Multi Via-Node Alternatives for Customizable Contraction Hierarchies

Bachelor's Thesis of

Scott Bacherle

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

Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Second reviewer:	Prof. Dr. Peter Sanders
Advisors:	Michael Zündorf
	Adrian Feilhauer

15.12.2023 – 15.04.2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

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

Karlsruhe, 15.04.2024

[illegible]

. . . .

(Scott Bacherle)

Abstract

Sometimes in a road network, the shortest path between two points is not the only desirable path for road users. The problem of finding alternative paths is well studied and there are many existing algorithms. Alternatives can be found quickly with customizable contraction hierarchies, however these algorithms are outperformed by others. We propose a novel method to find alternative paths by splitting a shortest path into smaller subpaths and searching for alternatives for each one. The alternatives for the subpaths are then linked to form alternatives for the complete path. We study its performance and success rate compared to existing CCH-based algorithms, determining that a combined approach finds more paths with only marginally higher average runtime.

Zusammenfassung

Der kürzeste Weg vom Start- zum Zielpunkt ist nicht immer der einzige erwünschte Weg von Teilnehmern im Straßenverkehr. Darum wird schon länger am Problem der alternativen Wege geforscht. Viele verschiedene Alorithmen existieren, darunter benutzen einige anpassbare Kontraktionshierarchien und können so schnell Resultate finden. Allerdings ist ihre Erfolgsquote niedriger im Vergleich zu anderen Methoden. Darum schlagen wir einen neuen Algorithmus vor, der auch Kontraktionshierarchien benutzt und Alternativen durch Aufteilung des kürzesten Weges in mehrere Segmente findet. Alternativrouten werden für jedes der Segmente gesucht. Durch Verknüpfung der einzelnen Segmente werden Alternativrouten für den gesamten Weg erzeugt. Es wird die Performance und die Erfolgsquote des Algorithmus untersucht und mit bestehenden Algorithmen, die Kontraktionshierarchien verwenden, verglichen. Dies führt zu dem Schluss, dass eine Kombination der Algorithmen verbesserte Ergebnisse liefern kann, ohne dass die Laufzeit zu stark verlangsamt wird.

Contents

1 Introduction

In this chapter, the motivation for this work is explained. We take a look at related work, which includes existing algorithms that use CCHs. Finally, a brief outline of the contents is given.

1.1 Motivation

Searching for alternative paths in addition to the shortest path is a well studied problem. Many algorithms exist to find alternative paths, however better results often come with increased runtimes. In particular, single via-node algorithms using Customizable Contraction Hierarchies (CCHs) perform comparably fast but find less paths [ADGW13]. We devise an algorithm using CCHs, that performs better in cases where the single via-node algorithm fails. This multi via-node algorithm exploits characteristics of these cases but is also useful in general. By running it when single via-node algorithms fail, the fast query times of CCHs provide alternative paths quickly in most cases, while still finding alternative paths otherwise. The proposed algorithm takes advantage of the short runtime of existing algorithms in CCHs by starting multiple searches on smaller segments of the shortest path.

1.2 Related Work

One method to find alternative paths between two points is to find overlapping branches in the search trees of a bidirectional search. This is known as the plateau method [CAM09]. To find the shortest paths between two nodes in a graph, searches are initiated at both. These branch out to find the node which minimizes the distance between both points. After the shortest path is found, the resulting trees also contain the shortest path to many other nodes. By looking for long overlapping sections between both trees, plateaus are found, which are themselves shortest paths in the graph. Paths containing long plateaus make good alternatives. This approach produces satisfying results, however searching for plateaus can take very long.

A more efficient version of this method is to find single via-paths like described by Abraham et al. [ADGW13]. Instead of searching for intersecting arcs in the search trees, only single via-nodes are compared. If a node appears in both search trees, the shortest path to it from both start and end node has already been found. By concatenating both paths, an alternative path is constructed. Different methods to speed up the bidirectional Dijkstra searches are evaluated. They analyze pruning and contraction hierarchies (CH) which both perform significantly faster but find less alternatives. To increase the success rate with CHs, they allow the algorithm to look "*downwards*" during the searches. However, it still finds significantly less alternative paths than the other studied approaches. Luxen et al. speed up the CH algorithm by preprocessing a set of candidate nodes which separate regions of a road network [LS15]. They reduce the number of via-node candidates which are evaluated by introducing a preprocessing phase during a query that finds nodes which are part of a separator between the region of the start and end of a search. This speeds up the algorithm by an order of magnitude and increases the number of alternative paths found.

Other methods besides plateaus and single via paths use penalties. The penalty method performs a shortest path search and afterwards increases the length of every arc included in the path [BDGS11]. This process can be repeated for each new alternative path found, increasing the arc weights of the previously found path. The resulting alternatives often snake around the shortest path, detaching and attaching many times. Plateau and single via paths allow alternatives to only detach and attach to the shortest path once, therefore the alternatives found by the different methods can vary widely. Like alternatives found with the penalty method, multi via-node paths that we examine in later analysis can split away and rejoin the shortest path multiple times. To prevent unattractive routing which is not locally optimal, Bader et al. propose multiple methods like penalizing adjacent arcs of the previous path or introducing a cost to re-joining the shortest path. We do not employ such techniques, opting instead to adapt the approach used for single via paths proposed by Abraham et al. [ADGW13].

1.3 Outline

In the following chapter, necessary graph theory and the used notation is introduced. Furthermore, the construction of CCHs is explained and nested dissection orders are characterized. Afterwards, what makes an alternative path approximately admissible is explained. We specify how to verify this in the fourth chapter and define an algorithm to find alternative paths using CCHs. In the fifth chapter, the multi via-node algorithm is introduced, its implementation is explained and multiple iterations are discussed. Afterwards, the recursive variant is explained. In chapter seven, the different algorithms are evaluated, results from multiple queries in road networks are compared. The eighth chapter is the conclusion, where results are summarized and future work is discussed.

2 Preliminaries

In the following chapter we define basic graph theory used in this work and introduce Customizable Contraction Hierarchies, which preprocess a graph to find shortest paths more efficiently. We talk about how CCHs are constructed and how shortest path queries using an elimination tree work. Lastly, nested dissection orders for road networks and their usage to rank nodes is explained.

2.1 Graph Theory

Road networks are modeled as directed graphs. A directed graph is defined as two sets $G = (V, A)$, where V contains vertices and A contains arcs (x, y) connecting the vertices. An arc (x, y) points from its *tail* vertex x to the *head* vertex y. For further analysis all graphs of road networks are regarded as simple, meaning loops are removed and multi-arcs are resolved by inserting vertices, as these features are not relevant to finding shortest paths. An undirected graph $G = (V, E)$ contains a set of edges $\{x, y\}$ instead of arcs, where all edges are bidirectional. Directed Graphs are converted to undirected graphs by ignoring the direction of each arc. Each arc has a non-negative *weight* $\ell : A \longrightarrow \mathbb{R}^+$ which may denote the travel time or distance in a road network. Moreover, when converting a directed graph to be undirected, asymmetric weights for arcs in opposite directions need to be preserved, so two different weight functions $\ell_{up} : E \longrightarrow \mathbb{R}^+$ and $\ell_{down} : E \longrightarrow \mathbb{R}^+$ are used. They are defined using a *rank* function on G. The function $rank : V \longrightarrow \{0, 1, \ldots, |V| - 1\}$ is bijective and assigns each vertex a unique integer value $rank(v)$. Given an edge $\{s, t\}$ with $rank(s) < rank(t)$, the weight functions are defined as $\ell_{up}(\{s, t\}) = \ell(s, t)$ and $\ell_{down}(\{s, t\}) = \ell(t, s)$. If there is no corresponding arc in the directed graph for a direction of travel, the matching length function for the edge is set to ∞.

The *neighborhood* $N : V \longrightarrow \mathcal{P}(V)$ of a vertex v contains all nodes which are connected via one arc with v, regardless of the direction. A *path* P in a graph connects two vertices, where s is its start and t its end vertex. It is a sequence of arcs $P = (a_0, a_1, \ldots, a_n)$ where the head vertex of the previous arc matches the tail vertex of the following arc. The *length* $\ell : \mathcal{P}(A) \longrightarrow \mathbb{R}^+$ of a path P is calculated as the sum of weights of all arcs in P. Given a path in an undirected graph, the decision of which weight $\ell_{up}(\{x, y\})$ or $\ell_{down}(\{x, y\})$ is added up for each edge is based upon the direction of travel along the edge $\{x, y\}$. In contrast to $\ell(P)$, the *hop-length* $|P|$ is the number of arcs contained in P. Sometimes the order of the arcs in a path is not actually needed, then P can be regarded as a set of arcs. Given a path P and another path or set of arcs S, the set-operators union $(P \cup S)$, intersection $(P \cap S)$ and subtraction $(P \setminus S)$ are defined as if P were a set of arcs. The *distance* $dist(s, t)$ of two vertices in G is the length of the shortest path P_{Opt} between them. If no path between them exists, $dist(s, t)$ is ∞.

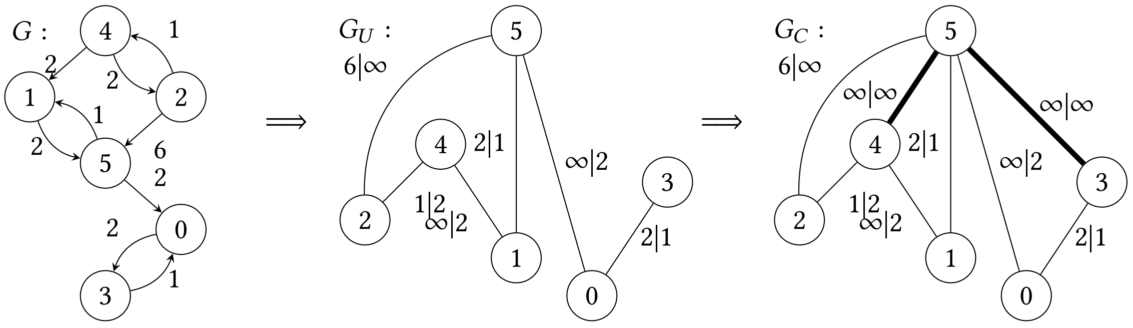


Figure 2.1: The metric independent phase of CCH construction. Vertices are labeled with their rank. At first, vertices are ordered according to their rank and all arcs are converted to edges with preserved directional weights. All edges e are labeled with their weights $\ell_{up}(e)|\ell_{down}(e)$. Lastly, shortcuts are inserted to construct a chordal graph.

