Alternative Routes in Road Networks

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck

Microsoft Research Silicon Valley
{ittaia, dadellin, goldberg, renatow}@microsoft.com

Abstract. We study the problem of finding good alternative routes in road networks. We look for routes that are substantially different from the shortest path, have small stretch, and are locally optimal. We formally define the problem of finding alternative routes with a single via vertex, develop efficient algorithms for it, and evaluate them experimentally. Our algorithms are efficient enough for practical use and compare favorably with previous methods in both speed and solution quality.

1 Introduction

We use web-based and autonomous navigation systems in everyday life. These systems produce driving directions by computing a shortest path (or an approximation) with respect to a length function based on various measures, such as distance and travel time. However, optimal paths do not necessarily match the personal preferences of individual users. These preferences may be based on better local knowledge, a bias for or against a certain route segment, or other factors. One can deal with this issue by presenting a small number of alternative paths and hoping one of them will satisfy the user. The goal of an algorithm is to offer alternative paths often, and for these alternatives to look reasonable.

Recent research on route planning focused on computing only a single (shortest) path between two given vertices (see [4] for an overview). Much less work has been done on finding multiple routes. Some commercial products (by companies such as Google and TomTom) suggest alternative routes using proprietary algorithms. Among published results, a natural approach is to use k-shortest path algorithms [8], but this is impractical because a reasonable alternative in a road network is probably not among the first few thousand paths. Another approach is to use multi-criteria optimization [13, 14], in which two or more length functions are optimized at once, and several combinations are returned. Efficient algorithms are presented in [5, 9]. Our focus is on computing reasonable alternatives with a single cost function. In this context, the best published results we are aware of are produced by the *choice routing* algorithm [2], which we discuss in Section 4. Although it produces reasonable paths, it is not fully documented and is not fast enough for continental-sized networks.

In this work, we study the problem of finding "reasonable" alternative routes in road networks. We start by defining in Section 3 a natural class of *admissible* alternative paths. Obviously, an alternative route must be substantially different from the optimal path, and must not be much longer. But this is not enough:

alternatives must feel natural to the user, with no unnecessary detours (formally, they must be *locally optimal*: every subpath up to a certain length must be a shortest path). Even with these restrictions, the number of admissible paths may be exponential, and computing the best one is hard. For efficiency, we focus on a more limited (yet useful) subset. Given an origin s and a target t, we restrict ourselves to *single via paths*, which are alternative routes obtained by concatenating the shortest paths from s to v and from v to t, for some vertex v. Section 4 discusses how the best such path can be computed in polynomial time using the bidirectional version of Dijkstra's algorithm (BD).

In practice, however, just polynomial time is not enough—we need sublinear algorithms. Modern algorithms for computing (optimal) shortest paths in road networks, such as those based on contraction hierarchies [10] and reach [11], are often based on pruning Dijkstra's search. After practical preprocessing steps, they need to visit just a few hundred vertices to answer queries on continental-sized graphs with tens of millions of vertices—orders of magnitude faster than BD. In fact, as shown in [1], their performance is sublinear on graphs with small highway dimension, such as road networks. Section 5 shows how to apply these speedup techniques to the problem of finding alternative routes, and Section 6 proposes additional measures to make the resulting algorithms truly practical.

Finally, Section 7 evaluates various algorithms experimentally according to several metrics, including path quality and running times. We show that finding a good alternative path takes only five times as much as computing the shortest path (with a pruning algorithm). Moreover, our pruning methods have similar success rates to a variant of choice routing, but are orders of magnitude faster.

Summarizing, our contributions are twofold. First, we establish a rigorous theoretical foundation for alternative paths, laying the ground for a systematic study of the problem. Second, we present efficient algorithms (in theory and in practice) for finding such routes.

2 Definitions and Background

Let $G = (V, E)$ be a directed graph with nonnegative, integral weights on edges, with $|V| = n$ and $|E| = m$. Given any path P in G, let $|P|$ be its number of edges and $\ell(P)$ be the sum of the lengths of its edges. By extension $\ell(P \cap Q)$ is the sum of the lengths of the edges shared by paths P and Q, and $\ell(P \setminus Q)$ is $\ell(P) - \ell(P \cap Q)$. Given two vertices, s and t, the *point-to-point shortest path problem* (P2P) is that of finding the shortest path (denoted by *Opt*) from s to t. Dijkstra's algorithm [7] computes $\text{dist}(s, t)$ (the distance from s to t in G) by scanning vertices in increasing order from s. Bidirectional Dijkstra (BD) runs a second search from t as well, and stops when both search spaces meet [3].

