Separator-Based Alternative Paths in Customizable Contraction Hierarchies

Scott Bacherle
Karlsruhe Institute of Technology, Germany

Thomas Bläsius
Karlsruhe Institute of Technology, Germany

Michael Zündorf
Karlsruhe Institute of Technology, Germany

Abstract

We propose an algorithm for computing alternatives to the shortest path in a road network, based on the speed-up technique CCH (customizable contraction hierarchy). Computing alternative paths is a well-studied problem, motivated by the fact that route-planning applications benefit from presenting different high-quality options the user can choose from. Another crucial feature of modern routing applications is the inclusion of live traffic, which requires speed-up techniques that allow efficient metric updates. Besides CCH, the other speed-up technique supporting metric updates is CRP (customizable route planning). Of the two, CCH is the more modern solution with the advantages of providing faster queries and being substantially simpler to implement efficiently. However, so far, CCH has been lacking a way of computing alternative paths. While for CRP, the commonly used plateau method for computing alternatives can be applied, this is not so straightforward for CCH.

With this paper, we make CCH a viable option for alternative paths, by proposing a new separator-based approach to computing alternative paths that works hand-in-hand with the CCH data structure. With our experiments, we demonstrate that CCH can indeed be used to compute alternative paths efficiently. With this, we provide an alternative to CRP that is simpler and has lower query times.

2012 ACM Subject Classification Theory of computation → Shortest paths; Mathematics of computing → Graph algorithms; Applied computing → Transportation

Keywords and phrases Alternative routes, realistic road networks, customizable contraction hierarchies, route planning, shortest paths

Digital Object Identifier 10.4230/OASIcs.ATMOS.2025.12

Supplementary Material *Software (Source Code)*: `https://github.com/mzuenni/Separator-Based-Alternative-Paths-in-CCHs`

Funding This work was supported by funding from the pilot program Core-Informatics of the Helmholtz Association (HGF).

1 Introduction

Computing shortest paths in a graph is a fundamental problem in computer science, with practical relevance in a wide range of applications. One of the most prominent examples is interactive navigation systems. For this application, the default solution for efficiently computing shortest paths – Dijkstra's algorithm [10] – is impractical due to the sheer size of real-world road networks. One can, however, make use of the fact that road networks do not change too frequently, which enables speed-up techniques that accelerate shortest-path queries via precomputation [3]. Beyond requiring fast shortest-path computations, modern navigation systems pose additional challenges. This includes real-time traffic updates and suggestions for alternative paths the user can choose from.

To formalize the concept of good alternatives to a shortest path Abraham, Delling, Goldberg, and Werneck [1] proposed a definition based on three properties.

1. The set of alternatives should be diverse, i.e., each proposed alternative should be sufficiently unique compared to the shortest path and other alternatives in the set.
2. The alternatives should have bounded stretch, i.e., they should never make big detours.
3. The alternatives should not contain obvious detours, i.e., any subpath that is sufficiently short should be a shortest path.

Abraham et al. [1] also proposed the concept of *via paths* as a technique to compute alternative paths. The idea here is to obtain a candidate path from the start s to the destination t by concatenating shortest paths from s to a via vertex v and from v to t. For each such candidate alternative, one then has to check whether it actually satisfies the conditions for being a good alternative.

A common method to produce promising candidates for via vertices uses the concept of *plateaus*, which are subpaths shared by the shortest path trees of a forward search from the start and a backward search from the destination. The resulting via vertex candidates are promising in the sense that a large plateau guarantees that an alternative via the plateau has no obvious detour (Property 3). While there are other approaches to alternative paths like the penalty method [7] and compact representations of a large set of multiple alternatives [2, 14, 17], most previous approaches are based on plateaus [1, 2, 8, 15, 16].

In its most basic form, the plateau approach works as follows [1]. Run Dijkstra's algorithm bidirectionally, i.e., forward from the start and backward from the destination. To just compute the shortest path, one could stop the searches once the shortest path has been found. To compute alternatives, let instead both searches run a bit longer, building larger and overlapping search trees, from which the plateaus can be extracted. With this, one obtains at least one admissible alternative for 95 % of shortest path queries [1]. As mentioned above, Dijkstra's algorithm is prohibitively slow on large road networks, and running it longer than necessary does certainly not improve this. It is thus not surprising that most work on alternative paths is concerned with using speed-up techniques to improve efficiency. Note that this leads to diametrically opposite objectives: On the one hand, speed-up techniques aim to prune as much of the search space as possible, ideally exploring only the vertices on the shortest path during a query. On the other hand, to get candidates for via vertices, we explicitly want to find vertices that are not on the shortest path.

Nonetheless, various speed-up techniques have been successfully used for computing alternative paths. Abraham et al. [1] demonstrate how to compute high-quality alternatives based on Reach [12] and Contraction Hierarchies [11]. Luxen and Schieferdecker [16] further improved the query times reported by Abraham et al. at the cost of excessive precomputation and a significantly reduced number of found alternatives. Bader, Dees, Geisberger, and Sanders [2] introduced the concept of an alternative graph to efficiently encode the union of many alternatives. Paraskevopoulos and Zaroliagis further engineered this approach to achieve faster runtime [17] and Kobitzsch used the alternative graph to efficiently extract new alternatives [14].

In their paper introducing the speed-up technique *CRP*, Delling, Goldberg, Pajor, and Werneck [8] also evaluate how well their algorithm is suited to compute alternatives. Similar to the basic bidirectional approach mentioned above, they observe that relaxing the stopping-criterion of the search leads to the discovery of plateaus, which result in at least one admissible alternative for 91 % of the queries. This comes at the cost of a slow-down of only slightly above 3 compared to a normal CRP query. We note that the result by Delling et al. [8] stands out in the sense that CRP (which is the abbreviation for *customizable route planning*) is the only approach that allows for efficient metric updates, which is essential as it enables features like real-time traffic updates.

While the technique *customizable contraction hierarchy (CCH)*, introduced by Dibbelt, Strasser, and Wagner [9], is an alternative to CRP that also allows metric updates, CCHs have not yet been studied in terms of alternative paths. This is unfortunate, as except for the lacking feature of alternative paths, CCH is generally preferable compared to CRP. It is easier to implement, achieves an order of magnitude faster queries with the same preprocessing time [9], and has a stronger theoretical foundation [4, 9, 19]. With this paper, we study how CCHs also can be used to compute alternative paths.

