drie jaar deelde ik het bureau met Leila. Bedankt Leila, voor het gezelschap, de nieuwjaarscadeautjes en alles wat je vertelde over je land en cultuur. Ik hoop nog steeds dat ik je ooit kan komen bezoeken!

DANKWOORD

Ook tijdens de eindsprint was de hulp van een aantal mensen zeer waardevol. Bert en Nico waren als recente ervaringsdeskundigen steeds bereid om hun ervaringen te delen. Karel las op de valreep nog enkele stukken tekst na en Virginie zag er nauwgezet op toe dat ik geen enkel praktisch detail over het hoofd zag. Glad was als LaTeX-goeroe steeds bereid om alle mogelijke LaTeX-problemen te helpen oplossen. Joke deed een schitterende job bij het ontwerpen van de cover en tante Rita nam een groot deel van de organisatie van de receptie op zich. Bedankt allemaal!

Ik wil ook mijn vrienden bedanken die steeds voor de nodige ontspanning en meer zorgen, en die er gelukkig begrip voor hadden dat ik de laatste maanden wat minder tijd had voor hen. Bedankt Evi, Heidi, Katrien, Sim, Siegried, Ragna, Taheen, Joram, Jan, Joke en Benny!

Mijn moeder en broer verdienen zeker een speciale vermelding. De voorbije periode viel ik vooral op door afwezigheid, en ik besef zeer goed dat dit voor jullie niet evident was. Zonder jullie onophoudelijke inzet zat de lezer nu niet in dit boekje te bladeren. Bedankt!

Tante Bee en nonkel Willy mogen ook niet onvermeld blijven. Ook al wonen jullie meer dan 6 000 km ver, jullie steun en interesse is nooit veraf. Op alle belangrijke momenten is er steeds een e-mail of telefoontje van jullie en dat wordt ten zeerste geapprecieerd. Bovendien bestaan er geen betere vakanties dan bij jullie in Atlanta!

De belangrijkste vermelding gaat echter ongetwijfeld naar mijn vader. Alles wat ik van hem geleerd heb, is zeker en vast bepalend voor mijn hele leven. Zijn optimisme en zijn manier om overal het beste van te maken, wat er ook gebeurt, doen weinigen hem na. Bovendien was tijdens mijn volledige school- en studieloopbaan geen enkele cryptische cursuspassage hem te veel, en fungeerde hij tijdens het schrijven van mijn masterthesis als zelfverklaarde “permanente leescommissie”. Ik kan hem niet genoeg bedanken, maar ik wil mijn best doen om in de buurt te komen.

DANKWOORD

Ik ben nu bijna op het eindpunt gekomen van vijf jaar bij de vakgroep Toegepaste Wiskunde en Informatica. Toen ik hier startte op 1 oktober 2007, had ik geen idee dat de vakgroep me zoveel meer zou bieden dan enkel een interessante onderzoeksomgeving. Spelletjesavonden, filmavonden, drie geslaagde edities van TWIkend, kajaktochten, lasershooting... het is te veel om op te noemen, maar vooral ook de samenhorigheid en vriendschap die er altijd geweest is en nog steeds is zorgden ervoor dat het echt een plezier was om hier te werken. Ik zal zeker nog vaak met heimwee terugdenken aan deze periode, maar tegelijk kijk ik ook uit naar de nieuwe start die voor me ligt, en heeft de TWI-traditie al lang aangetoond dat we elkaar zeker nog vaak terugzien op allerlei gelegenheden in de toekomst. Tot binnenkort!

Stéphanie Vanhove
november 2012

Contents

CONTENTS

1 Introduction

1.1 Routing algorithms

Routing algorithms have been studied for over 50 years. Around 1960 several algorithms were proposed, such as the algorithms of Dijkstra [25], Floyd-Warshall [30, 67] and Bellman-Ford [15, 31]. The late 90's and early 2000's saw the rise of interactive route planning applications, mobile navigation devices and digital GIS systems, for which efficient routing algorithms were needed which can run on large-scale road networks. The fact that online routing applications process many queries in a short time, and the fact that mobile devices usually run on a rather slow CPU increased the interest in efficient routing algorithms even more. The Floyd-Warshall approach is not very suitable for large road networks since it calculates and stores the shortest route between *all* pairs and is therefore both very time-consuming and memory-consuming. The Bellman-Ford algorithm can be used on road

networks. However, it is best suited for networks with both positive and negative arc weights. The algorithm of Dijkstra does not allow negative arc weights, which are usually not present in road networks, and has a better time complexity. This makes the algorithm of Dijkstra a better choice.

For this reason, countless efficient algorithms have been proposed which are based on the algorithm of Dijkstra, most of which aim at reducing the search space of the algorithm of Dijkstra. A comprehensive but incomplete overview is given by O'Brien [54]. Several of the methods described in this overview were proposed by Goldberg et al. [33, 32, 35, 34]. The *A* search* method, which is also described in Michalewicz and Fogel [49], is a variation on the algorithm of Dijkstra that uses an estimation of the distance to direct the search towards the target. One possibility to calculate such an estimation is by using *landmarks*. A small subset (e.g. 20) of the nodes in a network are chosen as landmarks. In a preprocessing step, the distance from and to every node and every landmark is calculated. This information can then be used to calculate estimations using the triangle inequality. Algorithms for selecting suitable landmarks are given in [35, 34]. Goldberg et al. also use the *reach* method, which is another pruning method for the algorithm of Dijkstra proposed by Gutman [37]. When the coordinates of the nodes are present, which is usually the case in road networks, they can also be used in a preprocessing step to speed up the algorithm of Dijkstra. An example of such a method is the *container* method presented by Willhalm [68]. Finally, Schultes [57] presents an impressive algorithm. It represents the road network in a hierarchical structure, distinguishing between different levels of "more important" roads (e.g. highways) and "less important" roads (e.g. small local streets). By taking advantage of this hierarchical structure, the Dijkstra search space is reduced massively.

While many efficient algorithms have been proposed, they do not always satisfy the needs of the user completely. We will call a *standard shortest path algorithm* an algorithm which calculates one shortest route between two given points while assuming that the length of the total route is the sum of the length of every road segment on this route. However, modern routing applications demand more and the algorithms should be adapted

to more realistic situations and queries. Therefore we believe it is interesting to investigate alternative routing problems rather than to optimize the standard shortest path algorithms even further.

1.2 Alternative routing algorithms

In this work we focus on three different alternative routing problems, each of which is described in the following sections.

Turn restrictions

An important difference between routing in road networks and in networks in general is the fact that turns can be forbidden in road networks. Roads can have lane dividers or forbidden turn signs which make a specific turn impossible. Furthermore, a turn can also have a certain *cost*. Typically, this is a waiting time spent at intersections, traffic lights or the time needed to physically take a sharp turn. This means that in some cases a detour can be necessary to avoid a forbidden or expensive turn, even if this means passing through the same intersection twice. In the decades before digital routing in road networks became popular, it was less necessary for shortest path algorithms to take this into account, which is why many standard shortest path algorithms assume that every turn is legal and has a zero cost. However, turn costs can have a large impact on the total travel time. Nielsen et al. [53] state that in Copenhagen 17% to 35% of the travel time for cars is spent waiting at intersections. In more congested cities, this ratio is probably even higher. We will use the term *turn restrictions* as a general term for turn costs and turn prohibitions. Currently, the integration of turn restrictions in routing systems is still not optimal. Many datasets do not contain data on turn restrictions. Even though some of the more detailed datasets include information on turn prohibitions, no datasets with realistic turn costs are available at this moment. Also, many routing algorithms still do not take turn restrictions into account. However, three methods

have been proposed: the node splitting method [43, 58], the line graph method [20, 10, 69, 29] and the direct method [36]. It would be of interest to learn which of these methods is best suited for a particular dataset, something which has remained unclear up to now. We will show that the performance of these algorithms is very dependent on the number of turns which are prohibited or have a (non-zero) cost. We present an experimental evaluation to discover how these algorithms compete with each other on road networks with different amounts of turn costs and turn prohibitions and present a guideline for choosing the right method for a specific road network.

k shortest paths

Another alternative routing problem is the k *shortest paths* problem, where not only the shortest path is desired, but also the 2^{nd} shortest path, the 3^{rd} shortest path etc. k shortest paths can e.g. be used for routing with additional constraints. A k shortest paths algorithm can be used to generate a ranking of shortest paths, from which the paths that best satisfy the other criteria (such as safety or beautiful scenery) are selected. Different exact algorithms [70, 48, 38] for the k shortest paths problem have been proposed, but they are all very time-consuming. On the Belgian road network, it can take up to 10 minutes to find the 100 shortest paths using an exact algorithm. Finding the 10 000 shortest paths can even take up to ten hours. This is definitely unacceptable in an interactive routing application. Therefore, faster heuristic methods which do not necessarilly find the optimal solution are necessary. Such heuristic approaches have been proposed by Roddity [55] and Bernstein [16]. However, these heuristics only provide a theoretical upper bound on the quality of the solution, and no experimental results are given. We will present a heuristic approach which finds a solution of good quality and which is much faster than the exact algorithms. We will also show that our heuristic outperforms the theoretical upper bound by Roddity and Bernstein in most cases.

Dissimilar paths

Nowadays routing applications and navigation systems typically do not only calculate a shortest route, but also present the user some alternatives. In this way, the user can choose one of these routes according to his own preferences. Google Maps is a good example of this. Different definitions for dissimilarity can be used. For a routing application, it can be sufficient to avoid sharing large road segments. A parallel road which is very close to the original road can be a valid alternative. In other contexts, such as the transportation of hazardous materials, the physical distance between the routes needs to be taken into account in the definition of dissimilarity. In this way, the risk can be spread over different areas. In a dissimilar paths algorithm, it is very important to make sure that the found solution is of good quality. On the one hand, the paths cannot be too much alike. On the other hand, they can also not be too long. The paths should of course not contain cycles. While different methods have been proposed, not all methods succeed in finding a solution with a quality that is acceptable for use in a realistic routing application. A major issue is that the routes often contain detours which are not natural to the user, e.g. leaving a major road and then joining it again. Only in recent years, attempts have been made to eliminate this problem. The concept of local optimality was introduced by Abraham et al. [11]. Still, the existing methods are either time-consuming or fail to find a solution in many cases. We will present a heuristic approach which is very fast at finding a solution consisting of locally optimal paths and rarely fails to find a solution.

Experiments

While a low theoretical complexity is important, we also attach great importance to the performance of our algorithms in practice. Many CPU hours were spent comparing and evaluating our algorithms to each other and to existing methods. The results of these experiments are presented in this work.

1.3 Overview

In Chapter 2 the concepts are introduced which are necessary for the remainder of this work. In Chapter 3 we present a computational experiment comparing methods for routing while taking turns into account. This work was published in *Operations Research Letters* [65]. In Chapter 4 we present our heuristic for the k shortest paths problem and show how well it performs based on our experiments. This work was published in *International Journal of Geographical Information Science* [64]. In Chapter 5 our algorithm for dissimilar paths is presented. Finally, the concluding remarks can be found in Chapter 6. Papers and abstracts regarding one or more chapters from this text have been presented at and appeared in the proceedings of international conferences, such as Cologne-Twente 2011 [63], ACM SIGSPATIAL 2010 [61], GIScience 2010 [62], InterCarto-InterGIS 2009 [60] and LBS 2008 [59]. Throughout this text many symbols are used, most of which can be looked up in the index at the end.

2
Preliminaries

In this chapter we will introduce the concepts and notations which are necessary for the remainder of this text. For routing algorithms it is of course crucial to have an efficient digital representation of the road network. A comprehensive description of the route calculation problem in road networks and the modeling of road networks in GIS systems is given by Miller and Shaw [51, 52]. A road network is essentially modeled as a graph. Many concepts from graph theory are used in the representation of a road network. In Section 2.1 we will describe the necessary graph theory concepts and introduce some notations. In Section 2.2 our graph implementation is described. Section 2.3 elaborates on the algorithm of Dijkstra, on which many routing algorithms are based. Section 2.4 describes the experimental setup which was used for all experiments. Particularly, the road networks which are used as test data are described. Problem-specific elements regarding the experimental setup are described in the corresponding chapters.

2.1 Basic concepts in graph theory

2.1.1 Graph

While there are many variants on the concept of a *graph*, a graph can be defined in general as follows:

Definition 2.1. *A graph G is an ordered pair (V, E) such that V is a finite set of nodes and E is a finite set of edges e such that $e = \{u, v\}$ is an unordered pair with $u \in V$ and $v \in V$.*

A *node* is often called a *vertex* in graph theory, but we will use the term *node* which is more common in a GIS context. Edges are *undirected* by definition. When E is a collection of ordered (u, v) pairs, the graph is called a *directed graph* and the ordered (u, v) pairs are called *arcs*. The node u is called the *tail* and the node v is called the *head* of the arc. We will always assume the graphs to be directed, so we will only consider arcs. Throughout this text, we will call n the number of nodes in the graph, i.e. $|V|$ and we will call m the number of arcs in the graph, i.e. $|E|$. A node $x \in V$ and an edge $\{u, v\}$ or arc $(u, v) \in E$ are called *incident* if and only if $x = u$ or $x = v$. Two nodes $u \in V$ and $v \in V$ are *adjacent* if and only if there exists an arc $(u, v) \in E$. The *degree* of a node $v \in V$ is the total number of arcs incident to v. In directed graphs, the *in-degree* and *out-degree* of a node $v \in V$ are the total number of incoming and outgoing arcs in v, respectively. A *weighted graph* assigns a number (= a *weight*) to every arc. The weight of an arc $e = (u, v)$ is denoted as $w(e)$ or as $w(u, v)$. A *subgraph* G' of a graph $G = (V, E)$ is an ordered pair (V', E') such that $V' \subseteq V$ and $E' \subseteq E$, where the arcs in E' are restricted to the nodes in V'.

2.1.2 Graph visualisation

Nodes are usually visualised by circles. An arc is visualised as an arrow from the tail node towards the head node of the arc. A line without an

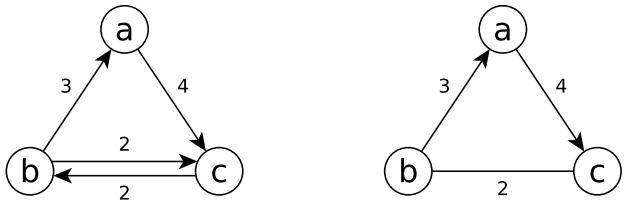


Figure 2.1: Example of a graph. Nodes are labeled with the letters a, b, c. The weights for the arcs are shown. There are two arcs with the same weight in both directions between b and c. This can be visualised as shown on the left or on the right.

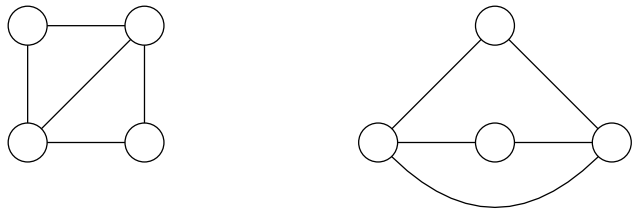


Figure 2.2: Two visualisations for the same graph. Both graphs are the same, since they have the same topology, even though they have a different embedding.

arrow represents two arcs with the same weight, one in each direction. Arc weights are shown as a number along the arc. When necessary, some or all nodes may be labeled with a number or letter inside the circle. An example is shown in Figure 2.1. It should be noted that only topology is defined for a graph. This means that one graph can have different visual representations. This is shown in Figure 2.2. A set of coordinates may be available for the nodes. Such a set of coordinates is called an *embedding*.

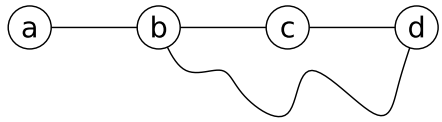


Figure 2.3: Notation for a path for which no details are shown. The graph contains two paths between node b and node d: the path $b{-}c{-}d$ and another path for which details are omitted (represented by the curved line).

2.1.3 Paths

A *walk* is a sequence of nodes (v_1,v_2,v_3,...,v_l) such that v_i and v_{i+1} are adjacent, for all i such that $1 \leq i < l$. If all arcs on a walk are different, the walk is called a *trail*. If all arcs and all nodes on a walk are different, it is called a *path*. The weight $w(P)$ of a path P is the sum of the weights of its arcs, i.e. $w(P) = \sum_{i=2}^{\ell} w(v_{i-1}, v_i)$. We will use the notation $P.nodeAt(i)$ to represent the node at position i in P. A closed path (v_1, v_2, v_3, ..., v_l) such that $v_1 = v_l$ is called a *cycle*. A graph is *connected* if a path exists between any pair of nodes in the graph. Otherwise the graph is *disconnected*. A directed graph is *strongly connected* if for every pair of nodes $\{x, y\}$ a path exists from x to y and from y to x. A *subpath* of a path P is a path which is part of the larger path P. We will denote the subpath in P from x to y as $P[x..y]$. Two paths $P_1 = (v_1, ..., x)$ and $P_2 = (x, ..., v_l)$ can be concatenated using the + operator, i.e. $P_1 + P_2 = (v_1, ..., x, ..., v_l)$, given that the last node of P_1 and the first node of P_2 are the same. In the same manner, a path $P = (v_1, ..., x)$ and an arc $e = (x, t)$ can be concatenated, i.e. $P + e = (v_1, ..., x, t)$, given that the last node of P and the tail node of e are the same. When the context is clear, we will sometimes refer to a path or subpath from x to y as the $x - y$ path or the $x - y$ subpath. We will use a curved line to represent paths for which no details are shown, as can be seen in Figure 2.3.

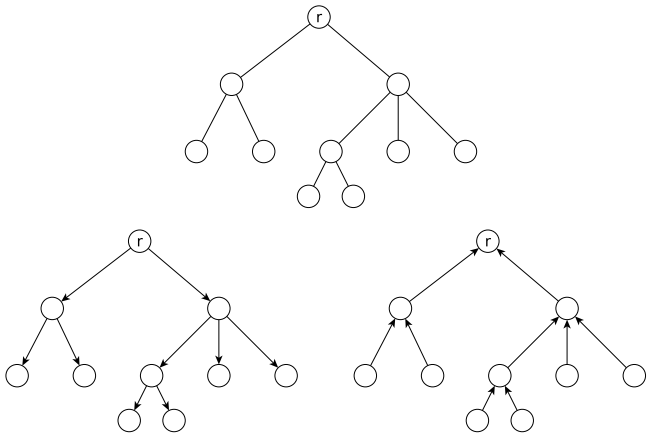


Figure 2.4: Undirected rooted tree (top), outward tree (left) and inward tree (right). The root nodes are labeled as r.

2.1.4 Trees

A *tree* T is a connected graph which has no cycles. One of the nodes can be assigned as the *root*. In this case the tree is called a *rooted tree*. In a directed tree, the arcs can be *outward* (i.e. away from the root) or *inward* (i.e. towards the root). We will call a rooted directed tree where are all arcs are outward or inward an *outward tree* or *inward tree*, respectively. In an outward tree, there is always exactly one path from the root to every node in the tree. In an inward tree there is always exactly one path from every node in the tree to the root. Figure 2.4 shows examples of an undirected rooted tree, an outward tree and an inward tree.

2.2 Representing road networks as a graph

A road network can be modeled as a weighted, directed graph. Nodes represent intersections or dead ends and arcs represent road segments. Arc

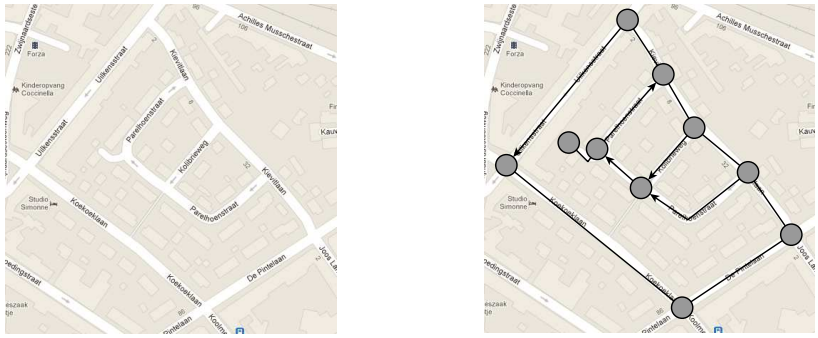


Figure 2.5: A road network (left) and part of its graph representation (right). One-way streets, two-way streets and a dead end can be seen.

weights usually represent either travel distances or travel times. In this work we will assume that arc weights are always non-negative, which is a realistic assumption in road networks. The graphs in this work are always directed. In this way, one-way streets and situations where $w(u, v) \neq w(v, u)$ can be modeled. Figure 2.5 shows an example of how a road network is modeled as a graph. Our graph implementation models the structure of the graph in an *adjacency list*. For every node, all of its outgoing arcs are stored in a Java `ArrayList`. This is the most commonly used Java implementation of a list. The data in the list are stored in an array. For performance and memory efficiency reasons, nodes are not represented by objects but by numbers from 0 to $n - 1$ (with n the number of nodes). This numbering also allows easy indexing in lists. Arcs are stored as Java objects, storing the tail and head node and the weight of the arc. In this implementation, outgoing arcs can be found fast, but it takes $\mathcal{O}(m)$ time to find incoming arcs. However, for some algorithms, it can be necessary to find the incoming arcs fast as well, e.g. when searching backwards. For this reason, the implementation can add *reverse data*, i.e. the incoming arcs, on demand. When the reverse data are no longer needed, they can be dismissed, again on demand.

2.3 Shortest path algorithms

2.3.1 The shortest path problem

First, we define the *shortest path problem*, for which different variations exist:

- The *single-pair shortest path problem* is the problem of finding the path with the lowest weight from a given node s to a given node t.
- The *single-source shortest path problem* is the problem of finding the path with the lowest weight from a given node s to all other nodes.
- The *single-destination shortest path problem* is the problem of finding the path with the lowest weight from all nodes to a given node t.
- The *all-pairs shortest path problem* is the problem of finding the path with the lowest weight between all pairs of nodes.

In routing applications, the user usually enters a *query* in which he specifies a start node s and a target node t, which makes it a single-pair shortest path problem. In this text we will always assume that queries have a start node called s and a target node called t. We will use the abbreviation SP for "shortest path". The weight of the shortest path is denoted by $w(SP)$. However, some algorithms in this text use the single-source and single-destination shortest path problem as well. In the context of routing applications, usually only paths are desirable, and not trails or walks, due to the possibility of repeated arcs and/or nodes. However, in Chapter 3 repeated nodes will be useful in some cases. Also, in Chapter 4 and Chapter 5 some algorithms will start with many trails, some of which may contain cycles, and then finally select a small cycle-free subset, i.e. a set of paths. For readability of the text, we have chosen for a slight abuse of terminology and use the term "path" even though in some cases a cycle may be present, which is either useful or will lead to the dismissal of the path.

2.3.2 The algorithm of Dijkstra

In this section we describe the algorithm of Dijkstra in more detail, since many algorithms, including ours, are based on the algorithm of Dijkstra. It can be used both for the single-source shortest path problem and for the single-pair shortest path problem. Intuitively, it searches in a circle around the start node s until all nodes have been reached, or until a given target node t has been reached. The search can be pruned, something which we will use in this text. The pseudocode is shown in Algorithm 2.1.
The algorithm of Dijkstra assigns *labels* to the nodes it visits. Let $label(v)$ be the label for a node v. A label indicates the shortest distance from s to this node found so far. This means that a shorter distance may or may not exist. A label can be a *temporary label* or a *permanent label*. A temporary label can still be improved, while a permanent label is guaranteed to be the shortest distance to this node. The algorithm also stores a *shortest path tree*.

Definition 2.2. *A shortest path tree SPT for a graph G and a start node s is a tree rooted in s, and is a subgraph of G, such that for every node v in SPT, the (only) path from s to v in SPT is also the shortest path from s to v in G.*

A shortest path tree can be complete or partial. If the algorithm visits every node, the shortest path tree will be complete and contains the shortest path to every node. If the algorithm stops earlier, it is partial and only contains a path to the nodes which had already been visited. For every node v its parent $parent(v)$ in SPT is stored. The algorithm visits the nodes in ascending order of distance from s. This means that when the shortest path to a node v is found, the shortest path to all nodes closer than v has also been found. For this purpose a *priority queue* is used which orders the nodes by their label.

The algorithm starts by assigning a label of 0 to the start node and $+\infty$ to all other nodes. Then, in every iteration, the node *current* with the

Algorithm 2.1 The algorithm of Dijkstra for single-source shortest path problems. It can be used as a single-pair shortest path algorithm by using the permanent labeling of a target node t as a stop condition.

Require: graph $G = (V, E)$, start node s
Ensure: shortest path tree storing the shortest path from s to all nodes

```
1: for all v ∈ V do
2:    label(v) ← +∞
3: end for
4: label(s) ← 0
5: parent(s) ← null
6: add s to priorityqueue
7: add s to SPT
8: while priorityqueue not empty and not STOP do
9:    current ← priorityqueue.poll()
10:   make current permanent
11:   for all nodes neighbour adjacent to current do
12:      if neighbour is not permanent and neighbour can not be pruned
         then
13:         if label(current) + w(current, neighbour) < label(neighbour)
            then
14:            label(neighbour) ← label(current) + w(current, neighbour)
15:            parent(neighbour) ← current
16:            add neighbour to priorityqueue
17:            add neighbour to SPT {override old entries for neighbour}
18:         end if
19:      end if
20:   end for
21: end while
```

smallest label is removed from the queue. The node *current* is made permanent and all of its non-permanent neighbours are examined. For every such node *neighbour*, a new label is calculated, i.e. $label(current) + w(current, neighbour)$. If this label is smaller than the neighbour's current label, $label(neighbour)$ is updated, *neighbour* is added to the priority queue, *neighbour* has its parent set to *current* and *neighbour* is added to the SPT. This overrides any previous entries for *neighbour* in the SPT.

The algorithm continues until all nodes have a permanent label, or until another stop condition has been reached. A very common stop condition is to stop when a given target node t has a permanent label. In this way, the algorithm of Dijkstra can be used as a single-pair shortest path algorithm. As can be seen in Line 11 in the pseudocode, the algorithm can also be pruned. Many algorithms based on the algorithm of Dijkstra make use of a pruning technique in this step, e.g. the A* algorithm [33, 32, 35, 49].

The algorithm of Dijkstra can also be run backward, starting in a target node t and finding the shortest path from all other nodes or from one particular start node s towards t. Therefore, the distinction needs to be made between a *forward shortest path tree* and a *backward shortest path tree*.

Definition 2.3. *A forward shortest path tree SPT_{OUT} for a graph G and a start node s is an outward tree rooted in s, and is a subgraph of G, such that for every node v in SPT_{OUT}, the (only) path from s to v in SPT_{OUT} is also a shortest path from s to v in G.*

Definition 2.4. *A backward shortest path tree SPT_{IN} for a graph G and a target node t is an inward tree rooted in t, and is a subgraph of G, such that for every node v in SPT_{IN}, the (only) path from v to t in SPT_{IN} is also a shortest path from v to t in G.*

Shortest path trees are often marked on a graph by coloring the arcs which are part of the shortest path tree. Figure 2.6 shows an example of a forward

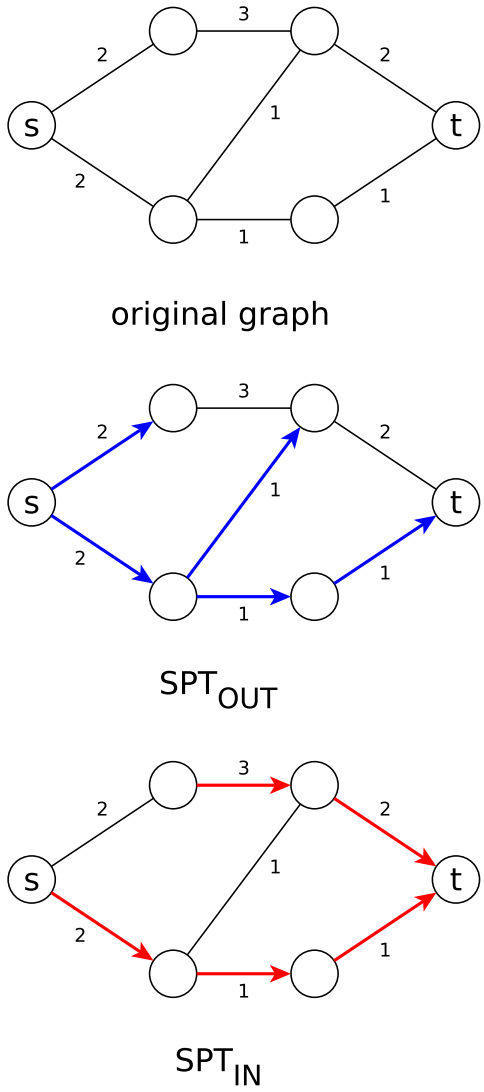


Figure 2.6: SPT_{OUT} (blue) and SPT_{IN} (red) for the given graph on top.

shortest path tree and a backward shortest path tree. The algorithm of Dijkstra can also be implemented in a bidirectional way. This is described by O'Brien [54].

2.3.3 Complexity

Using modern data structures, Dijkstra's algorithm can be implemented with a complexity $\mathcal{O}(m + n \log n)$ [14]. However, it should be noted that road networks are usually sparse. The maximum number of arcs at an intersection is bounded by a certain constant. This means that $m = \mathcal{O}(n)$, a fact which will be used throughout this text. For the algorithm of Dijkstra, it means that the complexity is reduced to $\mathcal{O}(n \log n)$ for road networks.

2.4 Experimental setup

Most of our experiments were run on real-world road networks. A first set of road networks was provided by the Institute of Theoretical Informatics at the Karlsruhe Institute of Technology (KIT), led by Dorothea Wagner. The data were originally intended for the 9th DIMACS Implementation Challenge [1] and were provided to the Institute of Theoretical Informatics by the PTV company in Karlsruhe [2]. These road networks are provided in a text format defined by DIMACS which can easily be read. The road networks are directed and for every arc the weight is provided, which is the distance with a precision of 10 meters. We will refer to these data as the *DIMACS road networks*.
A second set of road networks was provided by NAVTEQ [5, 6], a major American provider of GIS data and digital street maps. The data are provided as *shapefiles* [7], a file format which is very commonly used in a GIS context. An enormous amount of attributes is given, many more than would ever be needed in this work, so most of the attributes we do need are present. Besides the distance, a speed category is also given, which we have used to calculate travel times with a precision of 1 second. The

Road network	Country	# nodes	# arcs	provided by
CZE_MAX	Czech Republic	23 094	53 137	DIMACS
LUX_MAX	Luxembourg	30 047	69 382	DIMACS
IRL_MAX	Ireland	32 868	71 474	DIMACS
PRT_MAX	Portugal	159 945	368 935	DIMACS
BEL_MAX	Belgium	458 403	1 085,076	DIMACS
CHE_MAX	Switzerland	585 514	1 339,281	DIMACS
NAVTEQ_LUXEMBOURG	Luxembourg	39 883	89 594	NAVTEQ
NAVTEQ_PARIS	Paris	313 536	705 588	NAVTEQ
NAVTEQ_BELGIUM	Belgium	564 477	1 300 765	NAVTEQ
NAVTEQ_NETHERLANDS	The Netherlands	1 017 242	2 407 244	NAVTEQ

Table 2.1: Overview of our most commonly used test data. The names in the left column will be used throughout this text. For the DIMACS road networks, not the road network for the entire country is used, but the largest strongly connected subgraph. This explains the MAX in the names.

road networks are also directed. Furthermore, unlike the DIMACS road networks, information on illegal turns is also given, which will be very useful in Chapter 3. We will refer to these data as the *NAVTEQ road networks*.

Table 2.1 shows an overview of our most commonly used test graphs, allthough other graphs were used as well. Every graph in the table is given a name, which will be used throughout this text. It should be noted that for Luxembourg and Belgium we have two road networks. One of them is provided by DIMACS and uses distances as arc weights, while the other one is provided by NAVTEQ and uses travel times as arc weights. For all road networks, the embedding (i.e. the coordinates of the nodes) is present. While some routing algorithms use this embedding, our algorithms do not, so usually the embedding is not loaded into memory. However, the embedding was used for visualisation purposes. For the visualisation of a shortest route or several routes, a Google KML file [8] was created which contains the necessary coordinates to describe the path(s). This KML file was put as a layer on top of Google Maps. Many examples can be seen throughout this text, e.g. Figure 5.3 on Page 100.

All algorithms were implemented, compiled and executed in Java 1.6 [9]. An amount of memory of 1 800 MB was allocated. The experiments in Chapter 3 and 4 were run on a 2.13 GHz processor, while the experiments in Chapter 5 were run on a 2.8 GHz processor. Whenever a standard shortest path algorithm is needed, we have used the algorithm of Dijkstra, since most standard shortest path algorithms are a variation on the algorithm of Dijkstra.

3

Routing algorithms with turn restrictions

3.1 Introduction

Standard shortest path algorithms usually assume that the total weight of a route is simply the sum of the weight of all road segments on this route. However, this model is not always sufficient for a realistic road network. Usually not all turns can be taken in a road network, e.g., because of a lane divider or a forbidden turn sign, and even if a turn is legal, it usually implies an additional cost, e.g., the time needed to physically take a sharp turn, the time spent waiting at an intersection for other vehicles to pass, or the time spent waiting at traffic lights. Figure 3.1 shows two situations where a detour is the best option to avoid a forbidden or expensive turn. In both cases, a standard shortest path algorithm would see no harm in the

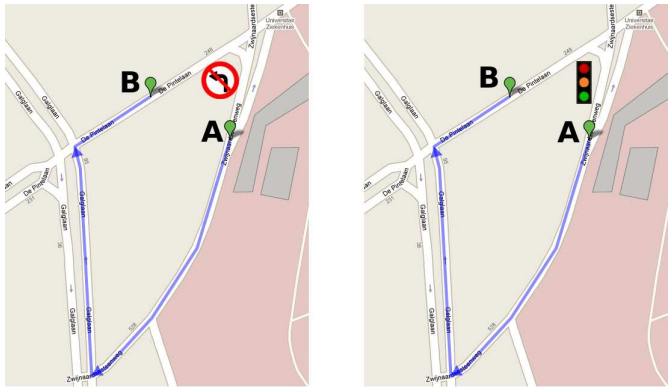


Figure 3.1: Two situations where a standard shortest path algorithm would take the obvious left turn to go from A to B. In reality, however, the best route is the detour shown in the pictures because of either a forbidden turn (left) or a long waiting time at traffic lights (right).

left turn and simply use it. These examples clearly show that it is crucial for a route planner to take such turn restrictions into account. However, standard shortest path algorithms assume that every turn is legal and has a zero cost.

Different methods have been proposed to overcome this problem. They will be discussed more thoroughly in the following sections. It would be of interest to learn how these techniques compete with each other.

The number of turn prohibitions in a road network is of course dependent on the topology of the network. Furthermore, in a real-life situation not all datasets may store a cost for *every* turn in the network. This can be for memory reasons, but also since visiting every turn to determine this cost would be a tremendous effort. Therefore, a dataset may store only the turns for the most important or busiest intersections. However, it is also possible that many or all turns are stored, e.g., if the turn costs are calculated automatically based on the angle. This motivates an experimental study to discover the influence of the amount of stored information on the performance of the algorithms. The results can be used as a guideline for

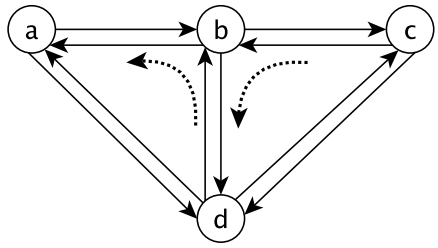


Figure 3.2: Graph with forbidden turns. The dashed arrows indicate forbidden turns: it is legal to move from arc (d, b) to (b, c) but it is illegal to move from arc (d, b) to (b, a).

choosing the best method for a particular dataset. While some studies comparing shortest path algorithms *without* turn restrictions can be found in the literature, e.g. [21, 71, 72, 45], no such study has been done for shortest path algorithms *with* turn restrictions until now.

A *turn* is defined as any transition from an arc to one of its succeeding arcs. This can be a left turn or a right turn, a U-turn, or even straight through. We will distinguish between two kinds of turn restrictions: *turn costs* and *turn prohibitions*. We will use the term *turn restrictions* as a general term for both. A turn cost often reflects the waiting time, e.g. at traffic lights, or the time needed to actually perform the turn, e.g. at very sharp angles. Turn prohibitions on the other hand reflect those situations where it is impossible or illegal to take a certain turn, e.g. due to a lane divider or a traffic sign forbidding the manoeuvre. Figure 3.2 shows an example of a graph with turn prohibitions. The solid arrows represent arcs. The dashed arrows indicate turn restrictions, in this case forbidden turns. In Section 3.2 we give a literature overview on routing algorithms with turn restrictions. Three known algorithms are described in detail in Section 3.3. In Section 3.4 certain implementation issues are addressed. In Section 3.5, 3.6, 3.7 and 3.8 we present detailed computational experiments and their results. Finally, in Section 3.9, we conclude with a guideline for choosing

the right method for a particular road network.

3.2 Literature overview

The literature considers two possible approaches for handling turn restrictions in shortest path algorithms. One approach consists in modifying the graph to integrate the information about the turn restrictions in the structure of the graph itself. Standard graph algorithms can then be applied to the modified graph, without even knowing about the turn restrictions. Examples of this approach are the node splitting method [43, 58] and the line graph method [20, 10, 69, 29].
Another approach is to develop a new algorithm which works on the original graph and explicitly takes the turn restrictions into account. We will call this approach the direct method. Gutiérrez and Medaglia [36] present a modified version of the well-known algorithm of Dijkstra [25], and Ziliaskopoulos and Mahmassani [73] present a similar method.
Besides the shortest path algorithms with turn restrictions themselves, it is also interesting to learn how turn restrictions are assigned to a network. Nielsen [53] gives a very detailed description of the calculation of turn costs, taking an impressive range of factors into account such as road capacity, road congestion, priority of the roads, green time at traffic lights, and green waves. Another method for assigning turn costs is described by Volker [66].
Not only shortest path algorithms can be affected by turn restrictions, many graph algorithms can be applied to road networks and should take turn restrictions into account. Micó et al. [50] present a very general heuristic and an exact method which can be used to include turn restrictions in many well-known graph problems such as the Chinese Postman Problem, the Travelling Salesman problem, and the Capacitated Arc Routing Problem. A more specific method is described by Bräysy et al. [19]. Clossey et al. [22] describe both a direct method and a graph transforming method for the Chinese Postman Problem.
Finally, an efficient implementation for shortest path calculation with turn

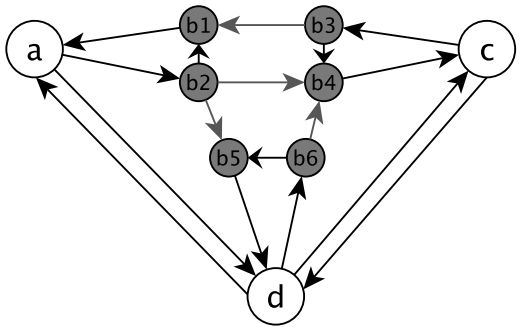


Figure 3.3: Node splitting: transformed graph for Figure 3.2. Node b is split into 6 nodes: b_1, b_2, b_3, b_4, b_5, b_6, each representing an arc incident with node b. Arcs between the split nodes represent legal turns. Split nodes and the arcs between them are shown in grey.

prohibitions in the specific case where a turn is only legal if its angle is between certain values, is proposed by Boroujerdi [17].

3.3 Known algorithms: overview

We have implemented and performed experiments with the node splitting method, the line graph method and the direct method, since all methods in the literature are instances of one of these three techniques. The following paragraphs describe these methods in more detail.

3.3.1 Graph transforming method: node splitting

Node splitting is an intuitive way to modify the graph. Figure 3.3 shows an example of node splitting for the graph in Figure 3.2. Nodes with turn restrictions are split into several nodes, one node for every incoming or outgoing arc. Legal turns are represented by arcs between these nodes. The

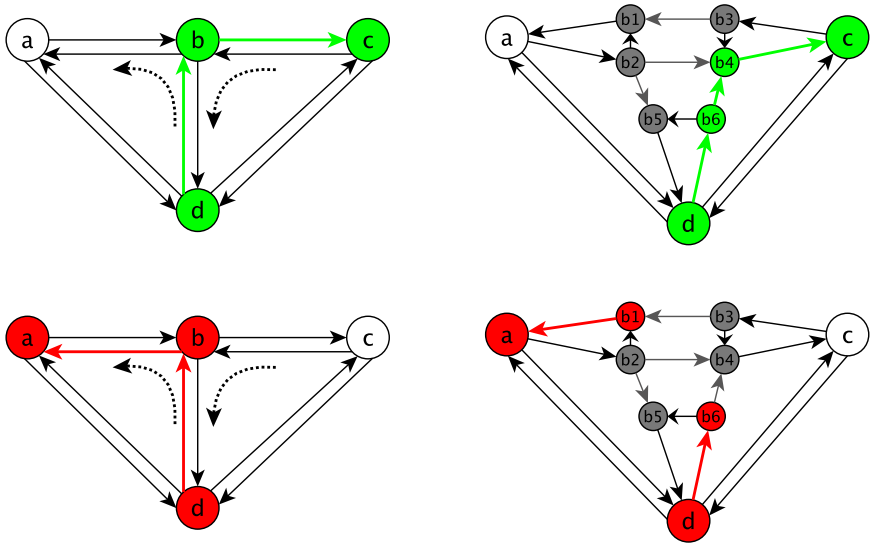


Figure 3.4: Top: legal turn in original graph (left) and in transformed graph (right). It is still possible to take this turn in the transformed graph. Bottom: illegal turn in original graph (left) and in transformed graph (right). The illegal turn cannot be taken in the transformed graph since there is no arc between b_6 and b_1.

cost of such an arc is equal to the cost of the represented turn. Forbidden turns are represented by simply *not* adding an arc between the corresponding pair of nodes, making it impossible to take this turn. This idea is illustrated in Figure 3.4. Nodes without turn restrictions are not split in the transformed graph. Therefore, the size of the transformed graph is dependent on the number of nodes with turn restrictions.

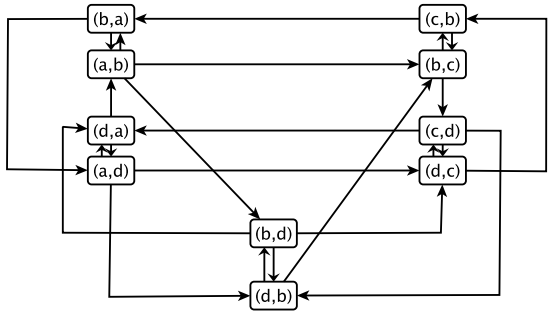


Figure 3.5: Line graph: transformed graph for Figure 3.2. Every node in the line graph represents an arc in the original graph. Arcs in the line graph represent legal turns in the original graph.

3.3.2 Graph transforming method: line graph

Another graph transforming method transforms the graph into its line graph. Figure 3.5 shows the line graph for the graph in Figure 3.2. Every node in the line graph represents an arc in the original graph, while every arc represents a legal turn in the original graph. The weight of such an arc in the line graph is the sum of the turn cost and the weight of the arc which the turn leads to. As with node splitting, illegal turns are simply represented by *not* adding an arc between the two corresponding nodes. This is shown in Figure 3.6. However, there is an important difference between the line graph and node splitting methods. As mentioned before, the node splitting method only splits nodes with turn restrictions, while the line graph method always transforms the entire graph, even if there are nodes without turn restrictions. Therefore, the number of nodes in the line graph is always equal to the number of arcs in the original graph, and the number of arcs in the line graph is always equal to the number of legal turns in the original graph.

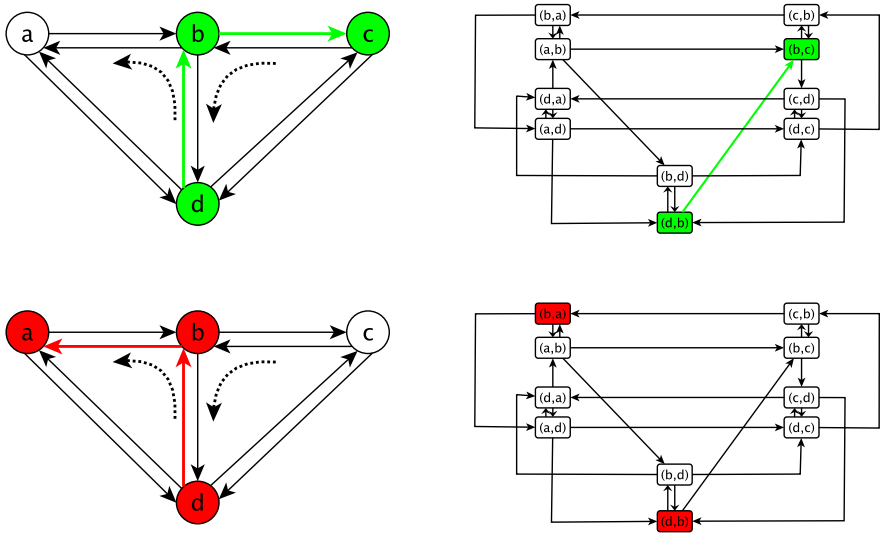


Figure 3.6: Top: legal turn in the original graph (left) and in the line graph (right). The legal turn can still be taken in the line graph. Bottom: illegal turn in the original graph (left) and in the transformed graph (right). The illegal turn cannot be taken in the transformed graph since there is no arc between the two red nodes.

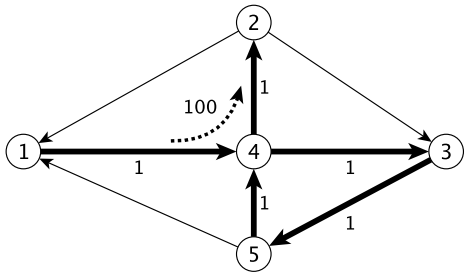


Figure 3.7: Situation where the shortest route contains a cycle. The shortest route without cycles from node 1 to node 2 is $(1, 4, 2)$ and has a weight of 102. However, the shortest route is $(1, 4, 3, 5, 4, 2)$ and has a weight of 5. This route has a cycle since node 4 is used twice. This example shows that, when turn restrictions are present, a shortest route can contain the same node more than once.

3.3.3 Algorithm for original graph: direct method

It is also possible to modify the algorithm instead of the graph. Gutiérrez and Medaglia [36] present a modified version of the algorithm of Dijkstra for finding shortest paths with turn prohibitions. We will call this method the *direct method*. The algorithm of Dijkstra calculates a shortest path which never contains the same node twice, so there are no cycles in the resulting path. However, if the graph has turn restrictions, the shortest route can sometimes include cycles. An example of this situation is shown in Figure 3.7.