2.2 Customizable Contraction Hierarchies

Customizable Contraction Hierarchies (CCH) provide an efficient way to run multiple shortest path queries on a given graph for different start and end pairs. Specifically using nested dissection orders, as described in [DSW16]. Given a rank function for a graph G, a CCH can be constructed for G by converting it to an undirected graph, inserting shortcuts, running basic customization and then deriving the elimination tree from the resulting graph. The algorithm used for inserting shortcuts is based on the work by Habib et al. [HMPV00]. Its implementation and use in CCHs is described in more detail by Buchold et al. [BWZZ20].

2.2.1 Metric Independent Construction

Firstly, G is converted to an undirected graph G_U. Then the $parent : V \to V$ of each vertex v is identified as the vertex with the smallest rank among all neighbors $w \in N(v)$ with $rank(v) < rank(w)$. If v has no neighbors with higher rank, $rank(v)$ is set to ∞. Afterwards, a chordal graph $G_C = (V, E_C)$ is constructed by inserting edges from $parent(v)$ to each neighbor $u \in N(v)$ with $rank(u) > rank(parent(v))$. A graph is chordal if all induced circles C with $|C| > 3$ have an arc between at least two non-adjacent vertices. The additional edges in G_C are called shortcuts and only inserted if the edge $\{parent(v), u\}$ is not already in G_U. Parents of vertices are updated while new edges are inserted. The length of the shortcuts ℓ_{up} and ℓ_{down} is set to ∞ and modified in the following step. Vertices are processed in ascending order of their rank. This phase is demonstrated for an example graph in Figure 2.1.

2.2.2 Customization

During basic customization, triangle inequality is established for all lower triangles regarding ℓ_{up} and ℓ_{down}. Given an edge $\{x, y\} \in E_C$, a lower triangle is formed by the vertices $\{x, y, z\}$ if the edges $\{z, x\}$ and $\{z, y\}$ exist and $rank(z) < min\{rank(x), rank(y)\}$. To fulfill triangle inequality, the length of all edges $\{x, y\}$ may not exceed the sum of the shortest edges $\{z, x\}$ and $\{z, y\}$ given all lower triangles $\{x, y, z\}$. Let $rank(x) < rank(y)$, for all lower triangles $\{x, y, z\}$ the equations

$$\ell_{up}(\{x, y\}) = min\big(\ell_{up}(\{x, y\}), \ell_{down}(\{z, x\}) + \ell_{up}(\{z, y\})\big)$$

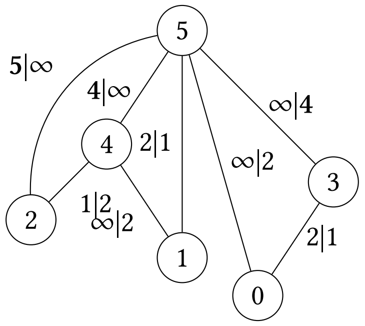


Figure 2.2: The chordal graph G_C after customization. Note the emphasized changed weights.

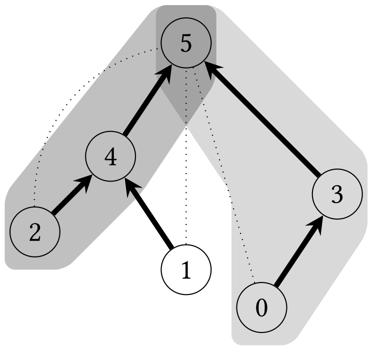


Figure 2.3: The elimination tree T_G, each arc oriented towards its parent vertex. The search spaces of vertex 2 and 0 are marked. Dotted edges are not part of the elimination tree.

and

$$\ell_{down}(\{x,y\}) = min\big(\ell_{down}(\{x,y\}), \ell_{up}(\{z,x\}) + \ell_{down}(\{z,y\})\big)$$

are set during basic customization. See figure Figure 2.2 for instance. Edges are customized in ascending order of their lower ranked vertex, this ensures that once the customization starts for an edge $\{x,y\}$, the edges in all lower triangles of $\{x,y\}$ already have their customized length.

2.2.3 Shortest Path Queries

To search for shortest paths in the original graph G, the *elimination tree* T_G is used as described in [BSW18]. The root v_{root} of T_G is the vertex with maximum rank, arcs point up from each vertex v to its parent in G_C. The arcs in the path from any vertex v to v_{root} consist of the vertices in the search space $SS(v) : V \longrightarrow \mathcal{P}(V)$ of v, like shown in Figure 2.3. When searching for a shortest path from s to t, the vertices in $SS(s) \cup SS(t)$ are relaxed in ascending order by climbing up in T_G.. During the search the current length and ancestor of all vertices in G is tracked twice. For the upward search starting at s current upward lengths $length_{up} : V \longrightarrow \mathbb{R}^+$ and upwards ancestors $anc_{up} : V \longrightarrow V$ are tracked, the downward search from t tracks downward lengths $length_{down} : V \longrightarrow \mathbb{R}^+$ and downward ancestors $anc_{down} : V \longrightarrow V$. At the start of the search all values are set to ∞, except $length_{up}(s)$ and $length_{down}(t)$, which are both set to 0.

A vertex v in the upward search starting at s is relaxed by updating $length_{up}$ and anc_{up} of each upper neighbor $w \in N(v)$ in G_C if $dist(s,v) + \ell_{up}(\{v,w\})$ is less than the current value of $length_{up}(w)$. Similarly, for the downward search starting at t, $dist(v,t) + \ell_{down}(\{v,w\})$ is compared with $length_{down}(w)$. The distances to v are read from the currently tracked lengths, as $dist(s,v) = length_{up}(v)$ and $dist(v,t) = length_{down}(v)$. This is because a vertex is only relaxed once the shortest path to it has already been found. After relaxation, the length values are reset. The searches meet as soon as the lowest ranked vertex out of the intersection of search spaces $SS(s) \cap SS(t)$ is reached. Upward and downward lengths for all vertices are then relaxed together until a vertex with no parent is reached. During this process, the vertex $x \in SS(s) \cap SS(t)$ that minimizes $length_{up}(x) + length_{down}(x)$ is kept track of.

After the root is reached, the final vertex x is contained in the shortest path. By descending the chain of its ancestors $anc_{up}(x)$ and $anc_{down}(x)$ until s or t is reached, the shortest $s - x$ and $x - t$ paths are constructed. Combined they form the shortest $s - t$ path in G_C, its length equals $length_{up}(x) + length_{down}(x)$ and is the same as the length of P_{Opt} in G.

However, some edges in the path were added to G_C as shortcuts and are therefore not contained in the original graph G. These edges have to be deleted and replaced by the original arcs to construct P_{Opt}. This process is called *path unpacking*, edges are recursively replaced until the path contains only edges that correspond directly to arcs in G. Each edge $\{x, y\}$ whose length was updated during customization is deleted from P_C and replaced by the edges of the lower triangle $\{x, y, z\}$ that caused the weight of $\{x, y\}$ to be updated during customization.

To reset the CCH after a search, the weights found by the upwards and downward search are reset. The weights found for vertices in $SS(s) \setminus SS(t)$ and $SS(t) \setminus SS(s)$ are reset right after each vertex is relaxed. Weights in the intersection of search spaces $SS(s) \cap SS(t)$ are only reset after all paths are constructed. They are reset by iterating through the intersection and setting all weights to ∞.

2.2.4 Nested Dissection Orders

The performance of the CCH queries depends largely on the size of the search spaces of s and t. In smaller search spaces less vertices need to be relaxed, leading to faster queries. Rank functions derived from nested dissection orders provide a way to restrict search spaces to sublinear size on most road graphs[BCRW16].

Using *nested dissection*, well balanced graph separators for road networks can be found by repeatedly subdividing a graph and finding separators in all subgraphs [LRT79]. It works by removing a set of vertices S from an undirected graph $G = (V, E)$ forming at least two unconnected subgraphs. Afterwards, separators are found in each of the resulting subgraphs, this process is repeated until all subgraphs only have one vertex remaining. A separator S is well balanced if $|S| \leq f(|V|)$ and $|V_j| \leq b \cdot |V|$, for all resulting subgraphs $G_j = (V_j, E_j)$. Whereby $b \in (0, 1)$ is a parameter and $f : \mathbb{N} \rightarrow \mathbb{R}$ is a monotonically increasing function with $f \in O(\sqrt{n})$. Usual values for these parameters in road networks are $b = \frac{2}{3}$ and $f \in O(\sqrt[3]{|V|})$ [DSW16]. A rank function derived from such separators will usually result in search spaces of sublinear size according to Bauer et al. [BCRW16].

The rank function is derived by assigning vertices in separators of higher order a higher rank. Separators of higher order are found first, the order of a separator is lower if it is found later. Given the number of nodes already included in separators B, the vertices v in a newly found separator S have ranks in the range of $|V| - B - |S| \leq rank(v) < |V| - B$.

Given a shortest path P_{Opt} in a graph, we identify a vertex v in the path as a *bottleneck*, if removing it from the graph increases the distances between all other vertices on the path by at least $(1 + \varepsilon)$ for $\varepsilon \in \mathbb{R}$. The parameter ε limits the stretch of an alternative path as described in Section 3.1. Navigating around a bottleneck comes with a significant cost and is therefore not feasible when searching for alternative paths, as the necessary detour is too long. Bottlenecks are often part of natural separators like bridges and tunnels. These are also good separators in road networks, leading to their inclusion in higher order separators and a high $rank(v)$ [EG08].

3 Alternative Paths

The problem of alternative paths consists of a start s and end node t in a graph G, the goal is to find viable alternative paths between the nodes. Simply searching for any paths from s to t does not yield appealing results for road users. Taking an alternative path is always a trade off compared to the shortest path. For this reason a path from s to t is deemed a viable alternative only if it seems reasonable compared to the shortest path. A path P from s to t must fulfill three criteria to be a viable alternative worth finding. They concern the length of the alternative, possible detours and the amount of shared sections between paths [ADGW13].

3.1 Uniformly Bounded Stretch

The path P may not be significantly longer than the shortest path P_{Opt} from s to t. There is an upper limit to the travel time road users are willing to spend taking an alternative path. Dividing the length of an alternative P by the length of the shortest path $dist(s,t)$ yields the stretch

$$stretch : \mathcal{A} \to \mathbb{R}^{+};$$

$$stretch(P) = \frac{\ell(P)}{dist(s,t)}$$

of P. By setting a maximum stretch, all alternatives with a higher stretch can be dismissed. The stretch of all sub-paths P' of P must also be lower than the maximum allowed stretch. If s' and t' are the start and end nodes of a subpath P', let P'_{Opt} be the shortest path between them. With $(1+\varepsilon)$ denoting the maximum allowed stretch, P is only viable if

$$\ell(P') \leq (1+\varepsilon) \cdot \ell(P'_{Opt}).$$

This must also be true for $P' = P$. For further analysis ε is chosen as 0.25, allowing alternative paths to be up to 1.25 times longer than the shortest route.