Challenges, Contribution, and Outline. As outlined above, a common approach to compute alternative paths is to run the normal shortest-path query longer than necessary to obtain plateaus. Our first observation is that this approach is not compatible with CCH: The CCH-query (at least in its basic form) runs upwards in a hierarchy defined by a so-called *elimination tree* until it reaches the root, without any preemptive stopping criterion; we refer the reader to Section 2.3 for a brief introduction to CCH. It follows that the strategy "run the query longer" is not well-defined for CCH, as there is nothing left to explore after reaching the root. Thus, instead of using plateaus, we propose a different approach to obtain good candidates for via vertices that is more aligned with the inner workings of CCH.

Our approach to find candidates for via vertices is based on separators. Assume that we are interested in alternative s-t-paths and that S is a separator that separates the start s from the destination t, i.e., removing the vertices in S from the graph places s and t in different connected components. Then any s-t-path must use one of the vertices from S, making S a set of promising candidates for via vertices. Interestingly, the above mentioned elimination tree of the CCH defines a hierarchy of separators. Moreover, the CCH query visits all vertices in the top-level separator that separates s from t. Thus, without any overhead compared to the normal CCH query, this provides us with a set of candidate via vertices. Additionally, the query already provides us with distances from and to the separator vertices, which already provides an upper bound to the stretch (Property 2). With little additional overhead (factor 2), we can check all three properties properly.

Interestingly, the previously mentioned opposition – alternative paths requiring via vertices that are not on the shortest path vs. speed-up technique trying to reduce the search space – becomes apparent for CCHs. We observe that only considering the top-level separator for potential via vertices yields an admissible alternative in only 65 % of the queries. To improve upon this, we propose an algorithm that not only considers the top-level separator but also separators on lower levels. Going down just one level already yields an admissible alternative in 84 % and we achieve 90 % by going even deeper, which is competitive to the state-of-the-art. Compared to a normal CCH query, this comes at the cost of a running time increase by factors of 4.5 and 9.4, respectively. This outperforms CRP, the only previous speed-up technique supporting alternative paths and efficient metric updates. Moreover, for finding two and three alternatives, we get success rates of 69 % and 45 %, which is a moderate improvement compared to CRP. Beyond the comparison to the state-of-the-art, we additionally provide experiments that foster the understanding of our algorithm, e.g., in terms of the trade-offs between running time and success rate.

After the preliminaries in Section 2, we introduce our separator-based approach for computing alternative paths with CCH in Section 3. Our experimental evaluation can be found in Section 4. We conclude with a discussion and some final remarks in Section 5.

2 Preliminaries

Let $G = (V, E)$ be a *directed graph* with *vertices* V and *edges* $E \subseteq V \times V$. Moreover, let G be weighted, i.e., we have a *cost function* $c : E \to \mathbb{R}_{\geq 0}$.[1] For a directed edge $e = (u, v) \in E$ we refer to u and v as the *tail* and *head*, respectively. A sequence $P = \langle e_0, \ldots, e_k \rangle$ of edges $e_i \in E$ is a *path* if the head of e_{i-1} equals the tail of e_i for $i \in [k]$. Let s be the tail of e_0 and t be the head of e_k. Then we call P an *s-t-path*. Slightly abusing notation, we also use P to refer to the set of edges in P. For $P' \subseteq P$, we call P' a *subpath* of P if P' appears consecutively in P. Moreover, if P' is an a-b-path for $a, b \in V$, then P is called *a-b-subpath* of P.

We extend the cost function c from individual edges to sets of edges (and in particular to paths), i.e., for $E' \subseteq E$ we define $c(E') = \sum_{e \in E'} c(e)$. For a path P, $c(P)$ is called the *length* of P. The *distance* between s and t in G is the minimum length of an s-t-path and is denoted by $d(s, t)$. In the remainder of this paper, we make the common assumption that there is only one s-t-path of length $d(s, t)$ and therefore call this path *the* shortest s-t-path and denote it with $P_{s,t}$.

2.1 Alternative Path Problem

Let $P_{s,t}$ be the shortest s-t-path for $s, t \in V$. Following the definition by Abraham et al. [1], we say that an s-t-path P is an *admissible alternative* if it has limited sharing, bounded stretch, and local optimality, which are defined as follows, depending on parameters γ, ε, and α, respectively.

1. P has *limited sharing* if $c(P \cap P_{s,t}) \leq \gamma \cdot d(s, t)$.
2. P has *bounded stretch* if for every a-b-subpath $P' \subseteq P$ it holds that $c(P') \leq (1+\varepsilon) \cdot d(a, b)$.
3. P is *locally optimal* if for every a-b-subpath $P' \subseteq P$ with $c(P') \leq \alpha \cdot d(s, t)$, it holds that P' is the shortest a-b-path, i.e., $c(P') = d(a, b)$.

In case we do not want just one but multiple alternatives, we require limited sharing (Property 1) to not only hold for the shortest path $P_{s,t}$ but for the union of $P_{s,t}$ and all other alternatives. Thus, we call a set $\mathcal{P}$ of alternative s-t-paths *admissible* if each $P \in \mathcal{P}$ has bounded stretch (Property 2), is locally optimal (Property 3), and

$$c\Big(P \cap \Big(\bigcup_{\substack{P' \in \mathcal{P} \\ P' \neq P}} P' \cup P_{s,t}\Big)\Big) \leq \gamma \cdot d(s, t).$$

We use the parameter values $\gamma = 0.8$, $\varepsilon = 0.25$, and $\alpha = 0.25$ typically used in the literature.

2.2 Checking Admissibility

As already noted in the literature [1], it is unknown how to check bounded stretch (Property 2) and local optimality (Property 3) without requiring quadratic time in the length of the path. It is thus common to not check the properties exactly but instead use the approaches described in the following.

For bounded stretch (Property 2), instead of checking every subpath $P' \subseteq P$, we only check the parts that deviate from the shortest path $P_{s,t}$. More precisely, we check the a-b-subpath P' if P' is maximal (with respect to inclusion) among the subpaths of P that are edge-disjoint with the shortest path $P_{s,t}$. For paths based on one via vertex v, i.e., if P

[1] For the purpose of the routing application, think of c as the cost to traverse an edge. Most commonly c is the travel time.