While the algorithm of Dijkstra assigns labels to nodes representing the distance from the start node, the direct method assigns labels to arcs instead of nodes. In this way, the direct method calculates a shortest path which never contains the same *arc* twice, but which can contain the same node more than once. An arc label represents the distance from the start node to this arc, including the arc's weight itself. While the authors only describe the direct method for use with turn prohibitions, we extend this method

for accommodating turn costs as well, as described in the next paragraph. The direct method is very similar to the algorithm of Dijkstra, for which the pseudocode is given in Algorithm 2.1 on Page 27. It starts by assigning a label to every outgoing arc of the start node, equal to the arc's weight, and a label with value infinity to all other arcs. However, every newly assigned label is only a *temporary* label since a shorter distance to this arc may still be found. In every iteration the algorithm selects the arc e with the smallest temporary label in the entire graph and makes this label *permanent*. The algorithm then visits all subsequent arcs e' of e. If e' already has a permanent label, e' is skipped. Otherwise, the algorithm looks up the turn restriction from e to e'. If this turn is forbidden, the arc e' is skipped as well. If the turn is legal, a potential new temporary label for e' is calculated which is the sum of the label of e, the turn cost from e to e' (which can of course be zero) and the weight of e'. If this new label is smaller than the current temporary label of e', the label of e' is updated and the parent of e' is set to e to facilitate the retrieval of the shortest path. It can be proven that when a label is permanent, no shorter path to this arc can be found, so the algorithm stops as soon as an arc incident with the target node receives a permanent label.
It should be noted that this direct method is similar to a standard Dijkstra algorithm performed on the line graph. When the algorithm of Dijkstra labels a node in the line graph, it assigns a label to a node which corresponds to an arc in the original graph. This is equivalent to the labeling of an arc in the original graph by the direct method. We may therefore expect a more or less similar behaviour for both algorithms.

3.3.4 Time complexity

As mentioned in Section 2.3.3, the algorithm of Dijkstra can be implemented with a time complexity of $\mathcal{O}(m + n \log n)$. For road networks, this is reduced to $\mathcal{O}(n \log n)$ since $m = \mathcal{O}(n)$. Let t be the number of turns in the graph, t_l the number of legal turns, t_f the number of forbidden turns, and t_c the number of turns with a non-zero turn cost. Let $t_r = t_f + t_c$

be the total number of turns with a turn restriction. Since the line graph method performs the algorithm of Dijkstra on a graph with m nodes and t_l arcs, the line graph method has a time complexity of $\mathcal{O}(t_l + m \log m)$, or $\mathcal{O}(m \log m)$ for road networks.
Even though the behaviour of the direct method is very similar to the line graph method, the complexity is slightly different. Our implementation of the direct method iterates over *all* turns and then checks whether they are legal or not. Because of this, a term t must be added to the complexity of the line graph, resulting in a complexity of $\mathcal{O}(t+t_l+m \log m) = \mathcal{O}(t+m \log m)$, or $\mathcal{O}(m \log m)$ for road networks, since $t = \mathcal{O}(m)$ in road networks.
The node splitting method splits every node which is restricted by a turn cost or turn prohibition into several nodes, one for every incoming or outgoing arc. The number of such restricted nodes in the original graph is bounded by $\mathcal{O}(t_r)$, so the number of nodes after node splitting is bounded by $\mathcal{O}(n + t_r d_{max})$, where d_{max} is the maximum number of incoming or outgoing arcs in a node. An arc is added for every legal turn at a node with turn restrictions, so the number of arcs is bounded by $\mathcal{O}(m + t_l)$. This leads to a time complexity of $\mathcal{O}((m + t_l) + (n + t_r d_{max}) \log(n + t_r d_{max}))$, or $\mathcal{O}((n + t_r d_{max}) (\log(n + t_r d_{max}))$ in road networks.

3.3.5 Memory complexity

Our graph implementation is described in Section 2.2. It stores one object for every arc. For the nodes, no specific object is stored except for a list containing all of its outgoing arcs. There are m objects representing arcs and n (empty or non-empty) lists of arcs. This leads to a memory complexity of $\mathcal{O}(n + m)$ for a graph with no turn restrictions. When adding turn restrictions, an entry is stored in a HashMap for every restricted turn. This results in a memory complexity of $\mathcal{O}(n + m + t_r)$ for a graph with turn restrictions as used by our implementation of the direct method. The line graph however is actually a transformed graph without any turn restrictions, so only its number of nodes m and its number of arcs t_l need to be taken into account. This leads to a memory complexity of $\mathcal{O}(m + t_l)$

for the line graph. Finally, for node splitting, the new number of arcs is bounded by $\mathcal{O}(m + t_l)$, so the memory complexity is $\mathcal{O}(n + m + t_l)$. From this theoretical complexity analysis, it can be expected that t_r and t_l have an important influence on the performance of the algorithms, as the results will confirm.

3.4 Implementation issues

The three methods described in the previous section were implemented in Java. As mentioned before, on a transformed graph any shortest path algorithm can be used. We chose the algorithm of Dijkstra as this gives the best basis for comparison with the direct method. The graph implementation is based on the implementation described in Section 2.2, but stores the turn restrictions as well. This enhanced graph implementation can immediately be used by the direct method, and serves as a basis for a graph transformation by the other methods. In order to store turn restrictions, a Java HashMap is stored for every arc which is followed by a restricted turn. The subsequent arc itself is the key, while the turn cost to travel from the current arc to the subsequent arc is the value. This turn cost is set to $+\infty$ for a forbidden turn. If the turn cost for a certain turn is zero (e.g. a very wide right turn) or if the turn cost is unknown, no key-value pair is added to the HashMap. This can potentially save a lot of memory, especially in those cases where turn costs are only known for large and busy intersections while no information on turn restrictions is available for less important intersections. The main advantage of this data structure is that it allows us to check whether a turn is legal and to retrieve a turn cost very fast.

3.4.1 Graph transformations

When dealing with transformed graphs, certain implementation issues arise. First of all, the shortest path found in the transformed graph needs conver-

sion. The transformed graph contains other nodes than the original graph. Figures 3.4 and 3.6 clearly show that the legal path, marked in green, is represented by a different sequence of nodes in the transformed graphs. For node splitting, the transformed graph contains split nodes which were not present in the original graph. For the line graph, all nodes are new and represent arcs in the original graph. In both cases, the resulting path consists of nodes from the *transformed graph*, not the original graph. Therefore, the path needs to be converted. For this purpose, some extra information needs to be stored. This is described in detail in the following paragraphs. Secondly, for both the line graph and node splitting, a node can be represented by more than one node in the transformed graph. Hence, a shortest path query can have multiple start and/or target nodes in the transformed graph. However, a standard shortest path algorithm such as the algorithm of Dijkstra cannot deal with this situation. As proposed by Winter [69], this can be solved quite easily by adding a *virtual start node* to the transformed graph with outgoing arcs to all start nodes. In the same way a *virtual target node* can be added to the transformed graph with incoming arcs from all target nodes. The outgoing arcs from the virtual start node and the incoming arcs for the virtual target node are called the *virtual arcs*. These virtual nodes and arcs need to be created before every query. Taking this into account, a shortest path algorithm on a transformed graph consists of three steps:

1. Create virtual nodes and arcs in the transformed graph
2. Run any shortest path algorithm on the transformed graph (e.g., the algorithm of Dijkstra)
3. Convert shortest path

We will refer to the time spent for Step 2 as the *strict query time*. However, Steps 1 and 3 may take some time as well, since they both require finding nodes in the transformed graph corresponding to a node in the original graph or vice versa. This raises the question if it would be interesting

to store additional data in memory in which these corresponding nodes can be looked up in constant time. We will call these additional data a *lookup table*. While this decreases the complexity of Steps 1 and 3, it also requires extra memory. There is a clear trade-off here between running time and memory usage. Rather than arbitrarily choosing one of both possibilities, we decided to implement both possibilities for the line graph method and for node splitting and compare them. We will refer to the line graph and node splitting method as LINE and SPLIT, respectively. The implementations with a lookup table will be marked with an asterisk, i.e. LINE* and SPLIT*. For the direct method, such a lookup table is not necessary since no virtual nodes or path conversion are needed. The direct method will only be denoted by DIR.

3.4.2 Line graph

The line graph is implemented in the same way as the original graph, but some additional data are stored. For every node in the line graph, its corresponding arc in the original graph is stored. This information is structured in a list L containing the corresponding original-graph arc for every line-graph node i at position i. In this way, it is possible to convert a path, consisting of line-graph nodes, to a sequence of arcs in the original graph, which can then be converted to a path consisting of original-graph nodes. Another issue is the necessity to find the correct start node(s) and target node(s) in the line graph for a given query. This information is not readily available. LINE* and LINE use different strategies for this, which will be described below.

LINE*

The LINE* implementation stores even more additional information. For every node in the original graph, a list of corresponding start nodes and a list of corresponding target nodes in the line graph is stored. Furthermore, the weights of the virtual start arcs need to be stored. This additional

information enables the retrieval of the required information in constant time.

LINE

The LINE implementation has to rely on the list L to find the start and target node(s). It searches through the entire list and selects all nodes corresponding to an arc in the original graph with the start node as tail or the target node as head. This takes linear time in the number of arcs in the original graph.

3.4.3 Node splitting

Like the line graph, the transformed graph for the node splitting method is implemented in the same way as the original graph, but stores some additional data. As mentioned before, the nodes in the original graph are numbered from 0 to $n-1$, n being the number of nodes in the original graph. Since the transformed graph has more nodes than the original graph because of the split nodes, the transformed graph will inevitably have some new nodes with a number $\geq n$. Nodes without turn restrictions correspond to one single node in the transformed graph, with the same node number. Nodes with turn restrictions are split and thus correspond to multiple nodes in the transformed graph, one of which is given the node number of the original node (to preserve a contiguous node numbering), while a contiguous range of new node numbers is assigned for the other nodes corresponding to this original node. This numbering system can be seen in Figure 3.8, which shows an original graph with turn restrictions on three nodes, and the resulting transformed graph. While a shortest path algorithm can be run on the transformed graph without knowing the meaning of these new split nodes, it is necessary for creating virtual nodes and arcs and for path conversion to know which split nodes in the transformed graph correspond to which node in the original graph and vice versa. The strategies used by SPLIT* and SPLIT are described below.

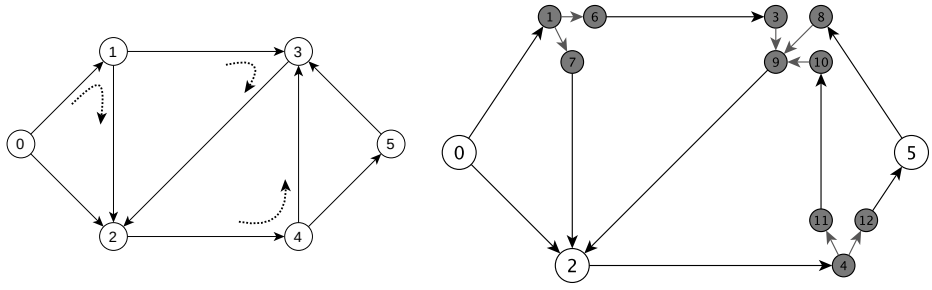


Figure 3.8: Original graph with three nodes with turn restrictions (left) and the resulting transformed graph (right). In this figure, a dashed arrow indicates a *legal* turn with a turn *cost* > 0.

SPLIT*

The data structure used by SPLIT* is quite straightforward. For every node in the transformed graph, its corresponding node in the original graph is stored. This allows access in constant time. It should be noted that the entries for the first n nodes are trivial and can therefore be omitted. The content of this data structure for the example in Figure 3.8 is shown in Figure 3.9. In the opposite direction, another map is stored which maps nodes in the original graph to their range of corresponding nodes in the transformed graph (not shown in the figure).

SPLIT

The information in the structure described above can be stored in a much more compact way if giving up constant-time access is acceptable. This representation uses the fact that the collection of split nodes always consists of the node with the original number together with a contiguous range of node numbers. Therefore it is sufficient to store an array with the node numbers of all nodes with turn restrictions, and a second array of the same length which stores the start of every contiguous range corresponding to

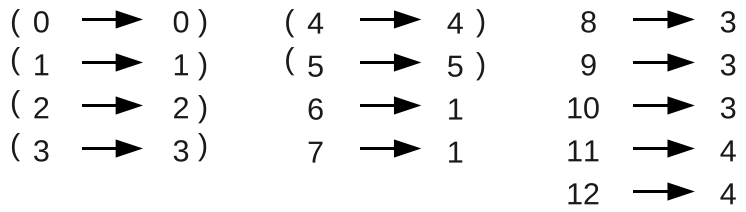


Figure 3.9: Data structure used by SPLIT* for the example in Figure 3.8. For every node in the transformed graph, its corresponding node in the original graph is stored. Entries between brackets are shown for clarity but are not actually stored.

original node	1	3	4
(split nodes)	(1, 6 - 7)	(3, 8 - 10)	(4, 11 - 12)
range start	6	8	11

Figure 3.10: Compact representation for the correspondence between transformed graph and original graph in Figure 3.8. The first row shows the array of nodes with turn restrictions in the original graph. The second row (with data between brackets) is added for clarity but is not actually stored. The third row shows the array of contiguous range starts.

the nodes in the first array. Position i in the second array stores the start of the contiguous range for the original node at position i in the first array. Figure 3.10 shows these arrays for the example in Figure 3.8.

This data structure avoids unnecessary memory overhead by *not* storing every node explicitly. Another interesting benefit of this data structure is the possibility to perform a binary search. Using a binary search it is possible to determine the start of the range in which a certain node lies, and thus also the correspondig node in the original graph. Since a binary search takes logarithmic time, this data structure enables virtual nodes and virtual arcs creation in logarithmic time.

3.5 Experimental setup

In the experiments we aim to discover the influence of the amount of turn restrictions on the performance on the algorithms. We also investigate how the algorithms perform on the NAVTEQ road networks with real-life turn prohibitions and on SPGRID networks for comparison with the results of Guttiérrez and Medaglia [36]. As mentioned before, we need to select a standard shortest path algorithm for calculating shortest paths in the transformed graphs. We chose the algorithm of Dijkstra, since virtually every route planning application is based on some variant of the algorithm of Dijkstra, but we expect similar behaviour for other shortest path algorithms.
In our experiments we study both running times and memory usage. Since Java cannot measure the exact size of an object, an estimation method is used. In short, this method consists of generating a number of copies of the graph object, storing them in an array, calculating the difference between the amount of used memory before and after, and finally dividing this difference by the number of copies created. While this method does not guarantee exact results, substantial efforts were taken to ensure that the measurements are as accurate as possible. Before the estimation starts, Java garbage collection is enforced in order to clear the memory from any remaining objects that are no longer in use. In this way, the measurements cannot be distorted by the garbage collector removing unrelated objects during the execution of the estimation method. Also, the first copy of the object to be measured is always discarded as a "warmup object", after which garbage collection is enforced again. For small objects, a large number of copies was generated and stored in the array mentioned earlier, spreading the error over all copies. For larger objects, memory was filled to capacity with as many copies as possible. It goes without saying that the machine had no other tasks running in the background. We verified the measurements on different 32 bit machines and different operating systems and always obtained equivalent results. Therefore, although this method is not exact, our test results confirm that it is in fact sufficiently accurate.

3.6 Influence of the amount of turn restrictions

For a fair comparison, only the amount of turn restrictions should be variable while all other factors should remain constant. For this purpose we applied different amounts of random turn restrictions to several of the DIMACS road networks. While the turn restrictions are random, the test graphs do represent real-life road networks for European countries. We present detailed results for BEL_MAX. This graph has 458 403 nodes, 1 085 076 arcs, and 2 922 504 turns. Similar results were obtained for the road networks of CHE_MAX, IRL_MAX, LUX_MAX, and PRT_MAX.
We did experiments where 5%, 10%, 15%, 20%, 25%, 50%, 75% and 100% of the turns in the same graph have a randomly chosen non-zero turn cost. For left turns we chose random costs between zero and three times the average arc weight taken over all arcs in the graph. Since waiting times for right turns are usually smaller, right turns were assigned random turn costs between zero and 1.5 times the average arc weight. Since forbidden turns are less common in road networks, lower percentages are chosen for forbidden turns. In this case, different versions of the same graph were assigned 0.1%, 0.5%, 1%, 5%, 10%, 15%, 20% and 25% randomly chosen forbidden turns.

3.6.1 Memory usage

For each of the test networks, we have measured the size in memory of the following: the original graph without any turn restrictions, the original graph + turn restrictions data for the direct method (DIR), the line graph without lookup table (LINE), the line graph with lookup table (LINE*), the transformed graph used by the node splitting method without lookup table (SPLIT) and the transformed graph used by the node splitting method with lookup table (SPLIT*).

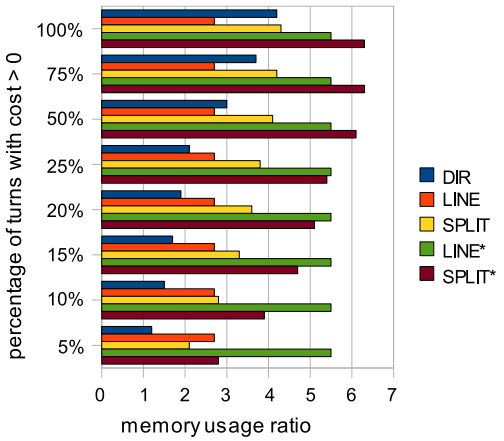
The results for BEL_MAX can be seen in Figure 3.11. The displayed values are not absolute values, but ratios compared to the original graph without any turn restrictions. For example, for a graph where 20% of the turns

imply a non-zero cost, the direct method takes 1.9 times more memory than the original graph. From these charts it is clear that using a lookup table requires a significant amount of extra memory, sometimes even up to a factor two.

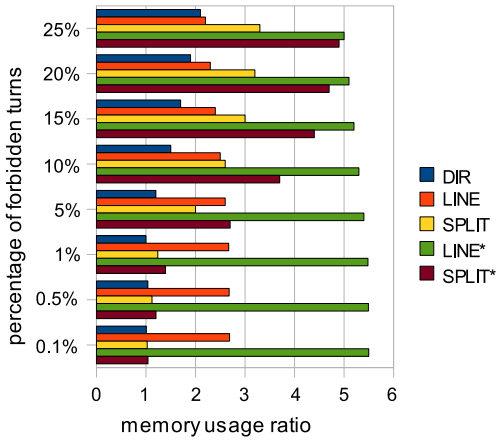
In the case of turn costs, memory usage for the line graph is constant, since this method always transforms the entire graph, even if there are only a few turn costs. On the other hand, memory usage for the direct and node splitting methods increases with the number of non-zero turn costs, since more information on turn costs needs to be stored and more nodes need to be split, respectively. Up to about 25% turn costs, the direct method requires the smallest amount of memory, while for graphs with more turn costs, the line graph (without lookup table) is more memory-efficient. In the case of turn prohibitions, the direct method is always most memory-efficient for the tested percentages. However, for very small percentages of turn prohibitions, the memory-efficiency of the node splitting method and direct method are similar, with the direct method clearly taking over around 5%. The direct method shows exactly the same behaviour for turn costs and turn prohibitions, since saving information on forbidden turns and turn costs takes the same amount of memory. However, memory usage for the line graph decreases with the number of turn prohibitions. This can be explained by the fact that fewer legal turns means fewer arcs in the line graph. As for node splitting, memory usage increases with the number of turn prohibitions, since more nodes need to be split. However, the increase is slower for higher numbers of turn prohibitions since more nodes are already split.

3.6.2 Running times

For the time measurements a set of one hundred random one-to-one queries was generated for each of the test networks. Every query in such a set has a random start node and a random target node. For each graph, time measurements were performed with different parameters. These measurements were all performed on the same set of one hundred queries which was gen-



(a) **Turn costs: memory usage ratio**



(b) **Turn prohibitions: memory usage ratio**

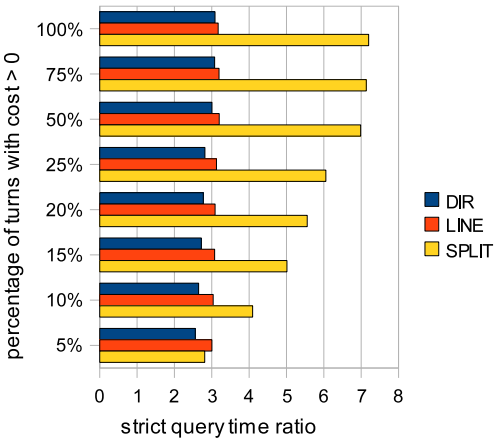
Figure 3.11: BEL_MAX: memory usage ratio for different percentages of turn costs (a) and turn prohibitions (b) for the direct method (DIR), line graph without lookup table (LINE), node splitting without lookup table (SPLIT), line graph with lookup table (LINE*) and node splitting with lookup table (SPLIT*). The size of the original graph without turn restrictions is 57.72 MB.

erated particularly for this graph. The parameters include: the method which was used, the type of turn restrictions (costs or prohibitions) and the number of turn restrictions. The average running time was calculated over these one hundred queries for each set of parameters. Furthermore, even the same query with the same set of parameters was executed 10 times and its average was taken, just to make sure that the time measurements are accurate. Loading the graph into memory is never included in the time measurements since this needs to happen only once for all the experiments on a graph. Figure 3.12 displays the results. They only concern the *strict query time*, i.e., Step 2 mentioned in Section 3.4. The running times for the other two steps were measured separately and will be discussed later. Also, for the time measurements no implementations using a lookup table are considered, since such a lookup table is not used in this step of the algorithm.

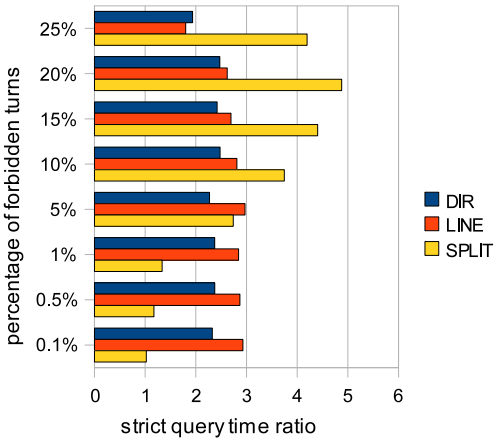
The displayed values in Figure 3.12 are again ratios compared to the average running time for the same queries for a standard Dijkstra algorithm on the original graph. Both charts show very similar query times for the direct method and the line graph, which can be explained by the similarities between both methods. For turn costs, the direct method and the line graph method clearly outperform node splitting. For turn prohibitions, node splitting is the fastest method for few turn prohibitions (less than 5%), but for road networks with many turn prohibitions, the direct method and line graph method are again the fastest. Since their running times are so similar, the actual choice between the direct method and the line graph method should be made based on memory usage.

3.6.3 Time needed for virtual nodes and path conversion

Running times for creating virtual nodes (Step 1) and path conversion (Step 3) were measured separately. For almost all methods, regardless of the usage of a lookup table, the average running times for Step 1 and Step 3 were below 1 millisecond. Keeping in mind that a one-to-one query of the standard Dijkstra algorithm on average takes 254.49 ms for this graph and



(a) **Turn costs: strict query time ratio**



(b) **Turn prohibitions: strict query time ratio**

Figure 3.12: BEL_MAX: average strict query time ratio for different percentages of turn costs (a) and turn prohibitions (b) for the direct method (DIR), the line graph (LINE) and node splitting (SPLIT). The average running time for a one-to-one query of the standard Dijkstra algorithm on the original graph is 254.49 ms.

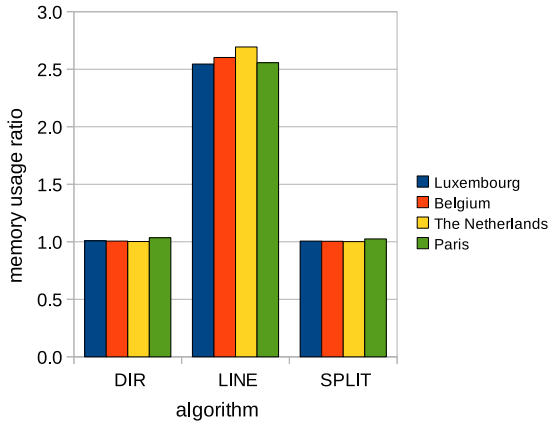
Road network	# nodes	# arcs	# turns	% TP	Size	Dijkstra
Luxembourg	39 883	89 594	226 840	0.14 %	4.85 MB	12.03 ms
Belgium	564 477	1 300 765	3 403 729	0.10 %	69.18 MB	389.62 ms
The Netherlands	1 017 242	2 407 244	6 496 909	0.04 %	128.16 MB	783.35 ms
Paris	313 536	705 588	1 809 714	0.57 %	38.04 MB	151.00 ms

Table 3.1: NAVTEQ real-world road network characteristics. The table shows the size and amount of turn prohibitions (% TP) for every road network. The column Dijkstra represents the average time in milliseconds for a shortest path calculation using the algorithm of Dijkstra.

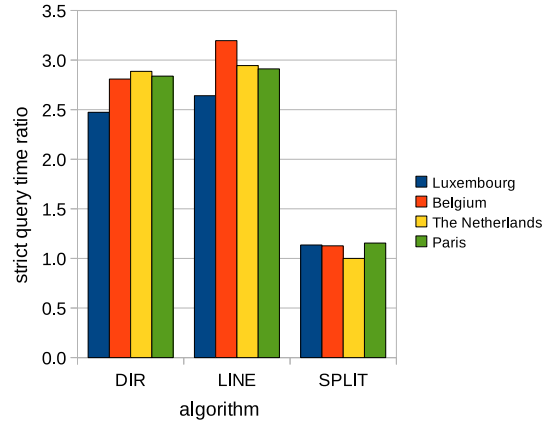
that the running times measured for the evaluated methods are a multiple of this, a running time smaller than 1 millisecond is definitely insignificant. However, there is one exception. The average running time for creating virtual nodes in the line graph method (without lookup table) is around 17 milliseconds. While the lookup table does provide some speedup here, it is rather small compared to the large amount of extra required memory and is generally not worth it unless gaining even a small amount of time is very critical.

3.7 Performance on real-world road networks with turn prohibitions by NAVTEQ

The NAVTEQ road networks, as described in Section 2.4, are very interesting since they contain information on turn prohibitions. This makes it possible to test the algorithm on road networks with real-world turn prohibitions. We had to limit ourselves to turn prohibitions only since, to the best of our knowledge, no datasets with real-life turn costs are available at this moment. Additional information on the NAVTEQ road networks, such as the number of turns and forbidden turns, can be seen in Table 3.1. As can be seen in the table, these road networks have very low percentages of turn prohibitions, something which can be expected in a realistic road net-



(a) **NAVTEQ road networks with real-world turn prohibitions: memory usage ratio**



(b) **NAVTEQ road networks with real-world turn prohibitions: strict query time ratio**

Figure 3.13: Results for the NAVTEQ road networks with real-world turn prohibitions. Memory usage ratio (a) compared to the original graph without turn restrictions and strict query time ratio (b) compared to the average time needed for running the algorithm of Dijkstra on the original graph. The absolute numbers for the original graphs can be found in Table 3.1.

work. Figure 3.13 shows the memory usage ratio and strict query time ratio for the four NAVTEQ networks. Only results for implementations without a lookup table are shown. The results clearly confirm that node splitting is the fastest method, and is also very memory-efficient. Furthermore, given the low percentages of turn prohibitions, the ratios for both memory and running times are very similar to the ratios obtained in the experiments in Section 3.6. It should also be noted that the results show absolutely no significant difference between NAVTEQ_PARIS, an urban road network, and the road networks representing entire countries.

3.8 Performance on SPGRID networks

Guttiérrez and Medaglia [36] also present experimental results in their work. In this section we will show that our results are similar. Guttiérrez and Medaglia have performed their experiments on SPGRID networks. The nodes of such a network are structured as a rectangular grid. The nodes on the rows are connected from left to right in a non-cyclic manner, while the nodes on columns are connected from top to bottom and from bottom to top in a cyclic manner. One additional node is connected to every node in the first column and acts as the start node. The authors chose these networks because of the resemblance to Manhattan-style urban road networks.
We have run experiments on these SPGRID networks as well. Just like Guttiérrez and Medaglia, we have used the SPGRID generator to generate 15 random test networks. This generator is described by Cherkassky et al. [21] and is available for download on the internet [3]. All of the following properties are exactly the same as in Guttiérrez and Medaglia:

- The grid sizes are 128x128, 256x256 and 512x512.
- Five instances of each size were generated.
- Arc weights were chosen randomly between 1 and 10 000.

SPGRID size	Avg. memory DIR (MB)	Avg. memory LINE (MB)	Improvement (vs. LINE)
128x128	2.4	5.9	60.2%
256x256	9.4	23.1	59.2%
512x512	41.0	92.5	55.6%

(a) Gutiérrez and Medaglia

SPGRID size	Avg. memory DIR (MB)	Avg. memory LINE (MB)	Avg. memory LINE* (MB)	Improvement (vs. LINE)	Improvement (vs. LINE*)
128x128	5.1	6.5	12.7	22.4%	60.1%
256x256	20.3	25.5	50.5	20.3%	59.7%
512x512	81.2	104.0	203.1	22.0%	60.0%

(b) Our implementation

Table 3.2: Comparison of the results for memory usage by Guttiérrez and Medaglia [36] and our results.The improvement for DIR is calculated against LINE (and against LINE* in our results).

- For each of the networks, 20% of the turns were randomly chosen and set as prohibited.

The results are presented in tables instead of charts, in order to resemble the tables presented by Guttiérrez and Medaglia.

3.8.1 Memory

Guttiérrez and Medaglia have calculated the memory usage for DIR and LINE, averaged for the different sizes of SPGRID networks. They also show the relative improvement of DIR compared to LINE. Their results are shown in Table 3.2a. This table was taken more or less literally from their paper. The authors find an improvement in memory usage of about 60%. Table 3.2b shows our results for this same experiment. However, we have included both LINE and LINE*. For LINE we find an improvement of about 21%. However, for LINE* we find a very similar improvement

SPGRID size	Average time DIR (ms)	Average time LINE (ms)	Improvement (vs. LINE)
128x128	38.50	44.80	14.06%
256x256	186.50	216.20	13.74%
512x512	998.50	1010.50	1.19%

(a) Gutiérrez and Medaglia

SPGRID size	Average time DIR (ms)	Average time LINE (ms)	Improvement (vs. LINE)
128x128	10.33	10.68	3.20%
256x256	50.22	46.51	-8.00%
512x512	300.75	306.23	1.79%

(b) Our implementation

Table 3.3: Comparison of the results for strict query time by Guttiérrez and Medaglia [36] and our results. The improvement for DIR is calculated against LINE.

of about 60%. This is also confirmed by our results in Section 3.6. For memory usage we had found a ratio of 1.9 for DIR, 2.3 for LINE and 5.1 for LINE*. This leads to an improvement of $1-1.9/2.3 = 17.4\%$ compared to LINE and $1-1.9/5.1 = 62.7\%$ compared to LINE*, which is again around 60%. This improvement is so similar that it leads us to believe that they too store some kind of lookup table or other additional data which can be helpful for the algorithm. However, we have shown that this lookup table does not provide a significant advantage.

3.8.2 Strict query time

For the time measurements we have determined the strict query time for 10 randomly selected queries for every network and averaged the results over the different sizes of SPGRID networks. Table 3.3a shows the results by Guttiérrez and Medaglia, which were taken from their paper. Our results

are shown in Table 3.3b. They seem less consistent than the results for memory usage, since sometimes the direct method is faster and sometimes the line graph method is faster. The results by Guttiérrez and Medaglia seem to have a similar issue, since the improvement is around 14% for the smaller graphs and around 1% for the larger graphs. No explanation for this is given in the paper. However, we can not neglect the fact that for the 512x512 graphs, our results do confirm their results. We obtain an improvement of 1.79% while Gutiérrez and Medaglia find an improvement of 1.2%. We think it is reasonable to assume that for small graphs the results are more dependent on the implementation, especially the 128x128 graphs which have only 16 384 nodes. For larger graphs we believe that our results do confirm the results in [36]. This is also confirmed by the fact that in Section 3.6 we had found a strict query time ratio of 2.5 for DIR and 2.6 for LINE. This leads to an improvement of $1 - 2.5/2.6 = 3.8\%$, which is definitely in the same order of magnitude as the results described above.

3.9 Guideline for real-world applications

The results can be summarized in the following guideline for real-world applications, which can also be seen in Figure 3.14. For road networks with turn costs, use the direct method if no more than 25% of the turns have a non-zero cost, and use the line graph method otherwise. For road networks with forbidden turns, use the node splitting method if less than 5% of the turns are forbidden, which is generally the case. In exceptional cases where 5% of the turns or more are forbidden, use the direct method. Using a lookup table with additional information is generally not worth the extra memory since the speedup is rather insignificant.

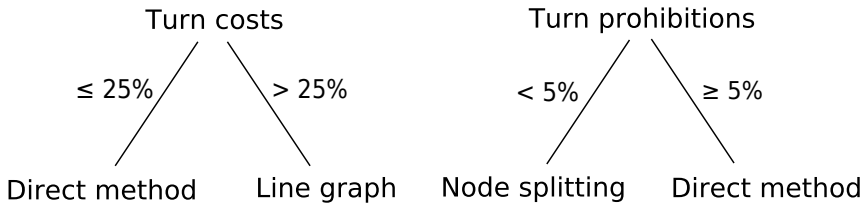


Figure 3.14: Guideline for choosing the right algorithm for a particular dataset.

4

k shortest paths

4.1 Introduction

While there are many algorithms for the shortest path problem, in some cases a ranking of several alternatives for the shortest path is desired. This problem is called the k shortest paths problem, where k is the number of paths to be calculated. Several applications of the k shortest paths problem are listed by Eppstein [26], who mentions finite state machines, biological sequence alignment and the evaluation of automatic translation systems. Also, several authors (e.g. Eppstein [26], Martins et al. [48] and Bernstein [16]) describe how k shortest paths algorithms can be used for solving problems with additional constraints. A k shortest paths algorithm can be used to generate a ranking of shortest paths, from which the path(s) that best satisfy the other criteria are selected. Kuby et al. [44] present a method for finding alternative routes based on a k shortest paths algo-

rithm. However, we will show in the next chapter that other methods are better suited for the specific problem of finding alternative routes. Several exact algorithms for the k shortest paths problem exist, e.g. Eppstein [26], Yen [70] and Hershberger et al. [38]. However, they are very time-consuming and in an interactive application, users expect the results to be shown very fast. This motivates looking for a method for the k shortest paths problem which is faster than the exact algorithms, but which does not aim to find an exact solution. This means that some paths may be missed, causing other (slightly) longer paths to climb up in the ranking. However, in many applications it may not be absolutely necessary to find every path in the ranking, especially in applications where only a small subset of the paths is actually used in the final result. Roddity [55] and Bernstein [16] give approximation algorithms with a theoretical performance guarantee, but until now little research has been done on practical heuristics for the k shortest paths problem.

In this chapter we present a new heuristic approach. By precalculating a backward shortest path tree towards the target node, we can avoid having to perform the many shortest path calculations that typically are part of k shortest path algorithms. Even though there is no guarantee that the exact k shortest paths are found, our results show that the heuristic finds a majority of the paths, with only a slight increase in path weight. Furthermore, the heuristic is much faster than any of the exact algorithms, typically several hundreds of times faster, sometimes even thousands of times faster.

This chapter is organised as follows. In Section 4.2 we give some preliminary background of the k shortest paths problem and a literature overview of existing algorithms. Section 4.3 describes the general principle behind deviation path algorithms, on which our heuristic is based, as well as the algorithm of Yen [70]. In Section 4.4 we present the ideas behind our heuristic and its efficient implementation, while Section 4.5 describes how the heuristic can be extended into an exact k shortest paths algorithm. An evaluation of our heuristic and exact algorithm tested on real-world road networks is given in Section 4.6.

4.2 The k shortest paths problem: an overview

4.2.1 General definitions

For a given integer $k \geq 1$, the k shortest paths problem is intended to determine successively the shortest path, the second shortest path, ..., until the k-th shortest path between a given pair of nodes.

4.2.2 k shortest *simple* vs. *non-simple* paths

There are two variants of the k shortest paths problem. In the first variant, the k shortest paths are not allowed to contain cycles, and this is called the k shortest simple paths problem. The second variant, which allows cycles, is called the k shortest non-simple paths problem. Figure 4.1 shows the difference between both versions of the problem. Naturally the shortest path from node s to node t is the same simple path for both algorithms, as can be seen at the top. However, the second simple shortest path (left) has a weight of 9 (left), which is higher than the second non-simple shortest path with a weight of 6 (right). This example clearly shows that results can be very different for both variants of the problem.

k shortest non-simple paths

For the k shortest non-simple paths problem, an influential algorithm was developed by Eppstein [26]. Hoffman and Pavley [40] also describe a method for the k shortest non-simple paths problem. The algorithm is based on looking up paths in a precomputed shortest path tree rather than performing many time-consuming shortest path calculations. Hoffman and Pavley clearly state that their method allows paths to have loops. They briefly mention that the method can be adapted for generating loopless paths, but give no details. However, Brander and Sinclair [18] give an adaptation of Hoffman and Pavley's algorithm for generating loopless paths. They use

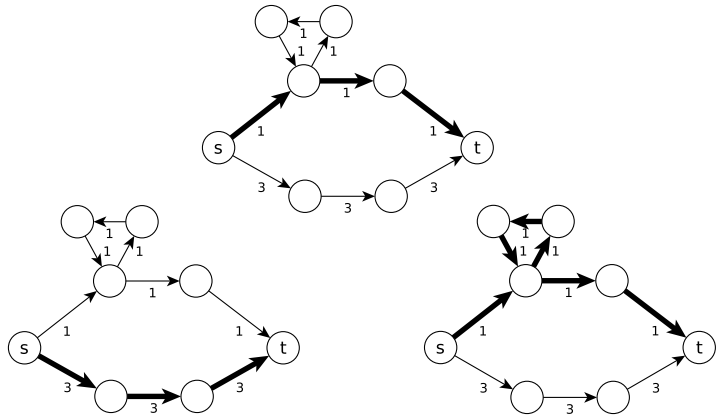


Figure 4.1: Shortest path from node s to node t (top), second simple shortest path (left) and second non-simple shortest path (right).

Hoffman and Pavley's algorithm to generate the complete ranking of paths, containing both loopless paths and paths with loops. All paths with loops need to be stored as well since they may still result in a loopless path later on. The algorithm does not stop when k paths are generated, but continues until k of the generated paths are loopless. Obviously, it can be expected that in many cases, the algorithm needs to generate a very large amount of paths until k loopless paths are found. We have performed an experiment to verify this. For different graphs and different values of k, one hundred random queries were performed and the number of paths which are stored in memory was determined. The minimum, average and maximum number for different values of k in CZE_MAX is shown in Table 4.1. The results clearly confirm this expectation. In the worst case more than 1.7 million paths are stored only to calculate the 100 shortest loopless paths. Furthermore, memory quickly fills up, sometimes causing the algorithm to run out of memory before it can finish. Table 4.2 shows how often this happens. For $k = 1\,000$ this happens very often and for $k = 10\,000$ the algorithm rarely actually finishes, which is unacceptable.

	$k = 100$	$k = 1\,000$	$k = 10\,000$
Minimum	2 484	6 350	524 711
Average	122 436	454 167	1 022 323
Maximum	1 769 552	1 344 353	1 414 923

Table 4.1: Number of paths stored in memory by the Hoffman-Pavley algorithm to calculate k *simple* shortest paths in CZE_MAX. Failed runs due to insufficient memory are not included in this table. The results are similar for other graphs.

Graph	$k = 100$	$k = 1\,000$	$k = 10\,000$
CZE_MAX	5%	43%	97%
IRL_MAX	4%	28%	97%
LUX_MAX	2%	47%	99%
PRT_MAX	0%	3%	85%
BEL_MAX	5%	54%	100%

Table 4.2: Percentages of Hoffman-Pavley runs which fail due to insufficient memory. The experiments were run on a machine with 2 GB RAM, 1 800 MB of which was allocated to the Java Virtual Machine.

Since cycles are usually not relevant for path finding in road networks, we focus on the k shortest *simple* paths problem. Since the Hoffman-Pavley algorithm is in fact a k shortest *non-simple* paths problem, and its adaptation is not suitable for large values of k, we do not consider this algorithm further. However, our heuristic does use the idea of precomputing a backward shortest path tree, like the Hoffman-Pavley algorithm.

k shortest simple paths

Unfortunately, the k shortest simple paths problem is computationally harder than the non-simple variant. Most known algorithms for the k shortest simple paths problem are based on an algorithm which was originally developed by Yen [70] and simultaneously by Lawler [46]. These algorithms are based on the fact that the i-th shortest path will always deviate at some node from a path previously found. For that reason they are called *deviation path algorithms*. Compared to the algorithm of Dijkstra [25], the complexity of deviation path algorithms is significantly higher, since they typically perform many single-source shortest path computations in order to compute the deviations.

Yen's algorithm essentially performs $\mathcal{O}(n)$ shortest path computations for each of the k output paths. Since the algorithm of Dijkstra can be implemented in $\mathcal{O}(m + n \log n)$ time, the algorithm of Yen can be implemented in $\mathcal{O}(nk(m + n \log n))$ worst-case time, or simply $\mathcal{O}(k\, n^2 \log n)$ worst-case time, when we assume $m = \mathcal{O}(n)$. Several improvements to Yen's algorithm have been proposed and implemented, often algorithms which have the same worst-case bound but perform well in practice.

A possible variant of Yen's algorithm, which is suggested by Martins et al. [48], updates the shortest path tree instead of calculating every shortest path from scratch. Another variant is introduced by Hershberger at al. [38] and performs k invocations of the replacement paths algorithm by Hershberger and Suri [39]. The authors claim that for GIS map data with about 5 000 nodes and 12 000 arcs, their algorithm is 4 to 8 times faster than Yen's algorithm.

Algorithm 4.1 Deviation path algorithms: general principle.

Require: graph G, number of shortest paths k, start s, target t
Ensure: sorted collection L of k shortest paths
1: $P \leftarrow$ calculate shortest path from s to t
2: add P to collection C of candidate paths
3: **for** i **from** 1 **to** k **do**
4: $P \leftarrow$ shortest path in C
5: remove P from C
6: add P to L
7: *calculate deviations of P and add them to C* {algorithms differ here, see Algorithms 4.2, 4.3 and 4.6}
8: **end for**

Approximation algorithms for the k shortest paths problem have been studied only recently. Such approximation algorithms typically try to prove a theoretical upper bound for the performance of their heuristic. An important performance measure in the case of k shortest paths is the ratio of the weight of the i-th path computed by an approximation algorithm over the weight of the exact i-th path; this is called the *stretch* of the path. The first approximation algorithm was published by Roddity [55]; it has a stretch of 3/2. Recently a $(1 + \epsilon)$ approximation algorithm was proposed by Bernstein [16].

4.3 Deviation path algorithms

4.3.1 General principle

The heuristic we present is based on the algorithm of Yen, and both can be classified as *deviation path algorithms*. Deviation path algorithms for k shortest simple paths are all based on the same principle, which is shown in pseudocode in Algorithm 4.1.

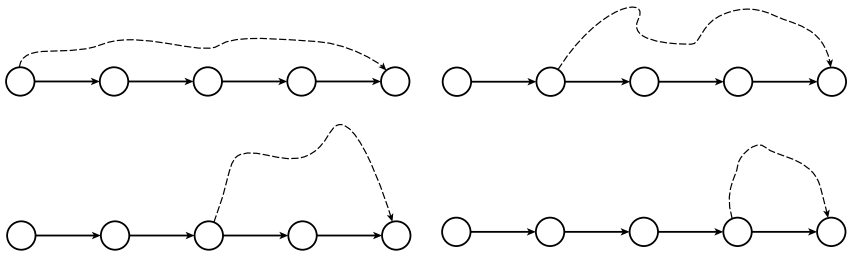


Figure 4.2: All possible deviations from a path with 5 nodes.

Two collections of paths are stored, a *collection* C of candidate paths and a *sorted collection* L containing the i ($< k$) paths found so far. Initially the collection C contains the shortest path from the start node s to the target node t, which can be calculated using any shortest path algorithm, such as Dijkstra's algorithm [25]. In every iteration, the shortest path P in C is fetched and removed from C and it is added to L. Next, deviations of P are calculated and added to C as new candidate paths.

For a given path P containing ℓ nodes, there are only $\ell - 1$ nodes where it is possible to deviate from this path. It is not useful to deviate from the path in the final node. Figure 4.2 shows all possibilities for a path of length 5. Without loss of generality, a path can also coincide with some other path, then deviate from it, and then coincide again. The various deviation path algorithms differ in their way of calculating deviations (Line 7 of Algorithm 4.1). It should be noted that it is possible that less than k paths between the given nodes exist. For readability of the algorithm, this case was omitted from the pseudocode in Algorithm 4.1, but of course it should be dealt with in the implementation.

4.3.2 Yen's algorithm

Yen's algorithm is an example of a deviation path algorithm and calculates deviations in the following way. The so-called *deviation node* v_d of a path P

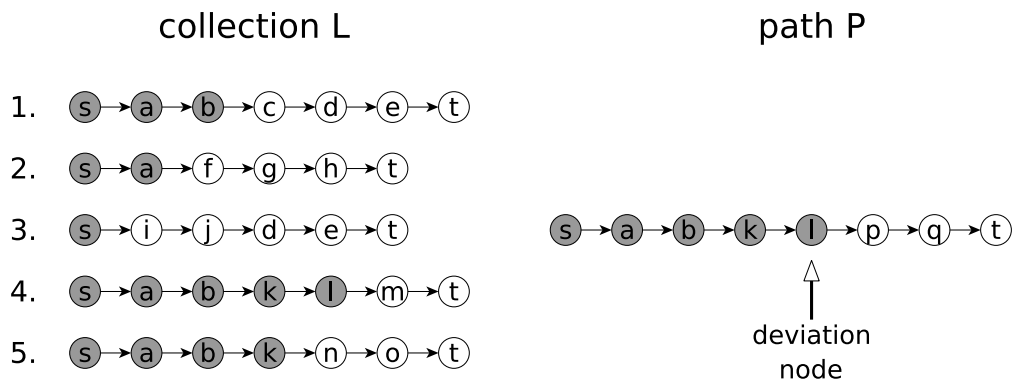


Figure 4.3: Deviation node for path P. Collection L contains the 5 paths from s to t found so far. Subpaths coinciding with P are marked in grey. Path 4 in L has the longest subpath coinciding with P. Therefore, ℓ is the deviation node of P. There is only one deviation arc for P: the arc (ℓ, m).

from s to t is the last node in the path for which a path exists in L which completely coincides from s up to v_d. In other words, v_d is the first node from which P deviates from all other paths in L. Furthermore, for every such coinciding path in L, there is one arc from v_d to its successor in this path. The set of such arcs is called the set of *deviation arcs* for P. An example is given in Figure 4.3.