The *reach* of v, denoted by $r(v)$, is defined as the maximum, over all shortest u–w paths containing v, of $\min\{\text{dist}(u, v), dist(v, w)\}$. BD can be pruned at all vertices v for which both $\text{dist}(s, v) > r(v)$ and $\text{dist}(v, t) > r(v)$ hold [12]. The insertion of *shortcuts* (edges representing shortest paths in the original graph) may decrease the reach of some original vertices, thus significantly improving the efficiency of this approach [11]. The resulting algorithm (called RE) is three orders of magnitude faster than plain BD on continental-sized road networks.

An even more efficient algorithm (by another order of magnitude) is *contraction hierarchies* (CH) [10]. During preprocessing, it sorts all vertices by importance (heuristically), then shortcuts them in this order. (To *shortcut* a vertex, we remove it from the graph and add as few new edges as necessary to preserve distances.) A query only follows an edge (u, v) if v is more important than u.

3 Admissible Alternative Paths

In this paper, we are interested in finding an alternative path P between s and t. By definition, such a path must be significantly different from *Opt*: the total length of the edges they share must be a small fraction of $\ell(Opt)$.

This is not enough, however. The path must also be *reasonable*, with no unnecessary detours. While driving along it, every local decision must make sense. To formalize this notion, we require paths to be *locally optimal.* A first condition for a path P to be T *locally optimal* (T-LO) is that every subpath P' of P with $\ell(P') \leq T$ must be a shortest path. This would be enough if P were continuous, but for actual (discrete) paths in graphs we must "round up" with a second condition. If P' is a subpath of P with $\ell(P') > T$ and $\ell(P'') < T$ (P'' is the path obtained by removing the endpoints of P'), then P' must be a shortest path. Note that a path that is not locally optimal includes a local detour, which in general is not desirable. (Users who need a detour could specify it separately.)

Although local optimality is necessary for a path to be reasonable, it is arguably not sufficient (see Figure 1). We also require alternative paths to have limited stretch. We say that a path P has $(1+\epsilon)$ *uniformly bounded stretch* ($(1+\epsilon)$-UBS) if every subpath (including P itself) has stretch at most $(1+\epsilon)$.

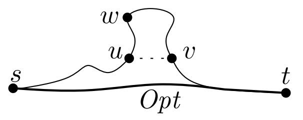

Fig. 1. Rationale for UBS. The alternative through w is a concatenation of two shortest paths, s–w and w–t. Although it has high local optimality, it looks unnatural because there is a much shorter path between u and v.

Given these definitions, we are now ready to define formally the class of paths we are looking for. We need three parameters: $0 < \alpha < 1$, $\epsilon \geq 0$, and $0 \leq \gamma \leq 1$. Given a shortest path *Opt* between s and t, we say that an s–t path P is an *admissible alternative* if it satisfies the following conditions:

1. $\ell(Opt \cap P) \leq \gamma \cdot \ell(Opt)$ (limited sharing);
2. P is T-locally optimal for $T = \alpha \cdot \ell(Opt)$ (local optimality);
3. P is $(1 + \epsilon)$-UBS (uniformly bounded stretch).

There may be zero, one, or multiple admissible alternatives, depending on the input and the choice of parameters. If there are multiple alternatives, we can sort them according to some objective function $f(\cdot)$, which may depend on any number of parameters (possibly including α, ϵ, and γ). In our experiments, we prefer admissible paths with low stretch, low sharing and high local optimality, as explained in Section 6. Other objective functions could be used as well.

Note that our definitions can be easily extended to report multiple alternative paths. We just have to ensure that the ith alternative is sufficiently different from the union of *Opt* and all $i-1$ previous alternatives. The stretch and local optimality conditions do not change, as they do not depend on other paths.

4 Single Via Paths

Even with the restrictions we impose on admissible paths, they may still be too numerous, making it hard to find the best one efficiently. This section defines a subclass of admissible paths (called *single via paths*) that is more amenable to theoretical analysis and practical implementation. Given any vertex v, the *via path through* v, P_v, is the concatenation of two shortest paths, s–v and v–t (recall that we are looking for s–t paths). We look for via paths that are admissible. As we will see, these can be found efficiently and work well in practice.

Note that single via paths have interesting properties. Among all s–t paths through v (for any v), P_v is the shortest, i.e., it has the lowest stretch. Moreover, being a concatenation of two shortest paths, the local optimality of P_v can only be violated around v. In this sense, via paths are close to being admissible.