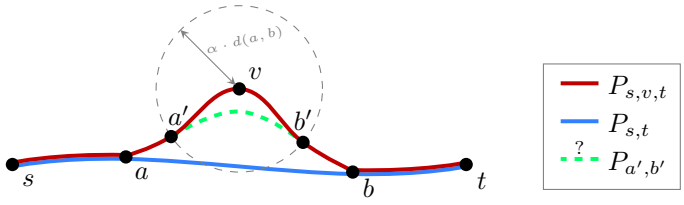

Figure 1 Visualization of via alternative $P_{s,v,t}$ and the vertices along it that are used for admissibility checks. The vertices a and b are the vertices where the shortest path $P_{s,t}$ diverges and meets the alternative. The vertices a' and b' are those vertices along the alternative that are used for the T-test. The dotted line between a' and b' symbolizes the possible existence of a shorter path between a' and b' not via v which in turn would imply that the T-test is not passed.

is the concatenation of the shortest paths $P_{s,v}$ and $P_{v,t}$, one can see that there is just one such a-b-subpath; see Figure 1. More generally, for multiple via-vertices there is at most one such subpath for each via vertex.

For the local optimality (Property 3), we use the so-called T-test, which guarantees local optimality with respect to α but sometimes falsely rejects alternatives if they are not locally optimal with respect to 2α [1]. For this, consider a via vertex v and let $P_{s,v,t}$ be the concatenation of the shortest paths $P_{s,v}$ and $P_{v,t}$. Moreover, let a' and b' be the vertices on $P_{s,v,t}$ at distance $\alpha \cdot d(a,b)$ from v; see Figure 1. The alternative $P_{s,v,t}$ passes the T-test if $P_{a',v,b'}$ is the shortest a'-b'-path.

2.3 Customizable Contraction Hierarchies

Here we introduce the concepts of CCH necessary to understand our alternative path algorithm. For a more general introduction and in-depth discussion, see the recent survey [5]. In general, CCH follows a three-phase approach: The first phase is a metric-independent preprocessing, i.e., a preprocessing of the graph G, ignoring the cost function c. This is the most costly phase, taking in the order of a few minutes for a continentally sized network, which is acceptable as the topology of G rarely changes. The second step is the customization phase, where the cost function c is preprocessed. Taking just a few seconds, this allows for frequent traffic updates. Finally, the third phase are the actual queries, which usually take less than a millisecond, exploiting the additionally information computed in the first two phases.

Given the graph G, the CCH preprocessing computes the following two additional structures. First, it computes a hierarchy of small balanced separators that splits the graph recursively into pieces; see Figure 2 (left) for an illustration. This hierarchy is represented by a rooted tree T called *elimination tree*; see Figure 2 (right). Let $S \subseteq V$ be one of the separators. Then S forms a path in T and only the bottom-most vertex of S has multiple children. The different subtrees below these children contain the vertices from the different connected components that were separated by S. From this, one can already make the following crucial observation: For two vertices s and t, let $A \subseteq V$ be the set of common ancestors of s and t in T. Then, A separates s from t (or contains one of the two) and thus any s-t-path goes through a vertex of A. Hence, to compute the distance $d(s,t)$, it suffices to compute the distances $d(s,v)$ and $d(v,t)$ for all $v \in A$ and then choose the vertex v that minimizes $d(s,v) + d(v,t)$.

To achieve this, we need the second additional structure computed in the precomputation; the *augmented graph* G^+. The augmented graph is obtained from G by inserting additional edges, which are called *shortcuts* and represent paths. Explaining why this works is beyond

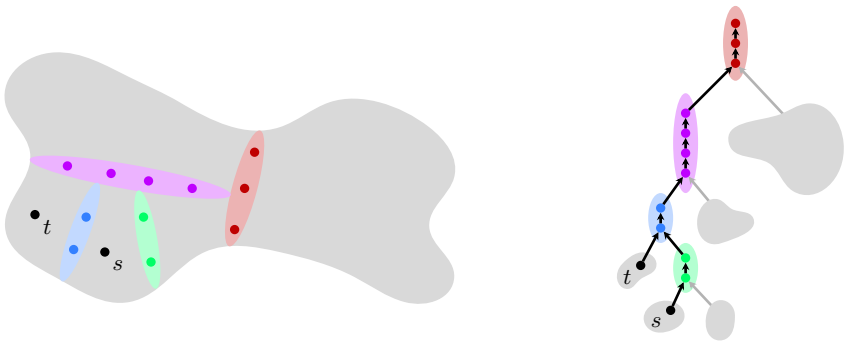

Figure 2 Sketch of separators inside a CCH (left) and the corresponding search spaces (right). Even though the union of the blue and lilac separators already separates s from t we still consider the red separator because it is also in the common search space of s and t.

the scope of this paper (see e.g., [5]), but adding shortcuts in a clever way yields the following crucial property. Let $s \in V$ be any vertex and let $\langle v_1, \dots, v_k \rangle$ be the path in the elimination tree T from $s = v_1$ to the root v_k. Now run Dijkstra's algorithm, but instead of using a priority queue to decide which vertices to relax, only relax the vertices $v_1, \dots, v_k$ in that order. Then this computes the correct distances $d(s, v_i)$ for all $i \in [k]$, i.e., for all ancestors of s in the elimination tree T. Running this type of search from s and (backwards) from t thus yields the distances $d(s, v)$ and $d(v, t)$ for every common ancestor of s and t, which yields the distance $d(s, t)$ as noted above.

3 Separator Based Alternatives with CCH

Let $G = (V, E)$ be the graph for which we have built the CCH and let T be the corresponding elimination tree. Moreover, let the start $s \in V$ and the destination $t \in V$ be given, and let A be the set of common ancestors of s and t in T. As outlined in Section 2.3, A separates s from t. The idea of our basic approach is to use A as the set of potential via vertices.

We believe that this is a rather natural set of candidates for via vertices: As A separates s from t, every s-t-path, and thus every alternative, has to go through a vertex in A. Moreover, the separator hierarchy used for the CCH attempts to use small and balanced separators. Small separators mean that we do not have too many candidates to check, making the runtime acceptable. Balanced separation means that the separators lie somewhat in the middle between many s-t-pairs, which intuitively makes for good via vertex candidates; see Figure 3 for examples. With the set of candidate via vertices A, it remains to check for admissibility. More precisely, we need to select a subset of A such that the corresponding alternatives form an admissible set of alternatives.