In order to calculate deviations from P, Yen's algorithm removes every arc (v_i, v_{i+1}) between the deviation node v_d and the last node t one by one. All the nodes preceding v_i in P are also removed. A new path Q from v_i to t is then calculated using an arbitrary shortest path algorithm. By concatenating Q and the $s - v_i$ subpath of P, i.e. $P[s..v_i]$, a new $s - t$ path is created and added to C. It should be noted that the resulting path does not contain cycles, since all nodes preceding v_i in P were removed from the graph. All nodes and arcs are restored to the graph after the entire procedure is finished. As mentioned before, the algorithm skips the arcs between

Algorithm 4.2 Algorithm for calculating deviations in Yen's algorithm.

Require: set of candidate paths C, path $P = (s = v_1, v_2, ..., v_l = t)$ fetched from C

Ensure: deviations from P added to C

1: $v_d \leftarrow$ deviation node of P
2: $E_d \leftarrow$ deviation arcs of P
3: **for** i **from** 1 **to** $d-1$ **do**
4: remove v_i from the graph
5: **end for**
6: remove all arcs in E_d from the graph
7: **for** i **from** d **to** l **do**
8: remove (v_i, v_{i+1}) from the graph
9: $Q \leftarrow$ calculate shortest path from v_i to t
10: $P' \leftarrow P[v_1..v_i] + Q$
11: add P' to C
12: remove v_i from the graph
13: **end for**
14: restore graph

the first node and the deviation node. This is because the $v_1 - v_d$ subpath of P coincides with some other path already in L, so this calculation has already been done for these arcs. The details in Algorithm 4.2 should make this clear. It should also be noted that, in an efficient implementation of the algorithm, nodes and arcs are usually not really removed from the graph, but rather marked as forbidden, thus allowing a fast reconstruction of the original graph.

4.4 Heuristic for calculating deviations

4.4.1 Method

The heuristic we propose is a deviation path algorithm based on the algorithm of Yen. The pseudocode for our heuristic is shown in Algorithm 4.3. While the algorithm of Yen performs many shortest path computations, the heuristic is a lot faster because it uses precomputed information instead. The algorithm starts by calculating a backward shortest path tree SPT_{IN} towards the target node t. This calculation can be done by running the algorithm of Dijkstra backwards and is performed only once, regardless of k. The tree SPT_{IN} stores the shortest path weight as well as the shortest path itself from every node to t and can thus be used by the algorithm to retrieve the shortest path from any node x to t very fast. Just like the algorithm of Yen, the heuristic removes every arc (v_i, v_{i+1}) between the deviation node and the target node in a path P and all the nodes preceding v_i in P from the graph in order to calculate deviations. However, while the algorithm of Yen would perform a shortest path calculation from v_i to t, the heuristic will now use a faster method. All outgoing arcs (v_i, x) of v_i can lead to a suitable detour, by concatenating the shortest path from x to t to the arc (v_i, x) and then to $P[v_1..v_i]$. The shortest path from every node x to t can simply be fetched from SPT_{IN}. This eliminates the need to perform shortest path calculations. Figure 4.4 illustrates this principle. However, a problem may occur here because the shortest path from x to t which was fetched from SPT_{IN} may contain nodes and arcs which were removed from the graph after SPT_{IN} was calculated. It is therefore necessary to check for every arc and node in the path if it still exists in the graph. Only if the entire path still exists in the graph, the path can be concatenated and added to C. Otherwise, the path is simply dropped rather than calculating a new shortest path. This is what causes the heuristic to be fast but possibly inexact. Figure 4.5 shows an example where the heuristic goes wrong. Assume that k shortest paths from s to t are sought. The shortest path is (s, a, t). When calculating deviations from this path, the heuristic looks at

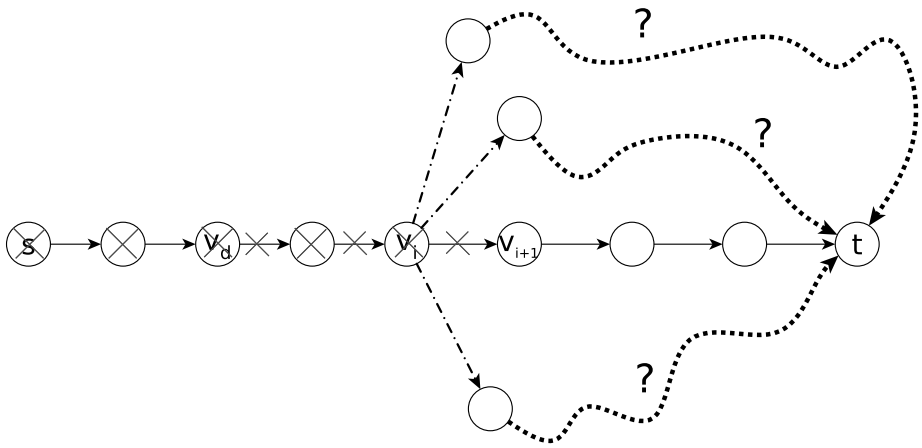


Figure 4.4: How the heuristic works. Solid lines indicate the current path P from s to t. Crosses indicate forbidden nodes and arcs. A detour is necessary from v_i to t. Dashed lines indicate other outgoing arcs from v_i. Dotted lines indicate paths from these neighbours to t, which can be fetched immediately from the backward shortest path tree SPT_{IN}. These paths are not allowed to pass through already forbidden nodes or arcs.

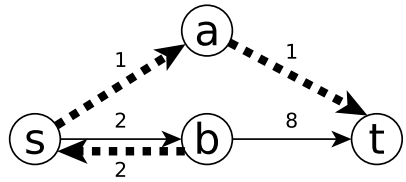


Figure 4.5: Small graph where the heuristic does not find exact results. Arcs in the backward shortest path tree to t are marked as bold dotted arrows. Looking for deviations from the shortest path (s, a, t), the node s is forbidden and the second shortest path (s, b, t) is not found.

Algorithm 4.3 Heuristic for calculating deviations.

Require: graph G, precalculated backward shortest path tree SPT_{IN}, set of candidate paths C, path $P = (s = v_1, v_2, ..., v_l = t)$ fetched from C
Ensure: deviations from P added to C

```
1:  v_d ← deviation node of P
2:  E_d ← deviation arcs of P
3:  for i from 1 to d − 1 do
4:      remove v_i from the graph
5:  end for
6:  remove all arcs in E_d from the graph
7:  for i from d to l do
8:      remove (v_i, v_{i+1}) from the graph
9:      for every outgoing arc (v_i, x) of v_i do
10:         Q ← fetch shortest path from x to t in SPT_IN
11:         if Q exists in G then
12:             P' ← P[v_1..v_i] + (v_i, x) + Q
13:             add P' to C
14:         end if
15:     end for
16:     remove v_i from the graph
17: end for
18: restore graph
```

the other outgoing arc from s: the arc (s, b) and fetches the shortest path from b to t in the backward shortest path tree: (b, s, a, t). Since the node s has been forbidden in the meantime, the path is simply dropped and the actual second shortest path (s, b, t) is never found.

Complexity of the heuristic

As mentioned earlier, road networks are usually sparse so it can be assumed that $m = \mathcal{O}(n)$, where m is the number of arcs and n is the number of nodes in the graph. The check on line 11 in Algorithm 4.3 takes $\mathcal{O}(n)$ time,

since all nodes and arcs in the path are checked. This check is performed $\mathcal{O}(nk)$ times: $\mathcal{O}(n)$ times by the loop on line 7 in Algorithm 4.3 and k times by the loop on line 3 in Algorithm 4.1. This leads to a complexity of $\mathcal{O}(n^2k)$. The lines 1-6 in Algorithm 4.3 have a complexity of $\mathcal{O}(n)$, so they do not add to the complexity, and neither does the initial calculation of the backward shortest path tree, which can be implemented with a complexity of $\mathcal{O}(n \log n)$ in road networks.

4.4.2 Efficient implementation of the heuristic

The heuristic avoids repeated calls of Dijkstra's algorithm, which is typically one of the bottlenecks in most exact k shortest paths algorithms. Most of the operations performed by the heuristic are light-weight operations. It is the computation of deviation nodes (and the associated deviation arcs), which is performed once in every call of the algorithm in Algorithm 4.3, which now becomes the bottleneck. Hence it is important to implement this operation efficiently.

The collection L

In a straightforward implementation, the collection L (see Section 4.3) is represented as a *list*. In order to find the deviation node and arcs, the algorithm then needs to iterate over all the paths in L and for every path over all its nodes, until the path no longer coincides with the current path P.

More efficiently, however, the collection L can be implemented as a *deviation tree*, an example of which is shown in Figure 4.6. The concept of a deviation tree was proposed by Roddity and Zwick [56]. Initially the first path is added entirely to the empty deviation tree D. After that, for every new path P which is added, the longest subpath $P[s..v_d]$ starting in s is sought in D. The remainder of the path P, i.e. $P[v_d..t]$ is then appended to the node representing v_d in D. The node v_d is called the deviation node. Looking up the deviation node can be done efficiently in a similar

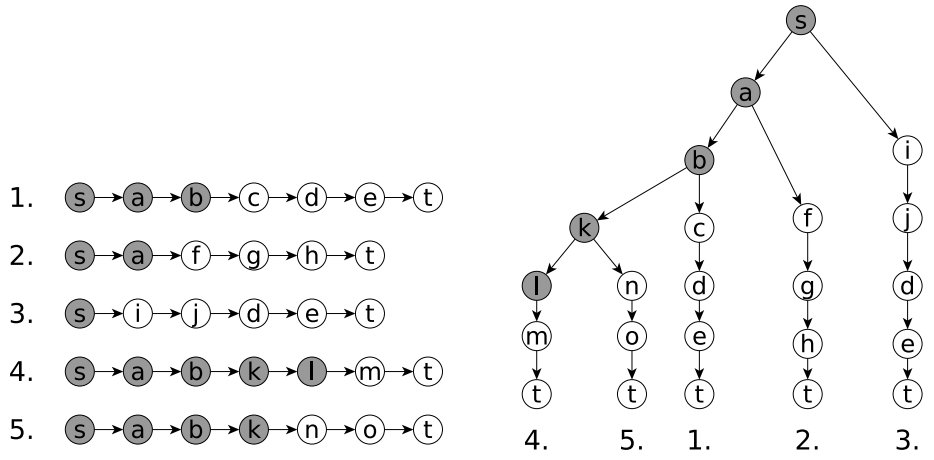


Figure 4.6: Deviation tree (right) for the given collection of paths (left).

way. Algorithms 4.4 and 4.5 show the pseudocode for adding a path to the deviation tree and for looking up a deviation node, respectively.

In order to get an impression of the effect of using the deviation tree versus the straightforward list implementation, we performed 10 random queries for $k = 10\,000$ on CZE_MAX. We measured running times for both the list and the deviation tree implementation. The results are shown in the first chart in Figure 4.7. It is clear that the use of the deviation tree implies a major speedup compared to the list implementation.

It is interesting to note that Yen's algorithm also needs to compute deviation nodes and arcs quite often, so Yen's algorithm can benefit from the efficient computations described in Algorithms 4.4 and 4.5 as well. The second chart in Figure 4.7 shows timing results for the same queries using Yen's algorithm. But here the speedup is hardly noticeable. This can be explained by the fact that for the heuristic the relative amount taken by the operation to find a deviation node is much higher, so the heuristic is much more affected by the efficiency of this operation. In our experiments

Algorithm 4.4 Algorithm for adding a path to a deviation tree.

Require: path P, deviation tree D
Ensure: P is added to D
1: **if** D is empty **then**
2: $D \leftarrow P$
3: **else**
4: $currentTreeNode \leftarrow D.root$
5: $index \leftarrow 2$ {indexing starts at 1}
6: **while** $index \leq P.length$ and
$currentTreeNode$ has child $P.nodeAt(index)$ **do**
7: $currentTreeNode \leftarrow currentTreeNode.child(P.nodeAt(index))$
8: $index \leftarrow index + 1$
9: **end while**
10: append $P[P.nodeAt(pathIndex)..t]$ to $currentTreeNode$
11: **end if**

Algorithm 4.5 Algorithm for finding the deviation node of a path in a deviation tree.

Require: path P, non-empty deviation tree D
Ensure: return deviation node in D for P
1: $currentTreeNode \leftarrow D.root$
2: $index \leftarrow 2$ {indexing starts at 1}
3: **while** $index \leq P.length$ and
$currentTreeNode$ has child $P.nodeAt(index)$ **do**
4: $currentTreeNode \leftarrow currentTreeNode.child(P.nodeAt(index))$
5: $index \leftarrow index + 1$
6: **end while**
7: return $P.nodeAt(index - 1)$

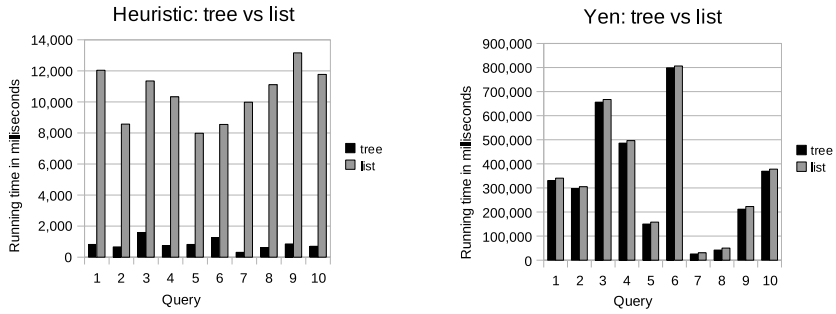


Figure 4.7: Running times for 10 random queries with $k = 10\,000$ on CZE_MAX, both for the heuristic and Yen's algorithm, using the list and deviation tree implementation for calculating deviation nodes.

(described in Section 4.6) the implementation with a deviation tree was used for both the heuristic and the algorithm of Yen.

The collection C

The collection C, which stores the candidate paths, is stored as a queue which sorts the paths in order of ascending path weight. However, the size of this queue is bounded by $k - i$, where i is the number of paths already in L. When the queue has reached its capacity, it no longer accepts paths which are longer than the longest path in the queue. This is because such a path can never be part of the k shortest paths. Candidate paths shorter than the longest path in the queue are accepted into the queue, but in such cases the longest path is removed from the queue. Not only does this save a lot of memory, but it also saves time, since paths which are too long for the queue can immediately be discarded instead of building them and testing if they exist in G (line 11 in Algorithm 4.3).

4.5 Exact calculation of shortest paths

A promising feature of the heuristic is the possibility to enhance it to an efficient exact algorithm. The pseudocode for this algorithm is shown in Algorithm 4.6. Just like the heuristic, the exact variant checks whether a generated path still exists in the modified graph (line 11 in Algorithm 4.3). If not, the exact algorithm calculates a new shortest path (line 14 in Algorithm 4.3), rather than simply dropping the path as the heuristic would do. Obviously this exact algorithm is slowed down by performing more shortest path calculations. But it is interesting to see how this algorithm performs with respect to the other exact algorithms. Another difference is that the exact algorithm only tries one alternative path (with smallest weight) from v_i to t while the heuristic tries a path for all of v_i 's outgoing arcs. We will now prove that this algorithm does indeed return the exact k shortest paths and present experimental results further in this chapter.

Lemma 4.1. *The described algorithm calculates an exact set of k shortest paths.*

Proof. It is known that Yen's algorithm calculates an exact set of k shortest paths. Yen's algorithm and this algorithm only differ in lines 9-10 (Algorithm 4.2) and lines 9-16 (Algorithm 4.6) respectively. All other lines are identical in both algorithms. The given lines operate on the exact same graph, since the graph is only modified in the identical lines. Lines 9-10 in Yen's algorithm simply calculate a shortest path from v_i to t and append it to the $v_1 - v_i$ subpath of P. Our algorithm first finds the shortest path from v_i to t in the precomputed shortest path tree SPT_{IN} (lines 9-10). Then, there are two possible cases:

1. The $v_i - t$ path which was found in SPT_{IN} is still valid in the modified graph. In this case it is definitely the shortest path. The shortest $v_i - t$ path in the modified graph cannot be shorter since no nodes or arcs were added, only removed.

Algorithm 4.6 Exact algorithm for calculating deviations.

Require: graph G, precalculated backward shortest path tree SPT_{IN}, set of candidate paths C, path $P = (s = v_1, v_2, ..., v_l = t)$ fetched from C

Ensure: deviations from P added to C

1: $v_d \leftarrow$ deviation node of P
2: $E_d \leftarrow$ deviation arcs of P
3: **for** i **from** 1 **to** $d-1$ **do**
4: remove v_i from the graph
5: **end for**
6: remove all arcs in E_d from the graph
7: **for** i **from** d **to** l **do**
8: remove (v_i, v_{i+1}) from the graph
9: $x \leftarrow$ find outgoing arc (v_i, x) from v_i such that $w(v_i, x) + T.getWeightToTarget(x)$ is minimized
10: $Q \leftarrow$ fetch shortest path from x to t in SPT_{IN}
11: **if** Q exists in G **then**
12: $P' \leftarrow P[v_1..v_i] + (v_i, x) + Q$
13: **else**
14: $Q \leftarrow$ calculate shortest path from v_i to t
15: $P' \leftarrow P[v_1..v_i] + Q$
16: **end if**
17: add P' to C
18: remove v_i from the graph
19: **end for**
20: restore graph

2. The $v_i - t$ path which was found in SPT_{IN} is no longer valid in the modified graph. In this case a new exact shortest path calculation is performed (line 14).

In both cases the concatenation of the $v_1 - v_i$ subpath of P and the shortest path from v_i to t is assigned to the variable P', just like in Yen's algorithm. Hence, we can conclude that this algorithm obtains the same results as Yen's algorithm and is therefore exact. □

4.6 Results and discussion

4.6.1 Experimental setup

In our experiments, we have compared our heuristic and its exact variant to three existing exact k shortest path algorithms, i.c. the algorithms of Yen [70], Martins et al. [48] and Hershberger et al. [38]. Where applicable, Dijkstra's algorithm was used for shortest path calculations. For each graph we performed 100 random queries for $k = 100$, $k = 1\,000$ and $k = 10\,000$ respectively, thus leading to a total number of 1 500 queries. Both the quality of the results and the time performance were evaluated. In this section we present summary results of all these experiments, as well as detailed results for PRT_MAX.

4.6.2 Quality of the paths found

Unlike the exact algorithms, our heuristic approach does not aim at generating an exact set of k shortest paths. This fact necessitates a comparison of the results of the heuristic with the exact results. When evaluating the quality of the paths generated by the heuristic, two aspects are important. On the one hand, we need to find out how many paths the heuristic "misses". If e.g. the 1 000-th path found by the heuristic is actually the 1006-th path in an exact set of k shortest paths, then the heuristic has

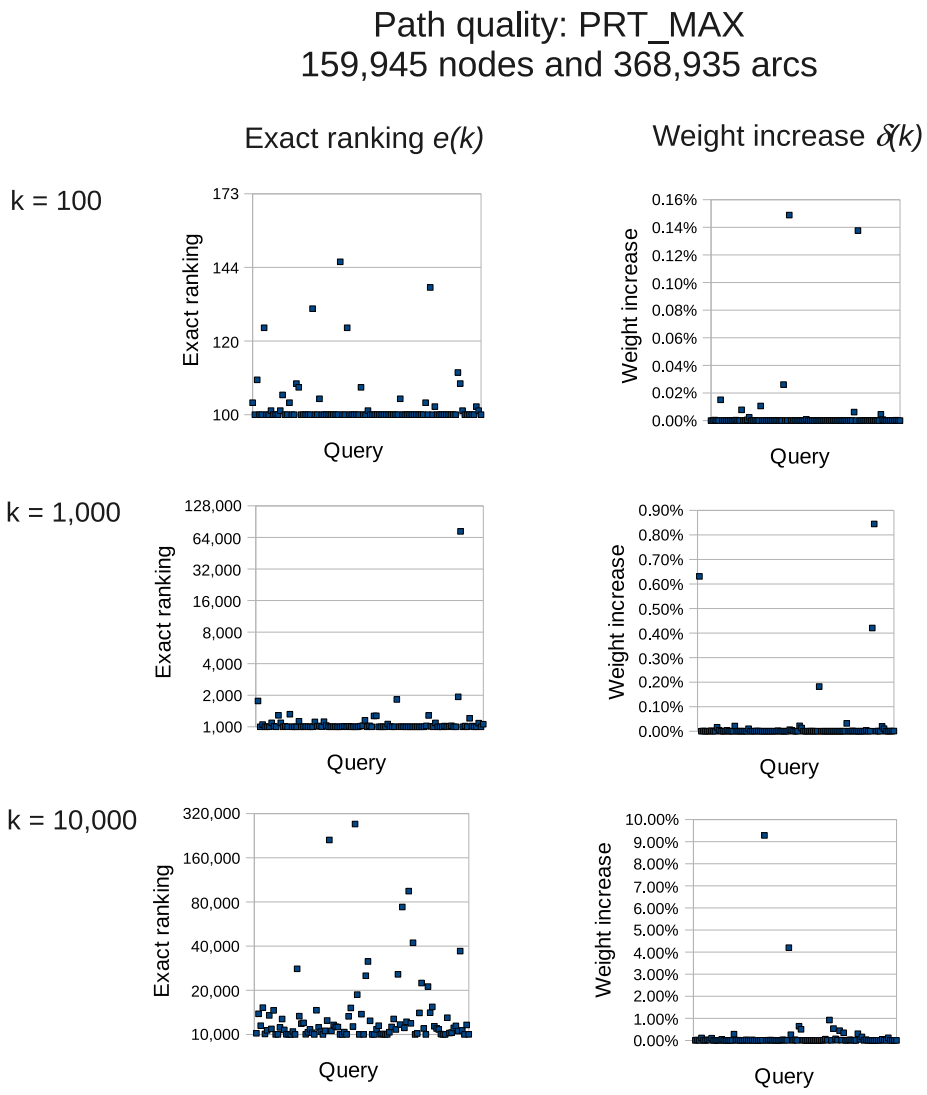


Figure 4.8: Quality of the paths found by the heuristic for PRT_MAX. The exact ranking $e(k)$ and the weight increase $\delta(k)$ are shown for different values of k, each time for 100 random queries.

missed six paths. On the other hand, it is also important how much the path weight increases due to the missed paths. If e.g. the weight of the 1 000-th path found by the heuristic is a travel time of 2 200 seconds, while the real 1 000-th path has a weight of 2 000 seconds, then the weight became 10% worse.

Let $e(k)$ be the ranking of the k-th path found by the heuristic in an exact set of k shortest paths. Let $\delta(k)$ be the percentual weight increase for the k-th path found by the heuristic compared to the real k-th shortest path. It is easy to see that $e(k) \geq k$ and $\delta(k) \geq 0$.

For several values of k, the values of $e(k)$ and $\delta(k)$ were calculated for a large number of random queries by comparing the results of the heuristic to the results of an exact algorithm. The charts in Figure 4.8 show the results for PRT_MAX. As can be seen in the charts, the dots often lie very close to the horizontal axis, which means for $e(k)$ that (almost) no paths were missed and for $\delta(k)$ that the weight increase is very small in many cases. There are however some outliers where quite a few paths are missed, but the number of missed paths remains within acceptable bounds.

Table 4.3 summarises the values of $e(k)$ for all experiments. Each line in the table corresponds to 100 queries for a given k in a given graph.

The column where $e(k) = k$ shows the number of queries where no paths were missed, i.e. the heuristic did find the exact solution. For $k = 100$ this is the case for 47.4% of all queries, but for larger k this number decreases drastically. However, the value of $e(k)$ remains within a small constant factor of k for almost all queries. Even for $k = 10\,000$ on average 93.6% of the queries have $e(k) < 4k$ and on average 97.7% of the queries have $e(k) < 10k$.

The effectiveness of the heuristic becomes even more clear when looking at the values of $\delta(k)$, which are summarised in Table 4.4. Again, each line in the table corresponds to 100 queries for a given k in a given graph and shows the number of queries where $\delta(k) < 1\%$, 2%, 5% or 10%. For example, for IRL_MAX and for $k = 100$, the weight increase is less than 1% for 90 out of 100 queries. For 91, 97 and 98 out of 100 queries, the weight

Graph	k	$e(k) = k$	$e(k) < 2k$	$e(k) < 4k$	$e(k) < 10k$
CZE_MAX	100	35	91	96	99
	1 000	3	83	95	99
	10 000	0	68	89	97
IRL_MAX	100	45	84	94	95
	1 000	7	81	89	95
	10 000	6	77	92	100
LUX_MAX	100	27	87	95	97
	1 000	8	71	84	94
	10 000	4	63	80	94
PRT_MAX	100	76	100	100	100
	1 000	53	99	99	99
	10 000	16	88	95	98
BEL_MAX	100	54	96	100	100
	1 000	18	90	99	100
	10 000	1	85	97	99

Table 4.3: For several graphs and several values of k, each time for 100 random queries, the number of queries where $e(k) = k$, and where $e(k) < 2k, 4k, 10k$ resp.

Graph	k	$\delta(k) < 1\%$	$\delta(k) < 2\%$	$\delta(k) < 5\%$	$\delta(k) < 10\%$
CZE_MAX	100	94	97	98	99
	1 000	93	98	99	99
	10 000	89	94	99	100
IRL_MAX	100	90	91	97	98
	1 000	86	90	95	98
	10 000	90	96	99	100
LUX_MAX	100	92	95	99	99
	1 000	85	89	94	99
	10 000	90	93	99	100
PRT_MAX	100	100	100	100	100
	1 000	100	100	100	100
	10 000	98	98	99	100
BEL_MAX	100	99	100	100	100
	1 000	98	99	100	100
	10 000	97	99	99	100

Table 4.4: For several graphs and several values of k, each time for 100 random queries, the number of queries where $\delta(k) < 1\%, 2\%, 5\%, 10\%$ resp.

increase is less than 2%, 5% and 10%, respectively. It is clear that, even for large k, the weight increase of the k-th path is very small, mostly under 1%. A 95.9% of all queries have $\delta(k) < 2\%$, while 98.5% have $\delta(k) < 5\%$. For routing applications this is very acceptable since a travel time increase of less than 2%, or even 5%, can almost be neglected.

Moreover, we can relate our measure $\delta(k)$ to the theoretical concept of *stretch*, as used in the approximation algorithms of Roddity [55] and Bernstein [16]. We recall that the stretch of the i-th path is defined as the ratio of the weight of the i-th path found by the approximation algorithm over the weight of the exact i-th path. On the one hand, the stretch 3/2 of the approximation algorithm of Roddity [55] corresponds to our $\delta(k) \leq 50\%$, an upper bound which is more than outperformed in almost all cases. On the other hand, the ϵ in the $(1+\epsilon)$ approximation algorithm of Bernstein [16] corresponds to our $\delta(k) \leq \epsilon$. For e.g. $\epsilon = 0.05 = 5\%$, this condition is also satisfied in most cases. Hence, although our heuristic gives no upper bound guarantee for the stretch of the paths, in practice it competes very well with the $(1+\epsilon)$ approximation algorithm of Bernstein [16] and the 3/2 algorithm of Roddity [55].

4.6.3 Time performance

Of course a heuristic approach is only beneficial if it provides a significant speedup in comparison with an exact algorithm. In order to evaluate this, we have compared our heuristic with three existing exact algorithms for 100 random queries and for different values of k. More specifically, for each individual query we also executed the algorithms of Yen [70], Martins et al. [48] and Hershberger et al. [38], and took the smallest running time to compare it with the running time of our heuristic. It should be noted that the algorithm of Hershberger et al. [38] is often implemented using a *threshold*. Therefore the experiments were performed with several smaller and larger thresholds and for each query the best running time was chosen individually. The results for PRT_MAX can be seen in Figure 4.9. For

	$k = 100$		$k = 1\,000$		$k = 10\,000$	
Graph	average	median	average	median	average	median
CZE_MAX	78	77	189	157	560	239
IRL_MAX	95	93	270	241	661	399
LUX_MAX	67	58	141	122	293	182
PRT_MAX	175	180	678	625	731	479
BEL_MAX	145	93	491	303	1 019	389

Table 4.5: Average and median speedup of heuristic versus fastest exact algorithm.

	$k = 100$		$k = 1\,000$		$k = 10\,000$	
Graph	best	worst	best	worst	best	worst
CZE_MAX	222	7	784	6	13 169	19
IRL_MAX	236	6	1 095	10	11 062	11
LUX_MAX	203	4	500	8	3 191	7
PRT_MAX	373	1	2 181	7	8 343	4
BEL_MAX	458	4	1 982	2	5 816	12

Table 4.6: Best and worst speedup of heuristic versus fastest exact algorithm.

readability of the chart, the results are sorted by the running time for the fastest exact algorithm.

The results clearly show that the heuristic is much faster than the exact algorithms. Indeed, for PRT_MAX, the heuristic has a typical running time of a few seconds even for $k = 10\,000$, while for $k = 100$ more than half of the queries for the exact algorithms have running times above 1 minute, for $k = 1\,000$ above 8 minutes, for $k = 10\,000$ above 18 minutes.

For the other networks similar behaviour can be seen. This is summarised in Tables 4.5 and 4.6. The smallest speedup is sometimes very small, only a factor 2 or 4, but as can be seen from the charts in Figure 4.9, these cases correspond to situations where the fastest exact algorithm runs exception-

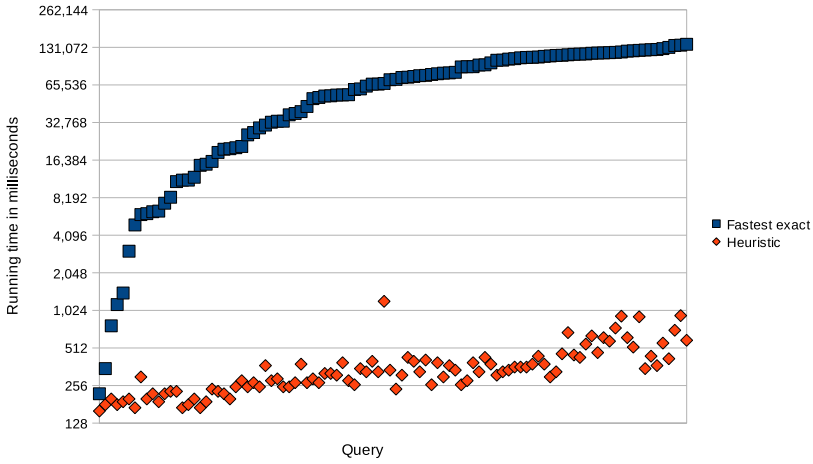
	$k = 100$		$k = 1\,000$		$k = 10\,000$	
Graph	average	median	average	median	average	median
CZE_MAX	1.40	1.38	1.30	1.28	1.19	1.13
IRL_MAX	1.16	1.13	1.15	1.14	1.15	1.09
LUX_MAX	1.25	1.18	1.29	1.14	1.20	1.10
PRT_MAX	1.10	1.08	1.21	1.19	1.52	1.46
BEL_MAX	2.52	1.84	2.94	1.82	2.62	1.90

Table 4.7: Average and median speedup of our exact algorithm versus fastest existing exact algorithm.

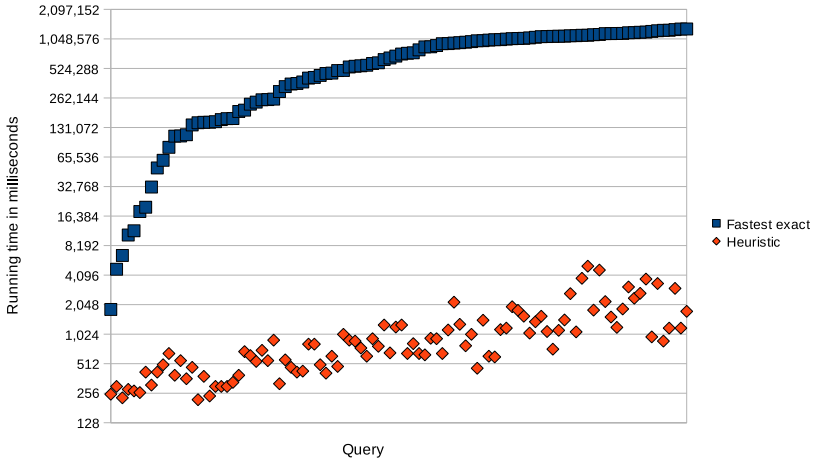
ally fast. On average the heuristic is several hundreds of times faster than the exact algorithms, in the best cases even thousands of times faster. In an interactive routing application, a speedup of this kind can very well be worth giving up the guarantee of exact results, given that the results are still of good quality.

For reasons of completeness, we also include the detailed timing results for the different exact algorithms. These results can be seen in Figure 4.10. The results are sorted by the running time for Yen's algorithm. This is only for the readability of the chart: dots above the dots for Yen's algorithm indicate a larger running time than Yen's algorithm and vice versa. Our experiments show that the algorithm of Martins et al. [48] is hardly ever the best choice, as it is typically the slowest of the 3 existing exact algorithms. Moreover Yen's algorithm in our implementation is often faster than the algorithm of Hershberger et al. [38]. Our own exact algorithm is also included in this chart, and appears to behave quite similar to the running times for Yen's algorithm, though a little faster.

Finally, Table 4.7 gives the average and median speedup of our exact algorithm versus the fastest existing exact algorithm. For all but BEL_MAX (where the speedup is 2.5 to 3), the speedup is about 10% to 20%, which makes our algorithm a valid alternative for the existing exact algorithms.

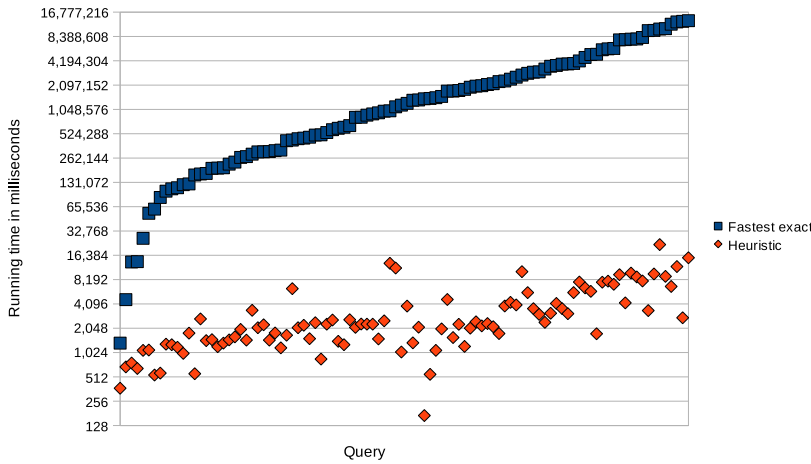


(a) PRT_MAX, $k = 100$



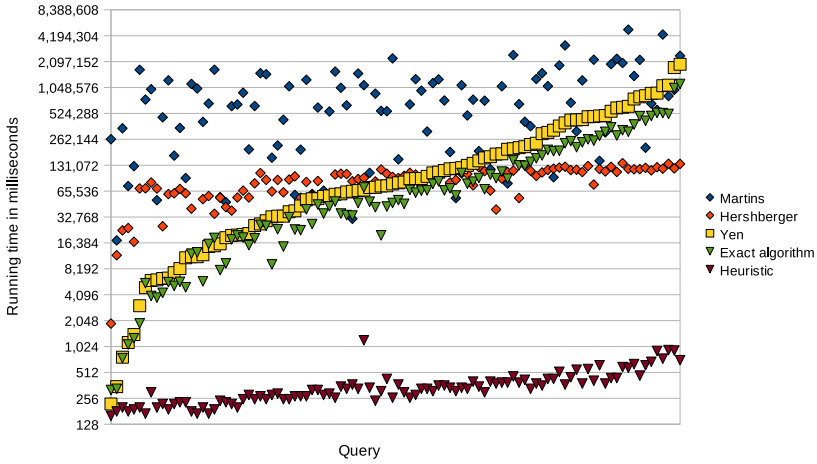
(b) PRT_MAX, $k = 1\,000$

Figure 4.9: Time performance for PRT_MAX.

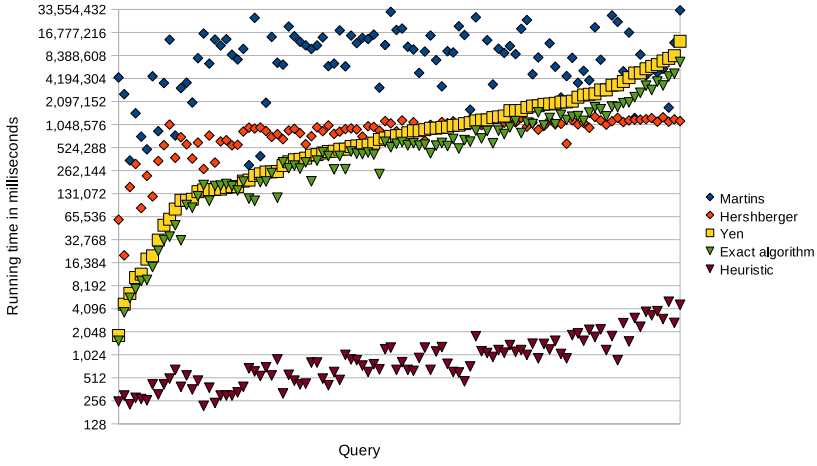


(c) PRT_MAX, $k = 10\,000$

Figure 4.9: Time performance for PRT_MAX. For each k, the heuristic was compared, for 100 random queries, to three existing exact algorithms: Yen [70], Martins et al. [48] and Hershberger et al. [38]. Only the best of the running times for the exact algorithms is shown. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for the fastest exact algorithm. (Continued)

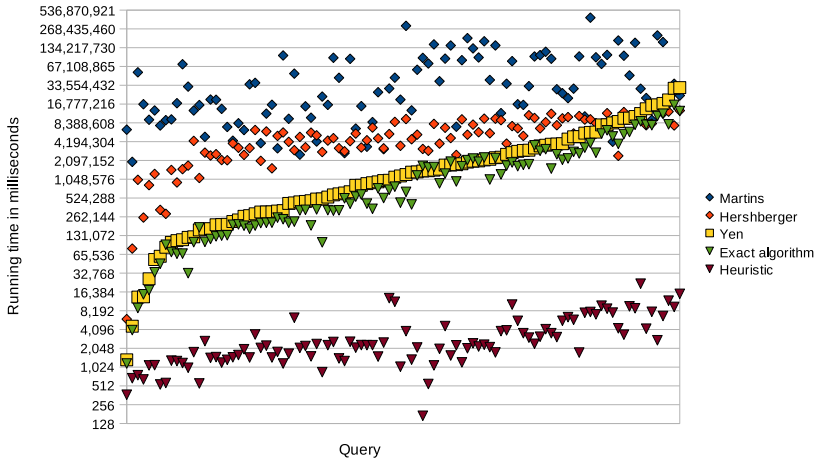


(a) PRT_MAX, $k = 100$



(b) PRT_MAX, $k = 1\,000$

Figure 4.10: Detailed time performance for PRT_MAX



(c) PRT_MAX, $k = 10\,000$

Figure 4.10: Detailed time performance for PRT_MAX. Five algorithms were tested for 100 random queries and different values of k: the three existing exact algorithms of Yen [70], Martins et al. [48] and Hershberger et al. [38], our exact algorithm and our heuristic. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for Yen's algorithm. (Continued)

4.7 Conclusion

A new heuristic has been proposed for the k shortest simple paths problem. Not only does this heuristic generate results of good quality, it is also very fast. The experiments clearly show that significant speedups can be achieved by compromising only slightly on path quality. The new heuristic certainly offers possibilities to serve as a basis for other algorithms and heuristics which make use of a large set of alternative shortest paths. Furthermore, the heuristic can be enhanced to an exact algorithm which is a good alternative to the existing exact algorithms.

5
Dissimilar paths

5.1 Motivation

Users of a route planning application often are interested not only in the shortest path, but also in a few good alternatives. This is because the user may have certain knowledge about a route, such as toll, scenery or probability of traffic jams, which the route planner does not take into account. Also, users may or may not like a particular road segment because of their personal preferences. Presenting more than one possibility gives the user the freedom to choose a route according to his own needs. An example is shown in Figure 5.1.

Naturally, these alternatives are only useful if they are not too much alike, i.e. if they are *dissimilar*. The alternatives should also be relatively short and not contain any detours which do not feel natural to the user. These criteria are discussed in more detail in Section 5.2.

Figure 5.1: Example of alternative routes. The shortest route is shown in red. (Image source: Google Maps)

Having dissimilar alternative routes is also interesting in the context of transporting hazardous materials, as described by Dell'Olmo et al. [24]. When one route fails due to bad weather conditions, one of the alternatives can be chosen. Also, the risk can be spread over different routes instead of just one, so that the risk is equally divided over the population along these routes.

It should be noted that even though the paths should be dissimilar, they should not necessarily be *disjoint*. Different paths often use the same route at the beginning and at the end, e.g. to reach a highway.

This chapter is organised as follows. In Section 5.2 we define what a "good" solution should look like. In Section 5.3 a number of known methods are described. In Section 5.4 we present our original algorithm, which is improved later. In the following sections, we describe our improved algorithm and describe the experiments on which some of our decisions were based. In Section 5.12 the algorithm is evaluated in terms of quality of the results as well as time performance (and the trade-off between the two). Finally, in Section 5.13, we give some concluding remarks and compare our results to results in the literature.

5.2 Evaluation of a solution

A solution S consists of a set of k paths, where k is the requested number of paths. The solution will always consist of the shortest path and $k-1$ alternatives. The quality of a solution is determined by the dissimilarity, local optimality, the weight of the paths and whether the paths contain cycles, each of which are described below. In our algorithm a solution is only feasible if it is locally optimal and contains no cycles, while our algorithm aims to optimize both dissimilarity and path weight.

Dissimilarity

In order to determine a set of dissimilar paths, some formal definition of dissimilarity between paths is needed. Several definitions have been proposed. Some definitions only consider whether paths coincide or not, while others also take into account how far the paths are away from each other geographically. While definitions of dissimilarity are easily interchangeable for most algorithms, it is still important that a suitable definition is chosen. Lombard and Church [47] calculate the area between the path and either the horizontal or vertical axis of the coordinate system for each path. The dissimilarity between two paths is then defined as the absolute difference between their areas.

Akgün et al. [12] propose another definition. They calculate a buffer of a certain width around the paths and use the area of the intersection of two buffer zones for the calculation of the dissimilarity between two paths. This is interesting in the context of transporting hazardous materials since it helps to avoid the selection of two different roads which are very close to each other. This would impose a risk on the same group of people and the two roads would be prone to the same weather conditions. Of course, a drawback of these last two methods is that they assume that the coordinates of the nodes are present. This takes additional memory and the coordinates may not always be present in a data set.

Let $w_s(P_i, P_j)$ be the total weight of the overlapping parts in P_i and P_j, i.e.

$$w_s(P_i, P_j) = \sum_{e \in P_i \cap P_j} w(e)$$

Let $w_n(P_i, P_j)$ be the total weight of the parts of P_i which do not overlap with P_j, i.e.

$$w_n(P_i, P_j) = \sum_{e \in P_i \setminus P_j} w(e)$$

A simple measure for dissimilarity could be $w_n(P_i, P_j)$. However, a drawback of this method is the fact that in most cases $w_n(P_i, P_j) \neq w_n(P_j, P_i)$, so this definition is not symmetric. Akgün et al. [12] describe how this can lead to different results when calculating dissimilar paths, and propose another measure which is symmetric. They define the *similarity* $S(P_i, P_j)$ between two paths P_i and P_j as

$$S(P_i, P_j) = \frac{w_s(P_i, P_j)/w(P_i) + w_s(P_i, P_j)/w(P_j)}{2}$$

The *dissimilarity* $D(P_i, P_j)$ is then defined as

$$D(P_i, P_j) = 1 - S(P_i, P_j)$$

The result is a number between 0 and 1 where 0 indicates that $P_i = P_j$, while 1 indicates that P_i and P_j are completely disjoint. However, we will be using a different dissimilarity measure, which will be explained in Section 5.8.

Local optimality

Even though an alternative path may not be a shortest path, every local instruction must still feel natural to the user. A path containing small

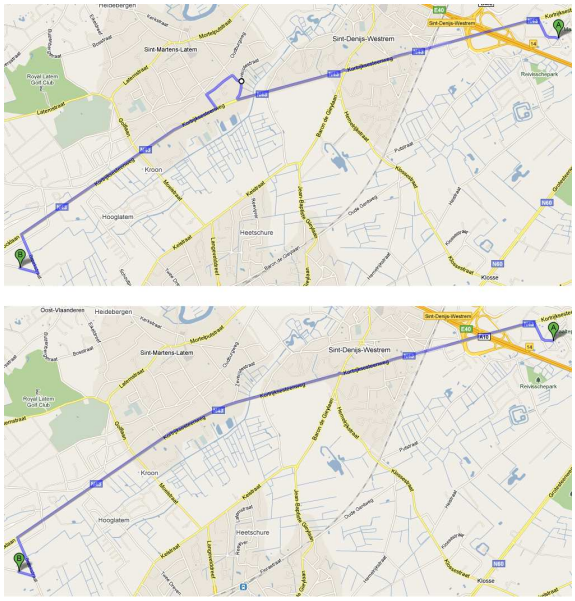
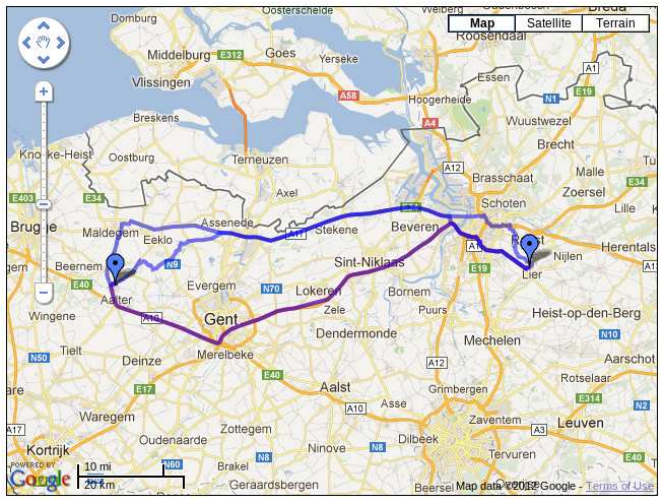


Figure 5.2: Top: Undesired situation where a route is not locally optimal for a certain value of T. The route leaves the main road and then joins it again. Bottom: Improved route which is locally optimal.

detours which make no sense to the user is not a good alternative. It is e.g. not a good idea to leave a major road briefly and then join it again. An example of such a situation is shown in Figure 5.2. This concept was formally introduced as *local optimality* by Abraham et al. [11].