3.2 Local Optimality

Following the path must feel natural, there should be no unnecessary detours. All local routing decisions must be optimal, otherwise taking a path is not appealing. Let P' be a sub-path of a path P and let $\ell(P')$ not exceed a previously chosen length. Local routing decisions are evaluated by checking wether $\ell(P') = \ell(P'_{Opt})$ is fulfilled, where P'_{Opt} is the shortest path between the start and end vertices of P'. The subpath P' must optimal if

$$\ell(P') \leq \alpha \cdot \ell(P_{Opt})$$

for $\alpha \in (0,1)$. A typically chosen value is $\alpha = 0.2$, which we also use in later analysis.

3.3 Limited Sharing

Finding many similar alternative paths is not valuable for a user, even if they fulfill the previous criteria. All paths must only share limited arcs with the shortest path P_{Opt} and all other alternative paths. The length of all arcs used in P that are shared with other paths $P_1, P_2, \dots P_i$ may not exceed $\gamma \cdot \ell(P_{Opt})$ for $\gamma \in [0, 1]$. If A_{Prev} is the set containing all arcs used for previously found viable paths, then

$$l(P \cap A_{Prev}) \leq \gamma \cdot \ell(P_{Opt})$$

is the requirement P must fulfill for limited sharing. We choose $\gamma = 0.8$ for later analysis.

Subsequently, the admissabilty of an alternative depends on all previous found paths, increasing the importance of the order in which paths are evaluated. Paths are ranked and evaluated according to a function $f(P)$, whose ideal definition arguably depends on the end user. Abraham et al. use a multi-dimensional approach, sorting paths in nondecreasing order among other variables according to their length and overlap [ADGW13]. In contrast, we always prefer shorter alternatives over longer ones, choosing $f(P) = \ell(P)$. We do this because shorter paths are more attractive to users.

3.4 Single Via-Node Alternatives

A subset of all alternative $s - t$ paths are *single via-node paths.* Given a node v, the single via-node path through v is constructed by concatenating the shortest $s - v$ and $v - t$ path. Finding paths this way provides a good set of candidates for viable alternative paths. To find such paths, first a set of via-nodes is identified, then all $s - v$ and $v - t$ paths are built and lastly the concatenated paths are verified. These paths can be found using any search algorithm, in the following sections we detail how to find them using CCHs.

3.4.1 Finding Via-Nodes

Via-node candidates are all nodes between s and t which could be used to construct a single via-node path. Using CCHs, they are found efficiently during a shortest path query in the intersection of search spaces $SS(s) \cap SS(t)$. The shortest path from s to t is routed via the vertex $z \in SS(s) \cap SS(t)$ that minimizes $dist(s, z) + dist(z, t)$; Each vertex $v \in SS(s) \cap SS(t)$ besides z is a via-node candidate. The length of each via-node path is known once all vertices in the intersection are relaxed, as it equals $dist(s, v) + dist(v, t)$. These distances are tracked by the upwards and downwards searches with $length_{up}(v)$ and $length_{down}(v)$. Therefore, a length limit to via-paths is enabled efficiently by dismissing all via-node candidates whose path length exceeds the limit. After all vertices in the intersection are filtered, the weights tracked by the searches are reset.

3.4.2 Shortest Paths

After all via-node candidates are identified, the resulting via-node paths are built in ascending order of path weight. Each path is constructed the same way as in Section 2.2.3, except the vertex z is replaced with a via-node candidate v and the construction stops if the path routes via any previously used via-node candidate. All via-nodes for which attempts were already made to construct the via-path, either successfully or not, are tracked. If during the

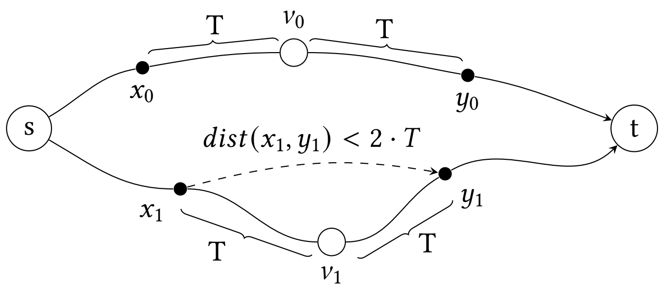


Figure 3.1: Two T-locally optimal single via-node paths with their respective via node. The first one passes the T-test, the second one fails because it is not $2 \cdot T$-locally optimal, as the shortest path between x_1 and y_1 (indicated by the dashed line) does not contain v_1.

construction of a path one of these prior vertices is included in the path, the construction stops and the via-node candidate is discarded. This prevents unnecessary detours, as for any via-path which includes a prior via-node candidate, a via-path with a shorter detour was already analyzed.

3.4.3 Verification

To determine whether any path P is a viable alternative path, the three criteria need to be checked with different tests. Limited Sharing can be directly verified by adding up the lengths of all arcs shared between P and previously found paths. Like described in Section 3.3, the viability of each alternative depends on the arcs used in all previous alternative paths S. All paths must therefore be verified in ascending order of their length. This is accomplished by sorting all candidates for alternative paths according to their length and verifying limited sharing in that order. To verify limited sharing, $\ell(P \cap S)$ is compared against $\gamma \cdot \ell(P_{Opt})$. If it is greater, the alternative path is dismissed.

Verifying local optimality and uniformly bounded stretch directly takes more effort, as a distance query is required for all sub-paths of P. This takes too long in practice, so the criteria are verified indirectly by checking local optimality at via-nodes and only calculating the stretch of P itself. According to Abraham et al., all alternative paths with a stretch of $(1 + \varepsilon)$ that pass a T-test for $T = \beta \cdot\ dist(s, t)$ with $0 < \varepsilon < \beta < 1$, have $\frac{\beta}{\beta-\varepsilon}$-uniformly bounded stretch [ADGW13]. We can therefore verify alternatives P by only calculating the stretch of P itself and performing a single T-test. The stretch of P is verified during the via-node candidate search, by setting the length limit of admissible via-nodes to $(1 + \varepsilon) \cdot \ell(P_{Opt})$.

However, this may discard viable alternatives which are actually locally optimal or include alternatives whose stretch is not uniformly bounded. A path is only guaranteed to not be locally optimal on a distance of $2 \cdot T$ if it fails a T-Test. This is illustrated in Figure 3.1. In contrast, a verified path is only guaranteed to have $\frac{\beta}{\beta-\varepsilon}$-uniformly bounded stretch, which may exceed ε. Therefore all verified alternative paths are only approximately admissible.

3.4.3.1 T-Test

With a T-test, local optimality is verified only at points along a path where detours can occur. It checks local optimality for a given length $T = \alpha \cdot \ell(P_{Opt})$ and only requires a single shortest path query. As a single via-node path consists of shortest $s - v$ and $v - t$ paths, detours can only occur around the via-node. The local optimality of a path P with via-node v is verified by identifying the first vertex x before v on P which is at least T away from v. The same is done for the vertex y on the other side of v. The vertices x and y are set to s and t respectively if the subpath has a length of less than T. Finally, a distance query is performed for x and y and the result $dist(x, y)$ is compared against the combined distance of x to y on path P. If $dist(x, y)$ is shorter, P includes a detour and fails the T-test. Otherwise, the path P is T-locally optimal. Once limited sharing is verified and the alternative P passes the T-test, all used arcs are added to the set of used arcs S. Once all alternative paths are verified, they are returned in ascending order of their length.

3.4.4 Detour Adjusted

Some paths which are T-locally optimal may fail the T-test, as it only guarantees paths to not be $2 \cdot T$-locally optimal if they do not pass. Therefore alternative paths which share up to 40% with the shortest path are all dismissed. This effectively reinforces the sharing parameter γ to 0.6 instead of 0.8. Paths P which share up to 80% with P_{Opt} may still be viable alternatives, so local optimality and uniformly bounded stretch are adjusted to only apply to the detour in P, a common solution proposed by Abrahams et al.[ADGW13]. With detour adjusted local optimality, a T-test is performed with

$$T = \alpha \cdot \ell(P \setminus P_{Opt}).$$

Uniformly bounded stretch is also applied only to the detour, with

$$\ell(P \setminus P_{Opt}) \leq (1 + \varepsilon) \cdot \ell(P_{Opt} \setminus P)$$

being the updated criteria. This can no longer be verified during the via-node candidate search directly, so an additional step is added afterwards to calculate the overlap between each alternative path and the shortest path and the updated criteria is checked. The initial guard during the candidate search is kept, as the stretch of the whole path will always be lower than the stretch of the detour.

4 Multi Via-Node Alternatives

In this chapter we propose a method to search for alternatives by splitting a shortest path at a node x and starting two separate searches. We discuss how this split node is selected and introduce an algorithm to find multi via-node alternative paths. With x being a node on the shortest path from s to t, the multi via-node algorithm splits the path at x, searches for alternatives on the section from s to x and from x to t, then concatenates the results to form alternatives for the whole path.

4.1 Selection of Split Node

The success rate of the multi via-node algorithm is largely influenced by the the selection of split node x. If x is a random vertex located close to s or t, the multi via-node algorithm may not perform better than the single via-node algorithm, as the section from s to x is too small to find any alternatives and the $x - t$ subpath is too similar to the whole $s - t$ path. Therefore the location of x in relation to the whole $s - t$ path is important.

However, the rank of x also influences the success rate of the multi-via node algorithm. If x is located near s or t and its rank is high, it may lead to a small intersection of search spaces $SS(s) \cap SS(s)$ Let x be a node with high rank which is located near s on the shortest path and far away from t. Because the upward search quickly reaches x compared to the downward search, $|SS(t) \setminus SS(s)|$ ends up relatively large. This can lead to some via-node candidates being contained in $SS(t) \setminus SS(s)$ instead of $SS(s) \cap SS(t)$, where they are not found by a single via-node alternative search. Splitting the path at x and searching for via-node candidates on both subpaths may yield better results.