3.1 Selection of Via Vertices

To maximize the number of found alternatives, we would ideally check for each separator vertex $v \in S$ whether the via vertex v yields an admissible alternative. From the resulting set of individually admissible alternatives, one would then have to extract a maximum set of alternatives that are also admissible together (recall that the limited sharing property for sets of alternatives also depends on the other selected alternatives). As this is computationally too expensive, we instead use the following greedy strategy, which is also commonly used in the literature regarding alternative paths.

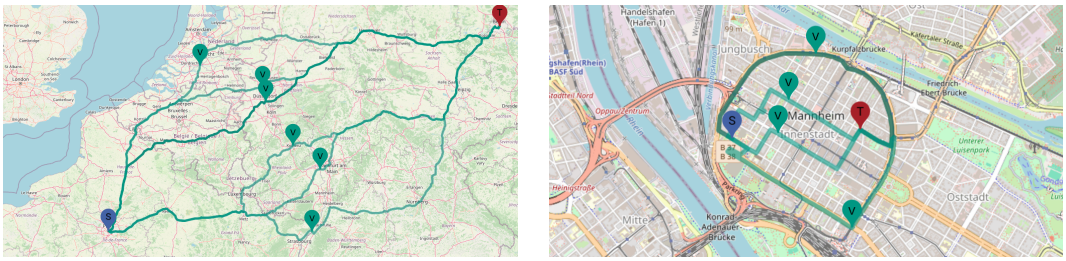

Figure 3 Alternative paths computed by the basic approach. Left: Alternative paths between Berlin and Paris. All paths have to cross the Rhine with some bridges. These bridges also form a small balanced separator between Berlin and Paris. Right: Alternative paths between two blocks in the german city of Mannheim. The corresponding separator vertices are highlighted.

We iteratively process the alternatives defined by vertices in A. When processing $v \in S$, we check whether the via path $P_{s,v,t}$ is admissible with respect to the already selected alternatives. If yes, we add it to the selection and discard it otherwise. We prioritize shorter alternatives, i.e., we process the candidates in the order of increasing length. We note that the CCH approach is very well suited for this. Recall that the CCH query computes $d(s,v)$ and $d(v,t)$ for every $v \in S$. Thus we already know the length of every via path $P_{s,v,t}$ without any overhead compared to the normal query.

It remains to check whether the next candidate path $P_{s,v,t}$ is admissible with respect to the already selected alternatives. To do this efficiently, we use four different checks that we apply in order of increasing computational cost. The reason for this is that, if an early cheap check already rules out a path, then we can save the later more expensive checks. The steps and how to implement them efficiently in the CCH approach are described in detail in the following. The first two steps check the bounded stretch requirement. The third step ensures limited sharing and the fourth and final step is the T-test checking for local optimality.

Total Stretch Pruning. If an alternative is already longer than $(1+\varepsilon) \cdot d(s,t)$ it will clearly violate the bounded stretch (Property 2). Since we process alternatives by increasing length this implies that all following alternatives will be too long as well, which allows us to return early. This property can also be used to speed up the standard CCH in the same fashion as proposed by Buchhold et al. [6]. They proposed that any vertex whose distance to either s or t already exceeds the length of the shortest path found so far can be skipped. In our setting, we can do the same pruning by additionally including the factor $1+\varepsilon$.

Bounded Stretch. For the bounded stretch, we need to find the a-b-subpath deviating from the shortest path as shown in Figure 1. In the following, we describe how to find a; determining b works analogously.

Let A be the common ancestors of s and t in the elimination tree and let $v \in A$ be the via vertex we are currently interested in. From the CCH query, we already obtain the shortest paths $P^+_{s,u}$ in the augmented graph G^+. Note that this path might still contain shortcut edges. The naive approach would be to unpack it and then check where the resulting path deviates from the shortest path. We can improve this by only unpacking the first edge on $P^+_{s,v}$ that deviates from the shortest path. We note that unpacking this edge once might again yield shortcut edges, which have to be unpacked recursively. However, in each step, we only have to unpack one edge. Once this deviating edge is an edge of G, i.e., not a shortcut, we have found the vertex a which is the tail of the edge. Moreover, during this unpacking,

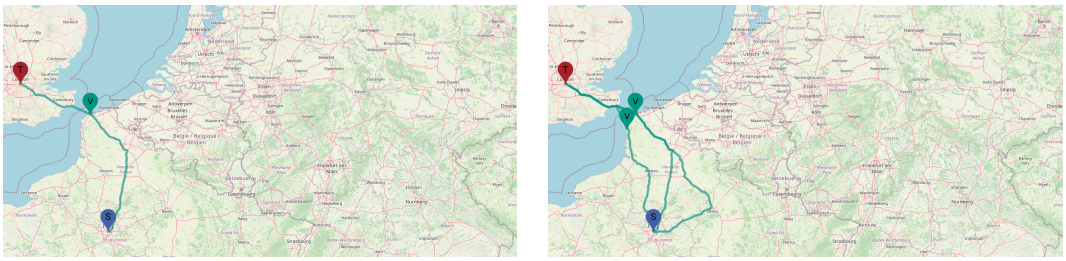

Figure 4 Alternatives between London and Paris. Left: the shortest path, which is also the only alternative that can be found by the basic approach because of the trivial separator. Right: the alternatives found by the two step approach. In this case all alternatives use the same path after using the Eurotunnel.

we can maintain the distance to v. Thus, we not only find a but also know $d(s, a)$ and $d(a, v)$. With this checking the bounded stretch (Property 2) boils down to checking whether $d(a, v) + d(v, b) \leq (1 + \varepsilon) \cdot (d(s, t) - d(s, a) - d(b, t))$.

Limited Sharing. To check limited sharing for the shortest path $P_{s,t}$, observe that from the previous checks, we already know

$$c(P_{s,v,t} \cap P_{s,t}) = c(P_{s,v,t}) - c(P_{a,v,b}) \ .$$

Thus, Property 1 can be checked with almost no overhead.