Although all $n-2$ via paths can be implicitly generated with a single run of BD (in $O(m + n \log n)$ time), not all of them must be admissible. For each via path P_v, we must check whether the three admissibility conditions are obeyed.

The easiest condition to check is sharing. Let $\sigma_f(v)$ be the sharing amount in the forward direction (i.e., how much s–v shares with *Opt*, which is known). Set $\sigma_f(s) \leftarrow 0$ and for each vertex v (in forward scanning order), set $\sigma_f(v)$ to $\sigma_f(p_f(v)) + \ell(p_f(v), v)$ if $v \in Opt$ or to $\sigma_f(p_f(v))$ otherwise (here p_f denotes the parent in the forward search). Computing $\sigma_r(v)$, the sharing in the reverse direction, is similar. The total sharing amount $\sigma(v) = \ell(Opt \cap P_v)$ is $\sigma_f(v) + \sigma_r(v)$. Note that this entire procedure takes $O(n)$ time.

In contrast, stretch and local optimality are much harder to evaluate, requiring quadratically many shortest path queries (on various pairs of vertices). Ideally, we would like to verify whether a path P is locally optimal (or is $(1+\epsilon)$-UBS) in time proportional to $|P|$ and a few shortest-path queries. We do not know how to do this. Instead, we present alternative tests that are efficient, have good approximation guarantees, and work well in practice.

For local optimality, there is a quick 2-approximation. Take a via path P_v and a parameter T. Let P_1 and P_2 be the s–v and v–t subpaths of P_v, respectively. Among all vertices in P_1 that are at least T away from v, let x be the closest to v (and let $x = s$ if $\ell(P_1) < T$). Let y be the analogous vertex in P_2 (and let $y = t$ if $\ell(P_2) < T$). We say that P_v *passes the T-test* if the portion of P_v between x and y is a shortest path. See Figure 2 for an example.

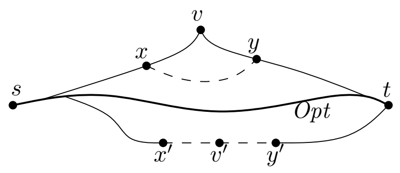

Fig. 2. Example for two T-tests. The T-test for v fails because the shortest path from u to w, indicated as a dashed spline, does not contain v. The test for v' succeeds because the shortest path from u' to w' contains v'.

Lemma 1. *If P_v passes the T-test, then P_v is T-LO.*

Proof. Suppose P_v passes the test and consider a subpath P' of P_v as in the definition of T-LO. If P' is a subpath of P_1 or P_2, then it is a shortest path. Otherwise P' contains v and is a subpath of the portion of P_v between x and y (as defined in the T-test), and therefore also a shortest path. □

This test is very efficient: it traverses P_v at most once and runs a single point-to-point shortest-path query. Although it may miss some admissible paths (a T-LO path may fail the T-test), it can be off by at most a factor of two:

Lemma 2. *If P_v fails the T-test, then P_v is not $2T$-LO.*

Proof. If P_v fails the test, then the x–y subpath in the definition of the test is not a shortest path. Delete x and y from the subpath, creating a new path P''. We know $\ell(P'') < 2T$ (v divides it into two subpaths of length less than T), which means P_v is not $2T$-LO. □

We now consider how to find the smallest ϵ for which a given path is $(1+\epsilon)$-UBS. We cannot find the exact value efficiently, but we can approximate it:

Lemma 3. *If a via path P_v has stretch $(1+\epsilon)$ and passes the T-test for $T = \beta \cdot \mathrm{dist}(s,t)$ (with $0 < \epsilon < \beta < 1$), then P_v is a $\frac{\beta}{\beta-\epsilon}$-UBS path.*

Proof. Consider a subpath P' of P_v between vertices u and w. If $v \notin P'$, or both u and w are within distance T of v, then P' is a shortest path (as a subpath of a shortest path). Assume v is between u and w and at least one of these vertices is at distance more than T from v. This implies $\ell(P') \geq T = \beta \cdot \mathrm{dist}(s,t)$. Furthermore, we know that $\ell(P') \leq \mathrm{dist}(u,w) + \epsilon \cdot \mathrm{dist}(s,t)$ (the absolute stretch of the subpath P' cannot be higher than in P_v). Combining these two inequalities, we get that $\ell(P') \leq \mathrm{dist}(u,w) + \epsilon \cdot \ell(P')/\beta$. Rearranging the terms, we get that $\ell(P') \leq \beta \cdot \mathrm{dist}(u,w)/(\beta - \epsilon)$, which completes the proof. □