Definition 5.1. *A path P is T-locally optimal if and only if every subpath P' of P with $w(P') \leq T$ is a shortest path.*

Intuitively, this means that every subpath shorter than T cannot contain detours. T is usually chosen as a fraction α of the shortest path weight, e.g. $\alpha = 25\%$. Many of the known methods generate paths which are not



(a) $\alpha = 10\%$

(b) $\alpha = 25\%$

Figure 5.3: Two solutions for the same query in NAVTEQ_BELGIUM with different values for α.

locally optimal or strive towards local optimality without guarantees. However, we believe that the alternatives must be locally optimal for the user to be satisfied, since any illogical detour may be regarded as disturbing by the user. Therefore, our algorithm only generates paths which are locally optimal for a given α. In many of our experiments, we will use $\alpha = 25\%$. This is the value which is also used by Abraham et al. [11] in their experiments and it leads to paths of good quality. To illustrate this, we have generated two solutions for the same query using our algorithm which will be described later. One solution is locally optimal for $\alpha = 10\%$ (but not for $\alpha = 15\%$) and the other solution is locally optimal for $\alpha = 25\%$. Figure 5.3 shows both solutions and gives an idea of the difference between solutions with different values for α. The paths which are locally optimal only for $\alpha = 10\%$ make some undesired detours around Eeklo and Schoten. This is not the case for the paths which are locally optimal for $\alpha = 25\%$.

Weight of the paths

The weight of the alternatives should obviously be within acceptable bounds. It makes no sense to present an alternative which is e.g. twice as long as the shortest path, unless there is no other option, which is very uncommon in practice. Most known methods eliminate all paths with a weight higher than a certain threshold, e.g. 30% longer than the shortest path weight. Abraham et al. [11] even require that every subpath is not more than a certain percentage longer than its corresponding shortest path. However, valuable alternatives may be dismissed by imposing a strict upper bound on the weight of the paths. Therefore, we aim to avoid this and optimize both dissimilarity and path weight.

Cycles

Of course, routes containing cycles are definitely suboptimal and should be avoided at all times. Therefore, our algorithm will eliminate any route

which contains a cycle.

5.3 Known methods

5.3.1 Iterative penalty

One of the first methods for finding dissimilar paths was proposed in 1993 by Johnson et al. [42] and is called the *iterative penalty* method. It starts by finding the shortest path, and then increases the weight of the shortest path by applying a penalty to it. This process is repeated until enough paths are found. While this method is very simple and efficient, it has some drawbacks.
There are many different ways for applying penalties to paths. Penalties can be applied to nodes or arcs, additively or multiplicatively, different values for the penalties can be used, nodes and arcs which are already penalized can be penalized again or not... This needs a lot of finetuning and is not guaranteed to work well on every network. Additive penalties have the disadvantage that they penalize paths consisting of many low-weight arcs more than paths consisting of few high-weight arcs. Also, the same path may be found twice and these duplicates need to be refused. The higher the penalty, the less duplicate paths need to be refused.
Another drawback is the fact that this method can easily enforce unnatural detours. Because of this, the paths are very unlikely to be locally optimal. Bader et al. [13] suggest an interesting approach to address this issue. They penalize not only the path itself, but also the arcs around it (decreasing with the distance to the path). In this way small detours are discouraged. However, a good alternative might be near the path. Another option they suggest is to penalize only the path and its incoming and outgoing arcs. They call this penalty the *rejoin penalty*.
Another issue is that this method provides no real evaluation of the found set of paths, so the paths may be very similar after all. Akgün et al. [12] have performed experiments with the iterative penalty method and found

that the dissimilarity of the paths is rather disappointing. However, they do suggest using this method as a basis for another algorithm, which will be described in Section 5.3.4.

5.3.2 K shortest paths

While k shortest path algorithms (e.g. Yen [70]) have their own applications, as described in Section 4.1, they are not the best choice as a standalone algorithm for dissimilar paths. This is because every small variation on the path can be included in the results as an alternative, but these variations can also be combined endlessly. Because of this, the paths tend to be very much alike and for a large graph a good alternative is probably not within the first 1 000 paths. Figure 5.4 shows the 1 000 shortest paths between two given points. The paths are so much alike that the difference can hardly be seen. Also, the alternatives often have small detours, so the paths are very unlikely to be locally optimal. However, k shortest paths algorithms serve as a basis for certain dissimilar paths algorithms, such as the algorithm by Kuby et al. [44] which is described in the next section. It should be noted that our k shortest paths heuristic presented in Chapter 4 could provide a large speed-up for such algorithms.

5.3.3 Minimax method

Kuby et al. present a method based on k shortest paths algorithms. They call it the *minimax* method. This method is one of the first methods which consider the dissimilarity between *every* pair of paths in a solution and not only between each alternative and the shortest path. A large collection of paths is generated using a k shortest paths algorithm and then a subset of short but dissimilar paths is chosen. For selecting a dissimilar subset of paths, a linear trade-off between the path weight and the dissimilarity is used. They define the *similarity* between two paths P_j and P_i as

$$w_s(P_j, P_i)$$

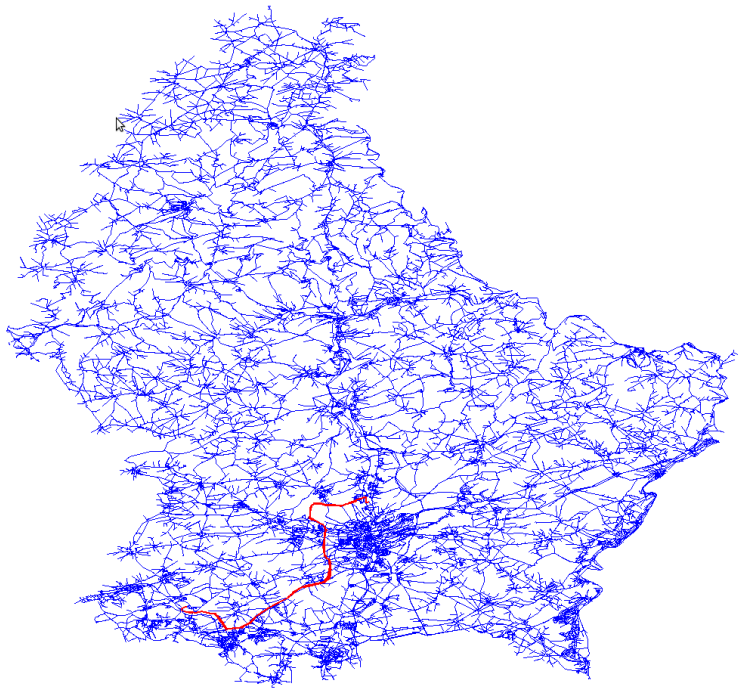


Figure 5.4: Ranking of the 1 000 shortest paths (marked in red) between two points in Luxembourg, calculated using a k shortest paths algorithm. The paths are so similar that there seems to be only one path.

while the weight of the path P_j is given by

$$w_s(P_j, P_i) + w_n(P_j, P_i)$$

Both of these values should be as low as possible, so their sum should be as low as possible too, i.e. their sum must be *minimized*. The authors use a linear trade-off by applying a weight of β to the similarity:

$$\beta \times w_s(P_j, P_i) + w_s(P_j, P_i) + w_n(P_j, P_i)$$

This can also be written as:

$$(1+\beta) \times w_s(P_j, P_i) + w_n(P_j, P_i)$$

Finally, this is scaled by dividing by the weight of the shortest path $w(P_1)$, resulting in the following objective function:

$$\frac{(1+\beta) \times w_s(P_j, P_i) + w_n(P_j, P_i)}{w(P_1)}$$

The shortest path P_1 is always included in the solution. The other paths are selected one by one and added to the solution. Suppose $j-1$ paths have already been chosen and the j-th path needs to be selected. The objective function is calculated for every possible path P_k for every path P_i already in the solution. The path P_j for which the highest value is as low as possible is added to the solution:

$$\underset{k}{\text{minimize}} \left[\max_{i=1,\ldots,j-1} \frac{(1+\beta) \times w_s(P_k, P_i) + w_n(P_k, P_i)}{w(P_1)} \right]$$

The outcome of course depends on β, so some finetuning is needed for the parameter β. Akgün et al. [12] later pointed out that this problem of choosing a small selection of paths is essentially a p-dispersion problem and this method of solving it is essentially the *greedy construction* heuristic as proposed by Erkut [28].

5.3.4 P-dispersion algorithms

Akgün et al. [12] present a method which first generates a large amount of paths, and then selects a small dissimilar subset. For generating a large amount of paths they use either the *iterative penalty* method or a *k shortest paths* algorithm. For the selection of a dissimilar subset, the authors point out that this can be modeled as a *p-dispersion problem*.

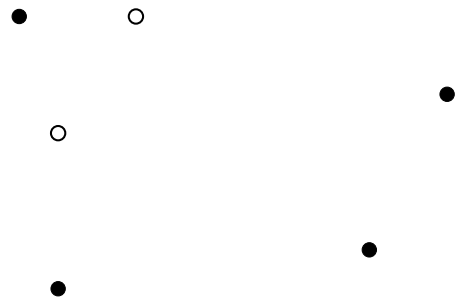


Figure 5.5: Example of a p-dispersion problem where p=4 points must be chosen out of 6 points. The set consisting of the points marked in black has the largest minimum distance between any pair of points.

Definition 5.2. *The p-dispersion problem is the problem of finding a subset of p points out of a set of n points such that the minimum distance between any pair of points in the subset is maximized.*

Figure 5.5 illustrates this principle in a two-dimensional setting. This can be translated to the problem of selecting dissimilar paths by letting the points correspond to paths. The distance between these points is then defined as the dissimilarity between their corresponding paths, e.g. any of the dissimilarity functions described in Section 5.2. Unfortunately, the p-dispersion problem is computationally hard and using an exact algorithm would definitely lead to high running times which are unacceptable in routing applications. However, there are many heuristic methods which find a good solution very fast, but not necessarily the optimal solution. Along with an exact branch-and-bound algorithm, many different heuristics for the p-dispersion problem are presented by Erkut et al. [27, 28]. A first family of heuristics are the construction algorithms, which use a set of points that are already chosen and another set with the points that are not (yet) in the solution. Examples are the *greedy construction* heuristic which starts with an empty set and adds points one by one, and the *greedy*

deletion heuristic which starts with a solution containing *all* points and then deletes points until a solution of the right size is found. Erkut et al. also present a *neighbourhood heuristic* which eliminates all points from the search within a certain radius around points that have already been chosen. Furthermore, an *interchange algorithm* is presented which starts with a random solution and tries to improve it by interchanging points in the solution with points not in the solution. Akgün et al. eventually used a two-phase heuristic described in [27] which constructs an initial solution and then improves it using local search.

The authors conclude that while the version using the iterative penalty method is much faster, it does not always find better results, e.g. in one case the iterative penalty method finds only 20 paths to choose from that are at most 40% longer than the shortest path, while the version using k shortest paths offers a lot more paths to choose from.
It should be noted that this method is a bit dated, since the paper was written more than ten years ago. The algorithm was tested on a graph with only 305 nodes and 854 arcs, while modern computers can easily run routing algorithms on graphs with more than 10 000 or even more than 100 000 nodes. This explains why good results are found when k is only 100 or 200. On larger graphs the first 100 or 200 paths would be very much alike. Also, the authors generate *all* paths up to a certain length. Depending on this length, this may be an enormous amount of paths for larger graphs. Furthermore, since the paths are generated by the iterative penalty method or by a k shortest paths algorithm, they are most likely not locally optimal.

5.3.5 Gateway shortest paths

Lombard and Church [47] introduce the *Gateway Shortest Paths* method which forces the paths through different nodes. While this method is very simple, it fails to consider the dissimilarity between every pair of paths in a solution and only takes the dissimilarity between the alternatives and the shortest path into account. This is of course a major drawback and can

lead to paths that are very similar after all. However, this method was designed only to find alternatives that are dissimilar to the shortest path, not to the other alternatives. This method also generates a lot of duplicate paths and a lot of paths with cycles.

5.3.6 Alternative graph

Bader et al. [13] present a new way for evaluating paths. The solution is stored in an *alternative graph*, which is the union of the shortest path and the alternatives, where arcs may represent subpaths. No constraints are used for the quality of the subpaths, since the authors believe this takes too long to evaluate. Instead, postprocessing is performed on the alternative graph. Arcs representing subpaths with a too high weight are eliminated.

5.3.7 Single via paths

Abraham et al. [11] impose very strict restrictions on candidate paths. The paths must be dissimilar, locally optimal for a given α and the weight of the paths and even their subpaths cannot be greater than $(1 + \epsilon)$ times the weight of the corresponding shortest path. Paths which satisfy these restrictions are called *admissible paths*. The authors present several heuristics which are based on *single via paths*. The *via path* through a node v is the concatenation of the shortest path from s to v and the shortest path from v to t. An advantage of these via paths is that they are guaranteed to be the shortest path through v. The *reach* method [37] is used for pruning to speed up the algorithm. However, while some of the heuristics are fast, the experiments in the paper show that they do not always find a solution.

5.4 A bidirectional heuristic for dissimilar paths

In this section we will describe our initial algorithm for finding dissimilar paths, which will be improved in the following sections. Our first idea

was to generate dissimilar paths first, and then modify the paths to make them locally optimal. This method finds good results that are very similar to what a real-world route planner would find. However, it also imposes some problems. We believe it is worth describing this algorithm briefly since it finds good results and since it was used as a basis for our improved algorithm. The algorithm consists of three phases, all of which will be discussed in the following paragraphs:

1. Generate many paths (with an acceptable weight)
2. Find a subset of dissimilar paths
3. Make the paths locally optimal

5.4.1 Phase 1: Generate many paths

Just as in Akgün et al. [12] and Kuby et al. [44], the algorithm starts by generating a large amount of paths from which a small subset will be selected later. A constant factor $\gamma > 1$ determines the maximum path weight $w_{max} = \gamma \times w(SP)$. Typically, $1 < \gamma \leq 2$. Rather than using a k shortest paths algorithm or an iterative penalty method, we use a faster heuristic which finds paths that are not as similar. The heuristic is based on the bidirectional version of the algorithm of Dijkstra [25], which runs two searches, one forward search from the start node and one backward search from the target node. Both searches take turns in labeling nodes.
Our heuristic is based on this idea. Both searches keep running even after the shortest path is found. Whenever both searches meet, i.e. a node x is labeled by both searches, there is a possibility for generating a new path. First, the weight of the possible path is checked. If this weight is not greater than w_{max}, the path is generated by concatenating the $s - x$ and $x - t$ paths. The new path is added to the collection of generated paths. Figure 5.6 illustrates this idea. The algorithm continues until the desired number of initial paths k is found, or until no more paths can be found. The advantage of this method is that both searches meet at different points

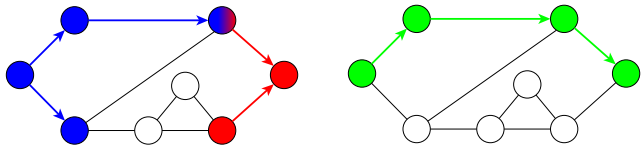


Figure 5.6: Bidirectional heuristic. Left: the forward (blue) and backward (red) shortest path tree meet in the blue/red node. Right: path generated after both searches meet. The algorithm will continue to generate paths in the same way.

in the shortest path trees, so the paths are likely to be dissimilar. The bidirectional algorithm of Dijkstra has a complexity of $\mathcal{O}(m + n \log n)$. Our heuristic adds a term of $\mathcal{O}(n\,k)$ to this, for actually generating the k paths, leading to a complexity of $\mathcal{O}(n\,k + m + n \log n)$, while k shortest paths algorithms have a significantly higher complexity. Furthermore, the searches in both directions are pruned at w_{max} since this branch in the search can never lead to a path with an acceptable weight.

5.4.2 Phase 2: Find a subset of dissimilar paths

The problem of finding a subset of dissimilar paths is modeled as a p-dispersion problem. The Greedy Construction method [28] is used. When applied to paths, this method starts by selecting one random path. Then, $p - 1$ times, the dissimilarity between all non-selected paths is calculated to all of the selected paths. The path with the highest minimum dissimilarity compared to all of the selected paths is chosen, and added to the selection. However, there is a slight adaptation in our algorithm. Rather than choosing the first path in the selection randomly, the shortest path is always chosen as the first path. This is because it makes sense for a routing application to always include the shortest path in the solution. The pseudocode is shown in Algorithm 5.1.

Algorithm 5.1 Greedy construction algorithm for selecting a subset of p dissimilar paths out of a set of n paths (including the shortest path).

Require: set $S = \{SP, P_1, P_2, ..., P_{n-1}\}$
Ensure: set S' containing p dissimilar paths, including SP

```
1: add SP to S'
2: for i from 2 to p do
3:    D_maxmin ← 0
4:    P_candidate ← null
5:    for every path P in S \ S' do
6:       D_min ← +∞
7:       for every path P' in S' do
8:          if D(P, P') < D_min then
9:             D_min ← D(P, P')
10:         end if
11:      end for
12:      if D_min > D_maxmin then
13:         D_maxmin ← D_min
14:         P_candidate ← P
15:      end if
16:   end for
17:   add P_candidate to S'
18: end for
```

Algorithm 5.2 Improve T-local optimality. This heuristic must be repeated iteratively until the path is T-locally optimal.

Require: path $P = (v_1, v_2, ..., v_n)$, parameter T
Ensure: T-local optimality of P is improved heuristically
1: **for** i **from** 1 **to** n **do**
2: $P' \leftarrow (v_i)$
3: $j \leftarrow 0$
4: **while** $w(P') < T$ AND $i + j < n$ **do**
5: $j \leftarrow j + 1$
6: $P' \leftarrow P' + v_{i+j}$
7: **end while**
8: P'' ←calculate shortest path from v_i to v_j
9: **if** $w(P'') < w(P')$ **then**
10: Replace P' by P'' in P
11: $i \leftarrow i + j$ {jump to end of new subpath}
12: **end if**
13: **end for**

5.4.3 Phase 3: Make the paths locally optimal

At this point, a small set of dissimilar paths is found, but the paths are not necessarily locally optimal. Therefore, the goal of this phase is to make the paths locally optimal. Our method for improving the local optimality checks for every node (except those at a distance $< T$ from t) in the path if the smallest subpath starting in this node with weight $\geq T$ is a shortest path by running the algorithm of Dijkstra. If so, the algorithm simply moves on to the next node. If not, this subpath is replaced by the shortest path and the algorithm continues at the endpoint of the new subpath. Figure 5.7 illustrates this idea. The pseudocode for the algorithm is shown in Algorithm 5.2. After this procedure the path is improved but not necessarily locally optimal. Therefore it is repeated until no more changes can be made. It should be noted that this procedure checks and improves local optimality at the same time. This means that, if no more changes can

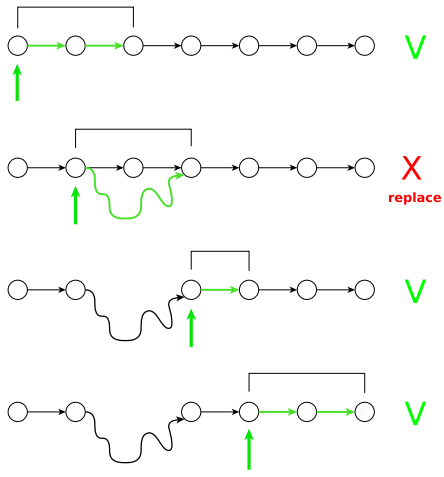


Figure 5.7: How the heuristic for improving T-local optimality works. For every node the smallest subpath starting in this node with weight $\geq T$ is found. If this subpath is a shortest path, it is approved. If not, it is replaced with the shortest path.

be made, the path has become locally optimal.

5.4.4 Shortcomings of this method

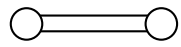
The algorithm finds good results. However, it does have some issues. Phase 3 performs $\mathcal{O}(n)$ Dijkstra runs in every iteration. As can be expected, this takes a lot of time. This needs to be done iteratively, making it even more time-consuming. Also, there is no guarantee on the number of iterations needed. Finally, two dissimilar paths might be transformed into the same path after Phase 3. Furthermore, this method imposes a strict upper bound on the weight of the paths. All of these issues are addressed in our improved algorithm which is presented in the next section.

5.5 Plateaus

Our improved algorithm generates locally optimal paths but does not need the very time-consuming phase where paths are modified to be locally optimal like in the previous algorithm. The need for checking local optimality is significantly reduced since part of the generated paths are guaranteed to be locally optimal. For the other paths, local optimality is only tested when necessary. The algorithm discards paths that are not locally optimal instead of performing the time-consuming routine to make the paths locally optimal. Just like the previous method, the bidirectional version of the algorithm of Dijkstra is used. Instead of creating a path whenever both searches meet, a path is only generated when a node v has been *permanently* labeled (and not just *temporary*) in both directions. In this way, the local optimality can only be violated around v, which facilitates testing for local optimality. This will be explained in more detail in Section 5.9. Taking it one step further, not only nodes that appear in both shortest path trees are considered, but also overlapping parts of both shortest path trees. Such overlapping parts are called *plateaus*.

Definition 5.3. *A plateau Pl for an $s-t$ query is a maximal path which appears in both SPT_{IN} and SPT_{OUT} such that all nodes in Pl have a permanent label in both shortest path trees.*

An early method based on plateaus was described briefly by the company CAMVIT [4] on their website. Also, Abraham et al. [11] prove some properties regarding plateaus. The authors present several heuristics for finding dissimilar paths, some of which are said to incorporate elements of plateaus for efficiency, but only few details are given. In what follows we investigate in detail how plateaus can be used for finding dissimilar paths and we suggest a robust algorithm based on plateaus. Arcs which are part of a plateau will be marked in figures as follows (with or without arrows):



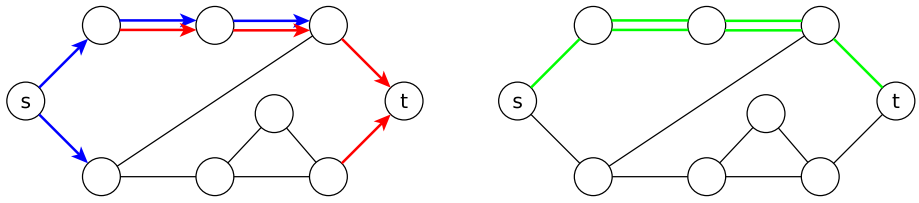


Figure 5.8: (Pruned) SPT's containing one plateau (left) and the resulting path (right). SPT_{OUT} is shown in blue, SPT_{IN} is shown in red.

A path can be created from a $v-w$ plateau Pl by concatenating the shortest $s-v$ path to the plateau and then to the shortest $w-t$ path. Both of these shortest paths are available in the SPT's. An example of a plateau and the resulting path can be seen in Figure 5.8. Whenever an $s-t$ path containing a plateau is mentioned in the remainder of this text, it refers to the shortest $s-t$ path containing this plateau (and not to an $s-t$ path which makes a detour before or after the plateau). We also consider a single node which has a permanent label in both shortest path trees, but which is not part of a larger plateau. This is a special case of a plateau, with weight zero and consisting of one node and no arcs.

5.5.1 Plateaus and local optimality

Abraham et al. [11] prove that any path containing a plateau is T-locally optimal, T being the weight of its plateau.

Property 5.1. *If a path P contains a plateau Pl, then P is $w(Pl)-$locally optimal.*

Therefore, if $w(Pl) \geq T$, P is guaranteed to be T-locally optimal. This is a very interesting property since it eliminates the need for testing local optimality for such paths.

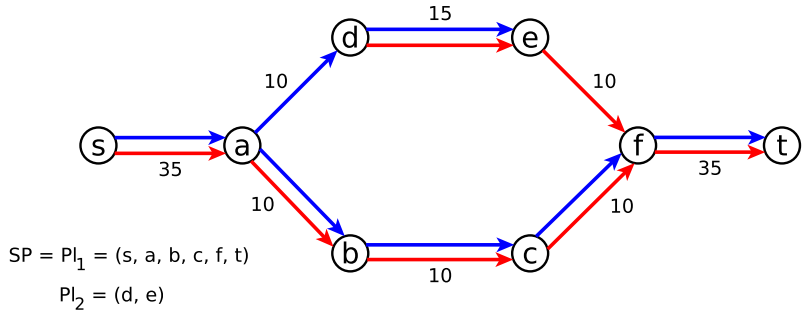


Figure 5.9: The path containing the plateau Pl_2 is T-locally optimal for $T > w(Pl_2)$. SPT_{OUT} is shown in blue, SPT_{IN} is shown in red.

Property 5.2. *A path P containing a plateau Pl can be T-locally optimal for $T > w(Pl)$.*

This means that paths containing a shorter plateau should not automatically be excluded when T-local optimality is required. Figure 5.9 shows an example of such a situation. The plateau Pl_2 has a weight of 15. However, while the corresponding path is definitely 15-locally optimal, it is also 34-locally optimal (but not 35-locally optimal) since every subpath with weight ≤ 34 is a shortest path. Up to now, the only methods which use plateaus, only consider the plateaus Pl such that $w(Pl) \geq T$ such that no local optimality testing is needed. In this work we will show that better solutions, or even a solution at all, can be found by taking the shorter plateaus into account as well.

5.5.2 Plateaus and dissimilarity

Another major advantage of using plateaus, is the fact that paths corresponding to long plateaus are likely to be dissimilar. First of all, plateaus cannot overlap, as is proved in the following lemma:

Lemma 5.1. *For a given $s-t$ query, there is no pair of plateaus Pl_1 and Pl_1 such that Pl_1 and Pl_2 share an $x-z$ subpath.*

Proof. Suppose Pl_1 is a p_1-q_1 plateau and $Pl_2 \neq Pl_1$ is a p_2-q_2 plateau. Suppose Pl_1 and Pl_2 are not disjoint, i.e. they share an $x-z$ subpath. Since $Pl_1 \neq Pl_2$, either p_1 or q_1 is not in Pl_2 or p_2 or q_2 is not in Pl_1. Let's assume without loss of generality that p_1 is not in Pl_2. The other cases can be proved in a similar way. There are two possibilities:

(i) p_1 is not on the shortest path from s to p_2 (see Figure 5.10). Because of the definition of plateaus, both the p_1-x and p_2-x subpaths are on the shortest path from s to x. However, since p_1 is not in Pl_2 and p_1 is not on the shortest path from s to p_2, there must be two shortest paths from s to x in SPT_{OUT}. This is impossible in a SPT, and therefore a contradiction.

(ii) p_1 is on the shortest path from s to p_2 (see Figure 5.11). In this case, the p_1-p_2 subpath is adjacent to Pl_2 and since it is part of a plateau, appears in both SPT_{IN} and SPT_{OUT}. This means that Pl_2 cannot exist in the given form but should have been extended with the p_1-p_2 subpath, which is a contradiction.

□

Another interesting property is the fact that different plateaus always lead to different paths. This eliminates the need to test whether a path already exists.

Lemma 5.2. *Let Pl_1 be a p_1-q_1 plateau which leads to the $s-t$ path P. Then P contains no other plateaus than Pl_1.*

Proof. Suppose P contains a p_1-q_1 plateau Pl_1 and a p_2-q_2 plateau Pl_2. Because of Lemma 5.1 it is known that Pl_1 and Pl_2 do not overlap. Let's assume without loss of generality that Pl_1 appears before Pl_2 in P. Then

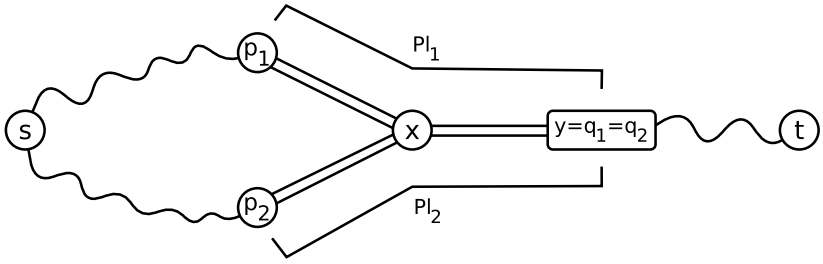


Figure 5.10: Example illustrating Lemma 5.1 where p_1 is not on the shortest path from s to p_2. There should be a shortest path in SPT_{OUT} from s to x via p_1 and via p_2, which is impossible.

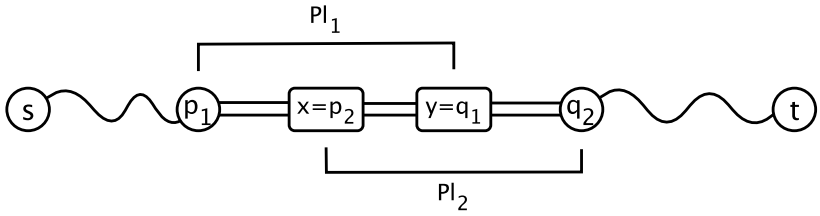


Figure 5.11: Example illustrating Lemma 5.1 where p_1 is on the shortest path from s to p_2. In this case the $p_1 - p_2$ subpath should have been part of Pl_2.

the $q_1 - p_2$ subpath appears both in SPT_{OUT} (since it is part of the path generated from Pl_2) and in SPT_{IN} (since it is part of the path generated from Pl_1). Therefore, the $q_1 - p_2$ subpath is a plateau as well, and the $p_1 - q_2$ subpath is one large plateau, which is a contradiction. □

It should be noted that a path resulting from a plateau can still contain arcs from other plateaus. An example of such a situation in shown in Figure 5.12. Since plateaus cannot overlap, the paths definitely differ from other paths in the plateaus themselves. The subpaths before and after the plateaus may or may not be similar to subpaths in other paths, depending on the location of the plateaus. Figure 5.13 shows all plateaus Pl such that $w(Pl) \geq 0.25 \times w(SP)$ for 4 random queries in NAVTEQ_BELGIUM. Because of Property 5.1, all of these plateaus correspond to a path which is guaranteed to be at least $0.25 \times w(SP)-$locally optimal. The figure clearly shows that this leads to some good options with high dissimilarity. Also, the number of such paths is very manageable, typically between 6 and 20 for Belgium, while enough possibilities to choose from are still provided. This can drastically reduce the time for making the final selection of paths. It should be noted that the set of plateaus may have different characteristics when T-local optimality for other values of T is required. This is examined more thoroughly in Section 5.7. Another remaining challenge is the fact that some plateaus correspond to paths which are clearly too long. This will be addressed in Section 5.8.

5.5.3 Plateaus and cycles

A path containing a plateau may contain a cycle if the subpath after the plateau crosses the subpath before the plateau. Such a situation is shown in Figure 5.14. The longer the plateau, the less likely it is for this situation to occur. Therefore, it is interesting to look for long plateaus, for reasons of local optimality, dissimilarity and avoiding cycles.

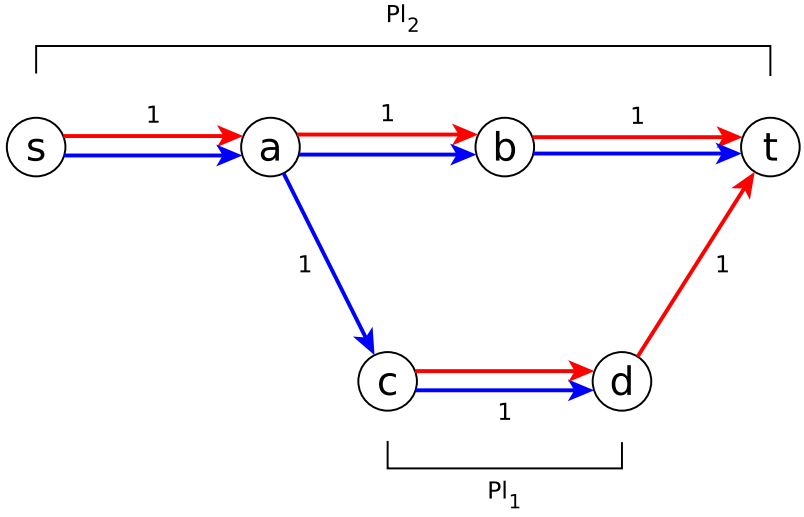


Figure 5.12: A path containing a plateau may contain arcs from other plateaus: the path containing Pl_1 contains the arc (s, a) which is part of Pl_2. SPT_{OUT} is shown in blue, SPT_{IN} is shown in red.

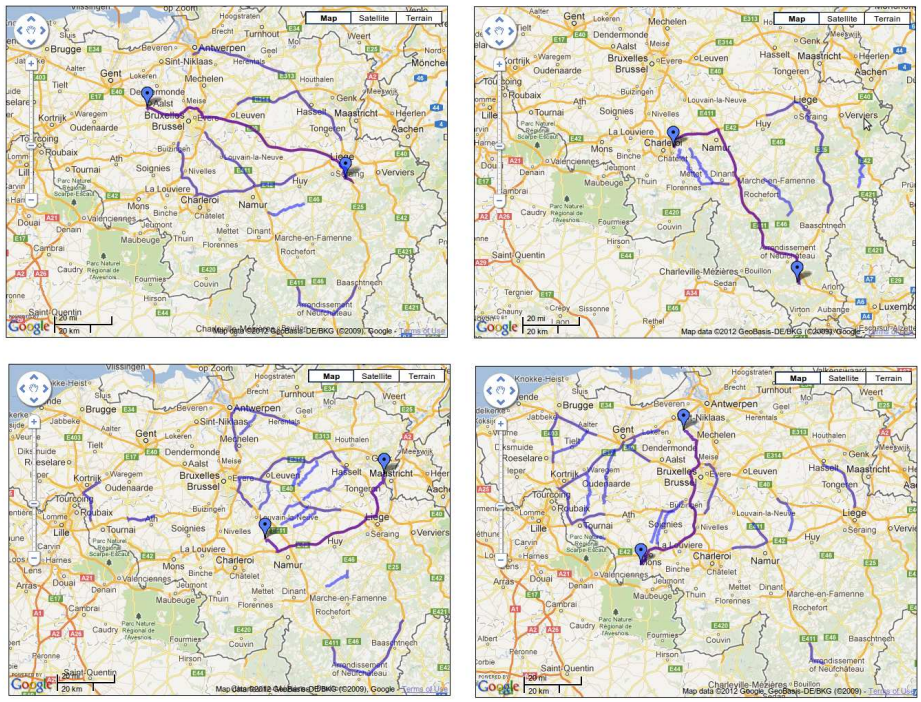


Figure 5.13: Four examples of plateaus for four random queries in NAVTEQ_BELGIUM. All plateaus larger than $\alpha \times w(SP)$ are shown for $\alpha = 25\%$.

5.6 Detecting plateaus

In this section we describe how to build a list containing all plateaus for a given query. The node sequence for every plateau is stored in this list, such that the list of plateaus can be used on its own without the shortest path trees. Plateaus are detected by determining the intersection of SPT_{OUT} and SPT_{IN}. First, the algorithm of Dijkstra is run in both directions. Both searches are pruned at $\beta \times w(SP)$ for a given value of $\beta \geq 1$. After

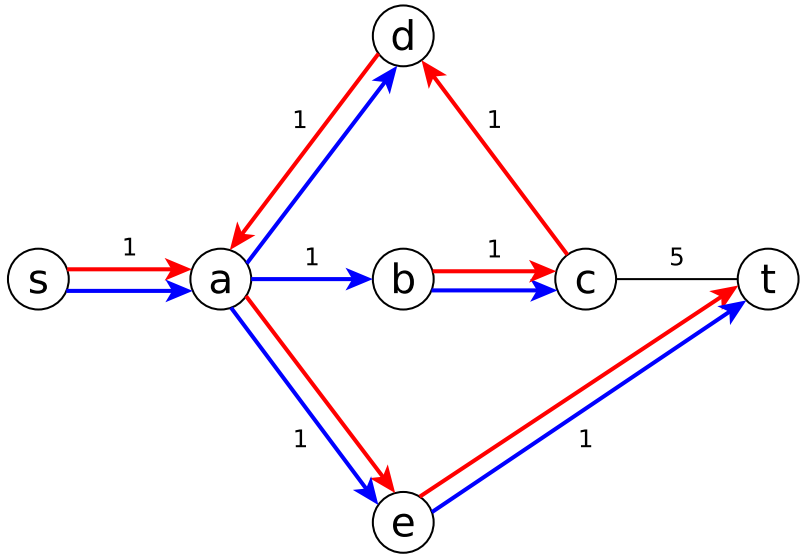


Figure 5.14: Situation where a cycle occurs: the path from s to the plateau (b, c) can be found in SPT_{OUT} and is (s, a, b). The path from the plateau to t can be found in SPT_{IN} and is (c, d, a, e, t). Both paths go through node a, which causes a cycle. SPT_{OUT} is shown in blue, SPT_{IN} is shown in red.

Algorithm 5.3 Algorithm for finding all plateaus for a given outward and inward shortest path tree for a query from s to t

Require: SPT_{OUT}, SPT_{IN}
Ensure: list *plateaus* containing all plateaus for the given SPT's

```
1: create childpointers
2: add s to stack
3: while stack is not empty do
4:    current ← stack.pop()
5:    for every child of current in SPT_OUT do
6:       {some nodes may have only temporary label due to pruning}
7:       if child has permanent label in SPT_OUT then
8:          stack.push(child)
9:       end if
10:   end for
11:   {check using Property 5.3}
12:   if current ≠ s AND preceding arc is part of plateau then
13:      plateaunr ← map.get(current.parent)
14:      {saves memory, value will not be needed again}
15:      map.remove(current.parent)
16:      map.put(current, plateaunr)
17:      currentplateau ← plateaus[plateaunr]
18:      add current to currentplateau
19:   else
20:      {this is the smallest plateau number not yet in use}
21:      plateaunr ← plateaus.size
22:      map.put(current, plateaunr)
23:      Pl ← createnewplateau
24:      add current to Pl
25:      add Pl to plateaus
26:   end if
27: end while
28: drop childpointers
```

this, the shortest path trees as well as the shortest path and its weight are available.
The pseudocode for the detection of plateaus is shown in Algorithm 5.3. SPT_{OUT} is traversed in preorder. For this purpose, childpointers are created since normally only pointers to the parents are stored in the SPT. The childpointers can be dropped when the plateaus have been determined. A stack is used to keep track of the traversal.
While the tree is being traversed, for every node in SPT_{OUT} there are three possibilities:

1. The node is not in SPT_{IN} and is therefore not part of a plateau.
2. The node has a permanent label in SPT_{IN} and the preceding arc is part of a plateau. In this case the node is part of the same plateau as its parent.
3. The node has a permanent label in SPT_{IN} but the preceding arc is not part of a plateau. In this case the first node of a new plateau is found.

For case 2, the preceding arc of the current node must be part of a plateau. To determine whether this is the case, the following property is used:

Property 5.3. *Let p be the parent of the current node x in SPT_{OUT}. Then the arc (p, x) is part of a plateau if and only if p has a permanent label in SPT_{IN} and the parent of p in SPT_{IN} is equal to x.*

To distinguish between the plateaus, every plateau is assigned a number. A HashMap is stored which maps nodes to the number of the plateau they belong to. In case 2, the plateau number for the parent can be found in the HashMap. In case 3, a new plateau is created and assigned a number, which is added to the HashMap. This plateau might be extended when the other nodes are visited but might also remain a single-node plateau. For memory efficiency, only the plateau numbers for the last node in each

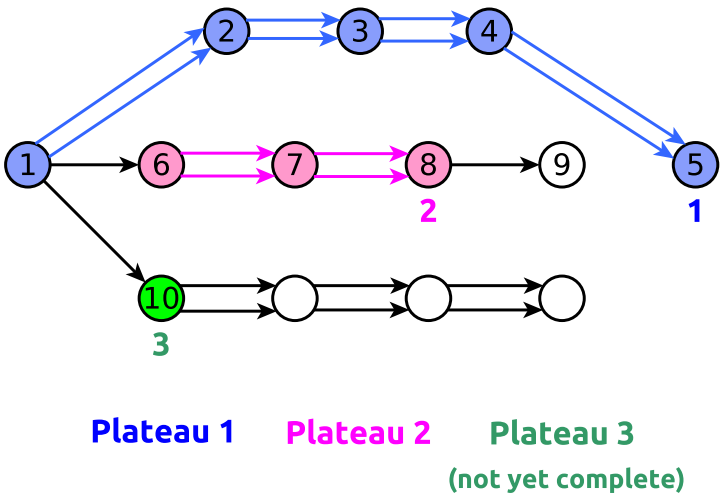


Figure 5.15: Detecting plateaus. A shortest path tree SPT_{OUT} is shown on which the algorithm is being performed. The algorithm has visited the first 10 nodes in the given order. It has detected Plateau 1 and 2 completely but has only detected the first node of Plateau 3 so far. Nodes 5, 8 and 10 are marked with their stored plateau number. No plateau numbers are stored for the other nodes at this point, since plateau numbers are only stored for the last visited node in their branch.

branch are stored. Whenever a node is assigned the plateau number of its parent (case 2), the plateau number for the parent is removed from the map since this information will no longer be needed. This is illustrated in Figure 5.15.

5.7 Plateaus: an in-depth look

It is interesting to take a closer look at plateaus and their properties. Using this information, the algorithm can be fine-tuned and appropriate values for

the parameters can be chosen. No such study has been done before. In this section we present in-depth results. One hundred random $s-t$ queries were chosen for each of five road networks. For each $s-t$ query, the complete set of plateaus was generated using Algorithm 5.3. This was done for different pruning factors ranging from 1 to 2, to discover the effect of the pruning factor on the plateaus. We study the number of plateaus, the weight of the plateaus and of their resulting paths, the occurrence of cycles in those paths and the local optimality of those paths. In this section, results are presented for the road networks NAVTEQ_LUXEMBOURG and PRT_MAX. Results for the other road networks were similar. In all tables in this section, the value ϵ is used to indicate "small but not zero". When "0" is used in the tables, the value is really zero and not a small value rounded to zero.

5.7.1 Conclusion 1: Many plateaus are generated. Most of them have a zero weight.

Table 5.1 shows the total amount of plateaus for the different pruning factors, averaged over the 100 $s-t$ queries (in the last rows of both tables). The number of plateaus is always quite high, thousands of plateaus are generated in every case. Also, the number of plateaus increases fast with the pruning factor. For example, 37 080 plateaus are generated on average for PRT_MAX with a pruning factor of 1. For a pruning factor of 2, this number rises to 84 914 on average. This means that the pruning factor needs to be chosen carefully. This will be studied further in Section 5.7.3. The average amount of plateaus within a given weight range (expressed as a percentage of $w(SP)$) is also shown. We can conclude that many of the plateaus (around 90% of all plateaus) have a zero weight. These are the nodes in which the shortest path trees touch but do not overlap. It is good to be aware of this fact, since longer plateaus are more likely to produce good results. An advantage of this, is that the amount of longer plateaus is very manageable, typically a few dozens for plateaus longer than $10\% \times w(SP)$.

Plateau weight	Pruning factor							
% of $w(SP)$	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	6 736	9 158	11 410	12 447	13 463	15 187	16 663	21 654
]0%,5%]	582	734	850	899	945	1 017	1 074	1 237
]5%,10%]	60	74	84	89	93	99	104	119
]10%,15%]	19	23	26	28	29	31	32	36
]15%,20%]	8	10	11	12	12	13	14	16
]20%,25%]	4	5	5	5	6	6	7	7
]25%,50%]	5	6	7	7	7	8	9	11
]50%,75%]	1	1	1	1	1	1	1	2
]75%,85%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]95%,100%[	0	0	0	0	0	0	ϵ	ϵ
100%	1	1	1	1	1	1	1	1
Sum	7 416	10 010	12 395	13 488	14 557	16 364	17 904	23 082

(a) NAVTEQ_LUXEMBOURG

Plateau weight	Pruning factor							
% of $w(SP)$	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	32 354	49 570	55 467	57 199	58 706	61 726	65 284	76 725
]0%,5%]	4 638	6 248	6 660	6 767	6 862	7 048	7 280	8 009
]5%,10%]	60	73	84	88	91	96	101	118
]10%,15%]	14	17	20	21	23	24	26	31
]15%,20%]	6	7	8	8	9	10	11	13
]20%,25%]	3	3	4	4	4	4	5	7
]25%,50%]	4	4	5	5	5	6	6	9
]50%,75%]	ϵ	ϵ	ϵ	ϵ	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1	1	1	1	1
Sum	37 080	55 924	62 248	64 094	65 701	68 916	72 715	84 914

(b) PRT_MAX

Table 5.1: Many plateaus are generated. Most of them have a zero weight. For different pruning factors (columns) and different plateau weights (rows) the total number of plateaus is shown. A value very close to zero is represented by ϵ.

5.7.2 Conclusion 2: Paths with short plateaus often contain cycles. Paths with longer plateaus are unlikely to contain cycles.

From the plateaus, their corresponding $s - t$ paths can be generated. Unfortunately, there is no guarantee that these paths do not contain cycles. In fact, many plateaus are very likely to produce paths with cycles. Table 5.2 shows the percentage of plateaus which produce paths with cycles, for different plateau weights and pruning factors, averaged over the 100 $s - t$ pairs. Almost every plateau with a zero weight produces a path with a cycle. However, almost none of the longer plateaus produces paths with cycles. It should also be noted that when a larger pruning factor is used, cycles are more likely to appear, which is another argument for choosing the pruning factor very carefully.