If we have already selected other alternatives, we also have to check the sharing with them. For this, we actually compute $P_{s,v,t}$ in G by fully unpacking the path found by the CCH, i.e., we transform a path in the augmented graph G^+ to a path in G by replacing shortcuts with paths. To efficiently check sharing, we maintain the invariant that all edges along the shortest path and along all previous selected alternatives are marked. Computing the sharing is then done by summing over all marked edges along the unpacked path $P_{s,v,t}$. Note that we can slightly improve the efficiency here by only summing over the detour and adding the distance $d(s, a)$ and $d(b, t)$ directly.

Local Optimality. Performing the T-test is conceptually easy even though it is computationally expensive. We first determine the vertices a' and b' along $P_{s,v,t}$ closest to v with distance at least $\alpha \cdot d(a, b)$. Then we perform a separate CCH query to compute $d(a', b')$ and check if it is equal to $c(P_{a',b'})$. Note that determining a', b' and $c(P_{a',b'})$ is trivial because we already unpacked the path in the previous step.

After all tests are passed we add the alternative to our selection and mark all edges to maintain the invariant required by the limited sharing check.

3.2 Two-Step Approach

The biggest advantage of road networks for efficiently computing shortest paths – its small natural separators – can become a problem for our basic algorithm described above if the separators are too small. Figure 4 shows the example of London and Paris, which are separated by only a few vertices. The shortest path uses the Eurotunnel and the only possible way to avoid this is by using ferries which take too long to provide admissible alternatives. A similar issue can arise if the separator is too close to either s or t. In such a case most separator vertices already violate the global stretch and get pruned.

The goal in the following is to extend our approach to also provide admissible alternatives in these situations. For this, we use the following intuitive observation. The above scenario happens if the separator vertex v used by the shortest path is too important to be avoided. Thus, we decide to keep v and instead look for alternatives in the subpaths from s to v and from v to t. The basic idea is to consider these two sides of the separator v as independent subproblems of the alternative path problem.

To make this more precise, let v be the vertex on the shortest path $P_{s,t}$ that is closest to the root in the elimination tree T. Moreover, let v_s and v_t be the two neighbors of v in $P_{s,t}$. Then we recursively call the basic algorithm to compute alternatives for s to v_s and v_t to t; see Figure 5. It remains to fill out the following details. First, we have to describe how to combine the subpaths that are returned by the recursive calls. Secondly, we want the resulting alternatives to be admissible for the original query of s and t. While admissibility of the resulting paths can be checked when combining subpaths, we also have to adjust the requirements for admissibility in the recursive calls to not falsely prune promising subpaths or return subpaths that can never be completed to admissible alternatives.

Path Combination. Assume we have recursively computed alternative paths from s to v_s and from v_t to t. As this typically does not yield too many paths, we consider every possible combination of these paths to obtain alternative s-t-paths. As in the basic approach, the order in which we consider these paths might have an impact on the resulting set of alternatives. As before, we greedily add alternatives ordered by their length. We note that we can save some time here by sorting the paths by their length before unpacking them and stopping as soon as the constructed paths become larger than $(1+\varepsilon)\cdot d(s,t)$.

For each combination we consider, we have to check, whether it is admissible. For the bounded stretch, running the recursive call with the same ε already provides the check as described in Section 2.2, so there is nothing to do there.Checking the sharing is straight forward, by unpacking the path. Finally, for the local optimality, adjusting the α-value appropriately in the recursive call (see one of the following paragraphs) makes sure that there are no local detours within the subpaths.Here we additionally run the T-test at the vertex v to also ensure overall local optimality.

Limited Sharing in the Recursive Call. Although limited sharing is ultimately tested when we combine the subpaths of the recursive calls, we can still exclude some alternative subpaths based on their sharing with the shortest path. For this, we only consider sharing with the shortest path (and not with other alternatives) in the recursive calls. Here we describe how we adjust γ in the recursive calls.

Let s and v_s be the start and destination of the recursive call. Our goal is to choose a γ' such that the following property holds. If an alternative s-v_s-path P shares at least $\gamma'\cdot d(s,v_s)$ with the shortest path P_{s,v_s}, then it should be safe to exclude P as every combination using P would have too much sharing with $P_{s,t}$. We claim that this is achieved by setting

$$\gamma' = \frac{\gamma\cdot d(s,t) - d(v_s,v_t)}{d(s,v_s)}\ .$$

To see this, assume that the path P is excluded due to the fact that $c(P\cap P_{s,v_s}) \geq \gamma'\cdot d(s,v_s)$. Plugging in the above definition for γ' and slightly rearranging yields $c(P\cap P_{s,v_s})+d(v_s,v_t)\geq \gamma\cdot d(s,t)$. Note that any combination involving P clearly shares $c(P\cap P_{s,v_s})$. Moreover, the subpath from v_s to v_t (consisting of just two edges) is also shared in every combination. Thus, it is fair to exclude P as any combination with P will fail the bounded shared check anyways.

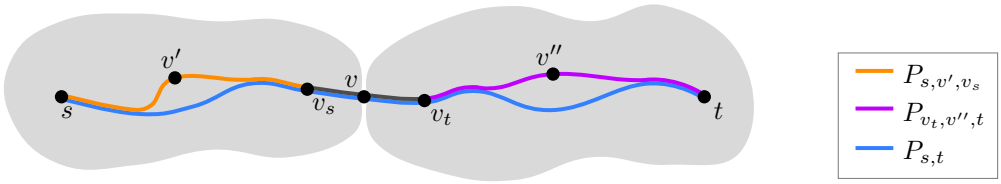

Figure 5 Visualization of the two-step approach. We consider v as a separator and therefore solve the alternative route problem from s to v and from v to t separately. More precisely, we consider the neighbors of v for this to be able to use the basic algorithm again. In the end, the alternatives of the subproblems can be used to create an alternative from s to t.

Local Optimality in the Recursive Call. Recall that the requirement for local optimality is relative to the distance $d(s,t)$, i.e., optimality is required for all subpaths between vertices at distance $\alpha \cdot d(s,t)$. To maintain the same guarantee in the recursive call, we run it with

$$\alpha' = \alpha \frac{d(s,t)}{d(s,v_s)} .$$

Note that being locally optimal for α' relative to $d(s,v_s)$ in the recursive call is equivalent to being locally optimal for α relative to $d(s,v_s)$. We note that α' can become larger than 1. In this case we can stop the recursive call and just report the shortest path.