With $rank(x)$ preferably being higher, simply starting and ending the search at x likely produces unbalanced search-trees. Because x has a high rank, the search space $SS(s) \setminus SS(x)$ would include many via-node candidates, as its size is significantly larger than $SS(x) \setminus SS(s)$. This leads to less alternatives being found, it is therefore assumed in the following that splitting the route at x and starting two searches actually terminates these at the vertices located next to x in the shortest path. Given $(x_{Pre}, x), (x, x_{Succ}) \in P_{Opt}$, the recursive searches actually go from s to x_{Pre} and x_{Succ} to t. An alternative of the whole $s - t$ path is then constructed from three parts, an $s - x_{Pre}$ path, the shortest path from x_{Pre} to x_{Succ} including x and an $x_{Succ} - t$ path. All alternative paths P found this way have the same structure:

$$P = ((s, v_0), \ldots, (x_{Pre}, x), (x, x_{Succ}), \ldots, (v_n, t))$$

4.1.1 Middle

Generally, more alternatives are found for longer paths. Choosing the split node x in the middle of the shortest path leads to x having the maximum possible distance from s and t. This could lead to the most paths being found on both sides of x, resulting in more alternatives being found for the whole path. However, the selected vertex will have a random rank which can lead to unbalanced sizes of the search spaces of $|SS(s)|$ and $|SS(t)|$ with less alternatives

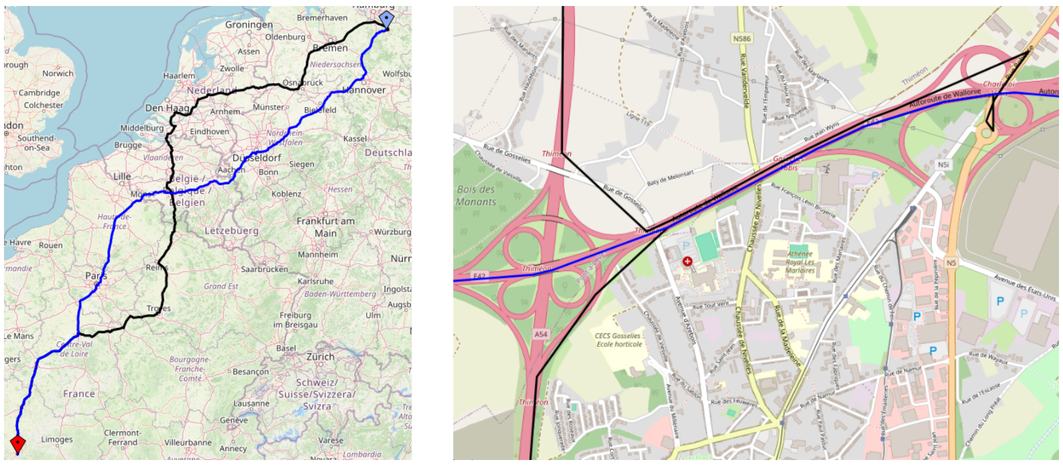


Figure 4.1: Results of a multi via-node search with split node chosen as the node with the middle index. No T-tests are performed. The shortest path is drawn blue. Left is an overview of both paths, right is zoomed in. The black alternative seems viable from the overview but it includes a local detour through the split node. (Map data obtained from OpenStreetMap: openstreetmap.org/copyright)

being found. The rank of the vertex can also be very low, forcing all alternative paths to include a minor vertex in their routing even if faster alternatives exist, like in Figure 4.1. This increases the number of paths being found which fail a T-test later, as the routing via x can be a detour.

4.1.1.1 Geographic Middle

Choosing the vertex on P_{Opt} furthest away from s and t ensures that both subpaths have the maximum possible length. The distance of x from s and t is calculated with the same metric as the length of the whole path $\ell(P_{Opt})$. Because road networks have discrete distance values instead of being continuous, both sub-paths are not guaranteed to have the same length. The exact node chosen is either the first node that has a distance of at least $\frac{1}{2} \cdot \ell(P_{Opt})$ from s or its predecessor, depending on which node x minimizes $|dist(s,x) - dist(x,t)|$. This ensures that the node x balances the length of both subpaths.

4.1.1.2 Medium Index

Maximizing the number of vertices on both sides of the split node x can be done via the hop length of the shortest path $|P_{Opt}|$. The arc in the middle of P_{Opt} has the index $\lfloor \frac{1}{2} \cdot |P_{Opt}| \rfloor$. The split vertex x can then be chosen as the head of the arc $P_{Opt}(\lfloor \frac{1}{2} \cdot |P_{Opt}| \rfloor)$. This disregards the actual distance between vertices, so it does not maximize the size of both sub-paths but it provides an approximation. Choosing x this way also includes the maximum number of vertices in both subpaths which can also lead to more alternatives being found. Each vertex is a potential intersection where alternative paths can diverge or converge back with P_{Opt}.

4.1.2 Highest

Choosing a vertex with a higher rank as split node makes routing through it less likely to be a detour. Vertices with a higher rank are part of higher order separators which makes the existence of a shorter path between two alternatives for the different sub-paths around the split node x less likely. Because if such an alternative path exists, it must also route via a vertex from the same separator, so it will likely be found by single via-node searches. If no single via-node alternative paths are found, then P_{Opt} may contain a bottleneck. Like described in Section 2.2.4, a bottleneck is often part of a high order separator and therefore has a high rank. The shortest path P_{Opt} is split at a bottleneck by choosing the node with the highest rank x as the split node. Given all arcs $(v, w) \in P_{Opt}$, the split vertex x must fulfill

$$rank(x) \geq max(rank(v), rank(w)).$$

This can lead to one of the subpaths being much shorter than the other, resulting in no alternatives being found for the shorter subpath. However, alternatives found for the longer subpath can still be combined with the shorter subpath to form viable alternatives.

4.1.3 Highest in the Middle

Short subpaths can be avoided by limiting the search for the vertex with highest rank to vertices with a sufficient distance from s and t. The split node x is still the node with the highest order but only if $dist(s, x)$ and $dist(x, t)$ is greater than $\delta \cdot \ell(P_{Opt})$ for $\delta \in [0, \frac{1}{2})$. The same principle can be applied to hop length, the hop length of the shortest subpaths from s to x and x to t must be at least $\delta \cdot |P_{Opt}|$. This guarantees a minimum length for both subpaths while still choosing a potential bottleneck as a split point. This also makes it less likely for alternatives to include a detour through x, as the rank of x is higher compared to choosing the vertex with random rank in the middle of the path. A bottleneck close to s or t may still be preferred as a split vertex over one creating more equal subpaths, as the existence of a bottleneck on a path leads to less alternatives being found with single via-node searches. The parameter δ is chosen as 0.3. This ensures a sufficient distance from the start and end node while leaving enough room to choose a high ranked vertex.

To limit the selection of vertices to that with sufficient length from s and t, the first vertex $v \in P_{Opt}$ with $dist(s, v) \geq \delta \cdot \ell(P_{Opt})$ is identified. Then the first vertex $w \in P_{Opt}$ with $dist(w, t) \geq \delta \cdot \ell(P_{Opt})$ starting from t is calculated. Let P_v be the shortest path from s to v and P_w be the shortest path from w to t. A linear search to find the vertex x with maximum rank is performed on the set $P_{Opt} \setminus P_v \setminus P_w$ of remaining arcs. Concerning hop length, the selection of nodes is limited by starting the search for the highest-ranked vertex x at index $\lfloor \delta \cdot |P_{Opt}| \rfloor$ and ending it at $\lfloor |P_{Opt}| - \delta \cdot |P_{Opt}| \rfloor$.

4.2 Performing Separate Searches

After choosing a split node x, identifying its predecessor x_{Pre} and successor x_{Succ}, single via-node searches on both subpaths are performed. The first one searches for paths from s to x_{Pre}, the second for paths from x_{Succ} to t. These searches run independently from each other, the found alternatives are combined afterwards. During the combination phase, the requirements for alternative paths are checked and only routes that fulfill all three are returned. The algorithm used for the single via-node search is described prior in Section 3.4 and only needs small adjustments.

The parameters ε, α and γ define the requirements uniformly bounded stretch, local optimality and limited sharing of alternative paths P_{Alt}. These same requirements do not need to be fulfilled by all subpaths of P_{Alt}. For instance, choose a maximum stretch of 1.25 and let P'_L, P'_R be alternative paths to the $s - x$ and $x - t$ subpaths P_L and P_R of P_{Opt} respectively. Let the stretch of the alternatives be $stretch(P'_L) = 1.3$ and $stretch(P'_R) = 1.1$ with $\ell(P_L) = \ell(P_R)$. The alternative P'_L will be discarded by single via-node search using the same parameter $\varepsilon = 1.25$ as the multi via-node search. However, combined with P'_R, the whole alternative has sufficiently low stretch of $stretch(P'_L \cup P'_R) = 1.2$.

4.2.1 Relaxed Stretch

To find the most alternatives with a multi via-node search, the maximum allowed stretch ε is increased for both single via-node searches. The prior example assumes $\ell(P_L) = \ell(P_R)$, in general the stretches are weighed against the length of $\ell(P_L)$ and $\ell(P_R)$ in relation to $dist(s, t)$ before adding them up. To account for the path in the middle from x_{Pre} to x_{Succ}, let P_M be the shortest $x_{Pre} - x_{Succ}$ path. The maximum allowed stretch for the multi via-node search is limited by

$$\ell(P'_L \cup P_M \cup P'_R) = \ell(P'_L) + \ell(P_M) + \ell(P'_R) \le (1 + \varepsilon) \cdot dist(s, t)$$

To find the maximum allowed stretch for P'_L, we assume a minimal stretch of 1 for P'_R, which means $\ell(P'_R) = \ell(P_R)$ and therefore

$$\ell(P'_L) \le (1 + \varepsilon) \cdot (\ell(P_L) + \ell(P_M) + \ell(P_R)) - (\ell(P_M) + \ell(P_R)) .$$

By solving

$$(1 + \varepsilon_L) \cdot \ell(P_L) = (1 + \varepsilon) \cdot (\ell(P_L) + \ell(P_M) + \ell(P_R)) - (\ell(P_M) + \ell(P_R))$$

for $(1 + \varepsilon_L)$ we get the maximum allowed stretch for the first search

$$1 + \varepsilon_L = 1 + \varepsilon \cdot \frac{\ell(P_L) + \ell(P_M) + \ell(P_R)}{\ell(P_L)} = 1 + \varepsilon \cdot \frac{dist(s, t)}{\ell(P_L)} .$$

The maximum stretch for the second search $(1 + \varepsilon_R)$ is calculated the same way by assuming $\ell(P'_L) = \ell(P_L)$:

$$1 + \varepsilon_R = 1 + \varepsilon \cdot \frac{dist(s, t)}{\ell(P_R)} .$$

4.2.2 Relaxed Sharing

The same principle is applied to limited sharing, defined by the parameter γ. If the same value for γ is used in the single via-node search, paths that share more than $\gamma \cdot dist(s, x_{Pre})$ or $\gamma \cdot dist(x_{Succ}, t)$ are discarded even though they could fulfill the requirement when combined with other paths. To prevent this, the sharing parameter γ needs to be raised for the single via-node searches. Let S contain all arcs from other paths, let P'_L and P'_R be paths from and to x that form a viable alternative with limited sharing $\ell((P'_L \cup P_M \cup P'_R) \cap S) \le \gamma \cdot dist(s, t)$.