BDV algorithm. We now consider a relatively fast BD-based algorithm, which we call BDV. It grows shortest path trees from s and into t; each search stops when it advances more that $(1+\epsilon)\ell(\mathit{Opt})$ from its origin. (This is the longest an admissible path can be.) For each vertex v scanned in both directions, we check whether the corresponding path P_v is *approximately admissible*: it shares at most $\gamma \cdot \ell(\mathit{Opt})$ with Opt, has limited stretch ($\ell(P_v) \leq (1+\epsilon)\ell(\mathit{Opt})$), and passes the T-test for $T = \alpha \cdot \ell(\mathit{Opt})$. Finally, we output the best approximately admissible via path according to the objective function.

The choice routing algorithm. A related method is the *choice routing algorithm* (CR) [2]. It starts by building shortest path trees from s and to t. It then looks at *plateaus*, i.e., maximal paths that appear in both trees simultaneously. In general, a plateau u–w gives a natural s–t path: follow the out-tree from s to u, then the plateau, then the in-tree from w to t. The algorithm selects paths corresponding to long plateaus, orders them according to some "goodness" criteria (not explained in [2]), and outputs the best one (or more, if desired). Because the paths found by CR have large plateaus, they have good local optimality:

Lemma 4. *If P corresponds to a plateau v–w, P is* $\mathrm{dist}(v, w)$*-LO.*

Proof. If P is not a shortest path, then there are vertices x, y on P such that the length of P between these vertices exceeds $\mathrm{dist}(x, y)$. Then x must strictly precede v on P, and y must strictly follow w. This implies the lemma. □

Note that both CR and BDV are based on BD and only examine single-via paths. While BDV must run one point-to-point query to evaluate each candidate path, all plateaus can be detected in linear time. This means CR has the same complexity as BD (ignoring the time for goodness evaluation), which is much faster than BDV. It should be noted, however, that local optimality can be achieved even in the absence of long plateaus. One can easily construct examples where BDV succeeds and CR fails. Still, neither method is fast enough for continental-sized road networks.

5 Pruning

A natural approach to accelerate BD is to prune it at unimportant vertices (as done by RE or CH, for example). In this section, we show how known pruning algorithms can be extended to find admissible single-via paths. The results of [1] suggest that pruning is unlikely to discard promising via vertices. Because of local optimality, an admissible alternative path P contains a long shortest subpath P' that shares little with *Opt*. Being a shortest path, P' must contain an "important" (unpruned) vertex v.

For concreteness, we focus on an algorithm based on RE; we call it REV. Like BDV, REV builds two (now pruned) shortest paths trees, out of s and into t. We then evaluate each vertex v scanned by both searches as follows. First, we perform two P2P queries (s–v and v–t) to find P_v. (They are necessary because some original tree paths may be suboptimal due to pruning.) We then perform an approximate admissibility test on P_v, as in BDV. Among all candidate paths that pass, we return the one minimizing the objective function $f(\cdot)$.

The main advantage of replacing BD by RE is a significant reduction in the number of via vertices we consider. Moreover, auxiliary P2P queries (including T-tests) can also use RE, making REV potentially much faster than BDV.

An issue we must still deal with is computing the sharing amount $\sigma(v)$. RE adds shortcuts to the graph, each representing a path in the original graph. To calculate $\sigma(v)$ correctly, we must solve a *partial unpacking* subproblem: given a shortcut (a, c), with $a \in Opt$ and $c \notin Opt$, find the vertex $b \in Opt$ that belongs to the a–c path and is farthest from a. Assuming each shortcut bypasses exactly one vertex and the shortcut hierarchy is balanced (as is usually the case in practice), this can be done in $O(\log n)$ time with binary search.

Running time. We can use the results of [1] to analyze REV. Our algorithms and their implementations work for directed graphs, but to apply the results of [1] we assume the input network is undirected in the analysis.

Suppose we have a constant-degree network of diameter D and highway dimension h. The reach-based query algorithm scans $O(k \log D)$ vertices and runs in time $O((k \log D)^2)$, where k is either h or $h \log n$, depending on whether preprocessing must be polynomial or not. For each vertex v scanned in both directions, REV needs a constant number of P2P queries and partial unpackings. The total running time is therefore $O((k \log D)^3 + k \log D \log n)$, which is sublinear (unlike BDV or CR). The same algorithm (and analysis) can be applied if we use contraction hierarchies; we call the resulting algorithm CHV.