3.3 Recursive Approach

A natural extension to the two step approach is to not compute the paths from v to v_s and v_t to t with the basic approach but with the two step approach itself. More precisely, we first use the base algorithm, if that does not yield sufficiently many alternatives we go one recursion step deeper and afterwards combine the paths in the same way as in Section 3.2 This approach ensures that even multiple small separators between s and t can be handled while only doing more work if necessary. As a stopping condition for the recursive descend, we propose to stop as soon as the distance between the current start s' and destination t' becomes too small. More precisely we introduce the new parameter μ. The recursive call returns just the shortest path if $d(s',t')$ is less than $\mu \cdot d(s,t)$.

4 Experimental Evaluation

The main objective of this section is to evaluate the performance of our algorithmic approaches in regards to quality – measured by the number of found alternatives – and runtime. For this, we start with an algorithm comparison in Section 4.2. We focus on the comparison with CRP, which is the main competitor due to the fact that no other speed-up technique also supports efficient metric updates. Additionally, we will also see that our different variants provide a trade-off between success rate and runtime.

Afterwards we provide a more detailed look into the inner workings of our algorithm. In Section 4.3, we provide statistics on how many candidates for alternative paths are considered and due to which checks they are filtered out. There we will also see that almost no candidates are rejected due to too much sharing with previously selected paths, which serves as a justification to greedily select alternatives. In Section 4.4, we consider the impact of the separator hierarchy used in the CCH on the success rate. Finally, in Section 4.5, we study the trade-off between success rate and runtime the parameter μ in our recursive approach provides. Before we start with the actual evaluation, we briefly specify our experiment setup.

Table 1 Comparison of various alternative route techniques. We report the success rate and the run time for all approaches for the first up to $k = 3$ alternatives. The runtime is the accumulated time needed to find the first k alternatives. Note that the reported times have to be taken with a grain of salt since the tests were executed on different machines. X-BDV, X-REV and X-CHV are due to Abraham et al. [1]. X-CHASEV and its variants are due to Luxen et al. [16]. HiDAR is due to Kobitzsch [14]. CRP is due to Delling et al. [8]. The bottom three rows represent the different variants of our algorithm (the recursive variant uses $\mu = 0.3$). Only the algorithms in the second part (CRP and our algorithms) allow efficient metric updates.

	first		second		third	
	success [%]	time [ms]	success [%]	time [ms]	success [%]	time [ms]
X-BDV	94.5	26 352.0	81.1	29 795.0	61.6	33 443.0
X-REV-1	91.3	20.4	70.3	33.6	43.0	42.6
X-REV-2	94.2	24.3	79.0	50.3	56.9	64.9
X-CHV-3	90.7	16.9	70.1	20.3	42.3	22.1
X-CHV-5	92.9	55.2	77.0	65.0	53.3	73.2
X-CHASEV	75.5	0.5	40.2	0.7	14.2	1.0
two-level	80.5	0.1	50.8	0.3	24.8	0.4
multi-level	81.2	0.1	51.2	0.3	25.0	0.4
HiDAR	91.5	18.2	75.5	18.2	55.9	18.2
CRP	90.9	5.8	65.4	3.3	39.2	3.4
SeArCCH	65.3	0.5	37.3	0.7	17.2	1.0
two-step	84.1	1.1	62.9	1.6	38.7	2.3
recursive	90.0	2.3	68.6	4.2	44.7	6.0

4.1 Experimental Setup

The input instance for all our experiments is the DIMACS Europe graph with travel-time as edge weights. The resulting graph has 18 million vertices, 42 million edges and represents the complete road network of west Europe around the year 2000. We chose to use this graph even though larger networks are available since it is the main benchmark for various other route planning techniques and has also been used to evaluate all other alternative path techniques.

To make the comparison with the related work easier we copy the methodology introduced by Abraham et al. [1]. That is, we test our algorithm with the same parameters. More precisely, we require alternatives to be locally optimal for $\alpha = 25\,\%$, have at most $\gamma = 80\,\%$ sharing, and bounded stretch of $1 + \varepsilon = 125\,\%$. Note that these are hard constraints and any alternative satisfying them is considered good. Thus, the quality of the algorithm can be measured by the success rate of finding alternatives. Additionally, we will consider the time required to find those alternatives. If not stated otherwise all numbers are averaged over 10^5 random queries.

Our algorithms are implemented in C++17 and compiled with the GNU compiler 13.3.1 using optimization level 3. Our test machine runs Fedora 39 (kernel 6.8.11), and has 32 GiB of DDR4-4266 RAM and a AMD Ryzen 7 PRO 5850U CPU with 16 cores clocked at 4.40 Ghz and 8×64 KiB of L1, 8×4 MiB of L2, and 16 MiB of L3 cache.

4.2 Algorithm Comparison

Unfortunately, no implementations of the competitor algorithms are freely available and the queries used during their respective evaluations were not published. To nonetheless make a comparison possible, we report the success rates and computation times from the related work in Table 1. The success rate for $k \in \{1, 2, 3\}$ represents the fraction of queries, for which an admissible set of alternatives of size at least k was found. Even though the numbers are based on different sets of sampled queries, the success rates should be comparable since they are averaged over at least 10^4 queries. The timings on the other hand should not be compared directly; see the more detailed discussion of the running time below.

We want to note that CRP and our approach are arguably the most practically relevant approaches due to their support for efficient metric updates. We thus focus our discussion on the comparison to CRP. The other approaches listed in Table 1 are included for completeness.

Success Rates. The algorithm X-BDV can be considered as the baseline in terms of success rate, since it checks all admissible alternatives that use a single via vertex. Even though this approach is infeasible for real applications it shows what success rates are achievable. Based on the related work we consider as success rate of 90 % as desirable for the first alternative. As we can see in Table 1 this is achieved by our recursive approach (with $\mu = 0.3$). Moreover, the two-step approach is with 84.1 % already close to this goal while being significantly faster. For the second and the third alternative, our approaches performs slightly worse than the baseline. However, compared to CRP, we obtain slightly higher success rates: We get an improvement from 65.4 % to 68.6 % and from 39.2 % to 44.7 % for the second and third alternative, respectively.