As an aside, it is assumed here that the subpaths are disjunct $(P'_L \cap P'_R) = \emptyset$ but because both searches are performed separately this cannot be guaranteed. However, paths that are not disjunct must include a local detour and will later fail a T-test that prevents them from being included in the results, see Section 4.3. Therefore

$$\ell\big((P'_L \cup P_M \cup P'_R) \cap S\big) = \ell(P'_L \cap S) + \ell(P_M \cap S) + \ell(P'_R \cap S)$$

is assumed. The maximum admissible amount of sharing for the first search γ_L is calculated by assuming $\ell(P'_R \cap S) = 0$,

$$\ell\big((P'_L \cup P_M \cup P'_R) \cap S\big) = \ell(P'_L \cap S) + \ell(P_M \cap S) \leq \gamma \cdot dist(s,t)$$

and using $dist(s,t) = \ell(P_L) + \ell(P_M) + \ell(P_R)$ for the shortest $s - x_{Pre}$ and $x_{Succ} - t$ paths P_L, P_R:

$$\gamma_L \cdot \ell(P_L) = \gamma \cdot (\ell(P_L) + \ell(P_M) + \ell(P_R)) - \ell(P_M \cap S).$$

Solving for γ_L results in

$$\gamma_L = \gamma \cdot \frac{\ell(P_L) + \ell(P_M) + \ell(P_R)}{\ell(P_L)} - \frac{\ell(P_M \cap S)}{\ell(P_L)} = \gamma \cdot \frac{dist(s,t) - \ell(P_M \cap S)}{\ell(P_L)}.$$

For the second search, γ_R is calculated accordingly as

$$\gamma_R = \gamma \cdot \frac{dist(s,t) - \ell(P_M \cap S)}{\ell(P_R)}.$$

4.2.3 Relaxed Local Optimality

The length of the subpath on which a T-test needs to be performed $\alpha \cdot dist(s,t)$, does not change between the multi via-node search and both single via-node searches. The parameter α is adjusted to

$$\alpha_L = \alpha \cdot \frac{dist(s,t)}{\ell(P_L)}$$

for the first search. This way

$$\alpha_L \cdot \ell(P_L) = \alpha \cdot \frac{dist(s,t)}{\ell(P_L)} \cdot \ell(P_L) = \alpha \cdot dist(s,t)$$

remains unchanged. The parameter α_R of the second search is set to

$$\alpha_R = \alpha \cdot \frac{dist(s,t)}{\ell(P_R)}.$$

Subsequently, if $\ell(P_L)$ or $\ell(P_R)$ is smaller than $2 \cdot \alpha \cdot dist(s,t)$ most alternative paths found by the respective search will be discarded. This is because alternatives in the single via-node searches can share more than $\gamma = 80\%$ with P_{Opt}, so they always fail a T-test with $\alpha = 0.2$.

To avoid this, detour adjusted local optimality like described in Section 3.4.4 can be used. To verify local optimality for subpaths without discarding paths that can be combined to form viable alternative paths, the distances T_L and T_R tested by the single via-node searches need to be relaxed. The distance $T = \alpha \cdot \ell(P_{Alt} \setminus P_{Opt})$ verified by the multi via-node search depends on the amount of sharing of P_{Alt} with P_{Opt} and is therefore unknown before the combination phase. Discarding viable alternatives is avoided by setting T_L and T_R relative to an ideal corresponding alternative in the other search. For the first search, this assumes a maximum overlap of P_M and P'_R with P_{Opt}, $\ell(P_M \setminus P_{Opt}) = \ell(P'_R \setminus P_{Opt}) = 0$. Therefore the parameter α_L is set to α, as $T_L = \alpha_L \cdot \ell(P'_L \setminus P_L)$ then equals

$$T_L = \alpha \cdot \big(\ell(P'_L \setminus P_{Opt}) + \ell(P_M \setminus P_{Opt}) + \ell(P'_R \setminus P_{Opt})\big) = \alpha \cdot \ell\big((P'_L \cup P_M \cup P'_R) \setminus P_{Opt}\big) = T.$$

Like in Section 4.2.2 we assume no sharing between paths from both searches. Additionally, the stretch of the detour is bounded by $(1+\varepsilon_L) = (1+\varepsilon)$, which can be shown by also assuming no overlap between P_M, P'_R and P_{Opt}:

$$(1+\varepsilon_L) \cdot \ell(P_L \setminus P'_L) = (1+\varepsilon_L) \cdot \ell\big((P_L \cup P_M \cup P_R) \setminus (P'_L \cup P_M \cup P_R)\big) = (1+\varepsilon) \cdot \ell\big((P_{Opt}) \setminus (P'_L \cup P_M \cup P_R)\big).$$

The parameters for the second search are set correspondingly to $\alpha_R = \alpha$ and $(1+\varepsilon_R) = (1+\varepsilon)$.

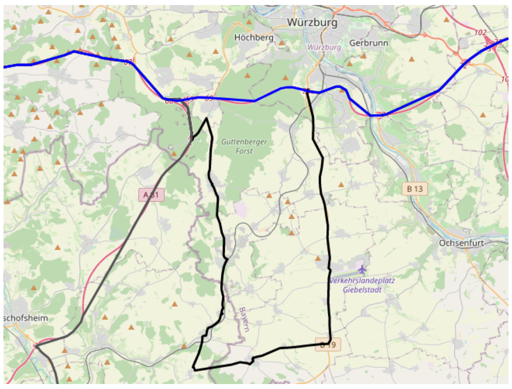


Figure 4.2: The results of a single via-node search with adjusted parameters before T-tests are performed. The shortest path is blue. The black alternative path splits off from blue and takes a long detour before rejoining blue. The routing of black is unappealing because it is not locally optimal. (Map data obtained from OpenStreetMap: openstreetmap.org/copyright)

4.2.4 Trade-Offs

Adjusting the parameters ε, α and γ is necessary when trying to find as many alternative paths as possible. It also increases the number of alternatives taken into consideration during the combination phase, hindering performance. Paths P found by single via-node searches with $stretch(P) > (1+\varepsilon)$ or sharing higher than γ often take long detours and are not viable alternatives, regardless of what other path they are combined with. For instance, see Figure 4.2. Such paths are not locally optimal and will therefore be discarded by a T-test. A more efficient solution would be to filter them out sooner in the process by not relaxing the parameters for the single via-node searches, using $\varepsilon_1 = \varepsilon_2 = \varepsilon$ and $\gamma_1 = \gamma_2 = \gamma$ instead. However, this also decreases the total number of alternatives found.

4.3 Combining Paths

During the combination phase, alternative paths from both single via-node searches are combined to form alternatives for the whole path P_{Opt}. All single via-node paths from one search are combined with the single via-node paths from the other search and the three criteria for alternative paths are checked. To make this process more efficient, the weight calculated by the single via-node searches is reused. The order in which alternatives are combined and checked is relevant to which alternative paths are returned.

4.3.1 Order

Like described in Section 3.3, the order in which paths are verified changes the results, as sharing between paths depends on which paths are verified first. In multi via-node searches the importance is even greater as it has to be decided which paths are combined to form complete alternatives. We use the same function to evaluate paths as in Section 3.3, namely $f(P) = \ell(P)$. So shorter alternatives are always preferred to longer ones. Let the results of two searches be $R_L = (P_L^0, P_L^1, \cdots, P_L^n)$ and $R_R = (P_R^0, P_R^1, \cdots, P_R^m)$ be two sequences of paths sorted in ascending order of their length. They include the shortest $s-x$ path P_L^0 and the shortest $x-t$ path P_R^0.

Firstly, all paths are combined to form paths of the form $P_L^i \cup P_M \cup P_R^j$. A greedy algorithm minimizing $\ell(P_L^i \cup P_M \cup P_R^j)$ is used to combine them. At the beginning, the shortest path P_L^0 in R_L is concatenated with the middle segment P_M and the shortest path P_R^0 in R_B. Then P_L^0 is concatenated with P_M and the second shortest path in R_B. This process repeats until P_L^0 is combined with the longest path P_R^m in R_B or the length of the resulting path is higher than $(1+\varepsilon) \cdot dist(s,t)$. As all subpaths are sorted in ascending order, the length of subsequent combinations will also exceed this limit. By discarding such combinations, all paths have bounded stretch and no unnecessary T-tests or sharing calculations are performed. Afterwards, the second shortest path in R_A is combined with the paths in R_B in ascending order. This process stops after the longest path P_L^n in R_A is combined with the paths in R_B.

The length of combined paths can be calculated efficiently by reusing the lengths of the paths calculated by the single via-node searches, as $\ell(P_L^i \cup P_M \cup P_R^j) = \ell(P_L^i) + \ell(P_M) + \ell(P_R^j)$. To calculate the length of the middle segment $\ell(P_M)$, the lengths of the arcs (x_{Pre}, x) and (x, x_{Succ}) are added. To ensure shorter alternatives are verified before longer ones, all combined paths are sorted in increasing order according to $f(P) = \ell(P)$ before verification.

4.3.2 Verification

While the single via-node searches verify all found alternatives, this does not guarantee all possible combinations to be viable alternatives. Two combined paths which are viable alternatives for their respective segment of the shortest path may not be viable alternatives for the whole path. This is not only because of the relaxed search parameters but also because of the routing via a split node, which bears potential for local detours. As such, uniformly bounded stretch, limited sharing and local optimality are verified for the combined alternatives. Uniformly bounded stretch is already certified while ordering the paths, as described in Section 4.3.1.

4.3.2.1 Limited Sharing

Limited sharing is verified by intersecting each alternative P_{Alt} with the set of arcs used in previously verified paths S. Then the length of all arcs in the intersection is calculated and compared with the maximum permissible length

$$\ell(P_{Alt} \cap S) \leq \alpha \cdot \ell(P_{Opt})$$

The set S initially contains all arcs used in the shortest path, ensuring alternatives of one section can be combined with the shortest path of the other section, provided the resulting path has sufficiently low sharing. Unlike the length of combined paths, limited sharing needs to be calculated again and cannot be reused from the single via-node searches. As sharing is calculated in respect to all previously found paths, the set of previously used arcs can differ between the single and multi via-node search. For instance, let P_L^1 and P_L^2 be alternative paths found by the first search with shared sections, so $P_L^1 \cap P_L^2 \neq \emptyset$. If no combined path containing P_L^1 is admitted, its arcs are never added to S. Therefore the length of shared arcs between P_L^2 and S can decrease compared to the single via-node search.