5.7.3 Conclusion 3: 1.25 is a good pruning factor.

A good pruning factor does not generate more plateaus than necessary, but also does not miss too many good alternatives. A good alternative has no cycles, is locally optimal and is "not too long". Table 5.3 shows the number of such good alternatives for different values of α and for different weights of the alternatives, averaged over the 100 $s - t$ queries. It is important to point out that rather than the plateau weight, the weight of the $s - t$ alternatives is considered in this table, since the eventual goal is to have alternatives which are relatively short, not plateaus. Clearly, more good alternatives are found when using higher pruning factors. However, this is especially the case for alternatives with higher weights, which may not be desirable. For alternatives with a weight no higher than 1.5 times the shortest path, the same amount of good alternatives is always found for a pruning factor of 1.25 and a pruning factor of 2. This is the case for all examined road networks. This means that not a single good alternative shorter than $1.5 \times w(SP)$ which would have been found with a pruning factor of 2, is missed with a pruning factor of 1.25. Because of this, there is no reason to use a pruning factor of 2, which generates 21 082 plateaus for

Plateau weight % of $w(SP)$	Pruning factor							
	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	94%	95%	95%	96%	96%	96%	96%	97%
]0%,5%]	22%	24%	26%	26%	27%	28%	28%	31%
]5%,10%]	5%	5%	5%	5%	5%	5%	5%	6%
]10%,15%]	3%	4%	4%	4%	4%	4%	4%	5%
]15%,20%]	1%	1%	2%	2%	2%	3%	3%	4%
]20%,25%]	2%	3%	3%	3%	4%	4%	6%	6%
]25%,50%]	ϵ	1%	2%	3%	3%	3%	4%	7%
]50%,75%]	0%	0%	0%	0%	0%	1%	4%	13%
]75%,85%]	0%	0%	0%	0%	0%	0%	0%	0%
]85%,95%]	0%	0%	0%	0%	0%	0%	0%	0%
]95%,100%[	/	/	/	/	/	/	0%	0%
100%	0%	0%	0%	0%	0%	0%	0%	0%

(a) NAVTEQ_LUXEMBOURG

Plateau weight % of $w(SP)$	Pruning factor							
	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	90%	92%	93%	93%	93%	93%	93%	94%
]0%,5%]	22%	28%	30%	30%	30%	31%	32%	36%
]5%,10%]	6%	6%	7%	7%	8%	8%	9%	14%
]10%,15%]	5%	5%	6%	6%	6%	6%	7%	12%
]15%,20%]	4%	5%	5%	5%	5%	7%	7%	12%
]20%,25%]	1%	2%	2%	2%	3%	4%	4%	6%
]25%,50%]	1%	1%	2%	3%	3%	4%	5%	9%
]50%,75%]	0%	0%	2%	2%	2%	2%	2%	7%
]75%,85%]	0%	0%	0%	0%	0%	0%	0%	0%
]85%,95%]	0%	0%	0%	0%	0%	0%	0%	0%
]95%,100%[	0%	0%	0%	0%	0%	0%	0%	0%
100%	0%	0%	0%	0%	0%	0%	0%	0%

(b) PRT_MAX

Table 5.2: Paths with short plateaus often contain cycles. Paths with longer plateaus are unlikely to contain cycles. For different pruning factors (columns) and different plateau weights (rows) the percentage of plateaus which produce cycles is shown. A value very close to zero is represented by ϵ.

α	Path weight $\times w(SP)$	Pruning factor							
		1	1.1	1.2	1.25	1.3	1.4	1.5	2
10%	[1.0,1.1[	16	16	16	16	16	16	16	16
	[1.1,1.25[	38	41	41	41	41	41	41	41
	[1.25,1.5[	45	55	59	59	59	59	59	59
	[1.5,2[	30	47	64	71	77	84	85	85
	≥ 2	0	4	13	18	25	42	60	118
25%	[1.0,1.1[	4	4	4	4	4	4	4	4
	[1.1,1.25[	6	6	6	6	6	6	6	6
	[1.25,1.5[	9	10	10	10	10	10	10	10
	[1.5,2[	8	11	14	15	16	17	17	17
	≥ 2	0	1	4	5	7	12	16	35
50%	[1.0,1.1[	2	2	2	2	2	2	2	2
	[1.1,1.25[	1	1	1	1	1	1	1	1
	[1.25,1.5[	1	1	1	1	1	1	1	1
	[1.5,2[	1	1	2	2	2	2	2	2
	≥ 2	0	ϵ	1	1	1	2	2	6
total # of plateaus		7 416	10 010	12 395	13 488	14 557	16 364	17 904	23 082

(a) NAVTEQ_LUXEMBOURG

α	Path weight $\times w(SP)$	Pruning factor							
		1	1.1	1.2	1.25	1.3	1.4	1.5	2
10%	[1.0,1.1[	58	74	74	74	74	74	74	74
	[1.1,1.25[	24	25	25	25	25	25	25	25
	[1.25,1.5[	22	30	33	34	34	34	34	34
	[1.5,2[	13	22	31	37	41	46	47	47
	≥ 2	ϵ	2	6	8	11	20	31	73
25%	[1.0,1.1[	46	68	68	68	68	68	68	68
	[1.1,1.25[	3	6	6	6	6	6	6	6
	[1.25,1.5[	4	4	5	5	5	5	5	5
	[1.5,2[	4	5	7	7	8	8	8	8
	≥ 2	ϵ	1	2	3	4	6	8	21
50%	[1.0,1.1[	34	56	56	56	56	56	56	56
	[1.1,1.25[	1	5	5	5	5	5	5	5
	[1.25,1.5[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	[1.5,2[	1	1	1	1	1	1	1	1
	≥ 2	0	ϵ	ϵ	1	1	1	2	5
total # of plateaus		37 080	55 924	62 248	64 094	65 701	68 916	72 715	84 914

(b) PRT_MAX

Table 5.3: 1.25 is a good pruning factor. For different pruning factors (columns) and different weights of the paths corresponding to the plateaus and different values of α (rows), the number of "good" paths is shown, i.e. paths which are cycle-free and locally optimal. The last row shows the total number of plateaus for each pruning factor.

NAVTEQ_LUXEMBOURG, while a pruning factor of 1.25 produces only 13,488 plateaus and does not miss a single good alternative. For this reason we choose 1.25 as our default pruning factor.

5.7.4 Conclusion 4: A path with a plateau Pl such that $w(Pl) < T$ can still be T-locally optimal.

A path can only be taken into consideration by the algorithm if it is both locally optimal and cycle-free. If a path contains a plateau Pl, then this path is guaranteed to be T-locally optimal for $T = w(Pl)$ according to Property 5.1. Such a path does not need to be tested for local optimality since it is guaranteed. It only needs to be tested for cycles, which is a lot less time-consuming. This means that we could consider only looking for plateaus which are long enough, and in this way eliminating the need for testing for local optimality at all. However, it is interesting that there are quite a lot of paths with a plateau shorter than T which are still T-locally optimal and cycle-free. This is shown in Table 5.4. The table shows the total number of plateaus for different plateau weights, and how many of those result in *cycle-free* locally optimal paths for $\alpha = 10\%$, $\alpha = 25\%$ and $\alpha = 50\%$, averaged over all 100 $s - t$ queries. The pruning factor is fixed at 1.25 in this table. For example, three of the four paths containing a plateau with a weight in the range]20%,25%] are locally optimal and cycle-free for $\alpha = 25\%$ in the road network PRT_MAX. Even some of the plateaus with a zero weight are still locally optimal, even for $\alpha = 50\%$. We can conclude that it is definitely worth looking at the shorter plateaus too.

5.8 The objective function $Q(S)$

In Section 5.2 the following dissimilarity function by Akgün et al. [12] was described:

$$D(P_i, P_j) = 1 - \frac{w_s(P_i, P_j)/w(P_i) + w_s(P_i, P_j)/w(P_j)}{2}$$

Plateau weight % of $w(SP)$	α 0.1	0.25	0.5	total # of plateaus
0	23	3	ϵ	12 447
]0,5%]	77	7	1	899
]5%,10%]	54	7	ϵ	89
]10%,15%]	26	7	1	28
]15%,20%]	11	5	ϵ	12
]20%,25%]	5	3	ϵ	5
]25%,50%]	7	7	1	7
]50%,75%]	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ
]95%,100%[	0	0	0	0
100%	1	1	1	1
Sum	206	40	6	13 488

(a) NAVTEQ_LUXEMBOURG

Plateau weight % of $w(SP)$	α 0.1	0.25	0.5	total # of plateaus
0	30	28	25	57 199
]0,5%]	72	41	35	6 767
]5%,10%]	38	4	1	88
]10%,15%]	20	4	ϵ	21
]15%,20%]	8	3	ϵ	8
]20%,25%]	4	3	ϵ	4
]25%,50%]	5	5	1	5
]50%,75%]	ϵ	ϵ	ϵ	ϵ
]75%,85%]	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1
Sum	178	88	64	64 094

(b) PRT_MAX

Table 5.4: A path with a plateau Pl such that $w(Pl) < T$ can still be T-locally optimal. For different values of α (columns) and for different plateau weights (rows) the number of cycle-free locally optimal paths is shown. For example, for PRT_MAX, 4 plateaus with a weight between 10% and 15% of $w(SP)$ result in a cycle-free path which is locally optimal for $\alpha = 0.25$. The total number of plateaus in each weight class is shown in the last column. The pruning factor was set to 1.25.

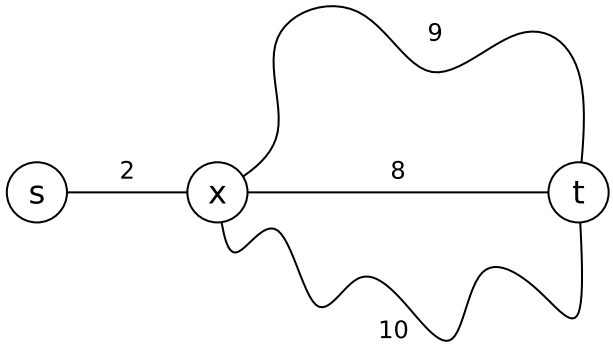


Figure 5.16: Example used for evaluating dissimilarity functions. The shortest path is the path (s, x, t) and has a weight of 10. The curved lines show two alternatives for the $x - t$ segment. Not all nodes and arcs are shown.

While this definition is very useful to evaluate dissimilarity as such, it tends to favor longer paths. Figure 5.16 illustrates this fact. The shortest path SP from s to t is the (s, x, t) path with a weight of 10. The curved lines represent two alternatives for the $x - t$ segment. Let P_1 be the $s - t$ alternative with weight 11 and let P_2 be the $s-t$ alternative with weight 12. Then we find

$$D(SP, P_1) = 1 - \frac{2/10 + 2/11}{2} = 0.809$$

and

$$D(SP, P_2) = 1 - \frac{2/10 + 2/12}{2} = 0.817$$

This means that a p-dispersion method would consider P_2 a better alternative than P_1. However, P_2 has exactly the same subpath in common with

SP as P_1 does, the only reason why it is "more dissimilar" from SP is the fact that it is longer.
Many existing methods bypass this problem by deleting every path which is longer than a certain threshold before running the p-dispersion heuristic. On the one hand, this can lead to eliminating paths which would have been good alternatives after all. On the other hand, once a relatively long path has been kept, it has a higher chance of being chosen by the p-dispersion method, which can affect the quality of the solution.
Therefore, we have defined a different objective function to be used by the p-dispersion heuristic which takes both dissimilarity and path weight into account and which does not favor longer paths like in the example above. The objective function consists of two components: the *adjusted dissimilarity* AD and the *weight ratio* WD .
Let $w_{min}(P_i, P_j)$ be the weight of the shortest path of P_i and P_j. Let $w_{max}(P_i, P_j)$ be the weight of the longest path of P_i and P_j. Then we define the *adjusted dissimilarity* AD as follows:

$$AD(P_i, P_j) = 1 - \frac{w_s(P_i, P_j)}{w_{min}(P_i, P_j)}$$

This measure is symmetric and gives a value between 0 and 1. A value of 0 represents paths which coincide completely, while a value of 1 represents completely dissimilar paths. Because of the division by the weight of the shortest path of both, longer paths are not favored anymore. For the example in Figure 5.16 the adjusted dissimilarity is as follows:

$$AD(SP, P_1) = 1 - \frac{2}{10} = 0.8$$

$$AD(SP, P_2) = 1 - \frac{2}{10} = 0.8$$

Using only the adjusted dissimilarity, P_1 and P_2 would have the same probability of being chosen by the p-dispersion heuristic. This is why the second component, the *weight ratio* WR is introduced:

$$WR(P_i, P_j) = \frac{w_{min}(P_i, P_j)}{w_{max}(P_i, P_j)}$$

Again, this measure is symmetric and gives a value between 0 and 1. Finally, the new dissimilarity function D' is a linear combination of both components, using a parameter β to determine the trade-off between the two:

$$D'(P_i, P_j) = \beta \times [1 - \frac{w_s(P_i, P_j)}{w_{min}(P_i, P_j)}] + (1 - \beta) \times \frac{w_{min}(P_i, P_j)}{w_{max}(P_i, P_j)}$$

This also results in a value between 0 and 1. A user study could determine a good value for β, but we have used $\beta = 0.5$ as this led to good results. With $\beta = 0.5$, for the example in Figure 5.16 we now get:

$$D'(SP, P_1) = \beta \times [1 - \frac{2}{10}] + (1 - \beta) \times \frac{10}{11} = 0.855$$

$$D'(SP, P_2) = \beta \times [1 - \frac{2}{10}] + (1 - \beta) \times \frac{10}{12} = 0.817$$

Using this definition, the path P_1 is favored over the path P_2, which was desired. Finally, the objective function $Q(S)$ defines the quality of a solution S as the minimum value of the dissimilarity function D' for every pair in the solution:

$$Q(S) = \min_{P_i, P_j \in S} \left[\beta \times [1 - \frac{w_s(P_i, P_j)}{w_{min}(P_i, P_j)}] + (1 - \beta) \times \frac{w_{min}(P_i, P_j)}{w_{max}(P_i, P_j)} \right]$$

k	Given paths	Result $Q_A(S)$ Result $Q(S)$	$Q_A(S)$	$Q(S)$
1	0% 6% 13% 15% 15% 21% 38% 60% 75%	0% 60% 75%	0.880	0.739
		0% 15% 15%	0.816	0.859
2	0% 1% 12% 31% 39% 47%	0% 12% 47%	0.861	0.792
		0% 12% 47%	0.861	0.792
3	0% 0% 2% 2% 3% 5% 7% 33%	0% 3% 33%	0.795	0.759
		0% 3% 5%	0.785	0.867
4	0% 0% 4% 6% 21% 23% 23% 30% 30% 89%	0% 23% 89%	0.907	0.705
		0% 6% 30%	0.893	0.824
5	0% 1% 2% 10% 18% 30% 39% 97% 99%	0% 18% 99%	0.806	0.726
		0% 2% 18%	0.579	0.778
6	0% 3% 3% 4% 8% 10% 16%	0% 10% 16%	0.711	0.778
		0% 3% 8%	0.708	0.836
7	0% 3% 13% 74% 85%	0% 13% 74%	0.560	0.509
		0% 3% 13%	0.423	0.694

Table 5.5: Comparison of the objective functions $Q(S)$ and $Q_A(S)$. The table shows a few representative examples. The 1^{st} column shows the number of the example. The 2^{nd} column shows the weight of the paths which were given as input to the p-dispersion algorithm, i.e. how much they are longer than $w(SP)$. The 3^{rd} column shows which paths were chosen by the p-dispersion algorithm using $Q(S)$ and $Q_A(S)$. Finally, the last two columns show the quality of these solutions, both according to $Q(S)$ and $Q_A(S)$.

Using the dissimilarity definition by Akgün et al. [12] the quality $Q_A(S)$ (where the A stands for Akgün) looks like this:

$$Q_A(S) = \min_{P_i,P_j \in S} \left[1 - \frac{w_s(P_i,P_j)/w(P_i) + w_s(P_i,P_j)/w(P_j)}{2} \right]$$

Table 5.5 shows a comparison of $Q(S)$ and $Q_A(S)$. For a number of $s-t$ queries, a set of possible paths (including the shortest path) was generated. This set is the set of cycle-free paths resulting from the plateaus larger than T, although another set of paths could have been used too since the goal of the experiment is only to evaluate which paths are chosen out of this set. A representative set of results is shown. The column "Given paths" shows a percentage for every path indicating how much longer it is than the shortest path. E.g. in the first example, the path with the percentage "0%" is the

shortest path, the path with the percentage "6%" is 6% longer than the shortest path and so on. Then there are two lines for every example. The first line shows the results when the p-dispersion heuristic uses Q_A. The second line shows the results when the p-dispersion heuristic uses Q. The third column shows which paths are chosen by both methods. As can be seen in the results, the p-dispersion heuristic always includes the shortest path. For comparative purposes, the value for both Q and Q_A is given for each solution. E.g., for the 4th example, the p-dispersion heuristic finds the shortest path and two alternatives which are 23% and 89% longer than the shortest path when Q_A is used, and 6% and 30% when Q is used. It is clear that the alternatives are much shorter when Q is used. The solution "0% 6% 30%" is a lot better than the solution "0% 23% 89%" when considering the Q value (0.824 vs. 0.705), but hardly any worse when considering the Q_A value (0.893 instead of 0.907). The other examples show a similar behaviour. On the first line, the long paths are favored, often the longest path is even included in the solution. On the second line, shorter paths are chosen which leads to a better value of Q and either an only slightly worse value of Q_A (Examples 3, 4 and 6) or a huge gain in path weight (Examples 1, 5 and 7). Sometimes both methods find the same result, as is the case in Example 2. Finally, Figures 5.17 and 5.18 show solutions of different quality for the same query. These figures can be used to get an idea of how "good" a solution is for a certain value of $Q(S)$. It is clear from these figures that solutions with a higher value for $Q(S)$ have more dissimilar and/or shorter paths.

(a) $Q(S) = 0.685$

(b) $Q(S) = 0.726$

Figure 5.17: Four solutions of different quality for the same query in NAVTEQ_BELGIUM. Solutions with a higher value of $Q(S)$ clearly contain more dissimilar and/or shorter paths.

(c) $Q(S) = 0.750$

(d) $Q(S) = 0.796$

Figure 5.17: Four solutions of different quality for the same query in NAVTEQ_BELGIUM. Solutions with a higher value of $Q(S)$ clearly contain more dissimilar and/or shorter paths. (Continued)

(a) $Q(S) = 0.690$

(b) $Q(S) = 0.790$

Figure 5.18: Two solutions of different quality for the same query in NAVTEQ_BELGIUM. Solutions with a higher value of $Q(S)$ clearly contain more dissimilar and/or shorter paths.

5.9 Methods for testing local optimality

Testing local optimality is a critical challenge since it can be very time-consuming. Both Abraham et al. [11] and Bader et al. [13] mention that testing local optimality efficiently remains an open question. Therefore, it is best to avoid testing local optimality as much as possible. However, sometimes local optimality does need to be tested. Some methods for testing local optimality for a path P from s to t in various situations are described below.

5.9.1 General method: brute force

The most straightforward method for checking local optimality is a simple brute force algorithm. It can be used for any path and is similar to Algorithm 5.2. For every node (except those at a distance $< T$ from t), the algorithm of Dijkstra is performed to see if the smallest subpath starting in this node with weight $\geq T$ is a shortest path. However, if this is not the case, the subpath is not replaced by its corresponding shortest path (Lines 10 and 11 in Algorithm 5.2), but the algorithm returns `false`. If the algorithm reaches the end without finding a subpath which is not a shortest path, the path is T-locally optimal and `true` is returned. While this algorithm is very time-consuming, there is currently no better known algorithm which can be used for any path. If more is known about the structure of the path, better algorithms can be used. This is described in the next paragraphs.

5.9.2 Exact algorithm around one central node

An algorithm for testing local optimality can benefit from the fact that there is a node x such that $s - x$ and $x - t$ are shortest paths. This case occurs when a node has a permanent label in SPT_{OUT} and in SPT_{IN}. Let v_i be the node in P between s and x which is the closest node to x such

that $w(P[v_i..x]) \geq T$), or s if no such node exists. Let v_j be the node in P between v and t which is the closest node to x such that $w(P[x..v_j] \geq T)$, or t if no such node exists. Let Q be $P[v_i..v_j]$.

Lemma 5.3. *If $Q = P[v_i..v_j]$ is T-locally optimal, then P is T-locally optimal.*

Proof. Suppose that P is not T-locally optimal. Then there are nodes v' and v'' in P such that the $v' - v''$ subpath P' has a weight $\leq T$ and is not a shortest path. P' cannot lie between s and x nor can it lie between x and t since the $s - x$ and $x - t$ subpaths are both shortest paths. Therefore, v' and v'' must be on different sides of x. Since $w(P[v_i..x]) \geq T$ (or $s = v_i$) and $w(P[x..v_j]) \geq T$ (or $v_j = t$), and $w(P') \leq T$, P' must be a subpath of Q. Since Q is T-locally optimal, P' must be a shortest path, which is a contradiction. □

Therefore, running the brute force algorithm only between v_i and v_j is sufficient. Depending on the value of T, this can save a lot of time. This method and the following methods are illustrated in Figure 5.19.

5.9.3 T-test around one central node

Abraham et al. [11] present a heuristic for testing local optimality. It requires only one run of the algorithm of Dijkstra, but the results may contain false negatives. It is called the *T-test*. Instead of running the brute force algorithm for Q they simply check if Q is a shortest path, which is a sufficient but not necessary condition. They prove that if a path P passes the T-test, it is definitely T-locally optimal. If P fails the T-test, P is definitely not $2T$-locally optimal and may or may not be T-locally optimal.

5.9.4 Exact algorithm around a plateau

The algorithm for testing local optimality around a central node can be extended as an algorithm for testing local optimality around plateaus. As

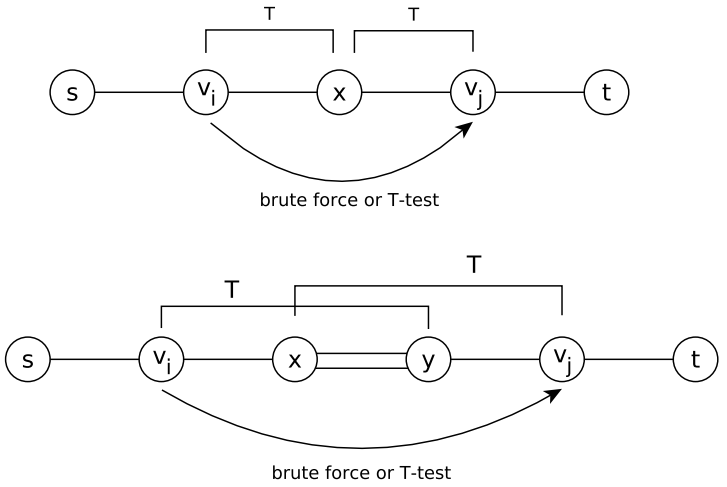


Figure 5.19: Testing local optimality around a central node (top) and around a plateau (bottom).

mentioned earlier, Abraham et al. [11] prove that a path is always T-locally optimal if it contains a plateau with weight at least T. It is only necessary to test the local optimality if the weight of the plateau is smaller than T. Assume that the plateau is bounded by the nodes x and y, such that $s-y$ is a shortest path which includes x and $x-t$ is a shortest path which includes y. Let v_i be the node in P between s and y which is the closest node to y such that $w(P[v_i..y] \geq T)$, or s if no such node exists. Let v_j be the node in P between x and t which is the closest node to x such that $w(P[x..v_j]) \geq T)$, or t if no such node exists. Let Q be $P[v_i..v_j]$. We now prove that it is sufficient to check the local optimality for the path Q, regardless of the size of the plateau.

Lemma 5.4. *If $Q = P[v_i..v_j]$ is T-locally optimal, then P is T-locally optimal.*

Proof. Suppose that P is not T-locally optimal. Then there are nodes v' and v" in P such that the $v' - v''$ subpath P' has a weight $\leq T$ and is not a shortest path. P' cannot lie between s and y nor can it lie between x and t since the $s - y$ and $x - t$ subpaths are both shortest paths. P' cannot be a subpath of the plateau since a plateau is always a shortest path. Therefore, v' and v'' must be on different sides of the plateau. If the weight of the plateau is higher than T, then $w(P')$ must be higher than T as well, which is a contradiction. If the weight of the plateau is smaller than T, P' must be a subpath of Q since $w(P[v_i..y]) \geq T$ (or $s = v_i$) and $w(P[x..v_j] \geq T$ (or $v_j = t$) and $w(P') \leq T$. Since Q is T-locally optimal, P' must be a shortest path, which is a contradiction. □

5.9.5 T-test around a plateau

The T-test, which was designed by Abraham et al. [11] to be used around one central node, can also be adapted for use around plateaus. Again, Q being a shortest path is sufficient (but not necessary). If P passes the T-test, it is T-locally optimal. If it fails the T-test, it is not $[2T - w(Pl)]$-locally optimal. This is a tighter bound than when checking local optimality around one node. This tighter bound is an interesting benefit of the use of plateaus, since there will be less false negatives.

Lemma 5.5. *If a path P around a plateau passes the T-test, it is T-locally optimal.*

Proof. If P passes the T-test, it means that the path Q is a shortest path longer than T and therefore T-locally optimal. Based on Lemma 5.3, we can conclude that P is also T-locally optimal. □

Lemma 5.6. *If a path P around a plateau fails the T-test, it is not $[2T - w(Pl)]$-locally optimal.*

Proof. The length of Q is $[2T - w(Pl)]$ (see Figure 5.20). If P fails the T-test, it means that Q is not a shortest path. According to the definition

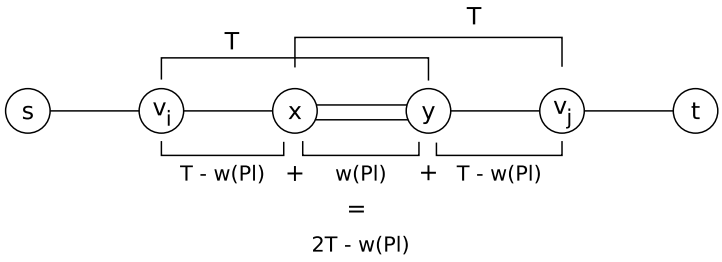


Figure 5.20: Calculation of the length of the path $Q = P[v_i..v_j]$ when running the T-test for a path with a plateau.

of local optimality, P is not $[2T - w(Pl)]$-locally optimal, since there is at least one path with length $\leq [2T - w(Pl)]$ which is not a shortest path. □

5.9.6 Experimental comparison of these methods

The question is raised whether it is more interesting to use the T-test, which may find false negatives, or the exact algorithm around a plateau. For this purpose, we designed an experiment which evaluates running times as well as the accuracy of the T-test. For five road networks and for 100 random $s - t$ queries for each road network, all plateaus were generated. For every plateau which is shorter than T and leads to a cycle-free path, both local optimality tests were run with $\alpha = 25\%$. Table 5.6 shows the average running time per path. It is clear that the running times are very similar, and sometimes the T-test is even slower than the exact algorithm. Furthermore, the success rate for the T-test is also shown. If the success rate is e.g. 46%, then the T-test returned `true` for 46% of the paths which are locally optimal, while the other 54% were false negatives. Typically, the success rate is around 50%, and for PRT_MAX it is even only 12%. Clearly, it is not worth sacrificing this accuracy if there is no gain in running time,

so we will use the exact algorithm for testing local optimality around a plateau.

Road network	Plateau exact (ms)	T-test (ms)	Success rate
CZE_MAX	2	3	50%
IRL_MAX	5	5	46%
NAVTEQ_LUXEMBOURG	4	4	51%
PRT_MAX	43	41	12%
NAVTEQ_BELGIUM	64	70	55%

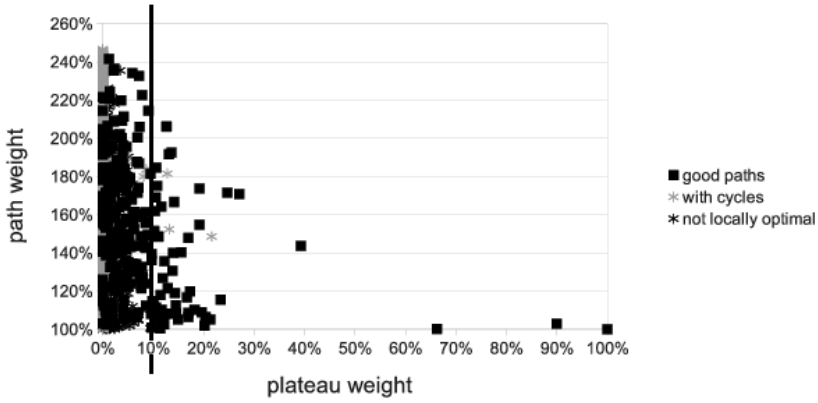
Table 5.6: Comparison of the exact algorithm for testing local optimality around a plateau and the T-test with $\alpha = 25\%$. The average running time in milliseconds per path is shown. The column "Success rate" shows for how many locally optimal paths the T-test returns `true`.

5.10 Strategy for testing plateaus

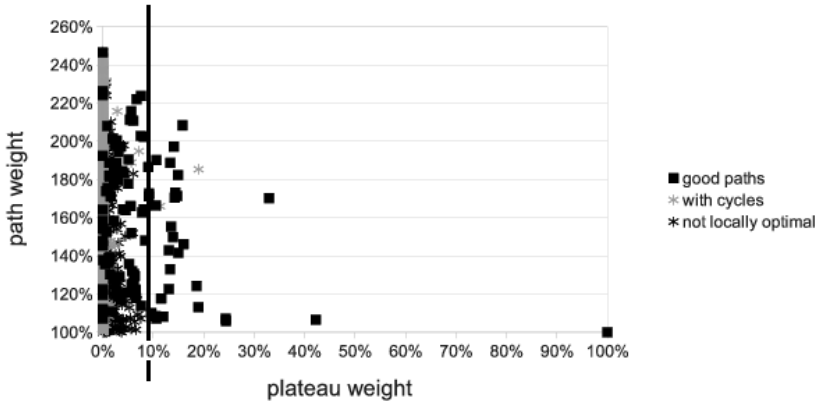
5.10.1 Motivation

Since testing local optimality requires several Dijkstra runs, it is a bottleneck for the algorithm and should be used sparingly. Testing every possible path for local optimality would generally be too time-consuming. Therefore, a parameter max is introduced which determines how many paths will be tested for local optimality. It can be seen as a measure for how long the user wants the algorithm to keep looking for a better solution. When $max = +\infty$ every single plateau is converted to its corresponding path, tested for cycles and tested for local optimality if no cycles were found. When $max = 0$, only the plateaus $\geq T$ are converted to their corresponding paths and tested for cycles. No local optimality testing is needed for these paths. However, as was shown in Section 5.7.4, shorter plateaus can lead to T-locally optimal paths as well. Therefore, it is worthwile to consider the shorter plateaus too in order to find a better solution, or sometimes in order to find a solution at all. A typical value for max would be between

5 and 100. E.g. if $max = 20$ the algorithm will continue until 20 paths have been tested for local optimality. Paths which have only been tested for cycles but not for local optimality, because they had a cycle, do not count since cycle testing is a lot faster than local optimality testing.
When only such a limited amount of paths are tested for local optimality, it is very important to make an informed decision on which plateaus to try first. Therefore, Figures 5.21, 5.22 and 5.23 each show a deep analysis for two single queries for different values of α. Every single plateau is shown in the charts. Its position in the chart is determined by its plateau weight and its corresponding path weight. Both the plateau weight and the path weight are shown as a percentage of $w(SP)$. Squares represent locally optimal paths without cycles. Light asterisks show paths which contain cycles. These paths do not need to be tested for local optimality. Dark asterisks represent paths which do not contain cycles but are not locally optimal either. These paths are time-consuming since they need to be tested for local optimality only to discover that the path cannot be used after all. Therefore, these paths should be avoided.
On each chart, a vertical line shows the value for T. All plateaus on the right of this line are guaranteed to correspond to locally optimal paths, although some plateaus might still result in paths with cycles. On the left of this line, many more squares can be seen in most charts. These are the good paths which we aim to find, so a strategy is needed to find as many of these paths as possible without spending too much time on "bad" paths. It is clear that the probability of finding a good path decreases with decreasing plateau weight. Especially close to the vertical axis many paths with cycles or which are not locally optimal can be found, so this area should be avoided. Therefore, it is clear that it is best to try the plateaus from right to left. On the other hand, it is also interesting to try the plateaus close to the horizontal axis first since they correspond to short paths, which is desirable. Therefore, we present the following strategy.

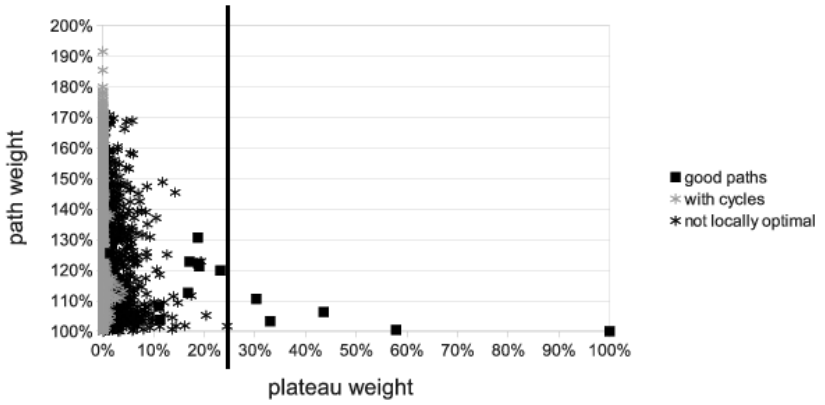


(a)

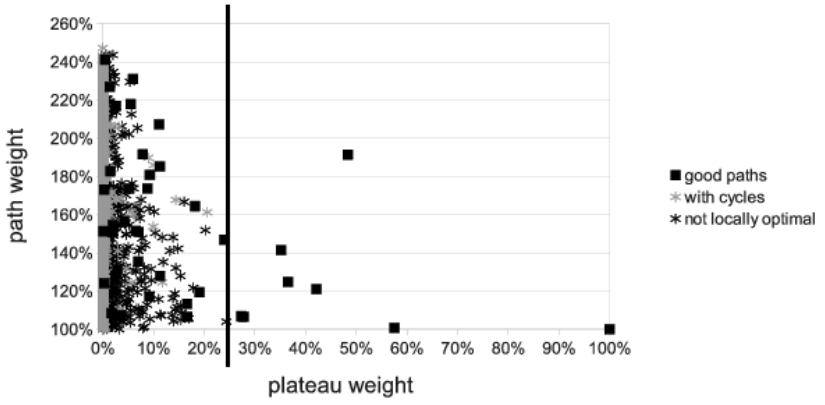


(b)

Figure 5.21: Deep analysis of two queries for NAVTEQ_LUXEMBOURG with $\alpha = 10\%$. For every plateau, both the plateau weight and the corresponding path weight are shown as a percentage of $w(SP)$. Squares indicate desirable paths, i.e. locally optimal paths without cycles. Paths on the right of the vertical line are guaranteed to be locally optimal.

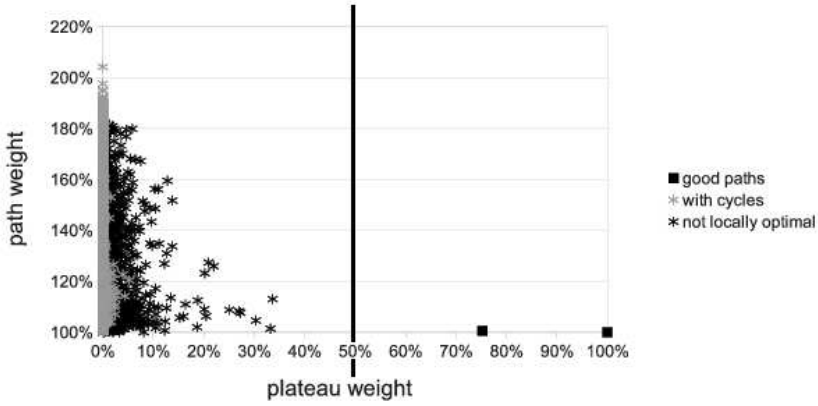


(a)

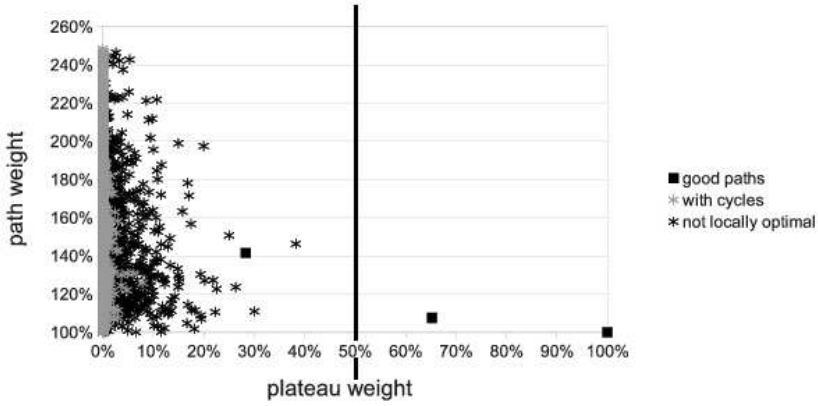


(b)

Figure 5.22: Deep analysis of two queries for NAVTEQ_LUXEMBOURG with $\alpha = 25\%$. For every plateau, both the plateau weight and the corresponding path weight are shown as a percentage of $w(SP)$. Squares indicate desirable paths, i.e. locally optimal paths without cycles. Paths on the right of the vertical line are guaranteed to be locally optimal.



(a)



(b)

Figure 5.23: Deep analysis of two queries for NAVTEQ_LUXEMBOURG with $\alpha = 50\%$. For every plateau, both the plateau weight and the corresponding path weight are shown as a percentage of $w(SP)$. Squares indicate desirable paths, i.e. locally optimal paths without cycles. Paths on the right of the vertical line are guaranteed to be locally optimal.

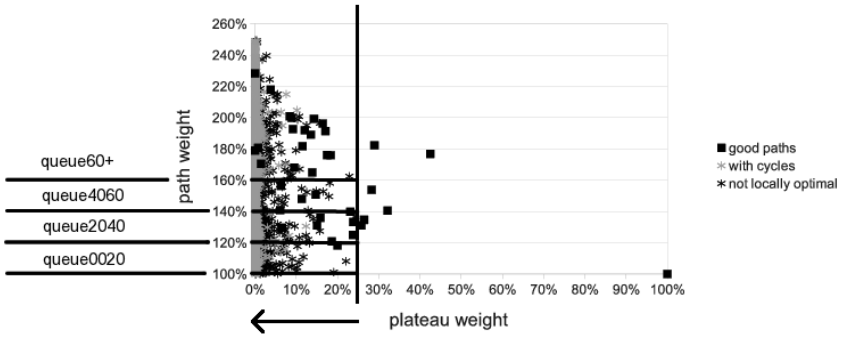


Figure 5.24: Strategy for testing plateaus. The queues are tried from bottom to top. Within the queues the plateaus are tried from right to left.

5.10.2 Strategy

First, all the plateaus on the right of the vertical line are added to a candidate set C from which the solution will be selected eventually. The plateaus on the left of the vertical line are partitioned into 4 queues as follows:

- The queue *queue*0020 contains all the plateaus corresponding to paths which are at most 20% longer than the shortest path.
- The queue *queue*2040 contains all the plateaus corresponding to paths which are between 20% and 40% longer than the shortest path.
- The queue *queue*4060 contains all the plateaus corresponding to paths which are between 40% and 60% longer than the shortest path.
- The queue *queue*60+ contains all the plateaus corresponding to paths which are more than 60% longer than the shortest path.

Each queue corresponds to a horizontal strip in the charts. Figure 5.24 illustrates this partitioning. Each strip will be tested from right to left, which means that the queues must be sorted by decreasing plateau weight.

The queue *queue*0020 is examined first since it contains the shortest paths. Paths are fetched from the queue and tested until less than 50% of the tried paths had no cycles and were locally optimal (given that at least 2 paths had been tried). When this occurs, the success rate is considered too low and the strategy moves on to the next queue, which is *queue*2040. Paths which contain no cycles and are locally optimal are added to the candidate set C. The same routine is performed for *queue*4060 and *queue*60+. After this, the desired success rate is lowered to 25%. The strategy now continues to search in a queue until less than 25% of the tried paths had no cycles and were locally optimal, given that at least 4 paths had been tried. After having searched in all queues, the success rate is lowered again to 12.5% and so on. Either way, the algorithm stops when max local optimality tests have been performed.

5.11 Improved algorithm for finding dissimilar paths: outline

In the previous sections all the necessary components were described, which can now be brought together in our improved algorithm for finding dissimilar paths. The pseudocode is shown in Algorithm 5.4 and Algorithm 5.5. First we describe the input needed from the user:

- the start node s and the target node t for a given road network
- the number of desired paths k
- the parameter for local optimality α
- the maximum number of LO tests max

5.11.1 Phase 1: Generate shortest path trees

A forward shortest path tree SPT_{OUT} is calculated using the algorithm of Dijkstra from the start node s and a backward shortest path tree SPT_{IN}

towards the target node t. Both Dijkstra searches are pruned at $1.25 \times w(SP)$ since it was determined in Section 5.7.3 that this is a good pruning factor. The shortest path weight $w(SP)$, which needs to be known for pruning, becomes clear during the execution of the algorithm of Dijkstra. Once the shortest path weight is known, the value for $T = \alpha \times w(SP)$ can also be calculated.

5.11.2 Phase 2: Generate plateaus.

Plateaus are generated using Algorithm 5.3, using SPT_{OUT} and SPT_{IN} as input.

5.11.3 Phase 3: Partition the plateaus.

The plateaus are partitioned into a set of plateaus $\geq T$ and 4 queues of plateaus $< T$ depending on their corresponding path weight. This is described in Section 5.10.2.

5.11.4 Phase 4: Local optimality testing and p-dispersion

The algorithm examines plateaus shorter than T in the order described in Section 5.10.2. Each plateau is converted to its correspondig path. This conversion can easily be done by looking up the shortest path from s to the plateau in SPT_{OUT} and the shortest path from the plateau to t in SPT_{IN}. The path is tested for cycles and, if it contains no cycles, is tested for local optimality. For local optimality testing, the exact method around a plateau is used (see Section 5.9.4). It was chosen rather than the T-test because our experiments which are described in Section 5.9.6 showed that the T-test is not faster and misses many locally optimal paths. Cycle-free paths which are locally optimal are added to a candidate set C. This continues until max paths have been tested for local optimality or until there are no more plateaus.

Algorithm 5.4 Improved algorithm for finding dissimilar paths

Require: start s, target t, number of desired paths k, parameter for local optimality α, maximum number of LO tests max, road network
Ensure: *solution* containing k paths

```
1: SPT_OUT ← run forward Dijkstra from s (prune at 1.25 × w(SP))
2: SPT_IN ← run backward Dijkstra from t (prune at 1.25 × w(SP))
3: T ← α × w(SP)
4: Generate plateaus using Algorithm 5.3
5: Add all plateaus Pl such that w(Pl) ≥ T to C
6: Partition other plateaus in queue0020, queue2040, queue4060, queue60+
7: For each queue, sort plateaus in order of ascending path weight.
8: nrlotests ← 0
9: stopratio ← 0.5
10: currentqueue ← queue0020
11: while nrlotests < max do
12:    add_paths_to_C(nrlotests, stopratio, currentqueue, max, C , T)
13:    currentqueue ← next queue (circular)
14:    if currentqueue is queue0020 then
15:       stopratio ← stopratio/2
16:    end if
17: end while
18: if C.size ≥ k then
19:    solution ← select dissimilar paths in C using Algorithm 5.1
20: end if
```

Algorithm 5.5 add_paths_to_C

Require: $nrlotests$, $stopratio$, $currentqueue$, max, C, T
Ensure: paths added to C, if any are found

```
1: nrtried ← 0
2: nrsuccess ← 0
3: while nrlotests < max and currentqueue not empty and (nrtried <
   1/stopratio or nrsuccess/nrtried >= stopratio) do
4:    nrtried ← nrtried + 1
5:    plateau ← currentqueue.poll()
6:    path ← create path from plateau
7:    if path contains no cycles then
8:       nrlotests ← nrlotests + 1
9:       if path is T-locally optimal then
10:         add path to C
11:         nrsuccess ← nrsuccess + 1
12:      end if
13:   end if
14: end while
```

Finally, the p-dispersion method is called to select a dissimilar set of paths, given that the number of available paths in C is at least k. For selecting a subset of dissimilar paths, a greedy construction heuristic for the p-dispersion problem is used which always includes the shortest path in the solution. The pseudocode can be seen in Algorithm 5.1. The objective function used by the heuristic is the objective function described in Section 5.8

5.12 Experiments

5.12.1 Experiment: choosing a stop condition

Experimental setup

We have designed an experiment to evaluate the performance of the algorithm for different stop conditions. We have performed the experiments on five road networks. We have chosen 16 different stop conditions, i.e. 16 different values for max ranging from 0 to $+\infty$. For each road network, we have selected 100 random $s - t$ queries. For each start-target pair, the algorithm was run 32 times: twice for each of the 16 stop conditions, once where the goal is to find a solution consisting of 3 paths (i.e. $k = 3$) and once for five paths (i.e. $k = 5$). The value of α was set to 25%. Table 5.7 shows the results for two road networks, but the results for the other road networks are similar.
For each stop condition, the average running time and the average quality of the found solution for the 100 start-target pairs is displayed. Furthermore the tables show how many of the 100 start-target pairs have a better solution compared to the previous stop condition. The average size of the improvement, if there is one, is also shown. Improvements of size 0 are not included in this average. For example, in Table 5.7b, when the algorithm is run with $max = 40$, 35 out of 100 start-target pairs have a better solution than when $max = 20$ is used. The average size of this improvement is

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	164	0.758			3
5	221	0.769	36	0.047	0
10	273	0.778	19	0.049	0
15	302	0.781	8	0.038	0
20	348	0.782	5	0.021	0
40	536	0.788	16	0.033	0
60	645	0.788	3	0.028	0
80	754	0.789	3	0.019	0
100	833	0.790	2	0.054	0
120	984	0.790	1	0.010	0
140	1 089	0.790	1	ϵ	0
160	1 202	0.790	0	/	0
180	1 144	0.790	0	/	0
200	1 312	0.790	1	0.007	0
500	2 700	0.792	3	0.051	0
$+\infty$	5 738	0.792	2	0.009	0

(a) NAVTEQ_LUXEMBOURG, $k = 3$

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	115	0.662			17
5	144	0.674	42	0.041	2
10	190	0.685	34	0.040	1
15	210	0.690	18	0.025	1
20	287	0.694	15	0.028	1
40	356	0.703	35	0.025	1
60	454	0.706	12	0.024	1
80	517	0.708	11	0.017	1
100	733	0.710	11	0.019	1
120	751	0.711	5	0.021	1
140	919	0.712	4	0.017	1
160	891	0.712	3	0.024	1
180	1 007	0.712	0	/	1
200	1 040	0.713	4	0.020	1
500	2 173	0.715	11	0.019	1
$+\infty$	5 574	0.716	5	0.024	1

(b) NAVTEQ_LUXEMBOURG, $k = 5$

Table 5.7: Results for different stop conditions with $\alpha = 25\%$. The average running time and average quality $Q(S)$ of the result is shown. The number of queries (out of 100) for which an improvement was found is shown, and the average size of this improvement if there is one. The last column shows the number of queries (out of 100) for which no solution was found.