Runtime. For the runtime comparison with CRP, our recursive variant is the most relevant, as it achieves comparable success rates. Additionally, it is also interesting to compare the runtime with our two-step approach, which has only slightly worse success rates. We want to note that, ideally, we could run comparison experiments for CRP on the same machine. Unfortunately, the implementation is not publicly available and building a competitive CRP implementation is highly non-trivial. Nonetheless, we believe that we can draw the conclusion that our algorithm outperforms CRP.

To make the comparison, we start with the base algorithms CCH and CRP and consider the slowdown encountered by additionally computing alternative paths. For the base algorithms, it is already well established that the query of CCH is a magnitude faster than CRP's query [9]. Moreover, for CRP, Delling et al. [8] report a slowdown of at least 3 for computing alternatives. Thus, even with a slowdown of 30, we would still obtain a competitive performance. We can report a runtime of 0.245 ms for the basic CCH query. Compared to this, our recursive algorithm computing alternatives leads to slowdowns of 9.4 for the first, 17.1 for the second, and 24.5 for the third alternative. With these numbers, we can safely conclude that our approach is at least on par with CRP and faster most of the time. We also note that our two-step approach with a success rate of 84.1 % for the first alternative has only a slowdown of 4.5. With the above estimation, we expect this to be at least 6 times faster than CRP. We note that the two-step variant forms the initial part of recursive. Thus, in the 84.1 % of the queries where two-step finds an alternative, recursive also finds the first alternative in the same time.

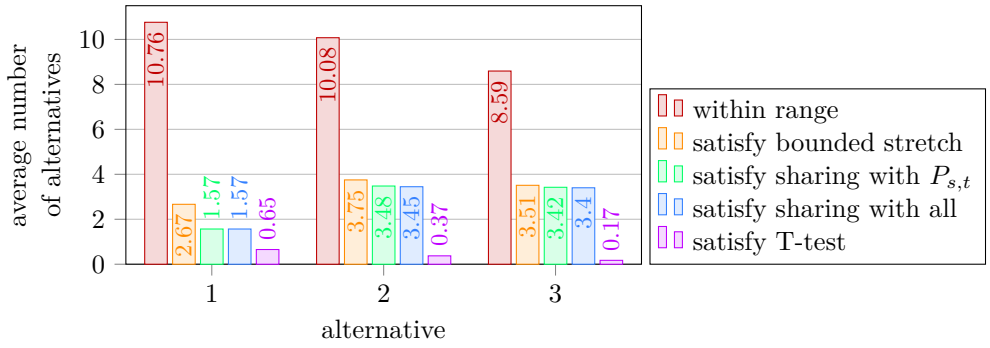

Figure 6 This plot visualizes the average number of alternatives remaining after each of the successive checks. We can observe that most alternatives are rejected due to bounded stretch or sharing with the shortest path. Almost none of the alternatives are rejected due to the sharing with previously selected alternatives.

4.3 Checking Admissibility of Candidates

Recall that our algorithm starts with a set of candidates for alternative paths, which are then filtered by different checks ensuring the resulting set of alternatives is admissible. Figure 6 shows for our basic (non-recursive) approach, how many candidates pass which check before we find the first, second, or third alternative (after finding the previous alternative). The first bar indicates the number of considered candidate paths that have overall length at most $(1+\varepsilon)\cdot d(s,t)$. The second bar shows how many of those have bounded stretch (property 2). The third bar shows the number of checked candidates additionally having limited sharing with the shortest path $P_{s,t}$ and the fourth bar is obtained by additionally requiring limited sharing with all previously selected alternatives (property 1). Finally, the fifth bar indicates the average number of paths also passing the T-test (property 3), which coincides with the fraction of queries finding at least one, two, or three alternatives.

One can clearly see that most considered candidates are rejected due to the stretch. For the second and third alternative, the T-test additionally rules out a big fraction of candidates. Note that only very few checked candidates are rejected due to too much sharing with previously selected alternatives.

We believe that the last observation is particularly interesting for the following reason. While the alternative path problem ensures high-quality alternatives by requiring admissibility (hard constraints), the optimization goal is to maximize the number of found alternatives. However, our approach follows the literature by greedily adding alternative paths to the set of alternatives, prioritizing shorter paths. A better approach to maximize the number of alternatives might be to somehow select the alternatives based on how much they share with candidates that are considered later. However, the fact that there is almost no difference between the third and fourth bars in Figure 6 shows that earlier selected alternatives very rarely make later candidates non-admissible due to a large overlap. This justifies the decision to select alternatives greedily by length instead of trying to explicitly maximize their number.

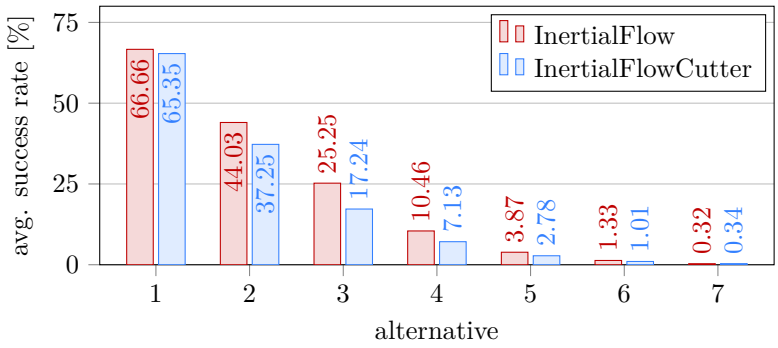

Figure 7 Impact of the contraction order on the success rate of finding k alternatives with the basic algorithm. We compare two contraction orders, one derived from InertialFlow, and the other from InertialFlowCutter. We can see that the order has negligible impact on the success rate for the first alternative but a slight impact on the subsequent alternatives.

4.4 Impact of the Separator Hierarchy

An important degree of freedom in CCH is the used separator hierarchy. Except when explicitly stated otherwise, we use the state-of-the-art algorithm InertialFlowCutter[2] engineered by Gottesbüren, Hamann, Uhl and Wagner in the default configuration [13]. This can be seen as the the default choice for high-performance CCHs. However, as already mentioned in Section 1, larger separators (which are worse for performance) provide more candidates for via vertices. For comparison, we thus additionally ran our alternative path algorithm using separators provided by the simpler InertialFlow algorithm proposed by Schild and Sommer [18], using the implementation in RoutingKIT[3] in the default configuration.