4.3.2.2 Local Optimality

After verifying the limited sharing of a combined path, a T-test is performed at the split node of the path. If it passes the test, all its arcs are added to the set of previously used arcs. In conjunction with the T-tests already performed by the single via-node searches, the single

T-test sufficiently proves that the path is locally optimal. Let x be the used split node, while P_L and P_R are the subpaths of a combined alternative P found by the single via-node searches. Furthermore, let v be the end node of path P_L, while w is the start node of path P_R.

Theorem 4.1: *If P_L and P_R are T-locally optimal and $P = P_L \cup \big((v, x), (x, w)\big) \cup P_R$ passes a T-test at x, then P is T-locally optimal.*

Proof. Suppose P passes the test and let P' be a subpath of P with $\ell(P') \leq T$. If P' does not contain x, then P' is a subpath of P_L or P_R and therefore a shortest path. Otherwise P contains x and is a subpath of the portion covered by the T-test performed at x and therefore also a shortest path. ■

Additional T-tests on P at all via-nodes would only confirm previous results and are therefore omitted.

4.3.2.3 Detour-Adjusted

Using detour adjusted local optimality and stretch requires some modifications. The basic process remains unchanged, alternative paths from both searches are combined to form alternatives for the whole path. The combined paths are filtered according to stretch and sorted in ascending order of their length. Limited Sharing and local optimality are verified and valid alternative paths are returned.

First alternative paths P'_L of one segment are combined with all paths P'_R from the other segment and P_M to form P_{Alt}. The stretch of P_{Alt} is the same as the combined stretch of the paths:

$$\ell(P_{Alt} \setminus P_{Opt}) = \ell\big((P'_L \cup P_M \cup P'_R) \setminus (P_L \cup P_M \cup P_R)\big) = \ell(P'_L \setminus P_L) + \ell(P'_R \setminus P_R).$$

We assume $P'_L \cap P_R = P'_R \cap P_L = \emptyset$. Graphs could be constructed where this assumption is broken, however we could not observe this in real road graphs. Because P_M is always included in all paths and has a fixed direction of travel, paths that share arcs with P_R before reaching x_{Pre} must include a detour. The same holds true for paths that share arcs with P_L after crossing x_{Succ}.

Combined alternatives whose stretch exceeds $(1+\varepsilon) \cdot (P_{Opt} \setminus P_{Alt})$ are discarded. Afterwards, the remaining alternatives are sorted in ascending order of their length. Limited sharing and detour adjusted local optimality are then verified for all alternatives. For local optimality, the T-tests run by the single via-nodes cannot sufficiently prove that the resulting combined alternative is T-locally optimal, as the exact length of t is unknown before combining alternatives. Let $T_L = \alpha \cdot \big((P'_L \cup P_R) \setminus P_L\big) =$ be the value of T used in the first search. If P'_L is then combined with an alternative path P'_R with $P'_R \cap P_R \neq \emptyset$, the distance for which P_{Alt} needs to be locally optimal T is greater than $min(T_L, T_R)$. Therefore additional T-tests are run at all via-nodes with parameter T. All paths which fulfill both limited sharing and local optimality are returned.

5 Recursion

A path can include multiple bottlenecks, which makes it difficult to find a single good split node. For instance, see Figure 5.1. If the path is split at a bottleneck x, the $s - x$ path or $x - t$ path still includes another bottleneck, so a single via-node search will likely not find any alternatives. Instead of running a single via-node search on the path including another bottleneck, a multi via-node search can be initiated. The resulting paths from the multi via-node search can then be combined with the paths from the other search to find alternatives for the whole $s - t$ path. This is not limited to only one side of x, both single via-node searches can be replaced with multi via-node searches. The process can also be repeated recursively, as each multi-via node search can initiate new multi via-node searches.

Multiple bottlenecks are not necessarily the only instance where more alternatives are found with recursion. A vertex does not need to be a bottleneck to have a high rank, as high order separators do not have to include bottlenecks. See Figure 5.1 for instance. Simply splitting a path at the node x with the highest rank does not guarantee x to be a bottleneck. If a path contains a bottleneck v, as well as a vertex w from a high order separator with $rank(v) < rank(w)$, then w is chosen as split vertex during a multi via-node search instead of v. A recursive multi via-node search will eventually choose v as the split node, increasing the likelihood of alternatives being found.

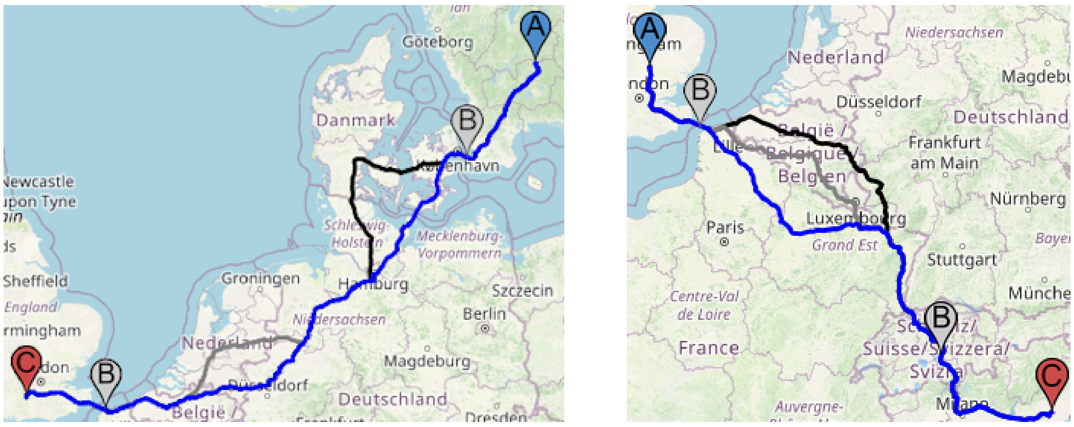


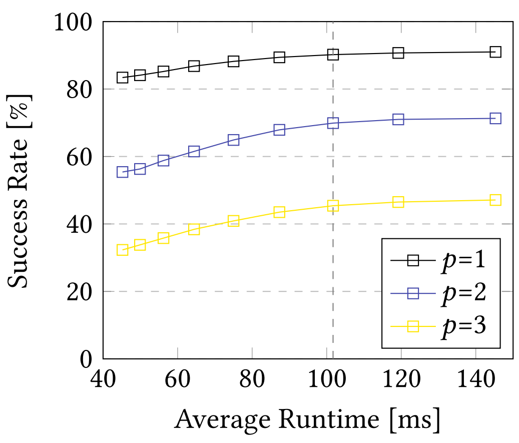
Figure 5.1: Successful recursive multi-via node searches. The shortest path on the left crosses two bottlenecks, their location is marked with **B**. The path on the right crosses multiple high order separators (location marked with **B**) with only the upper one being a bottleneck. Shortest paths are drawn in blue. (Map data obtained from OpenStreetMap: openstreetmap.org/copyright)

5.1 Termination

The recursion must terminate at some point, as generally less alternatives are found for shorter paths. However, we do not know an absolute length value for when diminishing returns are expected, so a minimal distance between start s' and end node t' is chosen relative to the length of the original path. If $dist(s', t') < \eta \cdot dist(s, t)$ for $\eta \in (0, 1)$, the recursion is stopped and a single via-node search is performed. With the vertices s and t being the start and end points of the original path, we call $\eta \cdot dist(s, t)$ the termination distance d of the recursive search.

Choosing a smaller η increases the number of initiated recursive searches, which leads to longer runtimes in exchange for more alternatives found. To determine a good value for η, 5000 random queries were ran for different $\eta \in (0, 1)$ using the same testing environment as in Chapter 6. The results are displayed in Figure 5.2. Choosing $\eta = 0.3$ guarantees at least a single alternative path is found in 90% of cases. Pushing η above 0.5 will allow at most a single new recursive search at each level, lowering the number of paths p being found. As η approaches zero, the success rate converges around 90% for $p = 1$, 70% for $p = 2$ and 45% for $p = 3$ but the average runtime increases by up to 45%. We choose $\eta = 0.3$ for further analysis as this presents a sensible compromise between success rate and runtime.

Figure 5.2: Tracking success rate and average runtime as η changes. A recursive search is successful if at least p alternative paths are found. Each point in the plot is the resultfor different η. The gray line marks $\eta = 0.3$.



η	Success Rate p=1 [%]	Success Rate p=2 [%]	Success Rate p=3 [%]	Average Runtime [ms]
0.1	91.0	71.3	47.1	145.3
0.2	90.7	71.0	46.5	119.2
0.3	**90.2**	**69.9**	**45.4**	**101.7**
0.4	89.4	67.9	43.5	87.3
0.5	88.2	64.9	40.9	75.0
0.6	86.8	61.5	38.4	64.4
0.7	85.2	58.8	35.8	56.2
0.8	84.1	56.3	33.8	49.9
0.9	83.4	55.4	32.3	45.2

5.2 Algorithm

Only minor modifications need to be made to the multi via-node algorithm to add recursion. Extra measures are added to keep track of the termination criteria. First an unmodified single via-node search is run and all paths are added to the set of potential alternative paths. Then the length of the shortest path is checked against the termination criteria and if it does not exceed d, the recursive search stops and returns the paths found prior.

Otherwise, a split node is chosen according to any of the methods detailed in Section 4.1. New smaller recursive searches are initiated for the segments on both sides of the split node. The smaller searches are both launched with relaxed parameters. Local optimality and stretch can also be adjusted to detour, like described in Section 4.2.3. The stretch of combined alternative paths found for the different segments is first verified, then they are sorted in ascending order of their length. The criteria for alternative paths are verified like in Section 4.3. Depending on whether detour adjusted local optimality is used, either a T-test is performed only at the split node or at each via-node. A version without adjusting to detour is sketched in Algorithm 5.1.

If local optimality is not detour-adjusted, all searches will already use the final distance parameter T in the T-test. Therefore, local optimality can only be violated at the split-node. However, with detour adjusted local optimality, only a minimal value for T is used during each search, therefore after combining two subpaths, another T-test is required at each via-node. Paths found from normal multi via-node searches have a single split node and up to two via-nodes originating from single via-node searches. The recursive algorithm can combine many more single via-node paths, leading to paths having many more via-nodes where detours can occur. We want to filter out non-viable alternative paths early, so a T-test is performed on each level of recursion at every via-node. As a result, verifying local optimality can take significantly more effort when adjusting for detour. Skipping these prior T-tests would not necessarily increase performance, as it increases the number of considered paths, leading to more T-tests in the final combination phase.