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	2 705	0.748			0
5	3 698	0.758	23	0.044	0
10	4 612	0.760	12	0.017	0
15	5 497	0.763	17	0.020	0
20	5 958	0.764	4	0.020	0
40	9 123	0.767	14	0.019	0
60	11 209	0.769	7	0.028	0
80	13 215	0.770	6	0.023	0
100	14 680	0.770	0	/	0
120	16 540	0.771	2	0.041	0
140	17 588	0.771	2	0.011	0
160	18 411	0.771	0	/	0
180	19 992	0.771	0	/	0
200	20 799	0.771	1	0.011	0
500	37 353	0.774	11	0.024	0
$+\infty$	702 773	0.777	11	0.026	0

(c) NAVTEQ_BELGIUM, $k = 3$

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	1 647	0.658			4
5	2 329	0.671	39	0.038	1
10	3 008	0.678	27	0.024	0
15	3 433	0.686	31	0.027	0
20	4 015	0.688	14	0.017	0
40	6 292	0.695	23	0.029	0
60	8 283	0.699	17	0.027	0
80	10 086	0.701	12	0.014	0
100	11 902	0.702	6	0.010	0
120	12 972	0.703	4	0.028	0
140	13 774	0.703	3	0.005	0
160	15 238	0.703	2	0.024	0
180	16 275	0.703	1	0.001	0
200	17 686	0.704	5	0.012	0
500	32 761	0.707	23	0.015	0
$+\infty$	625 568	0.712	29	0.016	0

(d) NAVTEQ_BELGIUM, $k = 5$

Table 5.7: Results for different stop conditions with $\alpha = 25\%$. The average running time and average quality $Q(S)$ of the result is shown. The number of queries (out of 100) for which an improvement was found is shown, and the average size of this improvement if there is one. The last column shows the number of queries (out of 100) for which no solution was found. (Continued)

0.025. Finally, the column "no solution" shows for how many of the 100 start-target pairs no solution was found. This occurs when at the end of the algorithm, the number of locally optimal cycle free paths is still less than k. In this case it is impossible to return a solution. For example, in the same table, no solution was found for 2 out of 100 start-target pairs for $max = 5$.

Discussion

The results clearly show that it is worth it to continue searching after applying the plateau method (i.e. $max = 0$). Especially for small graphs and for $k = 5$, the plateau method is likely not to find a solution at all. This is because the number of plateaus which generate cycle-free locally optimal paths is less than k. It is clear that trying even a small amount of extra plateaus significantly increases the probability of finding a solution. E.g. in Table 5.7b, the plateau method only finds a solution for 83 of the 100 queries. By trying 5 more paths, a solution can be found for 98 queries, and by trying 10 more paths a solution can be found for 99 queries. The algorithm still fails to find a solution for one query, but this is because there are simply not enough plateaus which lead to a good solution. Even checking every single plateau does not lead to a solution.
The results also show clear improvement in the quality of the solution by trying more plateaus. E.g. in Table 5.7d the quality is increased from 0.658 to 0.686 by trying 15 more paths. By trying every plateau, the solution even increases to 0.712. For the large road network NAVTEQ_BELGIUM the algorithm runs in just a few seconds for the smaller values of max. However, even for these smaller values, solutions of good quality are found. These running times are definitely acceptable, since the algorithm was tested on a 2.8 GHz CPU, a CPU which can be found in a basic home computer nowadays. For the smaller road network NAVTEQ_LUXEMBOURG, the algorithm is even faster and runs within less than 1 second for most of the tested values of max. For a smaller road network like this, one could even consider trying every plateau if quality of the solution is important, since

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	144	0.717	35		0
5	168	0.719	11	0.018	0
10	174	0.720	7	0.011	0
15	178	0.721	6	0.022	0
20	194	0.723	7	0.031	0
40	225	0.725	14	0.016	0
60	271	0.727	8	0.026	0
80	292	0.728	11	0.008	0
100	353	0.730	7	0.021	0
120	401	0.731	6	0.015	0
140	405	0.731	3	0.006	0
160	467	0.731	3	0.007	0
180	516	0.731	4	0.011	0
200	571	0.731	1	0.004	0
500	1 283	0.734	17	0.014	0
$+\infty$	3 178	0.735	9	0.017	0

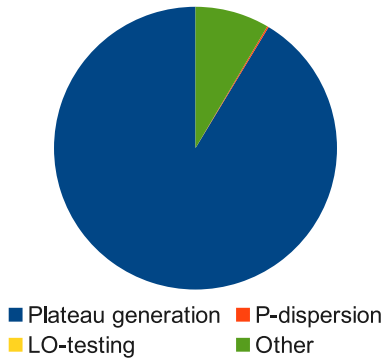
(a) NAVTEQ_LUXEMBOURG, $k = 5$

Table 5.8: Results for different stop conditions with $\alpha = 10\%$. The average running time and average quality $Q(S)$ of the result is shown. The number of queries (out of 100) for which an improvement was found is shown, and the average size of this improvement if there is one. The last column shows the number of queries (out of 100) for which no solution was found.

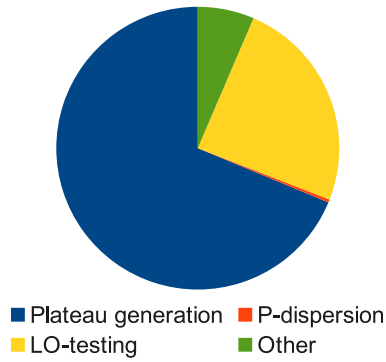
even this takes less than six seconds on average.
Clearly, there is a trade-off between running time and quality of the solution. When speed is most important, $max = 5$ could be a good stop condition since the probability of finding a solution is much higher than for $max = 0$, the quality of the solution is better, and the solution can still be found very fast. When quality of the solution and the guarantee of finding a solution are more important, max could be set to a value of 100. As a general rule of thumb, we suggest using $max = 15$ as a stop condition, since the quality of the solution is clearly much better than for $max = 0$, without compromising too much on speed.
Finally, Table 5.8 shows results for the same experiment, but where $\alpha = 10\%$ instead of 25%. In this case, the results show that the improvement is smaller. In all of our experiments a solution could be found for $max = 0$ and the quality cannot be improved as much by trying more plateaus than for $\alpha = 25\%$. This can be expected since there are a lot more plateaus with weight at least $\alpha \times w(SP)$ for $\alpha = 10\%$. However, as we have shown before, 25% proves to be a better value for α.

5.12.2 Experiment: how is the running time divided over different phases of the algorithm?

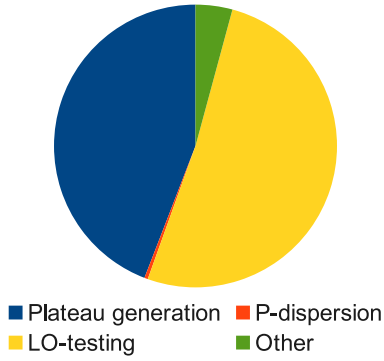
Figure 5.25 shows how much time the algorithm spends on the generation of the plateaus, testing local optimality, the p-dispersion heuristic and other aspects. Of course, for $max = 0$, no time is spent on local optimality testing. Most of the time is spent on the generation of the plateaus. However, as max increases, local optimality quickly takes over as the biggest bottleneck of the algorithm. This confirms the importance of an efficient strategy for testing local optimality. The algorithm definitely benefits from checking local optimality around a plateau (see Section 5.9.4) rather than using a brute force algorithm. It is also clear that the time spent on the p-dispersion heuristic is so small that it can be neglected. Therefore, efforts to further optimize the speed of the algorithm should definitely be focused on local optimality testing and not on the p-dispersion heuristic.



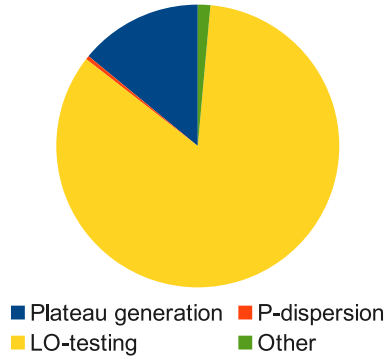
(a) $max = 0$ (total time: 2 705 ms)



(b) $max = 5$ (total time: 3 698 ms)



(c) $max = 15$ (total time: 5 497 ms)



(d) $max = 100$ (total time: 14 680 ms)

Figure 5.25: Division of the running time over the different phases of the algorithm. Running times are averaged for 100 random queries for NAVTEQ_BELGIUM with $k = 3$ and different stop conditions.

5.13 Comparison to results in the literature

Abraham et al. [11] present results in terms of quality of the solution as well as running time for their heuristics which aim to find locally optimal paths. Akgün et al. [12] present experimental results as well. However, these methods cannot be compared to our method since they do not aim for local optimality, since they were proposed before the concept of local optimality was introduced. All of the paths found by Abraham et al. are locally optimal for at least $\alpha = 29\%$, which is similar to the 25% used in our experiments. However, an important difference between the method by Abraham et al. and our method is the fact that Abraham et al. use hard constraints on the weight of the paths and the dissimilarity of the paths. Paths which are more than 25% longer than the shortest path are eliminated and a solution may not contain paths which share more than 80%. Our method aims to avoid such hard constraints. A path which is 26% longer than the shortest path, but very dissimilar from the other paths, can be a better choice than a path which is only 21% longer than the shortest path but which is very similar to another path in the solution. Also, these hard constraints can cause failure in finding a solution at all, while it would have been interesting if the best possible solution was still returned. Therefore, we have proposed an objective function which takes the trade-off between dissimilarity and path weight into account, and aims to optimize both without imposing hard constraints. Another difference is the fact that we use an exact method for local optimality testing, while Abraham et al. use the T-test which, as we have shown, is not faster and fails to find about half of the locally optimal paths.
Abraham et al. evaluate 3 methods which are named X-BDV, X-REV and X-CHV. The methods X-REV and X-CHV are very fast but also have very disappointing success rates, e.g. for $k = 4$ the X-CHV method can only find a solution in 10.9% of the cases. The X-BDV method is a lot more time-consuming and has a better success rate, but still only finds a solution in 81.1% of the cases for $k = 3$, while our algorithm had a success rate of 100% for $k = 3$ in our experiments. We believe that this lower success

rate can be explained on the one hand by the use of hard constraints, and on the other hand by the fact that only plateaus Pl such that $w(Pl) \geq T$ are considered while there are not always enough such plateaus to find a solution, and if a solution is found, it is suboptimal and can be improved by considering the other plateaus as well.

6

Concluding remarks

With the ever-growing interest in interactive route planning applications, mobile navigation devices and digital GIS systems, there is also a growing demand for very fast algorithms which are optimised for use on road networks. While many shortest path algorithms exist, there is a need to adapt these algorithms to the demands of the user. We have seen that, even with the increasing speed of computer systems nowadays, exact algorithms are often still too time-consuming for use on large road networks. This justifies looking for heuristic methods which are much faster. While these heuristic methods do not guarantee to find the optimal solution, we have shown that results of good quality can be found using heuristic methods. Each algorithm was developed while constantly comparing our implementation to other methods of our own and existing methods and we have performed many experiments to ensure both the efficiency on realistic road networks and the good quality of the results.

A first issue which is not addressed by standard shortest path algorithms, is the fact that realistic road networks usually contain illegal turns and turns which imply an additional cost. In Chapter 3 we have described three methods for routing with turn restrictions. Two of these, the line graph method and the node splitting method, make use of a graph transformation, after which any shortest path algorithm can be applied to the transformed graph. A third method, the direct method, can immediately be applied to the original graph. Different road networks may have different amounts of illegal turns and available turn costs. We have shown that the nature of the road network has a significant influence on the performance of the algorithms. Our experiments showed that for road networks where less than 5% of the turns are forbidden, node splitting is the best method. If more than 5% of the turns are forbidden, which is very rare, the direct method is the best choice. For road networks with turn costs, we concluded that the direct method performs best if no more than 25% of the turns have a non-zero cost, while the line graph method is a better choice otherwise. We investigated the possibility of storing additional data in a lookup table, but found that this is not worth the extra memory. No such study had been done before, and realistic route planning applications could benefit from this knowledge for choosing the method which is best suited for their specific road network.

In Chapter 4 we have presented a heuristic for the k shortest paths problem. We have described several exact algorithms for this problem, e.g. the algorithm of Yen, and performed experiments which showed that all of these exact algorithms are too slow for use in an interactive application. The heuristic is based on the algorithm of Yen. While the algorithm of Yen performs many shortest path calculations, the heuristic uses a precalculated backward shortest path tree for looking up these shortest paths. The shortest paths found in the shortest path tree may or may not lead to paths which contain a cycle. If such a cycle occurs, the path is simply dropped by the heuristic. Surprisingly, we have seen that this does not significantly affect the quality of the result. Our experiments have shown that the k-th path found by the heuristic is less than 5% longer than the actual k-th path

in 98.5% of all cases, which is definitely acceptable in road networks. The experiments have also shown that the heuristic is hundreds to even thousands of times faster than the fastest exact algorithm. Furthermore, the heuristic can be enhanced to an exact algorithm which is a valid alternative to the existing exact algorithms, with a speedup of about 10% to 20%.

One of the applications of k shortest paths algorithms is routing with additional constraints. A k shortest paths algorithm can be used to generate a ranking of shortest paths, from which the paths that best satisfy the other criteria are selected. Our heuristic can be very effective for speeding up these methods. One of the applications mentioned in the literature is the calculation of dissimilar alternative routes by selecting a small dissimilar subset from a large set of k shortest paths. Nevertheless, we have shown that a better set of paths can be generated by making use of *plateaus*. This is described in Chapter 5. We aim to find a small set of paths which contain no cycles and are dissimilar, not too long and locally optimal. First, we described our initial method which found paths of good quality, but spent too much time trying to improve the local optimality of the paths and had no guarantee that two dissimilar paths would not be transformed into the same path after improving local optimality. We have shown that these issues can be resolved by using plateaus. We can identify paths which are *guaranteed* to be locally optimal, and other paths which are *likely* to be locally optimal. This information is of great value since local optimality testing is a bottleneck in the algorithm. It allows us to find as many locally optimal paths as possible with as little local optimality testing as possible. Also, the paths containing longer plateaus are likely to be dissimilar and to contain no cycles. Even though paths can be found which are guaranteed to be locally optimal, we have shown that it is still useful to test some of the other paths for local optimality. By testing only 20 additional paths for local optimality, a solution can be found which is clearly of higher quality and the probability of not finding a solution can almost be reduced to 0 (given that a solution can be found by trying all plateaus).

We believe that our dissimilar paths heuristic is ready to be used in an interactive route planning application. Therefore, the next logical step would

be to build such a system. Currently, our software only allows the calculation of routes between two given node numbers. The software could be greatly improved by providing an interactive user interface. Users should be able to enter addresses or choose their starting point and destination by clicking on a map. For this purpose, the necessary conversions between addresses or coordinates and node numbers should be implemented. Another issue arises here due to the fact that not all available road networks contain address data. However, entering a query by clicking on a map should be possible for any road network for which coordinates are available.

Such an interactive routing application could also benefit from a user study to further fine-tune certain parameters, taking the preferences of the user into account. It would especially be of interest to find out what routes users consider to be good alternatives. Currently our dissimilar paths heuristic uses a 50%-50% trade-off between path weight and dissimilarity. By asking users to rate different sets of alternatives for the same query, we could discover the importance users attach to path weight compared to dissimilarity and choose a different value for this trade-off. It would also be interesting to find out if $\alpha = 25\%$ is indeed the value users prefer for local optimality.

Furthermore, the random turn costs should be replaced with real-life turn costs, which are currently not available. It is indeed not likely that all turns in a road network will be visited to discover the turn costs anytime soon. Therefore, realistic estimations should be made. The estimation method described by Nielsen [53] could be a good starting point. It would also be interesting to consider the possibility of deriving turn costs and, if necessary, turn prohibitions from GPS traces.

From an algorithmic point of view, the next logical step would be to include hierarchical routing in our algorithms. Hierarchical routing methods use different levels for representing a road network. At the highest level, the network is very sparse and only the most important nodes (e.g. highway nodes) are included. Typically, the more detailed levels are only used around the starting point and destination, while the higher levels are used in between. Hierarchical routing has proven to be very efficient for shortest path calculations. It would be interesting to examine whether a hierarchical

routing algorithm can be developed for the k shortest paths problem or for finding dissimilar paths. Furthermore, both our k shortest paths heuristic and our dissimilar paths heuristic make use of many shortest path calculations. Existing hierarchical routing methods could be used for speeding up these shortest path calculations. The dissimilar paths heuristic could especially benefit from even faster local optimality testing. Hierarchical routing should definitely be considered for this purpose.

Another challenge for the future will be to develop exact algorithms which can be run on large road networks. Even though our heuristics produce good results, it is still ambitious to find an optimal solution within a reasonable amount of time. Perhaps with improving hardware this will become possible. In this context it would also be interesting to consider parallel computing. For example, in our dissimilar paths algorithm a major speedup could be achieved by running the local optimality tests in parallel on different CPUs.

Finally, many other variants on standard shortest path algorithms are possible. Routing algorithms for public transportation networks (e.g. Huang [41]) are definitely very important as well. Also, algorithms which do not find a point-to-point route but a tour along certain streets (e.g. Corberán et al. [23]) are useful. Several ideas have been proposed and a lot of work remains to be done to combine all these ideas. For example, it would be interesting to find dissimilar alternatives for a tour along certain streets, or to combine the ideas from Chapter 4 and Chapter 5 to develop an algorithm which aims to find the k shortest locally optimal paths.

A

Supplementary material: Turn restrictions

In this Appendix additional results are presented for Chapter 3. The results are presented in tables showing the exact numbers instead of charts and more road networks are included.

Memory

Table A.1 shows additional results for Figure 3.11 on Page 53. The memory usage ratio is shown for different percentages of turn costs and turn prohibitions for the direct method (DIR), line graph without lookup table (LINE), node splitting without lookup table (SPLIT), line graph with lookup table (LINE*) and node splitting with lookup table (SPLIT*).

APPENDIX A. TURN RESTRICTIONS

Strict query time

Table A.2 shows additional results for Figure 3.12 on Page 55. The average strict query time ratio is shown for different percentages of turn costs and turn prohibitions for the direct method (DIR), the line graph (LINE) and node splitting (SPLIT). The average running time for a one-to-one query of the standard Dijkstra algorithm on the original graph is 254.49 ms.

Restriction	DIR	LINE	LINE*	SPLIT	SPLIT*
Turn prohibitions 0.1%	1.0	2.6	5.5	1.0	1.0
Turn prohibitions 0.5%	1.0	2.6	5.4	1.2	1.3
Turn prohibitions 1%	1.0	2.6	5.4	1.2	1.4
Turn prohibitions 5%	1.2	2.5	5.4	2.0	2.7
Turn prohibitions 10%	1.5	2.4	5.3	2.6	3.6
Turn prohibitions 15%	1.7	2.3	5.2	2.9	4.3
Turn prohibitions 20%	1.9	2.2	5.1	3.1	4.7
Turn prohibitions 25%	2.1	2.2	5.0	3.3	4.9
Turn costs 5%	1.2	2.6	5.5	2.1	2.7
Turn costs 10%	1.5	2.6	5.5	2.8	3.8
Turn costs 15%	1.7	2.6	5.5	3.2	4.6
Turn costs 20%	1.9	2.6	5.5	3.5	5.0
Turn costs 25%	2.1	2.6	5.5	3.7	5.4
Turn costs 50%	2.9	2.6	5.5	4.1	6.0
Turn costs 75%	3.6	2.6	5.5	4.2	6.1
Turn costs 100%	4.2	2.6	5.5	4.2	6.2

(a) CZE_MAX (original graph: 2.82 MB)

Restriction	DIR	LINE	LINE*	SPLIT	SPLIT*
Turn prohibitions 0.1%	1.0	2.6	5.5	1.0	1.0
Turn prohibitions 0.5%	1.0	2.6	5.5	1.1	1.2
Turn prohibitions 1%	1.0	2.6	5.4	1.2	1.4
Turn prohibitions 5%	1.2	2.5	5.4	2.0	2.6
Turn prohibitions 10%	1.5	2.4	5.3	2.6	3.7
Turn prohibitions 15%	1.7	2.4	5.2	2.9	4.2
Turn prohibitions 20%	1.9	2.3	5.1	3.1	4.6
Turn prohibitions 25%	2.1	2.2	5.0	3.2	4.8
Turn costs 5%	1.2	2.6	5.5	2.1	2.7
Turn costs 10%	1.5	2.6	5.5	2.8	3.9
Turn costs 15%	1.7	2.6	5.5	3.2	4.5
Turn costs 20%	1.9	2.6	5.5	3.5	5.0
Turn costs 25%	2.1	2.6	5.5	3.7	5.3
Turn costs 50%	2.9	2.6	5.5	4.1	6.1
Turn costs 75%	3.6	2.6	5.5	4.2	6.2
Turn costs 100%	4.2	2.6	5.5	4.3	6.3

(b) LUX_MAX (original graph: 3.70 MB)

Table A.1: Memory usage ratio for different road networks.

Restriction	DIR	LINE	LINE*	SPLIT	SPLIT*
Turn prohibitions 0.1%	1.0	2.5	5.4	1.0	1.0
Turn prohibitions 0.5%	1.0	2.5	5.4	1.1	1.2
Turn prohibitions 1%	1.0	2.6	5.4	1.2	1.4
Turn prohibitions 5%	1.2	2.4	5.4	1.9	2.5
Turn prohibitions 10%	1.4	2.4	5.3	2.5	3.6
Turn prohibitions 15%	1.7	2.3	5.2	2.8	4.1
Turn prohibitions 20%	1.9	2.2	5.1	3.0	4.5
Turn prohibitions 25%	2.0	2.1	5.0	3.1	4.7
Turn costs 5%	1.2	2.5	5.4	2.0	2.6
Turn costs 10%	1.4	2.5	5.4	2.7	3.7
Turn costs 15%	1.7	2.5	5.4	3.1	4.4
Turn costs 20%	1.9	2.5	5.4	3.4	4.8
Turn costs 25%	2.0	2.5	5.4	3.6	5.1
Turn costs 50%	2.9	2.5	5.4	4.0	5.9
Turn costs 75%	3.5	2.5	5.4	4.1	6.0
Turn costs 100%	4.1	2.5	5.4	4.2	6.2

(c) IRL_MAX (original graph: 3.87 MB)

Restriction	DIR	LINE	LINE*	SPLIT	SPLIT*
Turn prohibitions 0.1%	1.0	2.6	5.4	1.0	1.0
Turn prohibitions 0.5%	1.0	2.6	5.4	1.1	1.2
Turn prohibitions 1%	1.0	2.6	5.4	1.2	1.4
Turn prohibitions 5%	1.2	2.5	5.4	2.0	2.6
Turn prohibitions 10%	1.5	2.4	5.3	2.6	3.6
Turn prohibitions 15%	1.7	2.3	5.2	2.9	4.2
Turn prohibitions 20%	1.9	2.3	5.1	3.1	4.7
Turn prohibitions 25%	2.1	2.2	5.0	3.2	4.9
Turn costs 5%	1.2	2.6	5.5	2.1	2.7
Turn costs 10%	1.5	2.6	5.5	2.8	3.8
Turn costs 15%	1.7	2.6	5.5	3.2	4.5
Turn costs 20%	1.9	2.6	5.5	3.5	5.1
Turn costs 25%	2.1	2.6	5.5	3.7	5.4
Turn costs 50%	2.9	2.6	5.5	4.1	6.0
Turn costs 75%	3.6	2.6	5.5	4.2	6.2
Turn costs 100%	4.2	2.6	5.5	4.2	6.2

(d) PRT_MAX (original graph: 19.66 MB)

Table A.1: Memory usage ratio for different road networks. (Continued)

Restriction	DIR	LINE	LINE*	SPLIT	SPLIT*
Turn prohibitions 0.1%	1.0	2.7	5.5	1.0	1.0
Turn prohibitions 0.5%	1.0	2.7	5.5	1.1	1.2
Turn prohibitions 1%	1.0	2.7	5.5	1.2	1.4
Turn prohibitions 5%	1.2	2.6	5.4	2.0	2.7
Turn prohibitions 10%	1.5	2.5	5.3	2.6	3.7
Turn prohibitions 15%	1.7	2.4	5.2	3.0	4.4
Turn prohibitions 20%	1.9	2.3	5.1	3.2	4.7
Turn prohibitions 25%	2.1	2.2	5.0	3.3	4.9
Turn costs 5%	1.2	2.7	5.5	2.1	2.8
Turn costs 10%	1.5	2.7	5.5	2.8	3.9
Turn costs 15%	1.7	2.7	5.5	3.3	4.7
Turn costs 20%	1.9	2.7	5.5	3.6	5.1
Turn costs 25%	2.1	2.7	5.5	3.8	5.4
Turn costs 50%	3.0	2.7	5.5	4.1	6.1
Turn costs 75%	3.7	2.7	5.5	4.2	6.3
Turn costs 100%	4.2	2.7	5.5	4.3	6.3

(e) BEL_MAX (original graph: 57.72 MB)

Restriction	DIR	LINE	LINE*	SPLIT	SPLIT*
Turn prohibitions 0.1%	1.0	2.6	5.5	1.0	1.0
Turn prohibitions 0.5%	1.0	2.6	5.5	1.1	1.2
Turn prohibitions 1%	1.0	2.6	5.4	1.3	1.4
Turn prohibitions 5%	1.2	2.5	5.4	2.0	2.6
Turn prohibitions 10%	1.5	2.4	5.3	2.6	3.7
Turn prohibitions 15%	1.7	2.3	5.2	3.0	4.2
Turn prohibitions 20%	1.9	2.3	5.1	3.1	4.7
Turn prohibitions 25%	2.1	2.2	5.0	3.3	5.0
Turn costs 5%	1.2	2.6	5.5	2.1	2.7
Turn costs 10%	1.5	2.6	5.5	2.8	3.8
Turn costs 15%	1.7	2.6	5.5	3.2	4.5
Turn costs 20%	1.9	2.6	5.5	3.5	5.1
Turn costs 25%	2.1	2.6	5.5	3.7	5.5
Turn costs 50%	2.9	2.6	5.5	4.1	6.0
Turn costs 75%	3.6	2.6	5.5	4.2	6.2
Turn costs 100%	4.2	2.6	5.5	4.3	6.3

(f) CHE_MAX (original graph: 71.29 MB)

Table A.1: Memory usage ratio for different road networks. (Continued)

Restriction	DIR	LINE	SPLIT
Turn prohibitions 0.1%	2.0	2.4	1.0
Turn prohibitions 0.5%	1.9	2.2	1.0
Turn prohibitions 1.0%	1.9	2.2	1.1
Turn prohibitions 5%	1.9	2.3	2.0
Turn prohibitions 10%	1.9	2.2	2.6
Turn prohibitions 15%	1.7	1.8	2.7
Turn prohibitions 20%	1.4	1.5	2.4
Turn prohibitions 25%	0.3	0.4	0.6
Turn costs 5%	2.0	2.4	2.1
Turn costs 10%	2.1	2.4	3.0
Turn costs 15%	2.1	2.4	3.5
Turn costs 20%	2.2	2.5	3.8
Turn costs 25%	2.2	2.5	4.1
Turn costs 50%	2.4	2.5	4.7
Turn costs 75%	2.4	2.5	4.7
Turn costs 100%	2.3	2.5	4.7

(a) CZE_MAX (normal Dijkstra: 9.92 ms)

Restriction	DIR	LINE	SPLIT
Turn prohibitions 0.1%	2.2	2.2	1.0
Turn prohibitions 0.5%	2.2	2.4	1.1
Turn prohibitions 1.0%	2.2	2.4	1.3
Turn prohibitions 5%	2.0	2.4	2.1
Turn prohibitions 10%	2.1	2.3	2.8
Turn prohibitions 15%	2.0	2.2	3.0
Turn prohibitions 20%	1.6	1.7	2.6
Turn prohibitions 25%	0.2	0.2	0.3
Turn costs 5%	2.0	2.4	2.2
Turn costs 10%	2.1	2.5	3.0
Turn costs 15%	2.2	2.5	3.6
Turn costs 20%	2.3	2.5	4.1
Turn costs 25%	2.3	2.5	4.5
Turn costs 50%	2.4	2.6	5.3
Turn costs 75%	2.5	2.6	5.6
Turn costs 100%	2.5	2.5	5.6

(b) LUX_MAX (normal Dijkstra: 13.27 ms)

Table A.2: Strict query time ratio for different road networks.

Restriction	DIR	LINE	SPLIT
Turn prohibitions 0.1%	2.0	2.4	1.0
Turn prohibitions 0.5%	2.0	2.4	1.1
Turn prohibitions 1.0%	2.0	2.4	1.3
Turn prohibitions 5%	1.9	2.2	2.0
Turn prohibitions 10%	1.9	2.2	2.7
Turn prohibitions 15%	1.8	2.0	2.8
Turn prohibitions 20%	1.0	1.1	1.8
Turn prohibitions 25%	0.4	0.5	0.8
Turn costs 5%	2.0	2.4	2.1
Turn costs 10%	2.0	2.4	2.9
Turn costs 15%	2.1	2.4	3.4
Turn costs 20%	2.2	2.4	3.8
Turn costs 25%	2.2	2.5	4.2
Turn costs 50%	2.4	2.5	5.2
Turn costs 75%	2.4	2.5	5.5
Turn costs 100%	2.4	2.5	5.8

(c) IRL_MAX (normal Dijkstra: 12.27 ms)

Restriction	DIR	LINE	SPLIT
Turn prohibitions 0.1%	2.2	2.4	1.0
Turn prohibitions 0.5%	2.2	2.4	1.1
Turn prohibitions 1.0%	2.2	2.4	1.2
Turn prohibitions 5%	2.1	2.9	2.5
Turn prohibitions 10%	2.0	2.6	3.4
Turn prohibitions 15%	1.8	2.2	3.6
Turn prohibitions 20%	1.1	1.4	2.5
Turn prohibitions 25%	0.3	0.5	0.8
Turn costs 5%	2.2	3.0	2.7
Turn costs 10%	2.3	3.1	4.0
Turn costs 15%	2.3	3.1	4.8
Turn costs 20%	2.4	3.1	5.5
Turn costs 25%	2.4	3.1	6.0
Turn costs 50%	2.6	3.2	6.8
Turn costs 75%	2.7	3.2	7.2
Turn costs 100%	2.7	3.3	7.2

(d) PRT_MAX (normal Dijkstra: 87.04 ms)

Table A.2: Strict query time ratio for different road networks. (Continued)

Restriction	DIR	LINE	SPLIT
Turn prohibitions 0.1%	2.3	2.9	1.0
Turn prohibitions 0.5%	2.4	2.9	1.2
Turn prohibitions 1.0%	2.4	2.8	1.3
Turn prohibitions 5%	2.3	2.9	2.7
Turn prohibitions 10%	2.5	2.8	3.7
Turn prohibitions 15%	2.4	2.7	4.4
Turn prohibitions 20%	2.5	2.6	4.9
Turn prohibitions 25%	1.9	1.8	4.2
Turn costs 5%	2.6	3.0	2.8
Turn costs 10%	2.6	3.0	4.1
Turn costs 15%	2.7	3.1	5.0
Turn costs 20%	2.8	3.1	5.6
Turn costs 25%	2.8	3.1	6.1
Turn costs 50%	3.0	3.2	7.0
Turn costs 75%	3.1	3.2	7.1
Turn costs 100%	3.1	3.2	7.2

(e) BEL_MAX (normal Dijkstra: 254.49 ms)

Restriction	DIR	LINE	SPLIT
Turn prohibitions 0.1%	2.2	2.9	1.0
Turn prohibitions 0.5%	2.2	2.9	1.1
Turn prohibitions 1.0%	2.2	2.9	1.3
Turn prohibitions 5%	2.1	2.6	2.3
Turn prohibitions 10%	1.9	2.2	3.0
Turn prohibitions 15%	1.7	1.9	3.1
Turn prohibitions 20%	0.4	0.5	0.9
Turn prohibitions 25%	0.0	0.0	0.1
Turn costs 5%	2.2	2.8	2.6
Turn costs 10%	2.3	2.9	3.8
Turn costs 15%	2.3	2.9	4.6
Turn costs 20%	2.4	3.0	5.3
Turn costs 25%	2.4	3.0	5.6
Turn costs 50%	2.6	3.0	6.4
Turn costs 75%	2.7	3.0	6.6
Turn costs 100%	2.7	3.0	6.7

(f) CHE_MAX (normal Dijkstra: 328.00 ms)

Table A.2: Strict query time ratio for different road networks. (Continued)

B

Supplementary material: K shortest paths

In this Appendix additional results are presented for Chapter 4. The charts are similar to those in Chapter 4, but more road networks are included.

Path quality

Figures B.1, B.2, B.3, B.4 and B.5 show additional results for Figure 4.8 on Page 83.

Timing results

Figures B.6, B.7, B.8, B.9 and B.10 show additional results for Figure 4.9 on Page 91.

Detailed timing results

Figures B.11, B.12, B.13, B.14 and B.15 show additional results for Figure 4.10 on Page 93.

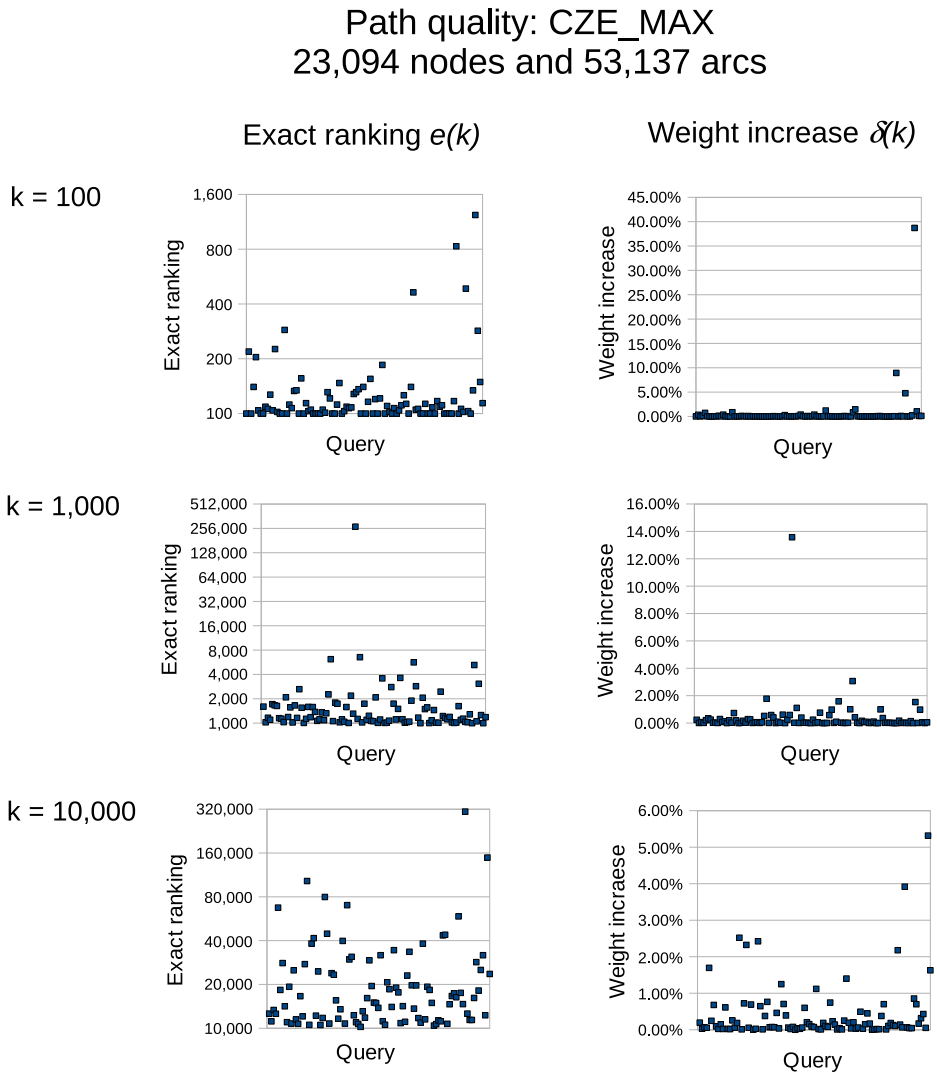


Figure B.1: Quality of the paths found by the heuristic for CZE_MAX. The exact ranking $e(k)$ and the weight increase $\delta(k)$ are shown for different values of k, each time for 100 random queries.

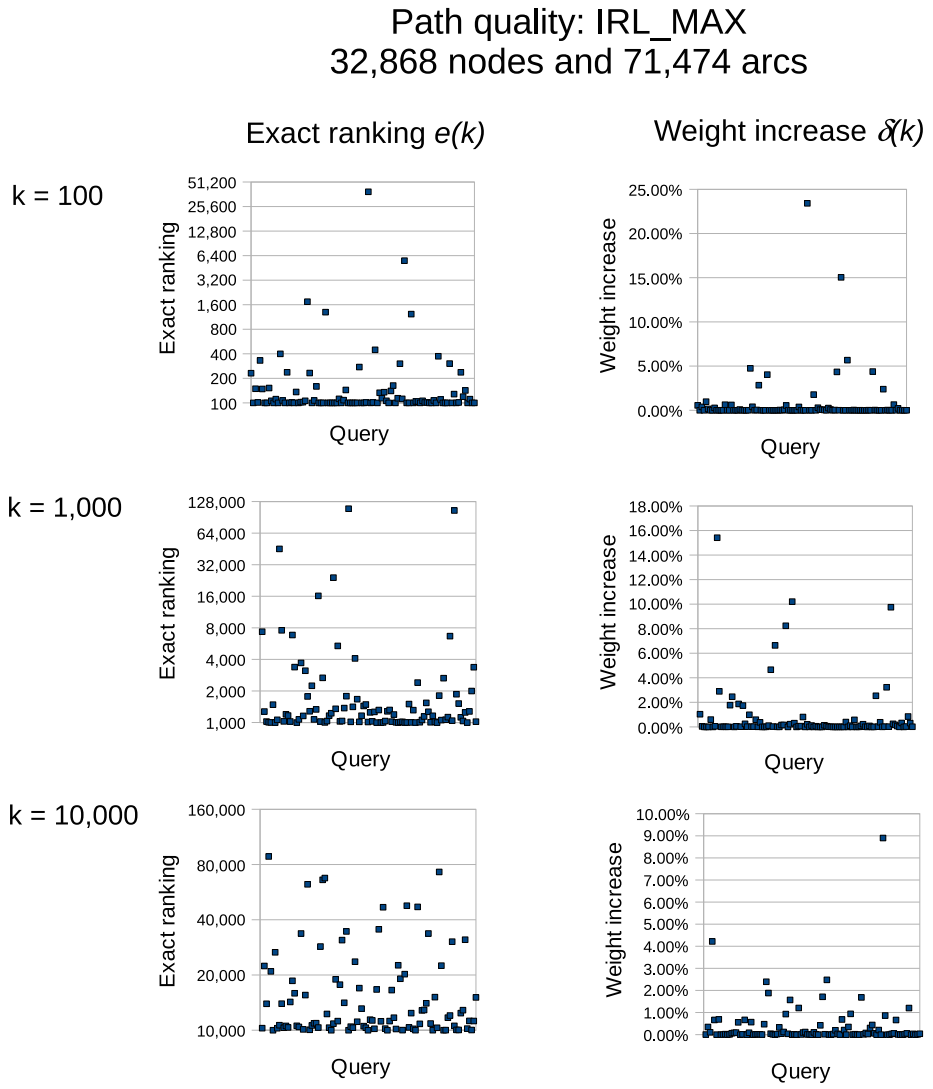


Figure B.2: Quality of the paths found by the heuristic for IRL_MAX. The exact ranking $e(k)$ and the weight increase $\delta(k)$ are shown for different values of k, each time for 100 random queries.

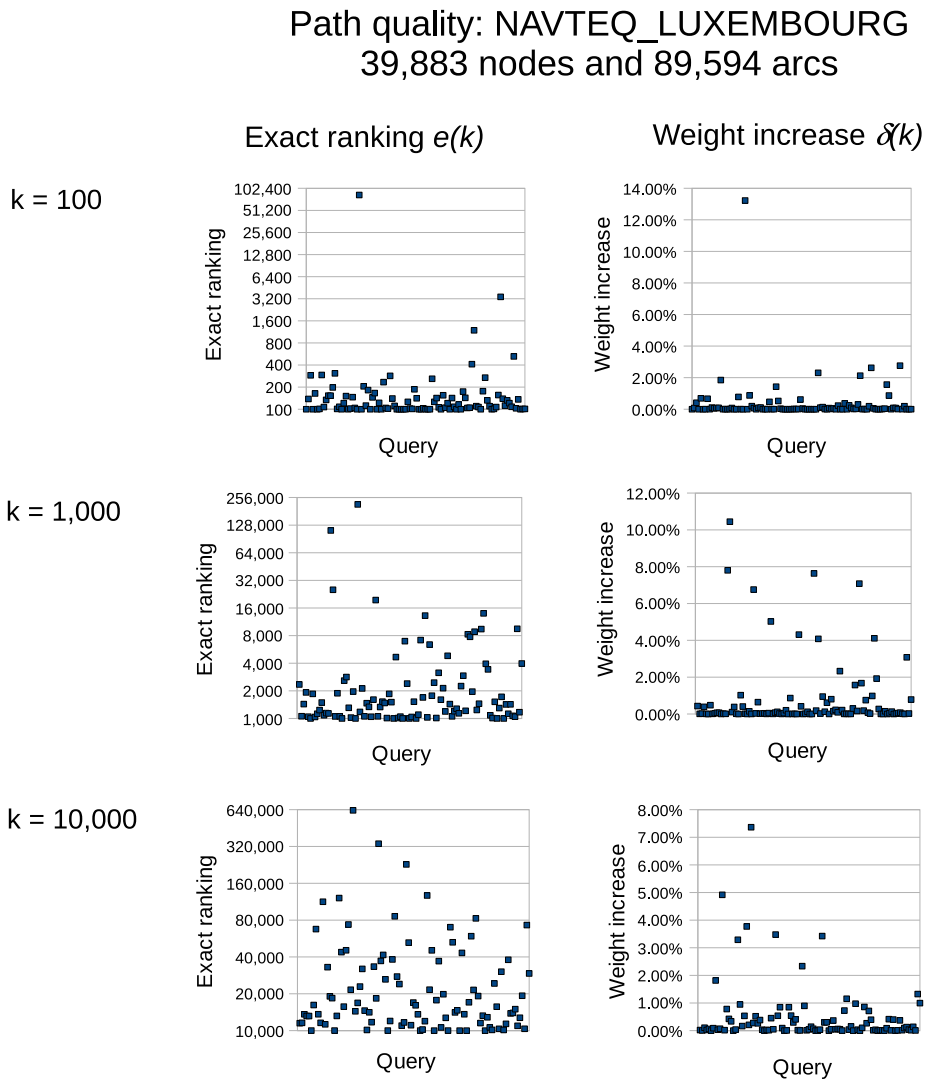


Figure B.3: Quality of the paths found by the heuristic for NAVTEQ_LUXEMBOURG. The exact ranking $e(k)$ and the weight increase $\delta(k)$ are shown for different values of k, each time for 100 random queries.

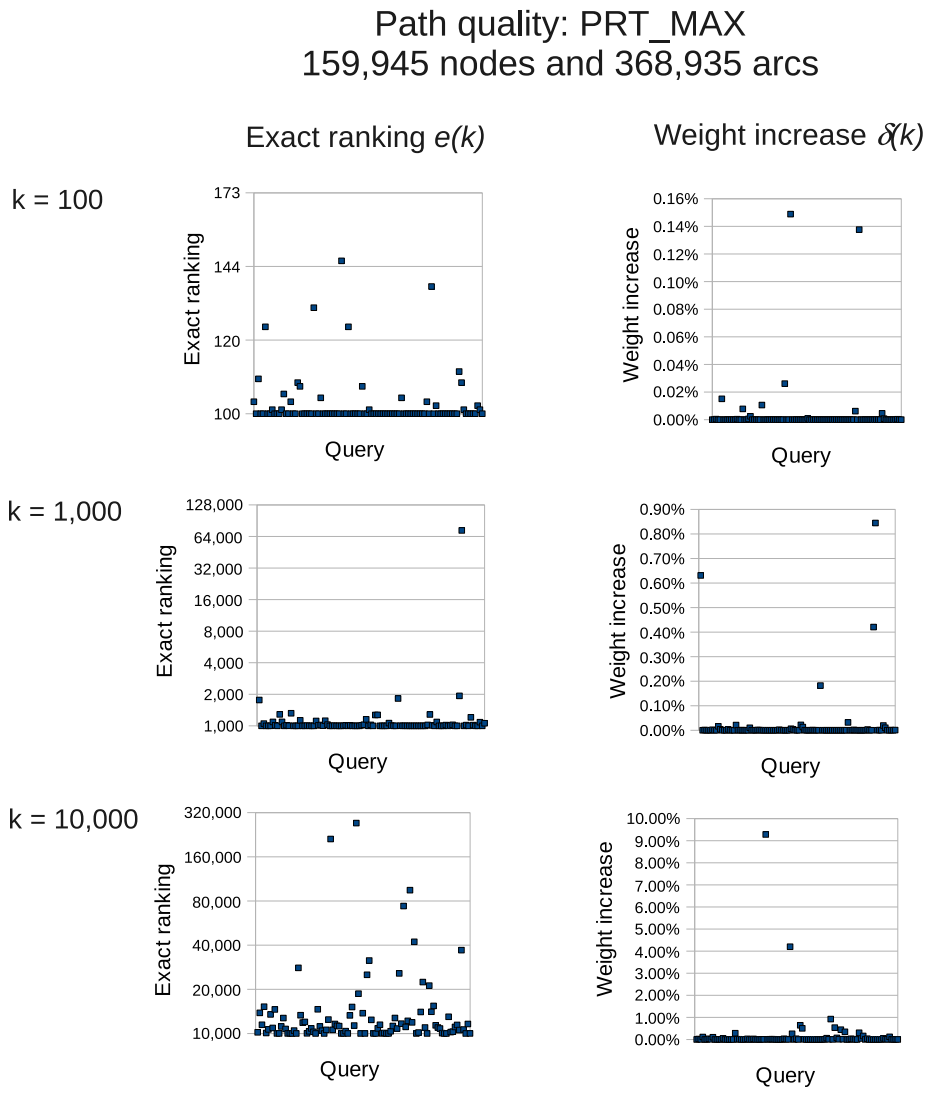


Figure B.4: Quality of the paths found by the heuristic for PRT_MAX. The exact ranking $e(k)$ and the weight increase $\delta(k)$ are shown for different values of k, each time for 100 random queries.