Figure 7 shows the number of alternatives found by the basic (non-recursive) variant of our algorithm for the different separator hierarchies. We can see that, indeed, the number of found alternatives slightly increases with the larger separators. Thus, in principle, choosing different separators provides a trade-off between quality and runtime. However, since the effect is quite small, we suggest using the InertialFlowCutter separators together with our recursive algorithm, which also provides a trade-off between quality and runtime, as discussed in the next section.

4.5 Impact of the Recursion Parameter μ

Figure 8 shows the trade-off between success rate and runtime of our recursive algorithm depending on the parameter μ. In the right plot, we can clearly see that less recursive calls (i.e., larger μ) yields lower runtimes. The plot on the left shows that the success rate already starts on a high level for large μ, which agrees with our previous observation that the two-step variant[4] already yields good results. From this high level, the success rate further increases for smaller μ. A success rate of at least 90% for the first alternative is obtained for $\mu \geq 0.3$ and with $\mu = 0.3$, we obtain a runtime of just below 6 ms. As 90% is comparable with the state-of-the-art, we suggest and use $\mu = 0.3$ as the default value.

[2] `https://github.com/kit-algo/InertialFlowCutter`
[3] `https://github.com/RoutingKit/RoutingKit`
[4] We note that the two-step variant is incomparable with the recursive variant. The two-step variant always makes two recursive calls, one for each subpath on the top level. The recursive variant might, e.g., make multiple nested recursive calls but always only for one of the two subpaths.

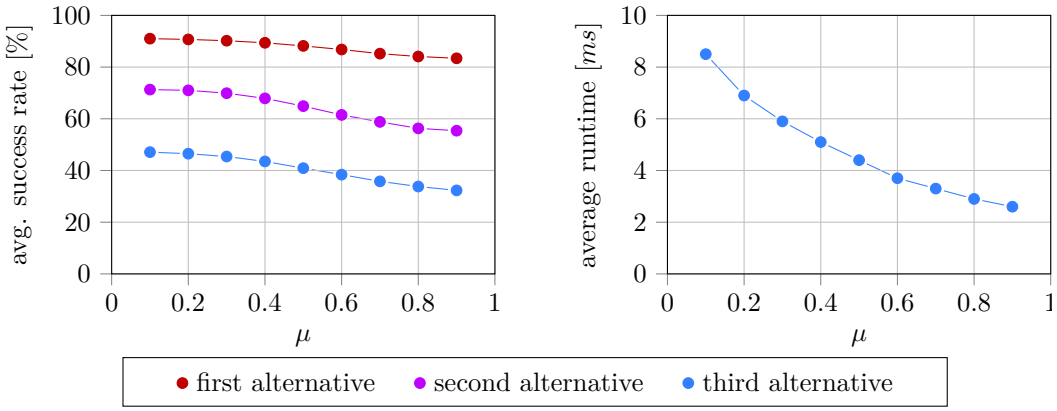

Figure 8 Impact on the stop parameter μ for the recursive algorithm on the success rate (left) and runtime (right).

5 Conclusion and Future Work

We have demonstrated that the speed-up technique CCH is well suited to compute alternative paths, resulting in an algorithm that is competitive with the state-of-the-art in performance and quality. With our implementation, we provide the first publicly available data structure that combines highly efficient queries for shortest paths and alternative paths with fast metric updates, which hopefully enables future research on the topic. As future directions, we believe it would be interesting to further explore how multiple via vertices can lead to more alternatives, in particular in cases where currently no single-via alternative is available. It would also be interesting to see if this can be used to beat the commonly used baseline algorithm, which is restricted to single-via alternatives. Concerning the baseline, we believe that it would also be interesting to develop exact algorithms for testing whether there exists an admissible alternative, even if this would prove computationally too expensive for actual applications.

References

1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative routes in road networks. *Journal of Experimental Algorithmics (JEA)*, 2013. doi:10.1145/2444016.2444019.

2 Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative route graphs in road networks. In *International Conference on Theory and Practice of Algorithms in (Computer) Systems*, 2011. doi:10.1007/978-3-642-19754-3_5.

3 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm Engineering: Selected Results and Surveys*, Lecture Notes in Computer Science. Springer, 2016. doi:10.1007/978-3-319-49487-6_2.

4 Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. *Theoretical Computer Science*, 2016. doi:10.1016/j.tcs.2016.07.003.

5 Thomas Bläsius, Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. Customizable contraction hierarchies – a survey, 2025. doi:10.48550/arXiv.2502.10519.

6 Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Real-time traffic assignment using engineered customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 2019. doi:10.1145/3362693.

7 Yanyan Chen, Michael GH Bell, and Klaus Bogenberger. Reliable pretrip multipath planning and dynamic adaptation for a centralized road navigation system. *IEEE Transactions on Intelligent Transportation Systems*, 2007.

8 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 2017. doi:10.1287/trsc.2014.0579.

9 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 2016. doi:10.1145/2886843.

10 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959. doi:10.1145/3544585.3544600.

11 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 2012. doi:10.1287/trsc.1110.0401.

12 Andrew V Goldberg, Haim Kaplan, and Renato F Werneck. Reach for A*: Shortest path algorithms with preprocessing. In *The shortest path problem*, 2006.

13 Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, 2019. doi:10.3390/a12090196.

14 Moritz Kobitzsch. An alternative approach to alternative routes: Hidar. In *European Symposium on Algorithms*, 2013. doi:10.1007/978-3-642-40450-4_52.

15 Cambridge Vehicle Information Technology Ltd. Choice routing, 2009. URL: http://www.camvit.com.

16 Dennis Luxen and Dennis Schieferdecker. Candidate sets for alternative routes in road networks. In *International Symposium on Experimental Algorithms*, Lecture Notes in Computer Science, 2012. doi:10.1007/978-3-642-30850-5_23.

17 Andreas Paraskevopoulos and Christos Zaroliagis. Improved alternative route planning. In *ATMOS-13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems-2013*, 2013. doi:10.4230/OASIcs.ATMOS.2013.108.

18 Aaron Schild and Christian Sommer. On balanced separators in road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, 2015. doi:10.1007/978-3-319-20086-6_22.

19 Ben Strasser and Dorothea Wagner. Graph fill-in, elimination ordering, nested dissection and contraction hierarchies. In *Gems of Combinatorial Optimization and Graph Algorithms*. Springer, 2015. doi:10.1007/978-3-319-24971-1_7.