Algorithm 5.1: A simplified recursive algorithm utilizing the median index to determine the split node.

```
Input:  A shortest path $P_{Opt}$, parameters $\alpha \in \mathbb{R}^+$ characterizing locally optimal
        distances, $\gamma \in \mathbb{R}^+$ defining maximum sharing, $\varepsilon \in \mathbb{R}^+$ bounding stretch and
        termination distance $d \in \mathbb{R}^+$.
Output: A set of alternative paths including $P_{Opt}$.

Function recursiveSearch($P_{Opt}$,$\alpha$,$\gamma$,$\varepsilon$):
    $R \longleftarrow singleViaSearch(P_{Opt},\alpha,\gamma,\varepsilon)$
    if $\ell(P_{Opt}) < d$ then
        return $R$
    $n \longleftarrow |P_{Opt}|$
    $m \longleftarrow \lfloor \frac{n}{2} \rfloor$
    $P_L \longleftarrow P_{Opt}[0, 1, \dots, m-2]$
    $P_M \longleftarrow P_{Opt}[m-1, m]$
    $P_R \longleftarrow P_{Opt}[m+1, \dots, n-1]$
    $R_L \longleftarrow recursiveSearch\left(P_L, \alpha \cdot \frac{\ell(P_{Opt})}{\ell(P_L)}, \gamma \cdot \frac{\ell(P_L)+\ell(P_R)}{\ell(P_L)}, \varepsilon \cdot \frac{\ell P_{Opt}}{\ell(P_L)}\right)$
    $R_R \longleftarrow recursiveSearch\left(P_R, \alpha \cdot \frac{\ell(P_{Opt})}{\ell(P_R)}, \gamma \cdot \frac{\ell(P_L)+\ell(P_R)}{\ell(P_R)}, \varepsilon \cdot \frac{\ell P_{Opt}}{\ell(P_R)}\right)$
    $R \longleftarrow sort(R \cup (R_L \times P_M \times R_R))$
    $S, V \longleftarrow \emptyset$
    forall $P \in R$ do
        if $\ell(P) \leq (1+\varepsilon) \cdot \ell(P_{Opt})$ and $\ell(P \cap S) < \gamma \cdot \ell(P_{Opt})$ then
            if $T - Test(P, \alpha)$ then
                $S \longleftarrow S \cup P$
                $V \longleftarrow V \cup \{P\}$
    return $V$
```

6 Evaluation

In this chapter we evaluate the different algorithms presented in prior chapters. The single-via node algorithm is denoted as **SV** and represents the baseline of our analysis. The algorithm **HV** splits at the highest node, **MV** and **GV** use either the medium index or the geographic middle as a split point respectively. Restricting the selection of highest nodes to the middle regarding hop-length is done by **HMV**, doing the same for the geographic middle yields **HGV**. Detour-adjusted algorithms are denoted by the prefix **D**, recursive variants use the prefix **R**.

We evaluate all methods on the DIMACS graph of *Europe* [DGJ09]. This graph has already been used in prior works, for instance to evaluate the single via-node algorithm in Abraham et al.[ADGW13]. The used length function is travel time. The graph is preprocessed using an order obtained through nested dissection.

The parameters characterizing admissible alternative paths are set to $\varepsilon = 0.25$, $\gamma = 0.8$ and $\alpha = 0.2$ for all queries. A total of 50000 queries are run for every test. The program was compiled with gcc version 12.3.0 using $-O3$- compliation. All experiments were performed on a machine with an Intel Xeon Skylake SP Gold 6144 CPU using 192 GB of DDR4 2666 MHz RAM. Initializing the CCH with the pre-computed order took 19.9 seconds.

6.1 Split Nodes

We find the best method to choose the split node by comparing all algorithms in Table 6.1. The success rate in relation to the number of desired alternatives p is recorded, along with the average and maximum query time. The average and maximum stretch of the whole alternative path is also tracked. Lastly, the average and maximum percentage of sharing between paths is included. We also include the baseline **SV** and **D-SV** to increase comparability.

The split node chosen by **D-HV** has the highest overall success rate, **MV** finds the least amount of routes. This confirms that vertices with high rank hinder single via-node searches, as cutting out a high ranked vertex from a shortest path increases success rates. This can also be seen by the increased number of routes found by **HMV** and **HGV** compared to their counterparts **MV** and **GV**. While **MV** and **GV** both choose vertices with random rank as a split point, **HMV** and **HGV** will favor higher ranked vertices. We also observe that adjusting local optimality to detour increases the success rate for every algorithm and lowers the average runtime at the cost of higher sharing.

While **D-HV** has the highest success rate, it also has the highest average runtime among the detour adjusted algorithms. The same is true for the non-detour adjusted algorithms, **HV** finds the most routes and needs longer on average than all comparable algorithms. However, the difference is not very high, **D-HV** takes ca. twice as long as baseline, the fastest algorithm **D-GV** takes ca. 1.4 times as long.

As a result of adjusting local optimality and stretch of every path to the length of its detour, the maximum stretch of alternatives found using detour-adjusted algorithms is reduced. All algorithms using normal local optimality find routes with a maximum stretch of 25%, however this is not the case for their detour adjusted counterpart. This can be explained by the stretch

Table 6.1: Comparison of multi via-node algorithms using different split nodes. Success rate is tracked as the number of desired alternative paths p varies.

	Success Rate			Time		Stretch		Sharing	
Algo	p=1 [%]	p=2 [%]	p=3 [%]	Average [ms]	Max [ms]	Average [%]	Max [%]	Average [%]	Max [%]
HV	41.9	12.7	2.7	52.3	217.3	2.9	25.0	23.9	80.0
MV	29.5	6.3	0.9	40.4	206.0	2.2	25.0	18.6	80.0
GV	31.8	7.6	1.3	42.7	198.0	2.5	25.0	18.9	80.0
HMV	37.9	9.7	1.9	45.7	203.2	2.7	25.0	23.5	80.0
HGV	40.1	10.8	2.1	47.4	200.1	2.8	25.0	25.0	80.0
D-HV	**54.5**	**19.7**	**5.5**	30.1	293.6	2.1	23.3	34.6	80.0
D-MV	35.4	9.6	1.7	19.8	216.3	1.2	20.1	23.7	80.0
D-GV	37.0	11.1	2.3	19.4	221.1	1.4	21.6	24.0	80.0
D-HMV	47.2	15.3	3.6	24.5	274.1	1.6	19.3	31.3	80.0
D-HGV	49.5	16.2	4.0	26.1	274.3	1.7	19.3	32.8	80.0
SV	52.9	26.3	9.6	15.1	123.1	4.6	25.0	23.4	80.0
D-SV	**61.4**	**33.8**	**13.8**	13.1	120.0	3.5	24.9	31.7	80.0

not being limited by the length of the path but by the length of its detour. To have a limited stretch of 25% for its whole length, the overlap of an alternative P with the shortest path needs to be zero, as

$$\ell(P \setminus P_{Opt}) \leq (1+\varepsilon) \cdot \ell(P_{Opt} \setminus P)$$

is equivalent to

$$\ell(P) \leq (1+\varepsilon) \cdot \ell(P_{Opt}) - \varepsilon \cdot \ell(P \cap P_{Opt}),$$

which implies $\ell(P) \leq (1+\varepsilon) \cdot \ell(P_{Opt})$. A multi via-node search will never find an alternative path with no overlap, as each path found contains the same middle section around the split node.

6.2 General Performance

We compare the performance of multi via-node searches against single via-node algorithms in Table 6.2. The structure of Table 6.2 is the same as explained in Section 6.1. The comparison is limited to the best performing multi via-node searches **HV** and **D-HV**. We also include the recursive algorithm.

All detour adjusted algorithms perform better than their non detour adjusted counterpart, as they have a higher success rate and lower average runtime. By focusing on the detour adjusted algorithms, we see that **D-HV** has a lower success rate than **D-SV**, while also taking approximately twice as long. The lower success rate is explained by the requirement of including the split node in every alternative path, limiting the available selection of via-node candidates.

Table 6.2: Comparison between the best performing single via-node, multi via-node and recursive algorithms. Success rate is tracked as the number of desired alternative paths p is varied.

	Success Rate			Time		Stretch		Sharing	
Algo	p=1 [%]	p=2 [%]	p=3 [%]	Average [ms]	Max [ms]	Average [%]	Max [%]	Average [%]	Max [%]
SV	52.9	26.3	9.6	15.1	123.1	4.6	25.0	23.4	80.0
HV	41.9	12.7	2.7	52.3	217.3	2.9	25.0	23.9	80.0
R-HV	76.0	49.4	28.1	189.0	589.5	5.5	25.0	41.0	80.0
D-SV	61.4	33.8	13.8	13.1	120.0	3.5	24.9	31.7	80.0
D-HV	54.5	19.7	5.5	30.1	293.6	2.1	23.3	34.6	80.0
DR-HV	**90.2**	**69.1**	**45.3**	101.2	803.2	3.9	24.9	55.4	80.0

The recursive algorithm finds the most alternative paths, with the success rate of **DR-HV** being 90% for $p = 1$. This is almost as high as the algorithms detailed by Abraham et al. [ADGW13]. **DR-HV** being more successful than **D-SV** is expected, as the former will find all via-node candidates of the latter. However, the average runtime of **DR-HV** is an order of magnitude higher than **D-SV**.

Therefore, the multi via-node search is not very attractive to use instead of a single via-node search, due to its lower success rate. The recursive search is also unattractive because of its significantly increased runtime. This leads us to conclude that multi via-node searches should not be performed instead of single via-node searches but in addition to them. If multi via-node algorithms find alternative paths in cases where single via-node algorithms fail, then a combined approach can increase success in return for longer runtimes. A multi via-node search may only be performed in cases where the single via-node search does not find the number of desired alternative paths.

6.3 Conditional Performance

To test whether multi via-node searches find alternative paths in cases where single via-node searches fail, we restrict our testing in this section to cases where **D-SV** finds no alternative paths. We restrict the analysis to detour-adjusted searches, as they have a higher success rate and lower runtime. The analysis contains algorithms with all methods to find different split nodes. For each different method we also include the recursive version. The result can be seen in Table 6.3. It is immediately clear that all recursive algorithms produce better results than their non-recursive counterpart but take roughly three times as long.

The results confirm our theories regarding bottlenecks and higher ranked vertices on shortest paths, as the success rate and performance of **D-HV** remains almost unchanged. In approximately 54% of cases it finds at least a single alternative, compared to 54.5% in general. This means that the performance of **D-HV** on a given shortest path is almost entirely unaffected by whether **D-SV** finds any alternatives for the path. Therefore, the usage of **D-HV** lends itself exceptionally well to a combination with **D-SV**. When **D-SV** does not find any alternatives, we can run **D-HV** to still return an alternative in more than half of all cases. The runtime of **D-HV** is approximately twice that of **D-SV** meaning a combined approach takes roughly three times longer than running **D-SV**.