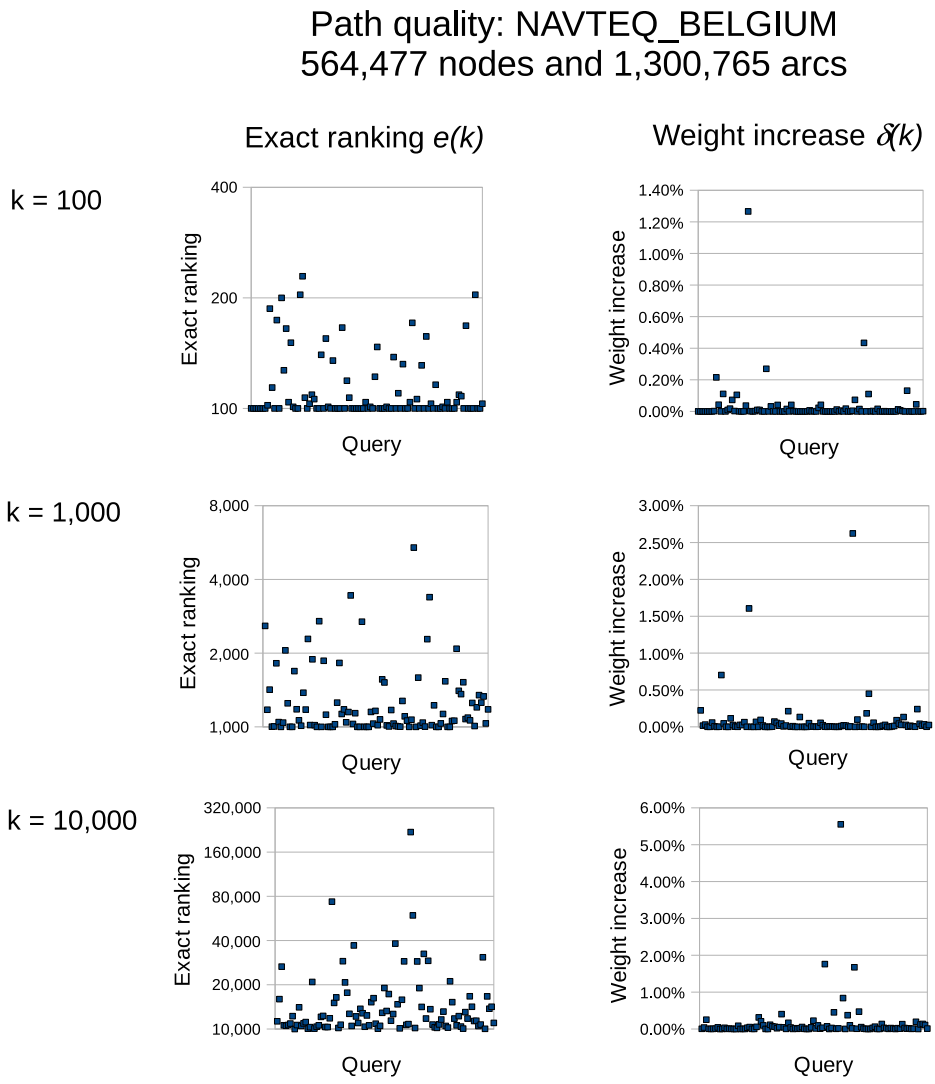
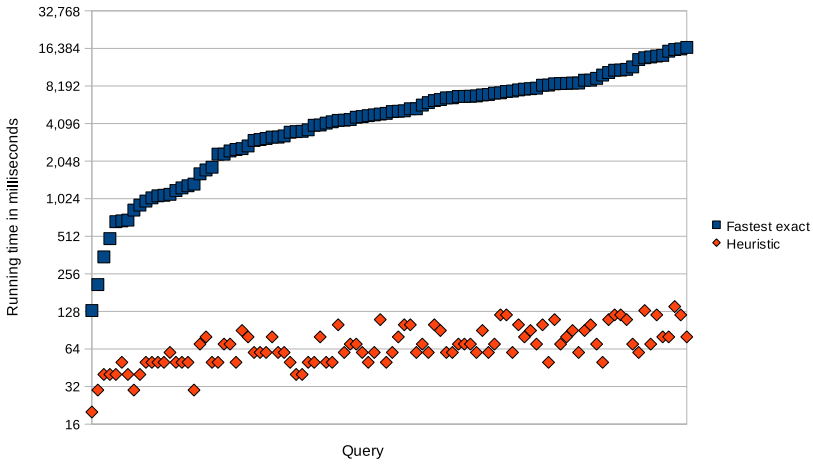
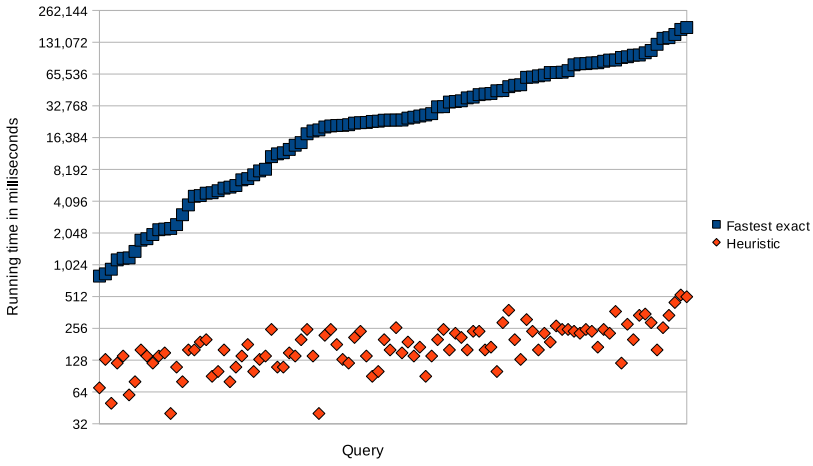


Figure B.5: Quality of the paths found by the heuristic for NAVTEQ_BELGIUM. The exact ranking $e(k)$ and the weight increase $\delta(k)$ are shown for different values of k, each time for 100 random queries.

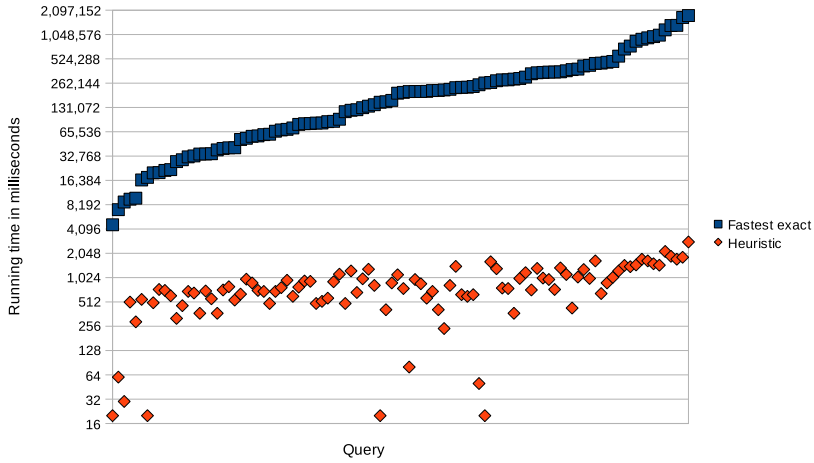


(a) CZE_MAX, $k = 100$



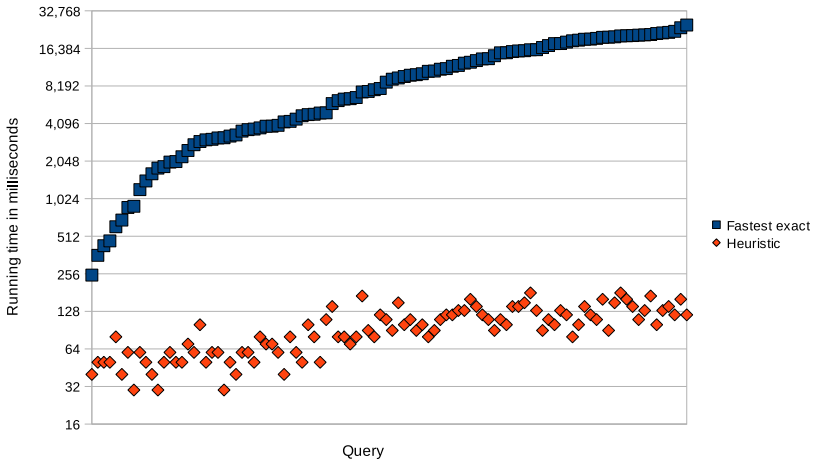
(b) CZE_MAX, $k = 1\,000$

Figure B.6: Time performance for CZE_MAX.

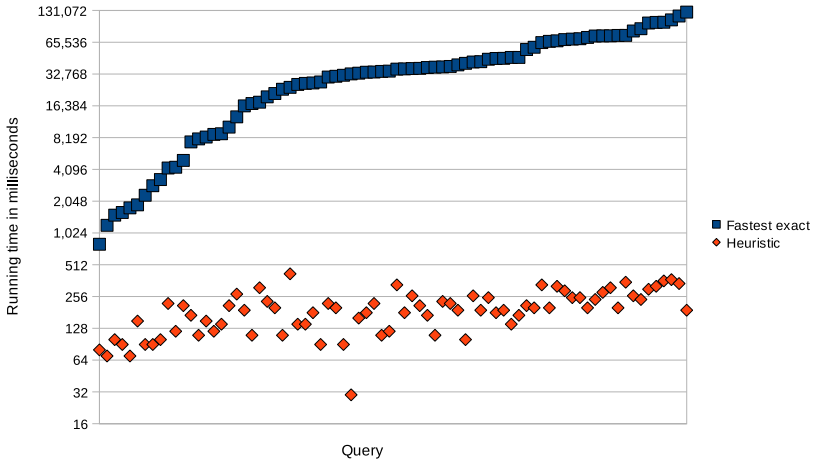


(c) CZE_MAX, $k = 10\,000$

Figure B.6: Time performance for CZE_MAX. For each k, the heuristic was compared, for 100 random queries, to three existing exact algorithms: Yen [70], Martins et al. [48] and Hershberger et al. [38]. Only the best of the running times for the exact algorithms is shown. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for the fastest exact algorithm. (Continued)

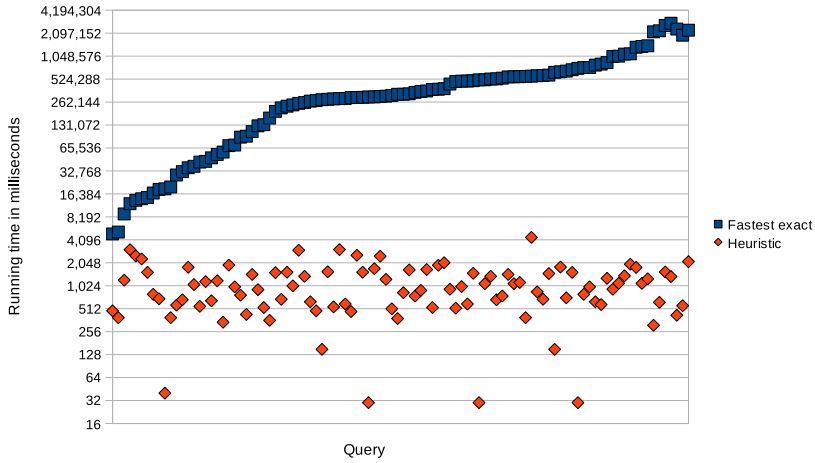


(a) IRL_MAX, $k = 100$



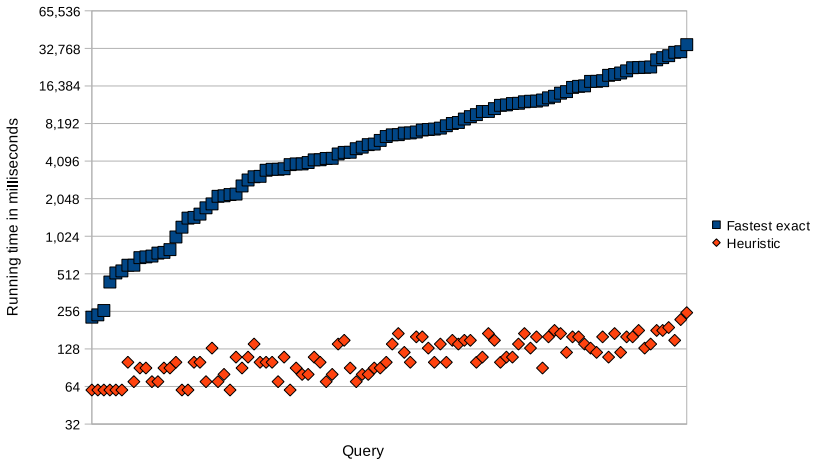
(b) IRL_MAX, $k = 1\,000$

Figure B.7: Time performance for IRL_MAX.

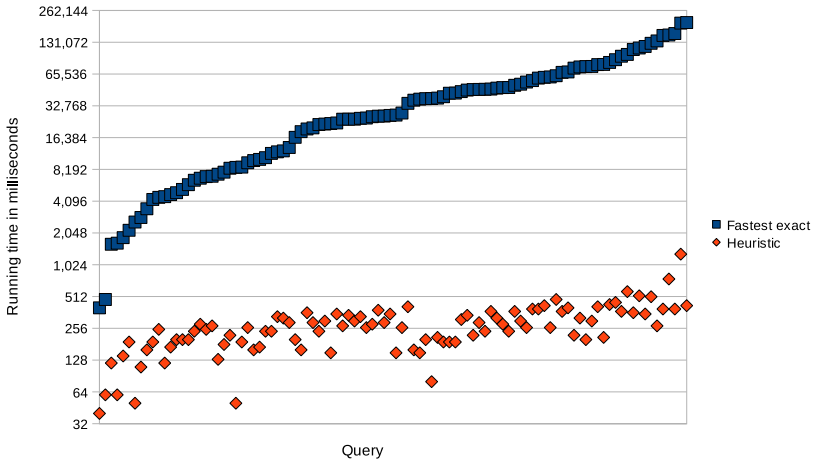


(c) IRL_MAX, $k = 10\,000$

Figure B.7: Time performance for IRL_MAX. For each k, the heuristic was compared, for 100 random queries, to three existing exact algorithms: Yen [70], Martins et al. [48] and Hershberger et al. [38]. Only the best of the running times for the exact algorithms is shown. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for the fastest exact algorithm. (Continued)

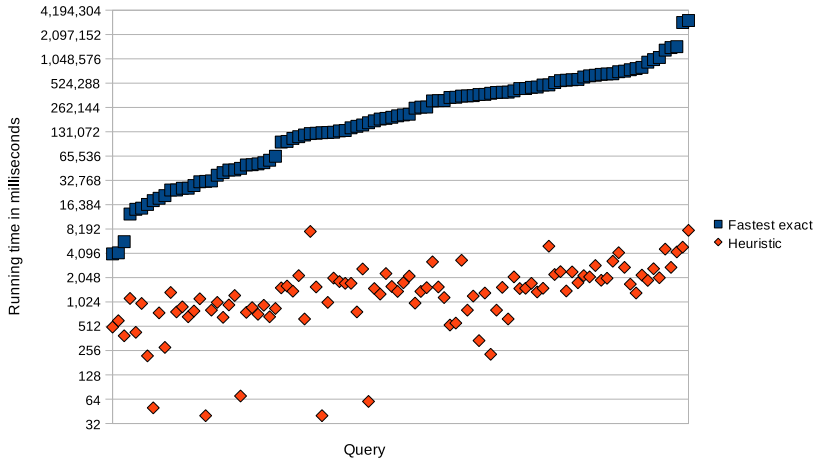


(a) NAVTEQ_LUXEMBOURG, $k = 100$



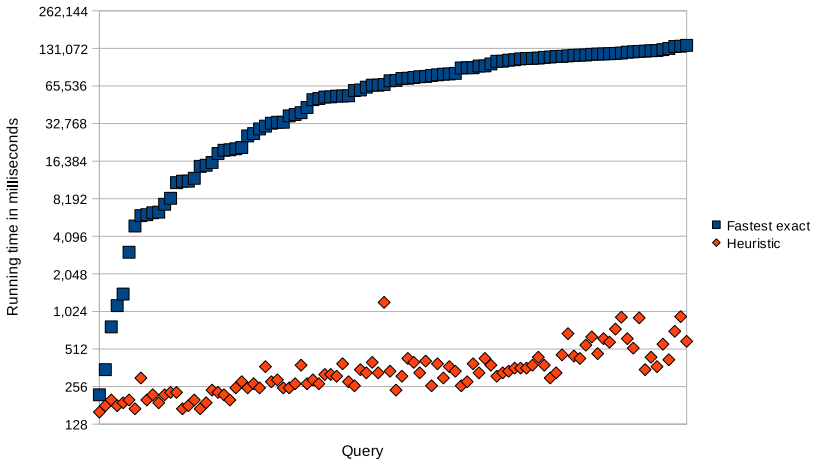
(b) NAVTEQ_LUXEMBOURG, $k = 1\,000$

Figure B.8: Time performance for NAVTEQ_LUXEMBOURG.

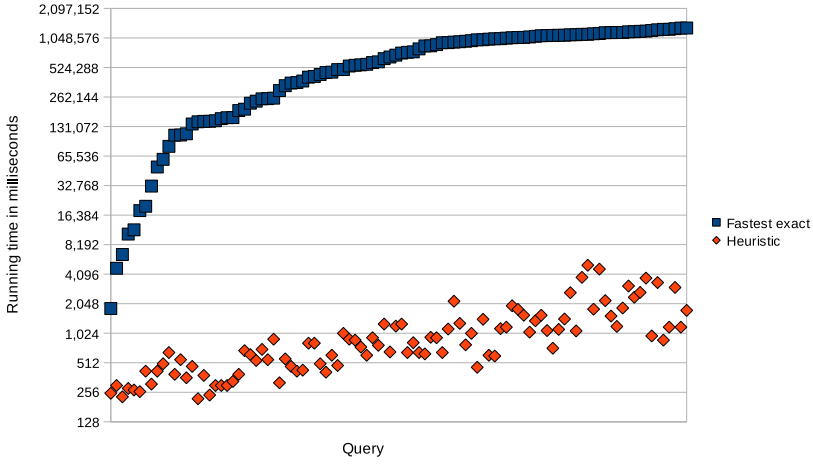


(c) NAVTEQ_LUXEMBOURG, $k = 10\,000$

Figure B.8: Time performance for NAVTEQ_LUXEMBOURG. For each k, the heuristic was compared, for 100 random queries, to three existing exact algorithms: Yen [70], Martins et al. [48] and Hershberger et al. [38]. Only the best of the running times for the exact algorithms is shown. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for the fastest exact algorithm. (Continued)

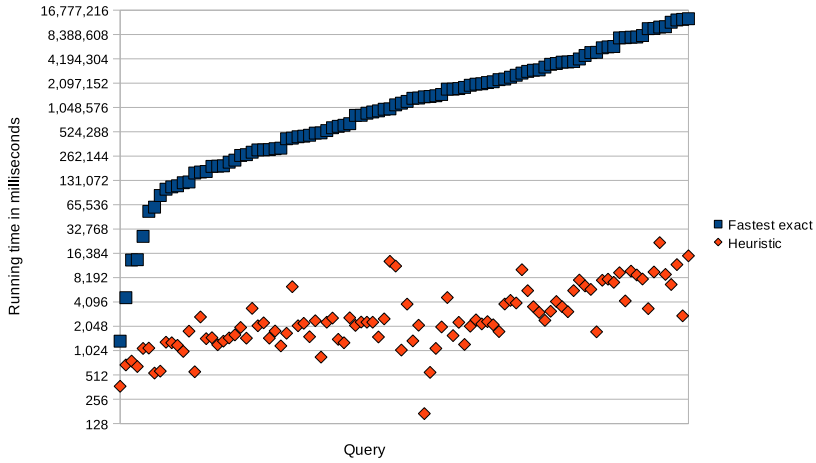


(a) PRT_MAX, $k = 100$



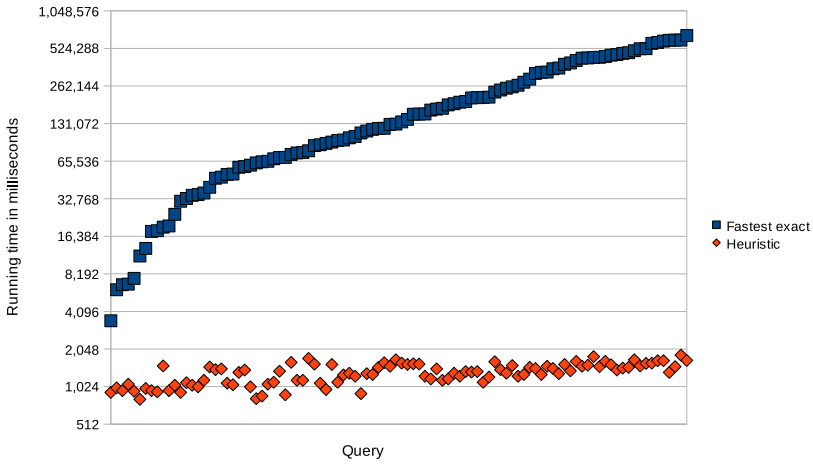
(b) PRT_MAX, $k = 1\,000$

Figure B.9: Time performance for PRT_MAX.

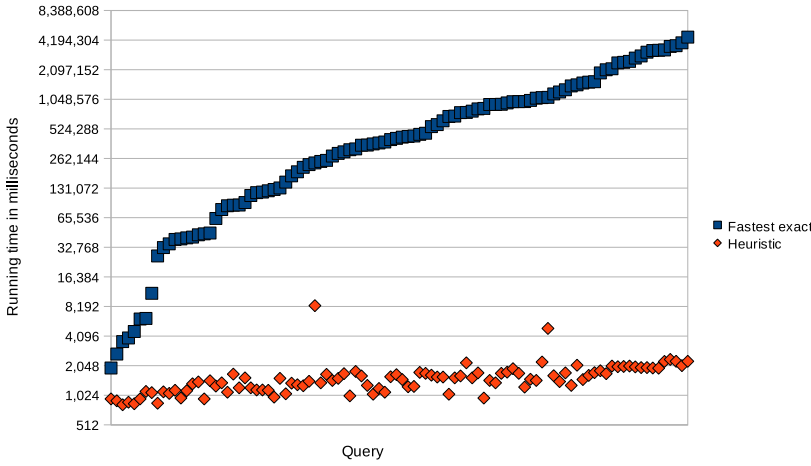


(c) PRT_MAX, $k = 10\,000$

Figure B.9: Time performance for PRT_MAX. For each k, the heuristic was compared, for 100 random queries, to three existing exact algorithms: Yen [70], Martins et al. [48] and Hershberger et al. [38]. Only the best of the running times for the exact algorithms is shown. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for the fastest exact algorithm. (Continued)

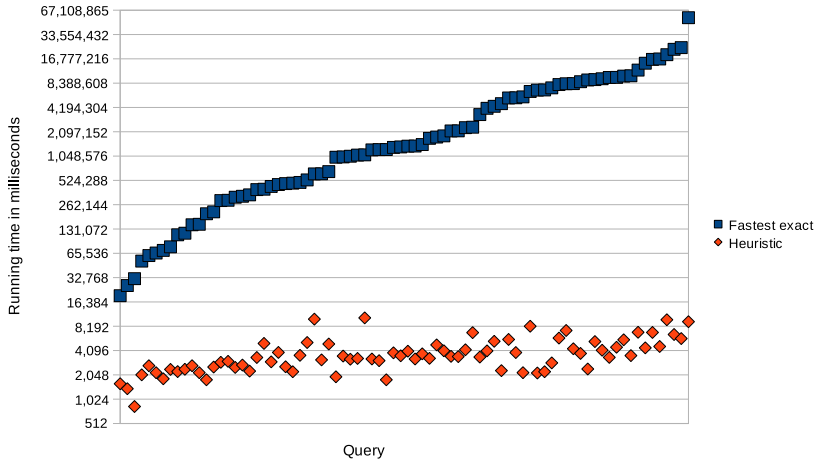


(a) NAVTEQ_BELGIUM, $k = 100$



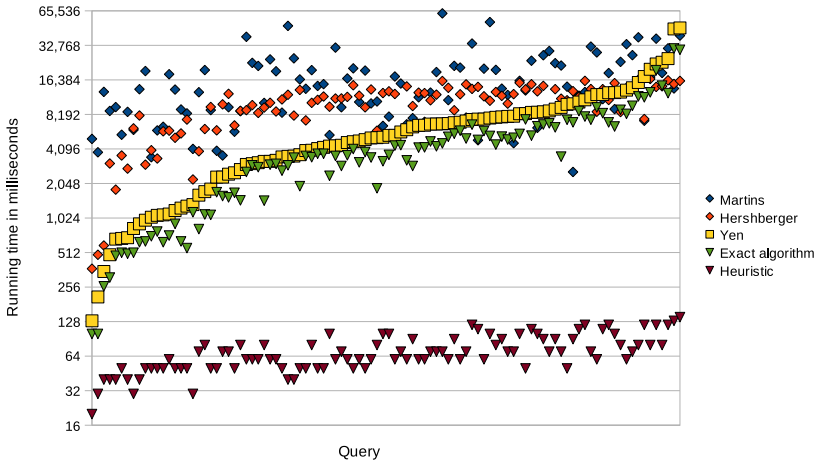
(b) NAVTEQ_BELGIUM, $k = 1\,000$

Figure B.10: Time performance for NAVTEQ_BELGIUM.

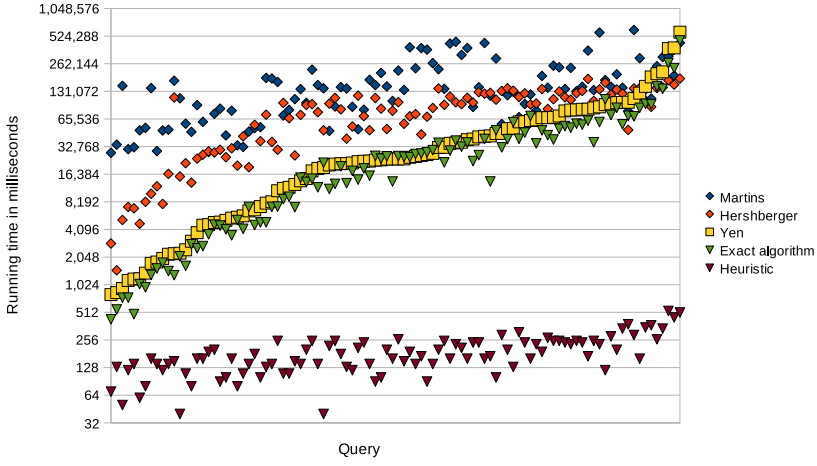


(c) NAVTEQ_BELGIUM, $k = 10\,000$

Figure B.10: Time performance for NAVTEQ_BELGIUM. For each k, the heuristic was compared, for 100 random queries, to three existing exact algorithms: Yen [70], Martins et al. [48] and Hershberger et al. [38]. Only the best of the running times for the exact algorithms is shown. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for the fastest exact algorithm. (Continued)

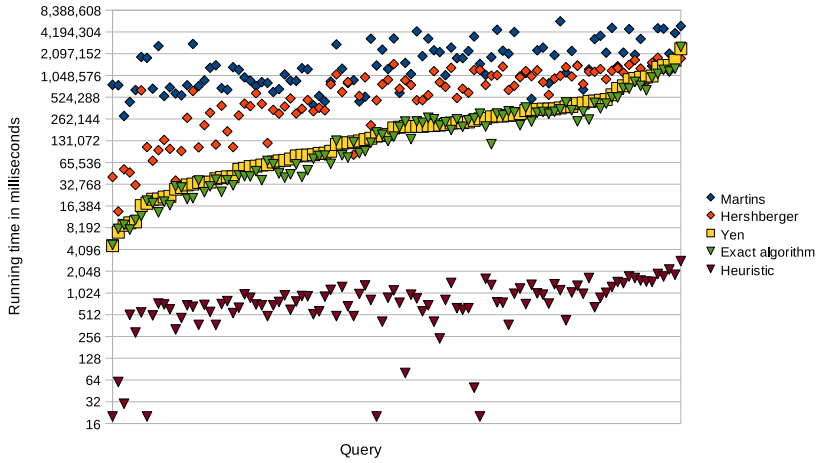


(a) CZE_MAX, $k = 100$



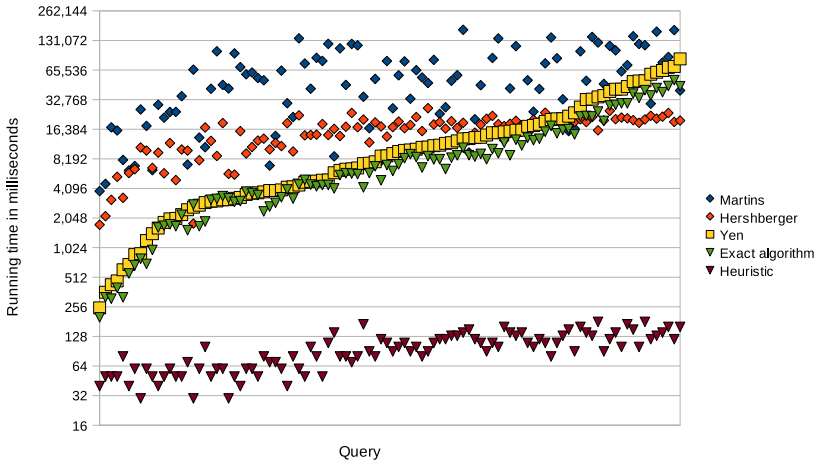
(b) CZE_MAX, $k = 1\,000$

Figure B.11: Detailed time performance for CZE_MAX

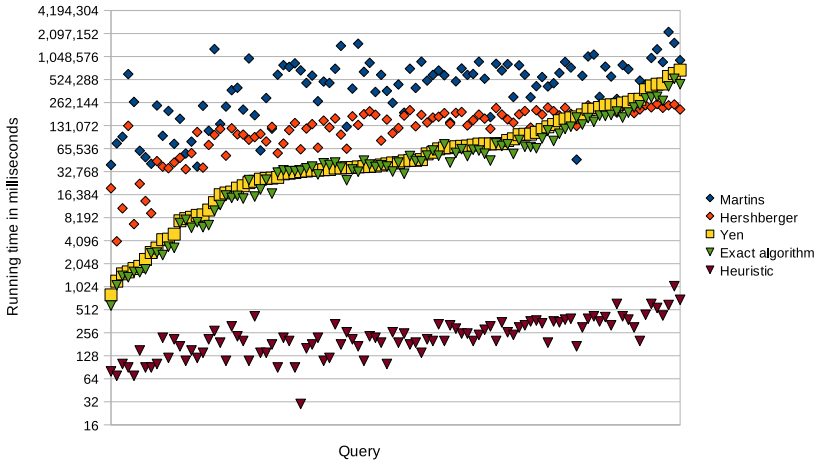


(c) CZE_MAX, $k = 10\,000$

Figure B.11: Detailed time performance for CZE_MAX. Five algorithms were tested for 100 random queries and different values of k: the three existing exact algorithms of Yen [70], Martins et al. [48] and Hershberger et al. [38], our exact algorithm and our heuristic. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for Yen's algorithm. (Continued)

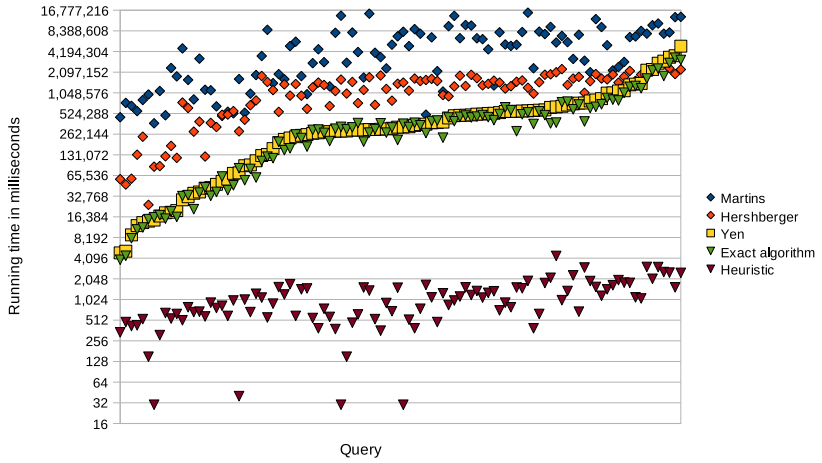


(a) IRL_MAX, $k = 100$



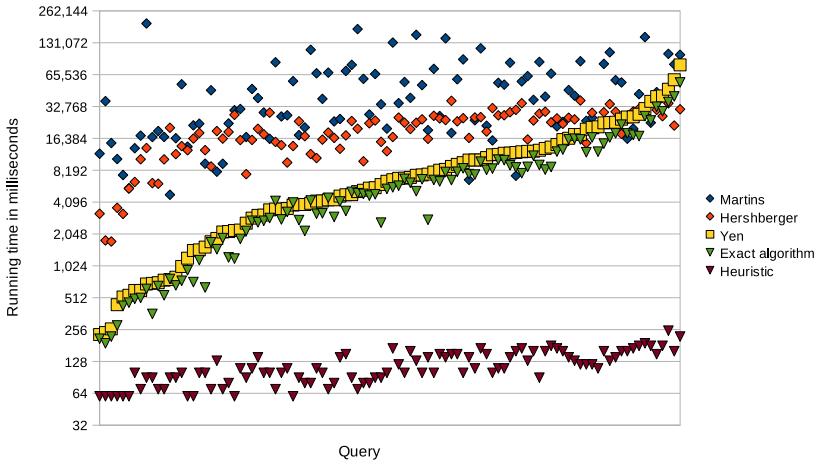
(b) IRL_MAX, $k = 1\,000$

Figure B.12: Detailed time performance for IRL_MAX

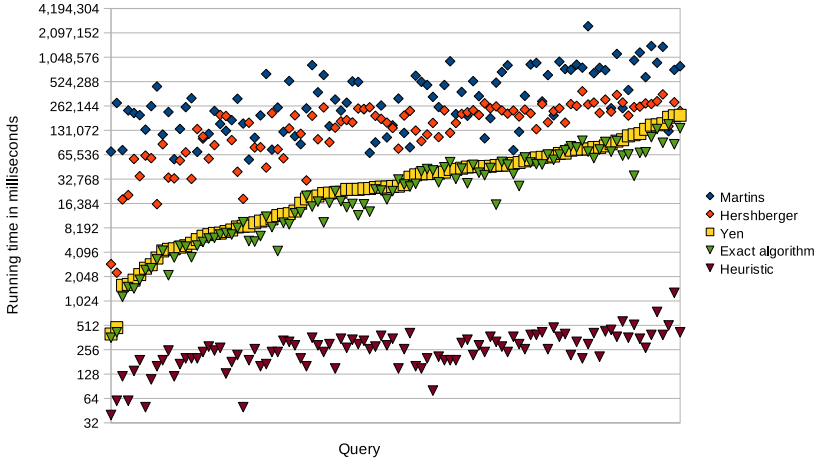


(c) IRL_MAX, $k = 10\,000$

Figure B.12: Detailed time performance for IRL_MAX. Five algorithms were tested for 100 random queries and different values of k: the three existing exact algorithms of Yen [70], Martins et al. [48] and Hershberger et al. [38], our exact algorithm and our heuristic. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for Yen's algorithm. (Continued)

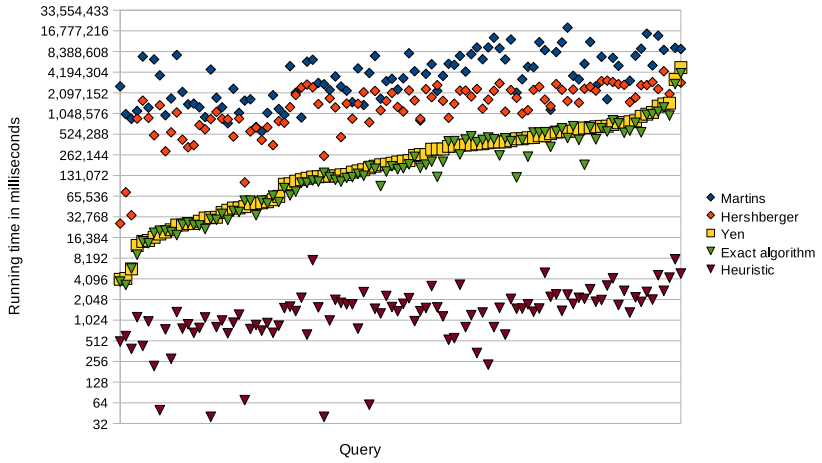


(a) NAVTEQ_LUXEMBOURG, $k = 100$



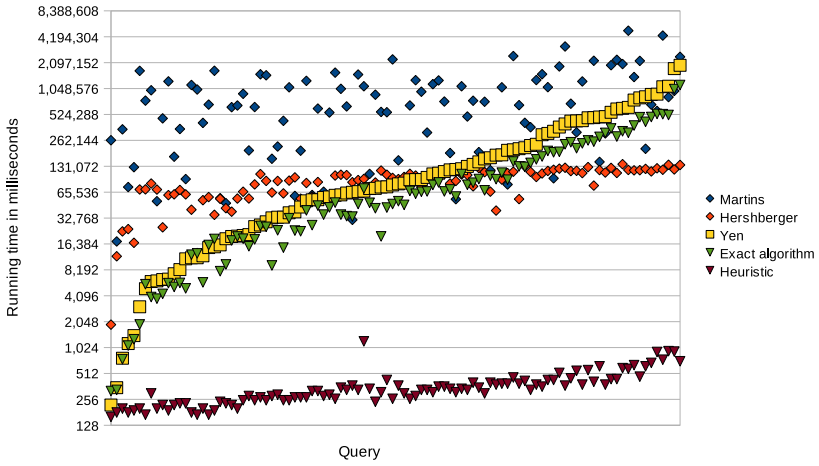
(b) NAVTEQ_LUXEMBOURG, $k = 1\,000$

Figure B.13: Detailed time performance for NAVTEQ_LUXEMBOURG

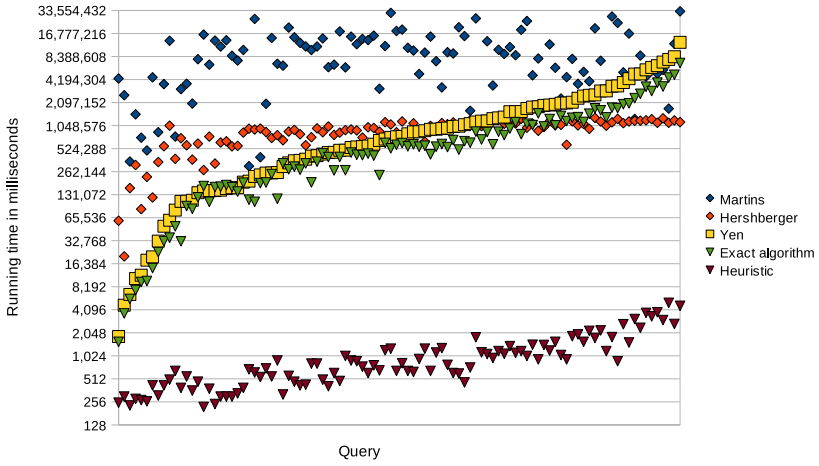


(c) NAVTEQ_LUXEMBOURG, $k = 10\,000$

Figure B.13: Detailed time performance for NAVTEQ_LUXEMBOURG. Five algorithms were tested for 100 random queries and different values of k: the three existing exact algorithms of Yen [70], Martins et al. [48] and Hershberger et al. [38], our exact algorithm and our heuristic. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for Yen's algorithm. (Continued)

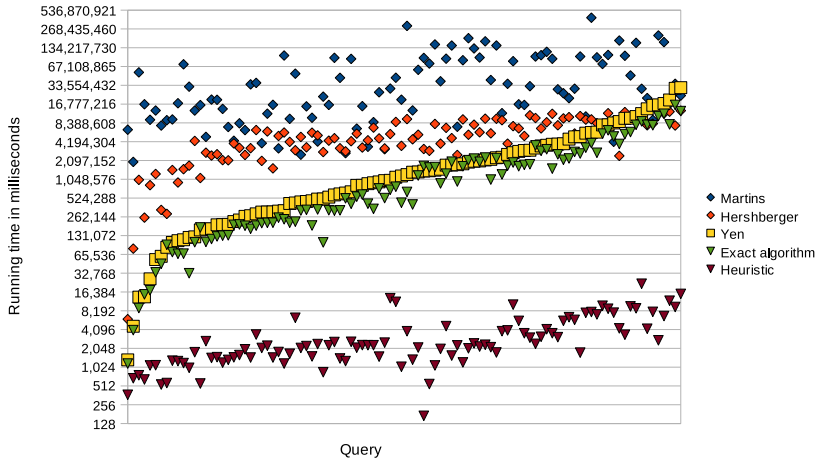


(a) PRT_MAX, $k = 100$



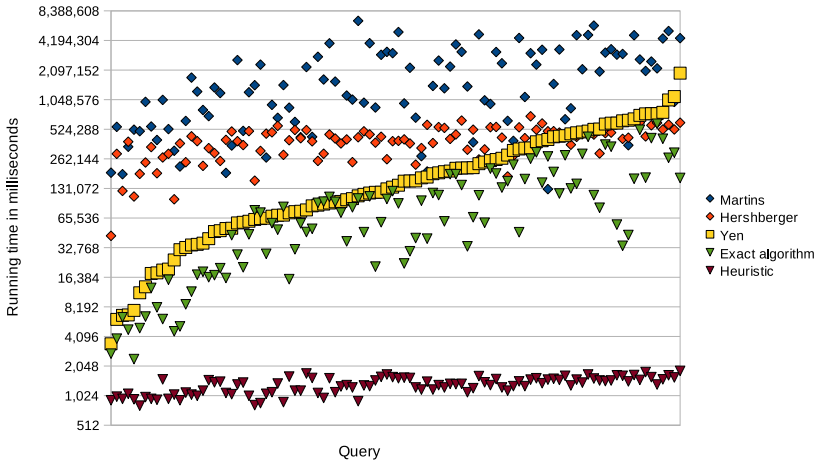
(b) PRT_MAX, $k = 1\,000$

Figure B.14: Detailed time performance for PRT_MAX

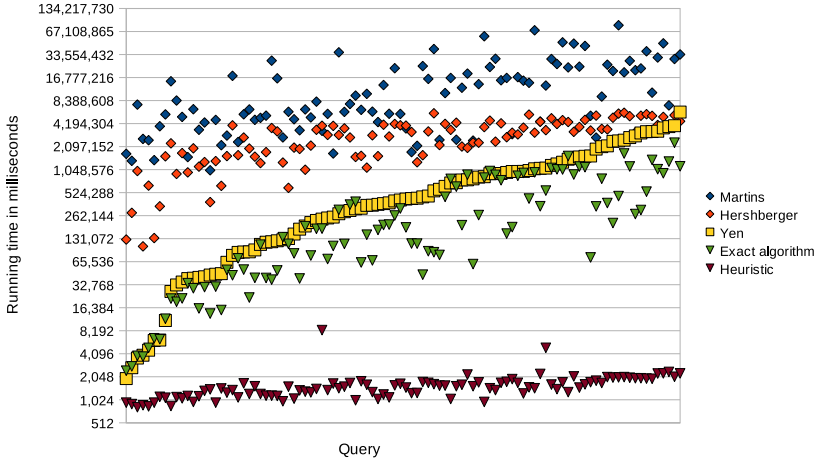


(c) PRT_MAX, $k = 10\,000$

Figure B.14: Detailed time performance for PRT_MAX. Five algorithms were tested for 100 random queries and different values of k: the three existing exact algorithms of Yen [70], Martins et al. [48] and Hershberger et al. [38], our exact algorithm and our heuristic. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for Yen's algorithm. (Continued)

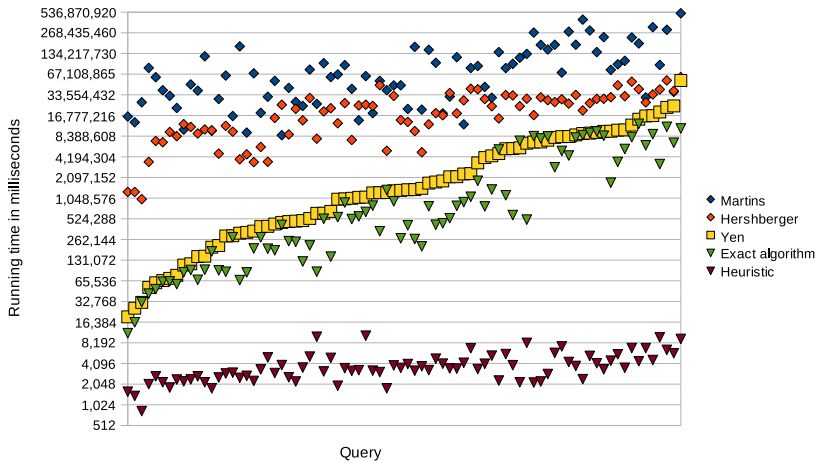


(a) NAVTEQ_BELGIUM, $k = 100$



(b) NAVTEQ_BELGIUM, $k = 1\,000$

Figure B.15: Detailed time performance for NAVTEQ_BELGIUM



(c) NAVTEQ_BELGIUM, $k = 10\,000$

Figure B.15: Detailed time performance for NAVTEQ_BELGIUM. Five algorithms were tested for 100 random queries and different values of k: the three existing exact algorithms of Yen [70], Martins et al. [48] and Hershberger et al. [38], our exact algorithm and our heuristic. The horizontal axis represents the 100 random queries. Results for the same query share their horizontal position. The vertical axis shows the running times in milliseconds and has a logarithmic scale. Results are sorted by the running time for Yen's algorithm. (Continued)

C

Supplementary material: Dissimilar paths

In this Appendix additional results are presented for Chapter 5. The tables are similar to those in Chapter 5, but more road networks are included.

Conclusion 1: Many plateaus are generated. Most of them have a zero weight.

Table C.1 shows additional results for Table 5.1 on Page 127.

Conclusion 2: Paths with short plateaus often contain cycles. Paths with longer plateaus are unlikely to contain cycles.

Table C.2 shows additional results for Table 5.2 on Page 129.

Conclusion 3: 1.25 is a good pruning factor.

Table C.3 shows additional results for Table 5.3 on Page 130.

Conclusion 4: A path with a plateau Pl such that $w(Pl) < T$ can still be T-locally optimal.

Table C.4 shows additional results for Table 5.4 on Page 132.

Experiment: choosing a stop condition

Table C.5 shows additional results for Table 5.7 on Page 158.

Plateau weight	Pruning factor							
% of $w(SP)$	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	4 698	6 113	7 438	8 054	8 629	9 651	10 540	13 201
]0%,5%]	767	926	1 050	1 104	1 150	1 226	1 288	1 452
]5%,10%]	48	59	68	72	75	82	87	104
]10%,15%]	15	18	21	23	24	26	28	34
]15%,20%]	6	8	9	10	10	11	12	16
]20%,25%]	3	3	4	4	5	5	6	9
]25%,50%]	3	4	5	5	5	6	7	10
]50%,75%]	1	1	1	1	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1	1	1	1	1
Sum	5 542	7 132	8 597	9 273	9 900	11 009	11 971	14 827

(a) CZE_MAX

Plateau weight	Pruning factor							
% of $w(SP)$	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	6 861	8 992	10 862	11 736	12 533	14 002	15 249	19 453
]0%,5%]	687	841	953	1 000	1 044	1 117	1 170	1 337
]5%,10%]	52	62	70	73	76	81	84	94
]10%,15%]	16	19	21	22	22	24	25	30
]15%,20%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]20%,25%]	6	7	8	9	9	9	10	12
]25%,50%]	3	4	4	4	4	4	5	6
]50%,75%]	4	4	4	4	5	5	5	6
]75%,85%]	1	1	1	1	1	1	1	1
]85%,95%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1	1	1	1	1
Sum	7 630	9 930	11 924	12 850	13 694	15 244	16 550	20 940

(b) LUX_MAX

Table C.1: Many plateaus are generated. Most of them have a zero weight. For different pruning factors (columns) and different plateau weights (rows) the total number of plateaus is shown. A value very close to zero is represented by ϵ.