Relaxed reaches. Pruning may cause REV and CHV to miss candidate via vertices, leading to suboptimal solutions. In rare cases, they may not find any admissible path even when BDV would. For REV, we can fix this by trading off some efficiency. If we multiply the reach values by an appropriate constant, the algorithm is guaranteed to find all admissible single-via paths. The resulting algorithm is as effective as BDV, but much more efficient. It exploits the fact that vertices in the middle of locally optimal paths have high reach:

Lemma 5. *If P is T-LO and $v \in P$, then $r(v) \geq \min\{T/2, \text{dist}(s,v), \text{dist}(v,t)\}$.*

Proof. Let v be at least $T/2$ away from the endpoints of P. Let x and y be the closest vertices to v that are at least $T/2$ away from v towards s and t, respectively. Since P is T-LO, the subpath of P between x and y is a shortest path, and v has reach at least $T/2$. □

Corollary 1. *If P passes the T-test and $v \in P$, then $r(v) \geq \min\{T/4, \text{dist}(s,v), \text{dist}(v,t)\}$.*

Let δ-REV be a version of REV that uses original reach values multiplied by $\delta \geq 1$ to prune the original trees from s and to t (i.e., it uses $\delta \cdot r(v)$ instead of $r(v)$). Auxiliary P2P computations to build and test via paths can still use the original $r(v)$ values. The algorithm clearly remains correct, but may prune fewer vertices. The parameter δ gives a trade-off between efficiency and success rate:

Theorem 1. *If $\delta \geq 4(1+\epsilon)/\alpha$, δ-REV finds the same admissible via paths as BDV.*

Proof. Consider a via path P_v that passes the T-test for $T = \alpha \cdot \text{dist}(s,t)$. Then by Corollary 1 for every vertex $u \in P_v$, $r(u) \geq \min\{\text{dist}(s,v), \text{dist}(v,t), \alpha \cdot \text{dist}(s,t)/4\}$. When $\delta \geq 4(1+\epsilon)/\alpha$, then $\delta \cdot r(u) \geq \min\{\text{dist}(s,v), \text{dist}(v,t), (1+\epsilon)\text{dist}(s,t)\}$. Therefore no vertex on P_v is pruned, implying that $\text{dist}(s,v)$ and $\text{dist}(v,t)$ are computed correctly. As a result P_v is considered as an admissible via path. □

The analysis of [1] implies that multiplying reach values by a constant increases the query complexity by a constant multiplicative factor. Hence, the asymptotic time bounds for REV also apply to δ-REV, for any constant $\delta \geq 1$. Note that Theorem 1 assumes that δ-REV and BDV are applied on the same graph, with no shortcuts.

6 Practical Algorithms

The algorithms proposed so far produce a set of candidate via paths and explicitly check whether each is admissible. We introduced techniques to reduce the number of candidates and to check (approximate) admissibility faster, with a few point-to-point queries. Unfortunately, this is not enough in practice. Truly practical algorithms can afford at most a (very small) constant number of point-to-point queries *in total* to find an alternative path. Therefore, instead of actually evaluating all candidate paths, in our experiments we settle for finding one that is good enough. As we will see, we sort the candidate paths according to some objective function, test the vertices in this order, and return the first admissible path as our answer. We consider two versions of this algorithm, one using BD and the other (the truly practical one) a pruned shortest path algorithm (RE or CH). Although based on the methods we introduced in previous sections, the solutions they find do not have the same theoretical guarantees. In particular, they may not return the best (or any) via path. As Section 7 will show, however, the heuristics still have very high success rate in practice.

The practical algorithms sort the candidate paths P_v in nondecreasing order according to the function $f(v) = 2\ell(v) + \sigma(v) - pl(v)$, where $\ell(v)$ is the length of the via path P_v, $\sigma(v)$ is how much P_v shares with Opt, and $pl(v)$ is the length of the longest plateau containing v. Note that $pl(v)$ is a lower bound on the local optimality of P_v (by Lemma 4); by preferring vertices with high $pl(v)$, we tend to find an admissible path sooner.