Combining **D-SV** with any other non-recursive algorithm is not really useful, as they perform worse than in general. Only **DR-HV** has a higher success rate than **D-HV**, but its average runtime is ca. three times longer than **D-HV** and almost 10 times longer than **D-SV**. The non-recursive algorithm **D-HV** is faster than every recursive variant. It also finds routes more often except for **DR-HV**. We conclude that the best trade-off between success rate and runtime is a combined approach of running **D-HV** in cases where **D-SV** fails.

Table 6.3: Comparison of multi via-node and recursive algorithms in cases where **D-SV** finds no alternative paths. Success rate is tracked as the number of desired alternative paths p varies.

	Success Rate			Time		Stretch		Sharing	
Algo	p=1 [%]	p=2 [%]	p=3 [%]	Average [ms]	Max [ms]	Average [%]	Max [%]	Average [%]	Max [%]
D-HV	**53.7**	**17.6**	**5.4**	33.8	209.6	1.7	23.0	34.2	80.0
D-MV	14.9	1.6	0.2	16.5	181.0	0.5	14.6	10.0	80.0
D-GV	10.6	1.6	0.4	15.4	199.5	0.3	18.2	7.0	80.0
D-HMV	31.7	7.7	1.7	22.6	178.4	0.9	17.1	21.1	80.0
D-HGV	38.9	10.5	2.7	26.9	199.1	1.1	15.6	25.8	80.0
DR-HV	74.3	40.0	17.7	99.8	722.7	2.3	23.0	49.5	80.0
DR-MV	17.1	2.1	0.3	44.2	334.2	0.5	14.6	11.6	80.0
DR-GV	18.9	4.7	1.4	51.8	487.1	0.6	18.2	12.8	80.0
DR-HMV	39.0	11.7	3.1	59.7	376.7	1.1	17.1	26.7	80.0
DR-HGV	45.4	14.5	3.9	64.5	381.1	1.2	15.6	30.9	80.0

6.4 Combined Performance

We evaluate a combination of the algorithms **D-SV** and **D-HV** called **D-CHV**. It works by first running **D-SV**. If it does not find the desired number of alternatives p, the results are stored and **D-HV** runs. All resulting and stored paths are sorted in ascending order of their length and the checks to only admit viable alternative paths like described in Section 4.3.2 are performed. All viable paths are then returned. Table 6.4 shows the result of running the algorithms for different desired numbers of alternatives p. We also include the combined recursive variant **DR-CHV** in the analysis.

The average runtime increases with p, this effect is more noticeable for **D-CHV** and **DR-CHV** compared to **D-SV**. Searching for at least $p = 2$ routes with **DR-CHV** takes 57% longer than for $p = 1$. This is a stark contrast to **D-SV**, which only takes approximately 13% longer. The increase for **D-CHV** lays between both values and is approximately 37%. Increasing p to 3 also incurs a more significant performance penalty with **DR-CHV** than with **D-SV** or **D-CHV**. Compared to $p = 2$, **DR-CHV** takes 28% longer, **D-CHV** takes 21% longer and the runtime of **D-SV** only increases by 9%. This is mostly due to the decreasing success rate of **D-SV**

Table 6.4: Comparison of **D-CHV** and **DR-CHV** with **D-SV**. Searches are stopped after the desired number of alternatives is found. Success rate is tracked as the number of desired alternative paths p varies.

			Time		Stretch		Sharing	
p	Algo	Success Rate [%]	Average [ms]	Max [ms]	Average [%]	Max [%]	Average [%]	Max [%]
1	D-SV	61.4	9.8	72.9	2.5	24.9	31.1	80.0
	D-CHV	**86.2**	22.5	234.2	3.4	24.9	47.4	80.0
	DR-CHV	90.0	46.2	756.9	3.4	24.9	50.0	80.0
2	D-SV	33.8	11.1	103.1	3.2	24.9	31.6	80.0
	D-CHV	**59.9**	31.3	307.4	3.9	24.9	49.8	80.0
	DR-CHV	68.6	71.2	821.2	3.9	24.9	52.9	80.0
3	D-SV	13.8	12.1	111.1	3.4	24.9	31.7	80.0
	D-CHV	**35.7**	37.9	308.7	4.0	24.9	51.1	80.0
	DR-CHV	44.7	91.4	842.3	3.9	24.9	54.4	80.0

with higher values of p. The less succesful **D-SV** is, the cases where a longer multi via-node search is initiated rises. With the number of cases rising, the runtime of the combined algorithms increases. By multiplying the average runtime of the multi via-node algorithm with the percentage of cases where **D-SV** fails, we get an estimate for the runtime of the combined algorithm in those cases. This product is added to the average runtime of **D-SV** to estimate the runtime of the combined algorithm. From this, the runtime changes due to the decreasing success rate of **D-SV** can be calculated, They are displayed in Table 6.5 and differ only slightly from our measurements. The higher increases in runtime of **D-CHV** and **DR-CHV** are therefore mostly caused by the decreasing success rate of **D-SV** and not because multi via-node searches take significantly longer to find more routes.

The recursive variant **DR-CHV** still has the highest success rate, however **D-CHV** comes close for $p = 1$, with alternative paths found in approximately 86% of cases compared to 90%. This gap increases for $p = 2$ and $p = 3$, where **DR-CHV** performs significantly better. Both combined approaches find routes in significantly more cases than **D-SV** but the average sharing is higher.

The single via-node approach is by far the fastest algorithm, with lower average and maximum runtime. To find a single alternative route, the runtime of **D-CHV** is more than twice as high on average and **DR-CHV** even needs at least four times as long. For a single alternative route, it therefore seems most effective to use **D-CHV**, as **DR-CHV** only finds marginally more routes but needs almost twice as long. With the number of desired routes increasing, the success rate of **DR-CHV** stays higher compared to the other algorithms, however its runtime is also double that of **D-CHV**. The algorithm **D-CHV** sits between **D-SV** and **DR-CHV** in terms of success rate and runtime, representing a sensible compromise.

Table 6.5: Comparison of inferred and measured runtime change of **D-CHV** and **DR-CHV** as p increases.

Increase of p	Algo	Runtime Change Inferrred [%]	Runtime Change Measured [%]
1-2	D-CHV	46.5	39.1
	DR-CHV	59.7	54.1
2-3	D-CHV	23.2	21.1
	DR-CHV	27.2	28.4

7 Conclusion

We detailed how multi via-node alternative paths can be successfully constructed using single via-node alternatives. The methods to confirm approximate admissibility were adapted to multi via-node paths and the resulting algorithm was detailed. Different methods to determine split nodes were introduced and the evaluation showed that vertices of higher rank make good split nodes. Splitting a path at the highest ranked vertex leads to the most paths being found compared to all other split nodes. We confirmed experimentally that more multi via-node paths can be found through recursion, exceeding the results possible with a single split node. However, the runtime of the recursive algorithms is an order of magnitude higher than that of existing single via-node algorithms.

Our evaluation also proved that multi via-node paths can be found in cases where the single via-node algorithm fails. In fact, we found no correlation between the performance of the multi via-node algorithm and the success rate of the single via-node algorithm. This leads us to conclude that multi via-node algorithms are best used in conjunction with single via-node algorithms. When a single via-node search fails, the multi via-node algorithm which splits the path at the highest ranked node can be initiated to still return a viable alternative path more than 50% of the time.

7.1 Future Work

This analysis proved that multiple via-nodes are a viable method to find alternatives using CCHs. Different approaches could be pursued to increase the success rate and optimize the used algorithm. Optimizations for the CCH and single via-node algorithm used during the analysis are possible. Like described by Dibbelt et al. the search can be pruned to reduce the number of nodes which are relaxed [DSW16]. This could also be implemented for our queries as it does not significantly effect the number of available via-node candidates. Another technique they discussed is using vectors to store multiple weights for each edge directly, which is also not implemented in our CCH.

Using recursion simplified the implementation of an algorithm using multiple split nodes, however this could also be achieved iteratively. The k-highest split nodes on a path could be identified directly, increasing predictability and preventing long runtimes. After the split nodes are identified, single via-searches could be initiated between them. Another possible improvement is to not discard alternative paths which fail the T-test and instead try to replace local detours with locally optimal paths. If a T-test finds a shorter path between two nodes, the shorter subpath could be inserted into the tested path to produce a locally optimal alternative.

Bibliography

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

[BCRW16] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. "Search-space size in contraction hierarchies". In: *Theoretical Computer Science* Volume 645 (2016), pp. 112–127. ISSN: 0304-3975. DOI: *https://doi.org/10.1016/j.tcs.2016.07.003*.

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

[BSW18] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. "Real-Time Traffic Assignment Using Fast Queries in Customizable Contraction Hierarchies". In: *17th International Symposium on Experimental Algorithms (SEA 2018)*. Edited by Gianlorenzo D'Angelo. Vol. 103. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 27:1–27:15. ISBN: 978-3-95977-070-5. DOI: *10.4230/LIPIcs.SEA.2018.27*.

[BWZZ20] Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zuendorf. "Customizable Contraction Hierarchies with Turn Costs". In: *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*. Edited by Dennis Huisman and Christos D. Zaroliagis. Vol. 85. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 9:1–9:15. ISBN: 978-3-95977-170-2. DOI: *10.4230/OASIcs.ATMOS.2020.9*.

[CAM09] CAMVIT. "Choice Routing". In: *(http://www.camvit.com)* (2009).

[DGJ09] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. *The shortest path problem: Ninth DIMACS implementation challenge*. Vol. 74. American Mathematical Soc., 2009.

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies". In: *Experimental Algorithms* (2016), 1.5:1–1.5:49. DOI: *10.1145/2886843*.

[EG08] David Eppstein and Michael T. Goodrich. "Studying (non-planar) road networks through an algorithmic lens". In: *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. Irvine, California: Association for Computing Machinery, 2008. ISBN: 9781605583235. DOI: *10.1145/1463434.1463455*.

[HMPV00] Michel Habib, Ross McConnell, Christophe Paul, and Laurent Viennot. "Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing". In: *Theoretical Computer Science* Volume 234 (2000), pp. 59–84. ISSN: 0304-3975. DOI: *https://doi.org/10.1016/S0304-3975(97)00241-7*.

[LRT79] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. "Generalized nested dissection". In: *SIAM journal on numerical analysis* Volume 16 (1979), pp. 346–358.

[LS15] Dennis Luxen and Dennis Schieferdecker. "Candidate Sets for Alternative Routes in Road Networks". In: *ACM J. Exp. Algorithmics* Volume 19 (2015). ISSN: 1084-6654. DOI: *10.1145/2674395*.