Plateau weight % of $w(SP)$	Pruning factor							
	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	7 820	10 343	12 348	13 221	14 000	15 264	16 348	20 167
]0%,5%]	646	772	859	891	920	967	1 005	1 127
]5%,10%]	41	50	57	59	62	67	70	83
]10%,15%]	11	13	15	16	16	18	19	23
]15%,20%]	5	5	6	7	7	8	8	11
]20%,25%]	2	3	3	3	3	4	4	5
]25%,50%]	3	4	4	5	5	6	6	8
]50%,75%]	ϵ	1	1	1	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1	1	1	1	1
Sum	8 529	11 191	13 294	14 203	15 015	16 334	17 461	21 428

(c) IRL_MAX

Plateau weight % of $w(SP)$	Pruning factor							
	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	6 736	9 158	11 410	12 447	13 463	15 187	16 663	21 654
]0%,5%]	582	734	850	899	945	1 017	1 074	1 237
]5%,10%]	60	74	84	89	93	99	104	119
]10%,15%]	19	23	26	28	29	31	32	36
]15%,20%]	8	10	11	12	12	13	14	16
]20%,25%]	4	5	5	5	6	6	7	7
]25%,50%]	5	6	7	7	7	8	9	11
]50%,75%]	1	1	1	1	1	1	1	2
]75%,85%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]95%,100%[	0	0	0	0	0	0	ϵ	ϵ
100%	1	1	1	1	1	1	1	1
Sum	7 416	10 010	12 395	13 488	14 557	16 364	17 904	23 082

(d) NAVTEQ_LUXEMBOURG

Table C.1: Many plateaus are generated. Most of them have a zero weight. For different pruning factors (columns) and different plateau weights (rows) the total number of plateaus is shown. A value very close to zero is represented by ϵ.(Continued)

Plateau weight % of $w(SP)$	Pruning factor							
	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	32 354	49 570	55 467	57 199	58 706	61 726	65 284	76 725
]0%,5%]	4 638	6 248	6 660	6 767	6 862	7 048	7 280	8 009
]5%,10%]	60	73	84	88	91	96	101	118
]10%,15%]	14	17	20	21	23	24	26	31
]15%,20%]	6	7	8	8	9	10	11	13
]20%,25%]	3	3	4	4	4	4	5	7
]25%,50%]	4	4	5	5	5	6	6	9
]50%,75%]	ϵ	ϵ	ϵ	ϵ	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1	1	1	1	1
Sum	37 080	55 924	62 248	64 094	65 701	68 916	72 715	84 914

(e) PRT_MAX

Table C.1: Many plateaus are generated. Most of them have a zero weight. For different pruning factors (columns) and different plateau weights (rows) the total number of plateaus is shown. A value very close to zero is represented by ϵ. (Continued)

Plateau weight % of $w(SP)$	Pruning factor							
	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	90%	91%	92%	92%	92%	93%	93%	94%
]0%,5%]	30%	33%	35%	35%	36%	38%	39%	42%
]5%,10%]	8%	10%	12%	12%	13%	15%	16%	21%
]10%,15%]	6%	7%	9%	10%	11%	13%	15%	19%
]15%,20%]	3%	4%	6%	7%	8%	9%	11%	16%
]20%,25%]	1%	3%	3%	4%	6%	7%	10%	19%
]25%,50%]	1%	1%	3%	3%	3%	4%	6%	13%
]50%,75%]	0%	0%	2%	3%	3%	6%	7%	12%
]75%,85%]	0%	0%	0%	0%	0%	0%	0%	8%
]85%,95%]	0%	0%	0%	0%	0%	0%	0%	0%
]95%,100%[	0%	0%	0%	0%	0%	0%	0%	0%
100%	0%	0%	0%	0%	0%	0%	0%	0%

(a) CZE_MAX

Plateau weight % of $w(SP)$	Pruning factor							
	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	92%	93%	94%	94%	94%	94%	95%	95%
]0%,5%]	15%	17%	18%	19%	20%	21%	22%	26%
]5%,10%]	3%	4%	4%	5%	5%	6%	7%	9%
]10%,15%]	4%	4%	6%	6%	7%	8%	10%	15%
]15%,20%]	2%	3%	4%	5%	6%	7%	9%	15%
]20%,25%]	1%	2%	3%	3%	4%	5%	6%	15%
]25%,50%]	0%	1%	2%	2%	2%	4%	6%	12%
]50%,75%]	0%	0%	0%	0%	0%	0%	0%	3%
]75%,85%]	0%	0%	0%	0%	0%	0%	0%	0%
]85%,95%]	0%	0%	0%	0%	0%	0%	0%	0%
]95%,100%[	0%	0%	0%	0%	0%	0%	0%	0%
100%	0%	0%	0%	0%	0%	0%	0%	0%

(b) LUX_MAX

Table C.2: Paths with short plateaus often contain cycles. Paths with longer plateaus are unlikely to contain cycles. For different pruning factors (columns) and different plateau weights (rows) the percentage of plateaus which produce cycles is shown. A value very close to zero is represented by ϵ.

Plateau weight	Pruning factor							
% of $w(SP)$	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	95%	95%	96%	96%	96%	96%	96%	97%
]0%,5%]	25%	28%	31%	32%	32%	34%	35%	39%
]5%,10%]	5%	7%	9%	10%	11%	12%	14%	18%
]10%,15%]	4%	5%	6%	7%	8%	9%	11%	18%
]15%,20%]	4%	4%	5%	7%	7%	10%	11%	19%
]20%,25%]	2%	3%	3%	4%	5%	7%	7%	13%
]25%,50%]	ϵ	1%	2%	3%	3%	5%	6%	15%
]50%,75%]	0%	0%	0%	0%	0%	0%	0%	6%
]75%,85%]	0%	0%	0%	0%	0%	0%	3%	9%
]85%,95%]	0%	0%	0%	0%	0%	0%	0%	0%
]95%,100%[	0%	0%	0%	0%	0%	0%	0%	50%
100%	0%	0%	0%	0%	0%	0%	0%	0%

(c) IRL_MAX

Plateau weight	Pruning factor							
% of $w(SP)$	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	94%	95%	95%	96%	96%	96%	96%	97%
]0%,5%]	22%	24%	26%	26%	27%	28%	28%	31%
]5%,10%]	5%	5%	5%	5%	5%	5%	5%	6%
]10%,15%]	3%	4%	4%	4%	4%	4%	4%	5%
]15%,20%]	1%	1%	2%	2%	2%	3%	3%	4%
]20%,25%]	2%	3%	3%	3%	4%	4%	6%	6%
]25%,50%]	ϵ	1%	2%	3%	3%	3%	4%	7%
]50%,75%]	0%	0%	0%	0%	0%	1%	4%	13%
]75%,85%]	0%	0%	0%	0%	0%	0%	0%	0%
]85%,95%]	0%	0%	0%	0%	0%	0%	0%	0%
]95%,100%[	/	/	/	/	/	/	0%	0%
100%	0%	0%	0%	0%	0%	0%	0%	0%

(d) NAVTEQ_LUXEMBOURG

Table C.2: Paths with short plateaus often contain cycles. Paths with longer plateaus are unlikely to contain cycles. For different pruning factors (columns) and different plateau weights (rows) the percentage of plateaus which produce cycles is shown. A value very close to zero is represented by ϵ. (Continued)

Plateau weight % of $w(SP)$	Pruning factor							
	1	1.1	1.2	1.25	1.3	1.4	1.5	2
0%	90%	92%	93%	93%	93%	93%	93%	94%
]0%,5%]	22%	28%	30%	30%	30%	31%	32%	36%
]5%,10%]	6%	6%	7%	7%	8%	8%	9%	14%
]10%,15%]	5%	5%	6%	6%	6%	6%	7%	12%
]15%,20%]	4%	5%	5%	5%	5%	7%	7%	12%
]20%,25%]	1%	2%	2%	2%	3%	4%	4%	6%
]25%,50%]	1%	1%	2%	3%	3%	4%	5%	9%
]50%,75%]	0%	0%	2%	2%	2%	2%	2%	7%
]75%,85%]	0%	0%	0%	0%	0%	0%	0%	0%
]85%,95%]	0%	0%	0%	0%	0%	0%	0%	0%
]95%,100%[	0%	0%	0%	0%	0%	0%	0%	0%
100%	0%	0%	0%	0%	0%	0%	0%	0%

(e) PRT_MAX

Table C.2: Paths with short plateaus often contain cycles. Paths with longer plateaus are unlikely to contain cycles. For different pruning factors (columns) and different plateau weights (rows) the percentage of plateaus which produce cycles is shown. A value very close to zero is represented by ϵ. (Continued)

α	Path weight $\times w(SP)$	Pruning factor							
		1	1.1	1.2	1.25	1.3	1.4	1.5	2
10%	[1.0,1.1[	18	18	18	18	18	18	18	18
	[1.1,1.25[	24	25	25	25	25	25	25	25
	[1.25,1.5[	26	31	33	33	33	33	33	33
	[1.5,2[	18	29	38	42	46	50	50	50
	≥ 2	0	2	8	11	15	25	35	68
25%	[1.0,1.1[	5	5	5	5	5	5	5	5
	[1.1,1.25[	4	4	4	4	4	4	4	4
	[1.25,1.5[	5	6	6	6	6	6	6	6
	[1.5,2[	5	8	10	11	11	12	12	12
	≥ 2	0	1	3	4	5	9	12	26
50%	[1.0,1.1[	3	3	3	3	3	3	3	3
	[1.1,1.25[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	[1.25,1.5[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	[1.5,2[	1	1	1	1	1	1	1	1
	≥ 2	0	ϵ	ϵ	1	1	1	2	5
total # of plateaus		5 542	7 132	8 597	9 273	9 901	11 009	11 971	14 827

(a) CZE_MAX

α	Path weight $\times w(SP)$	Pruning factor							
		1	1.1	1.2	1.25	1.3	1.4	1.5	2
10%	[1.0,1.1[	29	30	30	30	30	30	30	30
	[1.1,1.25[	31	33	33	33	33	33	33	33
	[1.25,1.5[	28	35	38	39	39	39	39	39
	[1.5,2[	17	28	37	42	46	51	52	52
	≥ 2	0	2	7	11	15	25	36	74
25%	[1.0,1.1[	8	9	9	9	9	9	9	9
	[1.1,1.25[	5	5	5	5	5	5	5	5
	[1.25,1.5[	4	5	5	5	5	5	5	5
	[1.5,2[	3	4	5	6	6	7	7	7
	≥ 2	0	ϵ	1	2	2	4	6	13
50%	[1.0,1.1[	4	4	4	4	4	4	4	4
	[1.1,1.25[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	[1.25,1.5[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	[1.5,2[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	≥ 2	0	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	1
total # of plateaus		7 630	9 930	11 924	12 850	13 695	15 244	16 551	20 940

(b) LUX_MAX

Table C.3: 1.25 is a good pruning factor. For different pruning factors (columns) and different weights of the paths corresponding to the plateaus and different values of α (rows), the number of "good" paths is shown, i.e. paths which are cycle-free and locally optimal. The last row shows the total number of plateaus for each pruning factor.

α	Path weight $\times w(SP)$	Pruning factor							
		1	1.1	1.2	1.25	1.3	1.4	1.5	2
10%	[1.0,1.1[	20	20	20	20	20	20	20	20
	[1.1,1.25[	21	22	22	22	22	22	22	22
	[1.25,1.5[	19	23	26	26	26	26	26	26
	[1.5,2[	15	24	32	35	39	42	43	43
	≥ 2	0	2	6	9	12	19	28	63
25%	[1.0,1.1[	5	5	5	5	5	5	5	5
	[1.1,1.25[	4	4	4	4	4	4	4	4
	[1.25,1.5[	4	5	5	5	5	5	5	5
	[1.5,2[	4	6	8	8	9	9	9	9
	≥ 2	0	1	2	3	4	6	9	21
50%	[1.0,1.1[	2	2	2	2	2	2	2	2
	[1.1,1.25[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	[1.25,1.5[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	[1.5,2[	1	1	1	1	1	1	1	1
	≥ 2	ϵ	ϵ	ϵ	1	1	1	2	5
total # of plateaus		8 529	11 191	13 294	14 203	15 015	16 334	17 461	21 428

(c) IRL_MAX

α	Path weight $\times w(SP)$	Pruning factor							
		1	1.1	1.2	1.25	1.3	1.4	1.5	2
10%	[1.0,1.1[	16	16	16	16	16	16	16	16
	[1.1,1.25[	38	41	41	41	41	41	41	41
	[1.25,1.5[	45	55	59	59	59	59	59	59
	[1.5,2[	30	47	64	71	77	84	85	85
	≥ 2	0	4	13	18	25	42	60	118
25%	[1.0,1.1[	4	4	4	4	4	4	4	4
	[1.1,1.25[	6	6	6	6	6	6	6	6
	[1.25,1.5[	9	10	10	10	10	10	10	10
	[1.5,2[	8	11	14	15	16	17	17	17
	≥ 2	0	1	4	5	7	12	16	35
50%	[1.0,1.1[	2	2	2	2	2	2	2	2
	[1.1,1.25[	1	1	1	1	1	1	1	1
	[1.25,1.5[	1	1	1	1	1	1	1	1
	[1.5,2[	1	1	2	2	2	2	2	2
	≥ 2	0	ϵ	1	1	1	2	2	6
total # of plateaus		7 416	10 010	12 395	13 488	14 557	16 364	17 904	23 082

(d) NAVTEQ_LUXEMBOURG

Table C.3: 1.25 is a good pruning factor. For different pruning factors (columns) and different weights of the paths corresponding to the plateaus and different values of α (rows), the number of "good" paths is shown, i.e. paths which are cycle-free and locally optimal. The last row shows the total number of plateaus for each pruning factor. (Continued)

α	Path weight $\times w(SP)$	Pruning factor							
		1	1.1	1.2	1.25	1.3	1.4	1.5	2
10%	[1.0,1.1[	58	74	74	74	74	74	74	74
	[1.1,1.25[	24	25	25	25	25	25	25	25
	[1.25,1.5[	22	30	33	34	34	34	34	34
	[1.5,2[	13	22	31	37	41	46	47	47
	≥ 2	ϵ	2	6	8	11	20	31	73
25%	[1.0,1.1[	46	68	68	68	68	68	68	68
	[1.1,1.25[	3	6	6	6	6	6	6	6
	[1.25,1.5[	4	4	5	5	5	5	5	5
	[1.5,2[	4	5	7	7	8	8	8	8
	≥ 2	ϵ	1	2	3	4	6	8	21
50%	[1.0,1.1[	34	56	56	56	56	56	56	56
	[1.1,1.25[	1	5	5	5	5	5	5	5
	[1.25,1.5[	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
	[1.5,2[	1	1	1	1	1	1	1	1
	≥ 2	0	ϵ	ϵ	1	1	1	2	5
total # of plateaus		37 080	55 924	62 248	64 094	65 701	68 916	72 715	84 914

(e) PRT_MAX

Table C.3: 1.25 is a good pruning factor. For different pruning factors (columns) and different weights of the paths corresponding to the plateaus and different values of α (rows), the number of "good" paths is shown, i.e. paths which are cycle-free and locally optimal. The last row shows the total number of plateaus for each pruning factor. (Continued)

Plateau weight % of $w(SP)$	α			total # of plateaus
	0.1	0.25	0.5	
0	13	2	1	8 054
]0 5%]	42	5	1	1 104
]5%,10%]	34	5	ϵ	72
]10%,15%]	20	5	ϵ	23
]15%,20%]	9	3	ϵ	10
]20%,25%]	4	3	ϵ	4
]25%,50%]	5	5	1	5
]50%,75%]	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1
Sum	130	29	5	9 273

(a) CZE_MAX

Plateau weight % of $w(SP)$	α			total # of plateaus
	0.1	0.25	0.5	
0	17	2	1	11 736
]0,5%]	56	5	1	1 000
]5%,10%]	43	4	ϵ	73
]10%,15%]	20	3	ϵ	22
]15%,20%]	8	3	ϵ	9
]20%,25%]	4	3	ϵ	4
]25%,50%]	4	4	1	4
]50%,75%]	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1
Sum	155	25	5	12 850

(b) LUX_MAX

Table C.4: A path with a plateau Pl such that $w(Pl) < T$ can still be T-locally optimal. For different values of α (columns) and for different plateau weights (rows) the number of cycle-free locally optimal paths is shown. For example, for PRT_MAX, 4 plateaus with a weight between 10% and 15% of $w(SP)$ result in a cycle-free path which is locally optimal for $\alpha = 0.25$. The total number of plateaus in each weight class is shown in the last column. The pruning factor was set to 1.25.

Plateau weight % of $w(SP)$	α			total # of plateaus
	0.1	0.25	0.5	
0	16	3	1	13 221
]0,5%]	36	3	1	891
]5%,10%]	30	3	ϵ	59
]10%,15%]	14	3	ϵ	16
]15%,20%]	6	3	ϵ	7
]20%,25%]	3	2	ϵ	3
]25%,50%]	5	5	1	5
]50%,75%]	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1
Sum	112	24	5	14 203

(c) IRL_MAX

Plateau weight % of $w(SP)$	α			total # of plateaus
	0.1	0.25	0.5	
0	23	3	ϵ	12 447
]0,5%]	77	7	1	899
]5%,10%]	54	7	ϵ	89
]10%,15%]	26	7	1	28
]15%,20%]	11	5	ϵ	12
]20%,25%]	5	3	ϵ	5
]25%,50%]	7	7	1	7
]50%,75%]	1	1	1	1
]75%,85%]	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ
]95%,100%[	0	0	0	0
100%	1	1	1	1
Sum	206	40	6	13 488

(d) NAVTEQ_LUXEMBOURG

Table C.4: A path with a plateau Pl such that $w(Pl) < T$ can still be T-locally optimal. For different values of α (columns) and for different plateau weights (rows) the number of cycle-free locally optimal paths is shown. For example, for PRT_MAX, 4 plateaus with a weight between 10% and 15% of $w(SP)$ result in a cycle-free path which is locally optimal for $\alpha = 0.25$. The total number of plateaus in each weight class is shown in the last column. The pruning factor was set to 1.25. (Continued)

Plateau weight % of $w(SP)$	α			total # of plateaus
	0.1	0.25	0.5	
0	30	28	25	57 199
]0,5%]	72	41	35	6 767
]5%,10%]	38	4	1	88
]10%,15%]	20	4	ϵ	21
]15%,20%]	8	3	ϵ	8
]20%,25%]	4	3	ϵ	4
]25%,50%]	5	5	1	5
]50%,75%]	ϵ	ϵ	ϵ	ϵ
]75%,85%]	ϵ	ϵ	ϵ	ϵ
]85%,95%]	ϵ	ϵ	ϵ	ϵ
]95%,100%[	ϵ	ϵ	ϵ	ϵ
100%	1	1	1	1
Sum	178	88	64	64 094

(e) PRT_MAX

Table C.4: A path with a plateau Pl such that $w(Pl) < T$ can still be T-locally optimal. For different values of α (columns) and for different plateau weights (rows) the number of cycle-free locally optimal paths is shown. For example, for PRT_MAX, 4 plateaus with a weight between 10% and 15% of $w(SP)$ result in a cycle-free path which is locally optimal for $\alpha = 0.25$. The total number of plateaus in each weight class is shown in the last column. The pruning factor was set to 1.25. (Continued)

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	44	0.753			3
20	81	0.770	48	0.041	0
40	117	0.775	12	0.040	0
60	153	0.776	4	0.035	0
80	197	0.778	3	0.068	0
100	217	0.780	5	0.035	0
120	249	0.781	4	0.028	0
140	284	0.781	2	0.010	0
160	313	0.781	0	/	0
180	347	0.781	0	/	0
200	478	0.781	0	/	0
500	1 183	0.782	3	0.026	0
$+\infty$	2 534	0.782	2	0.008	0

(a) CZE_MAX, $k = 3$

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	70	0.655			15
20	118	0.690	64	0.056	1
40	181	0.694	22	0.021	1
60	219	0.696	9	0.025	1
80	285	0.698	8	0.021	1
100	337	0.698	4	0.024	0
120	359	0.698	3	0.019	0
140	424	0.699	6	0.019	0
160	440	0.699	0	/	0
180	479	0.700	1	0.016	0
200	464	0.700	0	/	0
500	960	0.701	7	0.023	0
$+\infty$	2 032	0.703	4	0.039	0

(b) CZE_MAX, $k = 5$

Table C.5: Results for different stop conditions with $\alpha = 25\%$. The average running time and average quality $Q(S)$ of the result is shown. The number of queries (out of 100) for which an improvement was found is shown, and the average size of this improvement if there is one. The last column shows the number of queries (out of 100) for which no solution was found.

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	70	0.762			5
20	156	0.790	49	0.067	0
40	240	0.792	7	0.033	0
60	289	0.793	6	0.020	0
80	358	0.795	5	0.035	0
100	440	0.795	0	/	0
120	613	0.796	2	0.060	0
140	646	0.797	1	0.052	0
160	724	0.797	2	0.011	0
180	808	0.797	0	/	0
200	786	0.797	0	/	0
500	2 017	0.797	2	0.011	0
$+\infty$	4 292	0.797	0	/	0

(c) IRL_MAX, $k = 3$

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	79	0.650			32
20	168	0.694	57	0.068	0
40	254	0.701	26	0.025	0
60	344	0.706	13	0.036	0
80	448	0.707	7	0.013	0
100	486	0.707	3	0.005	0
120	588	0.708	5	0.022	0
140	630	0.708	1	0.036	0
160	766	0.708	1	0.006	0
180	764	0.709	2	0.014	0
200	828	0.709	0	/	0
500	2 034	0.710	6	0.018	0
$+\infty$	3 974	0.710	1	0.001	0

(d) IRL_MAX, $k = 5$

Table C.5: Results for different stop conditions with $\alpha = 25\%$. The average running time and average quality $Q(S)$ of the result is shown. The number of queries (out of 100) for which an improvement was found is shown, and the average size of this improvement if there is one. The last column shows the number of queries (out of 100) for which no solution was found. (Continued)

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	164	0.758			3
5	221	0.769	36	0.047	0
10	273	0.778	19	0.049	0
15	302	0.781	8	0.038	0
20	348	0.782	5	0.021	0
40	536	0.788	16	0.033	0
60	645	0.788	3	0.028	0
80	754	0.789	3	0.019	0
100	833	0.790	2	0.054	0
120	984	0.790	1	0.010	0
140	1 089	0.790	1	ϵ	0
160	1 202	0.790	0	/	0
180	1 144	0.790	0	/	0
200	1 312	0.790	1	0.007	0
500	2 700	0.792	3	0.051	0
$+\infty$	5 738	0.792	2	0.009	0

(e) NAVTEQ_LUXEMBOURG, $k = 3$

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	115	0.662			17
5	144	0.674	42	0.041	2
10	190	0.685	34	0.040	1
15	210	0.690	18	0.025	1
20	287	0.694	15	0.028	1
40	356	0.703	35	0.025	1
60	454	0.706	12	0.024	1
80	517	0.708	11	0.017	1
100	733	0.710	11	0.019	1
120	751	0.711	5	0.021	1
140	919	0.712	4	0.017	1
160	891	0.712	3	0.024	1
180	1 007	0.712	0	/	1
200	1 040	0.713	4	0.020	1
500	2 173	0.715	11	0.019	1
$+\infty$	5 574	0.716	5	0.024	1

(f) NAVTEQ_LUXEMBOURG, $k = 5$

Table C.5: Results for different stop conditions with $\alpha = 25\%$. The average running time and average quality $Q(S)$ of the result is shown. The number of queries (out of 100) for which an improvement was found is shown, and the average size of this improvement if there is one. The last column shows the number of queries (out of 100) for which no solution was found. (ContinuedFloat)

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	534	0.788			2
20	1 286	0.811	39	0.062	0
40	1 932	0.816	8	0.063	0
60	2 512	0.816	2	0.017	0
80	3 108	0.817	6	0.020	0
100	3 655	0.819	2	0.066	0
120	4 150	0.819	2	0.033	0
140	4 632	0.821	2	0.076	0
160	5 216	0.821	0	/	0
180	5 716	0.821	0	/	0
200	5 978	0.821	0	/	0
500	13 196	0.821	1	0.046	0
$+\infty$	212 424	0.823	3	0.062	0

(g) PRT_MAX, $k = 3$

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	514	0.689			27
20	1 143	0.723	52	0.066	4
40	1 717	0.727	27	0.024	2
60	2 257	0.730	13	0.025	1
80	2 828	0.732	10	0.024	1
100	3 364	0.733	4	0.008	1
120	3 830	0.733	3	0.013	1
140	4 362	0.733	1	0.001	1
160	4 732	0.734	3	0.020	1
180	5 543	0.734	2	0.027	1
200	5 956	0.735	2	0.030	1
500	13 422	0.737	9	0.018	1
$+\infty$	224 043	0.739	8	0.026	1

(h) PRT_MAX, $k = 5$

Table C.5: Results for different stop conditions with $\alpha = 25\%$. The average running time and average quality $Q(S)$ of the result is shown. The number of queries (out of 100) for which an improvement was found is shown, and the average size of this improvement if there is one. The last column shows the number of queries (out of 100) for which no solution was found. (Continued)

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	2 705	0.748			0
5	3 698	0.758	23	0.044	0
10	4 612	0.760	12	0.017	0
15	5 497	0.763	17	0.020	0
20	5 958	0.764	4	0.020	0
40	9 123	0.767	14	0.019	0
60	11 209	0.769	7	0.028	0
80	13 215	0.770	6	0.023	0
100	14 680	0.770	0	/	0
120	16 540	0.771	2	0.041	0
140	17 588	0.771	2	0.011	0
160	18 411	0.771	0	/	0
180	19 992	0.771	0	/	0
200	20 799	0.771	1	0.011	0
500	37 353	0.774	11	0.024	0
$+\infty$	702 773	0.777	11	0.026	0

(i) NAVTEQ_BELGIUM, $k = 3$

max	time (ms)	$Q(S)$	#improved	avg. improvement	#no solution
0	1 647	0.658			4
5	2 329	0.671	39	0.038	1
10	3 008	0.678	27	0.024	0
15	3 433	0.686	31	0.027	0
20	4 015	0.688	14	0.017	0
40	6 292	0.695	23	0.029	0
60	8 283	0.699	17	0.027	0
80	10 086	0.701	12	0.014	0
100	11 902	0.702	6	0.010	0
120	12 972	0.703	4	0.028	0
140	13 774	0.703	3	0.005	0
160	15 238	0.703	2	0.024	0
180	16 275	0.703	1	0.001	0
200	17 686	0.704	5	0.012	0
500	32 761	0.707	23	0.015	0
$+\infty$	625 568	0.712	29	0.016	0

(j) NAVTEQ_BELGIUM, $k = 5$

Table C.5: Results for different stop conditions with $\alpha = 25\%$. The average running time and average quality $Q(S)$ of the result is shown. The number of queries (out of 100) for which an improvement was found is shown, and the average size of this improvement if there is one. The last column shows the number of queries (out of 100) for which no solution was found. (Continued)

Nederlandstalige samenvatting

Routeringsalgoritmen bestaan al zeer lang. Rond 1960 werden al enkele algoritmen voorgesteld. Vanaf de jaren '90 was er een hernieuwde interesse in routeringsalgoritmen door de opkomst van navigatiesystemen, interactieve routeplanners en GIS-systemen. Er werden veel efficiënte algoritmen ontwikkeld die de kortste route berekenen tussen twee punten. Vaak vereist een realistische routeplanner echter meer dan enkel het berekenen van een kortste route. Er kan bvb. rekening moeten gehouden worden met wachttijden aan kruispunten, of de gebruiker wenst soms meer dan één route als resultaat te krijgen. Daarom focussen we ons in dit werk op het ontwikkelen van algoritmen voor een aantal alternatieve routeringsproblemen. Uiteraard is de theoretische complexiteit van de algoritmen belangrijk, maar we besteden ook de nodige aandacht aan de performantie van de algoritmen in de praktijk. De algoritmen werden uitgebreid met elkaar en met bestaande methoden vergeleken aan de hand van experimenten op bestaande wegennetwerken. De resultaten van deze experimenten worden uitgebreid besproken in dit werk.

In het **eerste hoofdstuk** wordt een inleiding gegeven. In het **tweede hoofdstuk** brengen we de nodige grafentheoretische concepten aan. Een wegennetwerk wordt voorgesteld als een graaf. Kruispunten en uiteinden van doodlopende straten worden voorgesteld als *toppen*. Straten worden voorgesteld als *bogen*, die deze toppen met elkaar verbinden. Deze bogen zijn gericht, waardoor eenrichtingsstraten kunnen voorgesteld worden. Een route zonder lussen van top s naar top t wordt in de grafentheorie een *pad* genoemd. Een zeer bekend algoritme voor het berekenen van het kortste

pad van s naar t is het algoritme van Dijkstra. Dit algoritme berekent een *kortstepadenboom* waarin het kortste pad van s naar t en ook naar een aantal andere toppen kan opgezocht worden. Het algoritme van Dijkstra kan ook achterwaarts uitgevoerd worden. In dit geval wordt een *achterwaartste kortstepadenboom* berekend. Hierin kan het kortste pad van s naar t en ook van een aantal andere toppen naar t opgezocht worden. Wanneer het algoritme van Dijkstra voorwaarts uitgevoerd werd, spreken we van een *voorwaartste kortstepadenboom*.

In het **derde hoofdstuk** beschrijven we het probleem van routering waarbij afslagen aan kruispunten verboden kunnen zijn of een bepaalde kost kunnen hebben, bvb. de wachttijd bij het voorrang verlenen of bij verkeerslichten, of de vertraging die ontstaat door het afremmen om de afslag te kunnen nemen. Standaard kortstepad-algoritmen zoals het algoritme van Dijkstra houden hier geen rekening mee en gaan ervan uit dat de kost van een route gelijk is aan de som van de kosten van de bogen op die route. Nochtans is het zo dat bvb. in Kopenhagen 17% tot 35% van de reistijd besteed wordt aan het wachten op kruispunten. In steden met drukker verkeer is dit wellicht zelfs nog meer. We bespreken drie methoden die hiermee rekening houden. Twee methoden maken gebruik van een graaftransformatie, namelijk de methode gebaseerd op de *lijngraaf* en de methode gebaseerd op het *splitsen van de toppen*. Op deze getransformeerde graaf kan dan een standaard kortstepad-algoritme uitgevoerd worden. Een derde methode kan rechtstreeks op de graaf zelf toegepast worden. Deze methode noemen we de *directe methode*. We bespreken een aantal implementatiedetails en geven de resultaten van gedetailleerde experimenten die als doel hebben na te gaan welke methode op welk type wegennetwerk het best presteert. We concluderen met een richtlijn voor het kiezen van de juiste methode voor een bepaald wegennetwerk.

In het **vierde hoofdstuk** presenteren we een heuristiek voor het bepalen van de k *kortste paden* tussen twee toppen. Bij dit probleem is het de bedoeling niet enkel het kortste pad, maar ook het tweede kortste, het derde kortste, ..., tot en met het k^{de} kortste pad te bepalen. Aangezien dit voor de hand ligt in wegennetwerken, beperken wij ons tot de variant van dit

probleem waarbij de paden geen lussen mogen bevatten. Dit worden de k kortste *simpele* paden genoemd. We bespreken een aantal exacte algoritmen voor dit probleem, maar deze hebben allemaal een hoge theoretische tijdscomplexiteit, waardoor ze vrij traag zijn. Wij presenteren een heuristiek voor het k kortste paden probleem, die niet noodzakelijk de exacte oplossing vindt, maar wel veel sneller is. Aan de hand van experimenten tonen we aan dat de heuristiek honderden tot zelfs duizenden keren sneller is dan de exacte algoritmen, en daarbij toch in de meeste gevallen een oplossing vindt die dicht bij de exacte oplossing ligt. Ook beschrijven we hoe de heuristiek kan omgezet worden in een relatief snel exact algoritme.

Een ander probleem, dat veel praktische toepassingen kent, is het vinden van *alternatieve routes*. Dit probleem wordt behandeld in het **vijfde hoofdstuk**. Een gebruiker van een routeplanner wenst vaak niet enkel de kortste route, maar ook enkele alternatieven zodat hij een keuze kan maken rekening houdend met zijn eigen voorkeuren. We beschrijven vier criteria die bepalen of een set alternatieven goed is. Ten eerste, moeten de routes *voldoende verschillend* zijn. We geven verschillende definities voor het "verschillend zijn" van routes, en geven uiteindelijk onze eigen definitie. Ten tweede, mogen de routes niet te lang zijn. Veel bestaande methoden lossen dit op door een strikte bovengrens te hanteren en alle routes die langer zijn dan deze bovengrens te elimineren. Wij willen een dergelijke strikte bovengrens echter vermijden, enerzijds aangezien er zich in sommige gevallen ook boven die grens nog goede alternatieven kunnen bevinden, en anderzijds omdat binnen de routes die wel behouden worden, de kortere routes vaak nog steeds de voorkeur genieten boven de langere. Daarom kiezen we voor een objectieffunctie Q die zowel rekening houdt met de lengte van de routes in een oplossing als met hoe verschillend ze zijn. Ten derde, mogen er uiteraard geen cykels voorkomen in de oplossing. Tenslotte, eisen wij ook dat de routes *lokaal optimaal* zijn. Intuïtief betekent dit dat een route geen korte onnodige omwegen mag bevatten die niet logisch zijn voor de bestuurder. Formeel is een pad T-lokaal optimaal, wanneer elk subpad van dit pad dat korter is dan T zelf ook een kortste pad is. Weinig bestaande methoden eisen dat de routes lokaal optimaal zijn. Nochtans is dit wel zeer

belangrijk aangezien onlogische omwegen absoluut niet wenselijk zijn. De bestaande methoden die hier wel rekening mee houden, gebruiken dan weer de harde bovengrens voor de lengte van de routes.

We geven een literatuuroverzicht en zetten kort onze oorspronkelijke methode uiteen, die later verbeterd werd. Daarna beschrijven we onze verbeterde heuristiek voor het probleem van alternatieve routes. Deze maakt gebruik van *plateaus*. Voor een gegeven $s - t$ query wordt zowel de voorwaartste kortstepadenboom vanuit s berekend als de achterwaartse kortstepadenboom vanuit t. Een plateau is dan een maximaal pad dat in beide kortstepadenbomen voorkomt. Met een $v - w$ plateau kan gemakkelijk een $s-t$ pad gevormd worden door het kortste pad van s naar v te concateneren met het $v - w$ plateau, en dit vervolgens te concateneren met het kortste pad van w naar t. Beide kortste paden kunnen gemakkelijk opgezocht worden in de kortstepadenbomen. Men kan bewijzen dat een dergelijk pad dat een plateau bevat met een lengte T minstens T-lokaal optimaal is. Dit is een zeer interessante eigenschap aangezien voor dergelijke paden niet meer moet getest worden of ze lokaal optimaal zijn, iets wat zeer tijdrovend is. We beschrijven hoe de plateaus kunnen gevonden worden uit de kortstepadenbomen en vervolgens gaan we in detail in op een aantal eigenschappen van plateaus, die we staven met resultaten van experimenten. We stellen vast dat voor een gegeven $s - t$ query meestal zeer veel plateaus bestaan, maar dat het overgrote deel daarvan slechts uit één top bestaat. We stellen ook vast dat de kans op cykels kleiner is bij langere plateaus, en dat de kortstepadenbomen best gesnoeid worden op 1.25 $\times$ het gewicht van het kortste pad. Ook stellen we vast dat heel wat paden die een plateau bevatten met een gewicht $\leq T$ toch nog T-lokaal optimaal zijn. Dit is een belangrijke vaststelling aangezien we deze paden kunnen gebruiken om de oplossing te verbeteren. We bespreken ook een aantal methoden om te testen of een pad lokaal optimaal is, en bepalen aan de hand van experimenten welke methode het meest geschikt is. We stellen vast dat het voor grote grafen veel te tijdrovend is om alle plateaus uit te proberen. Daarom voeren we een bovengrens max in voor het aantal paden waarvoor mag getest worden of ze lokaal optimaal zijn, en beschrijven we een strategie om te

bepalen welke plateaus best getest worden en in welke volgorde, wat ook weer gestaafd wordt met resultaten van experimenten. Tenslotte geven we een overzicht van het volledige algoritme en geven we experimentele resultaten die aantonen hoe snel de heuristiek functioneert voor verschillende stopcriteria en wat de kwaliteit is van de oplossing die gevonden wordt. We vergelijken deze resultaten ook met resultaten in de literatuur.

Tot slot concluderen we in het **zesde hoofdstuk** dat snelle heuristieken kunnen gevonden worden die een goede oplossing vinden voor routeringsproblemen, maar dat exacte algoritmen toch ambitieus blijven. Dit is een van de uitdagingen voor de toekomst. Wellicht zou het parallelliseren van de algoritmen hierbij van pas kunnen komen. Ook zou het interessant zijn nog andere varianten op routeringsproblemen te bestuderen en combineren. Verder zou het ook nuttig zijn de parameters die in dit werk gebruikt worden verder te finetunen aan de hand van een gebruikersstudie.

Bibliography

[1] 9th DIMACS Implementation Challenge on Shortest Paths. *http://www.dis.uniroma1.it/~challenge9/*, 2005.

[2] PTV Company Karlsruhe. *http://www.ptv.de/*, 2005.

[3] Andrew Goldberg's Network Optimization Library. *http://www.avglab.com/andrew/soft.html*, 2011.

[4] Cambridge Vehicle Information Technology Ltd (CAMVIT). Choice Routing. *http://www.camvit.com*, 2011.

[5] NAVTEQ. *http://www.navteq.com*, 2011.

[6] NAVTEQ Network for Developers. *http://www.nn4d.com*, 2011.

[7] ESRI Shapefile Technical Description. *http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf*, 2012.

[8] Google KML documentation. *https://developers.google.com/kml/*, 2012.

[9] Java 6 API. *http://docs.oracle.com/javase/6/docs/api/*, 2012.

[10] J. Añez, T. De La Barra, and B. Pérez. Dual graph representation of transport networks. *Transportation Research Part B: Methodological*, 30(3):209–216, 1996.

[11] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Alternative routes in road networks. In *Proceedings of the 9th International Symposium on Experimental Algorithms*, pages 23–34. Springer, 2010.

[12] V. Akgün, E. Erkut, and R. Batta. On finding dissimilar paths. *European Journal of Operational Research*, 121(2):232–246, 2000.

[13] R. Bader, J. Dees, R. Geisberger, and P. Sanders. Alternative route graphs in road networks. In *Proceedings of the First international ICST conference on Theory and practice of algorithms in (computer) systems*, pages 21–32. Springer, 2011.

[14] M. Barbehenn. A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices. *IEEE Transactions on Computers*, 47(2):263, 1998.

[15] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[16] A. Bernstein. A nearly optimal algorithm for approximating replacement paths and k shortest simple paths in general graphs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 742–755, 2010.

[17] A. Boroujerdi and J. Uhlmann. An efficient algorithm for computing least cost paths with turn constraints. *Information Processing Letters*, 67(6):317–321, 1998.

[18] A. W. Brander and M. C. Sinclair. A comparative study of k-shortest path algorithms. In *Proceedings of 11th UK Performance Engineering Workshop*, pages 370–379, 1995.

[19] O. Bräysy, E. Martínez, Y. Nagata, and D. Soler. The mixed capacitated general routing problem with turn penalties. *Expert Systems with Applications*, 2011.

[20] T. Caldwell. On finding minimum routes in a network with turn penalties. *Communications of the ACM*, 4(2):107–108, 1961.

[21] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[22] J. Clossey, G. Laporte, and P. Soriano. Solving arc routing problems with turn penalties. *Journal of the Operational Research Society*, 52(4):433–439, 2001.

[23] A. Corberán, R. Martí, E. Martínez, and D. Soler. The rural postman problem on mixed graphs with turn penalties. *Computers & Operations Research*, 29:887–903, 2002.

[24] P. Dell'Olmo, M. Gentili, and A. Scozzari. Finding dissimilar routes for the transportation of hazardous materials. In *Proceedings of the 13th Mini-EURO Conference on Handling Uncertainty in the Analysis of Traffic and Transportation Systems*, pages 785–788, 2002.

[25] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[26] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28:652–673, 1998.

[27] E. Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46:48–60, 1990.

[28] E. Erkut, Y. Ülküsal, and O. Yenicerioğlu. A comparison of p-dispersion heuristics. *Computers & Operations Research*, 21(10):1103–1113, 1994.

[29] M. Fidler and G. Einhoff. Routing in turn-prohibition based feedforward networks. *NETWORKING 2004, Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications*, pages 1168–1179, 2004.

[30] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345–, 1962.

[31] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[32] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. Technical report, Microsoft Research, 2004.

[33] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165, 2005.

[34] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better landmarks within reach. *Experimental Algorithms*, pages 38–51, 2007.

[35] A. V. Goldberg and R. F. Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments (ALENEX05)*, pages 26–40, 2005.

[36] E. Gutiérrez and A. L. Medaglia. Labeling algorithm for the shortest path problem with turn prohibitions with application to large-scale road networks. *Annals of Operations Research*, 157(1):169–182, 2008.

[37] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments (ALENEX04)*, pages 100–111, 2004.

[38] J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: a new algorithm and its implementation. *ACM Transactions on Algorithms*, 3(4), 2007.

[39] J. Hershberger and S. Suri. Vickrey prices and shortest paths: What is an edge worth? In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*, pages 252–259, 2001.

[40] W. Hoffman and R. Pavley. A method for the solution of the nth best path problem. *Journal of the Association of Computing Machinery*, 6(4):506–514, 1959.

[41] R. Huang. A schedule-based pathfinding algorithm for transit networks using pattern first search. *GeoInformatica*, 11(2):269–285, 2007.

[42] P. E. Johnson, D. S. Joy, D. B. Clarke, and J. Jacobi. HIGHWAY 3.1: An enhanced HIGHWAY routing model: Program description, methodology, and revised user's manual. Technical report, Oak Ridge National Laboratory, Tenessee (USA), 1993.

[43] R. F. Kirby and R. B. Potts. The minimum route problem for networks with turn penalties and prohibitions. *Transportation Research*, 3:397–408, 1969.

[44] M. Kuby and X. Zhongyi. A minimax method for finding the k best differentiated paths. *Geographical Analysis*, 29(4), 1997.

[45] U. Lauther. An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. *The Shortest Path Problem: Ninth Dimacs Implementation Challenge*, 2006.

[46] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[47] K. Lombard and R. L. Church. The gateway shortest path problem: generating alternative routes for a corridor location problem. *Geographical Systems*, 1(1):25–45, 1993.

[48] E. Q. V. Martins and M. M. B. Pascoal. A new implementation of Yen's ranking loopless paths algorithm. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1:121–133, 2003.

[49] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2nd edition, 2004.

[50] J. C. Micó and D. Soler. The capacitated general windy routing problem with turn penalties. *Operations Research Letters*, 39(4):265–271, 2011.

[51] H. J. Miller and S. Shaw. GIS-T data models. *Information Systems*, 2000.

[52] H. J. Miller and S. Shaw. *Geographic information systems for transportation : principles and applications*. Oxford University Press, 2001.

[53] O. A. Nielsen, R. D. Frederiksen, and N. Simonsen. Using expert system rules to establish data for intersections and turns in road networks. *International Transactions in Operational Research*, 5(6):569–581, 1998.

[54] M. O'Brien. How MapQuest works. pages 1–26, 2006.

[55] L. Roddity. On the k-simple shortest paths problem in weighted directed graphs. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 920–928, 2007.

[56] L. Roddity and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. In *Proceedings of the International Conference on Automata, Languages and Programming (ICALP)*, pages 249–260. Springer Verlag, 2005.

[57] D. Schultes. Fast and exact shortest path queries using highway hierarchies. 2005.

[58] L. Speičys, C. S. Jensen, and A. Kligys. Computational data modeling for network-constrained moving objects. In *Proceedings of the Eleventh ACM International Symposium on Advances in Geographic Information Systems*, pages 118–125, 2003.

[59] S. Vanhove and V. Fack. Routing algorithms with turn restrictions in road networks. In *Proceedings of the 5th Symposium on Location Based Services & Telecartography*, pages 42–45, 2008.

[60] S. Vanhove and V. Fack. k shortest paths with turn restrictions in road networks. In *Proceedings of InterCarto-InterGIS 15*, pages 251–258, 2009.

[61] S. Vanhove and V. Fack. Applications of graph algorithms in GIS. *ACM SIGSPATIAL Special*, 2(3):31–36, 2010.

[62] S. Vanhove and V. Fack. Fast generation of many shortest path alternatives. In *Extended Abstracts Volume, GIScience 2010*, 2010.

[63] S. Vanhove and V. Fack. Locally optimal dissimilar paths in road networks. In *Proceedings of the 10th Cologne-Twente Workshop on Graphs and Combinatorial Optimization*, 2011.

[64] S. Vanhove and V. Fack. An effective heuristic for computing many shortest path alternatives in road networks. *International Journal of Geographical Information Science*, 26(6):1031–1050, 2012.

[65] S. Vanhove and V. Fack. Route planning with turn restrictions: A computational experiment. *Operations Research Letters*, 40(5):342 – 348, 2012.

[66] L. Volker. Route planning in road networks with turn costs. *Student research project, Universität Karlsruhe*, 2008.

[67] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[68] T. Willhalm. Engineering shortest paths and layout algorithms for large graphs. 2005.

[69] S. Winter. Modeling costs of turns in route planning. *GeoInformatica*, 6(4):345–361, 2002.

[70] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[71] F. B. Zhan. Three fastest shortest path algorithms on real road networks: Data structures and procedures. *Journal of Geographic Information and Decision Analysis*, 1(1):69–82, 1997.

[72] F. B. Zhan and C. E. Noon. Shortest path algorithms: an evaluation using real road networks. *Transportation Science*, 32(1):65–73, 1998.

[73] A. K. Ziliaskopoulos and H. S. Mahmassani. A note on least time path computation considering delays and prohibitions for intersection movements. *Transportation Research Part B: Methodological*, 30(5):359–367, 1996.

Index