We are now ready to describe X-BDV, an experimental version of BDV that incorporates elements of CR for efficiency. Although much slower than our pruning algorithms, this version is fast enough to run experiments on. It runs BD, with each search stopping when its radius is greater than $(1+\epsilon)\ell(Opt)$; we also prune any vertex u with $\text{dist}(s,u) + \text{dist}(u,t) > (1+\epsilon)\ell(Opt)$. In linear time, we compute $\ell(v)$, $\sigma(v)$, and $pl(v)$ for each vertex v visited by both searches. We use these values to (implicitly) sort the alternative paths P_v in nondecreasing order according to $f(v)$. We return the first path P_v in this order that satisfies three hard constraints: $\ell(P_v \setminus Opt) < (1+\epsilon)\ell(Opt \setminus P_v)$ (the detour is not much longer than the subpath it skips), $\sigma(v) < \gamma \cdot \ell(Opt)$ (sharing is limited), and $pl(v) > \alpha \cdot \ell(P_v \setminus Opt)$ (there is enough local optimality). Note that we specify local optimality relative to the detour only (and not the entire path, as in Section 3). Our rationale for doing so is as follows. In practice, alternatives sharing up to 80% with Opt may still make sense. In such cases, the T-test will always fail unless the (path-based) local optimality is set to 10% or less. This is too low for alternatives that share nothing with Opt. Using detour-based local optimality is a reasonable compromise. For consistency, ϵ is also used to bound the stretch of the detour (as opposed to the entire path).

The second algorithm we tested, X-REV, is similar to X-BDV but grows reach-pruned trees out of s and into t. The stopping criteria and the evaluation of each via vertex v are the same as in X-BDV. As explained in Section 4, RE trees may give only upper bounds on $\text{dist}(s,v)$ and $\text{dist}(v,t)$ for any vertex $v \notin Opt$. But we can still use the approximate values (given by the RE trees) of $\ell(v)$, $\sigma(v)$, and

$pl(v)$ to sort the candidate paths in nondecreasing order according to $f(v)$. We then evaluate each vertex v in this order as follows. We first compute the actual via path P_v with two RE queries, s–v and v–t (as an optimization, we reuse the original forward tree from s and the backward tree from t). Then we compute the exact values of $\ell(v)$ and $\sigma(v)$ and check whether $\ell(P_v \setminus Opt) < (1+\epsilon)\ell(Opt \setminus P_v)$, whether $\sigma(v) < \gamma \cdot \ell(Opt)$, and run a T-test with $T = \alpha \cdot \ell(P_v \setminus Opt)$. If P_v passes all three tests, we pick it as our alternative. Otherwise, we discard v as a candidate, penalize its descendants (in both search trees) and try the next vertex in the list. We penalize a descendant u of v in the forward (backward) search tree by increasing $f(u)$ by $\text{dist}(s, v)$ ($\text{dist}(v, t)$). This gives less priority to vertices that are likely to fail, keeping the number of check queries small.

A third implementation we tested was X-CHV, which is similar to X-REV but uses contraction hierarchies (rather than reaches) for pruning.

7 Experiments

We implemented the algorithms from Section 6 in C++ and compiled them with Microsoft Visual C++ 2008. Queries use a binary heap as priority queue. The evaluation was conducted on a dual AMD Opteron 250 running Windows 2003 Server. It is clocked at 2.4 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. Our code is single-threaded and runs on a single processor at a time.

We use the European road network, with 18 million vertices and 42 million edges, made available for the 9th DIMACS Implementation Challenge [6]. It uses travel times as the length function. (We also experimented with TIGER/USA data, but closer examination revealed that the data has errors, with missing arcs on several major highways and bridges. This makes the results less meaningful, so we do not include them.) We allow the detour to have maximum stretch $\epsilon = 25\%$, set the maximum sharing value to $\gamma = 80\%$, and set the minimum detour-based local optimality to $\alpha = 25\%$ (see Section 6).

X-REV and X-CHV extend the point-to-point query algorithms RE [11] and CH [10]. Since preprocessing does not change, we use the preprocessed data given in [10, 11]. In particular, preprocessing takes 45 minutes for RE and 25 for CH.

We compare the algorithms in terms of both query performance and path quality. Performance is measured by the number of vertices scanned and by query times (given in absolute terms and as a multiple of the corresponding P2P method). Quality is given first by the *success rate*: how often the algorithm finds as many alternatives as desired. Among the successful runs, we also compute the average and worst uniformly-bounded stretch, sharing, and detour-based local optimality (reporting these values requires $O(|P|^2)$ point-to-point queries for each path P; this evaluation is not included in the query times). Unless otherwise stated, figures are based on 1 000 queries, with source s and target t chosen uniformly at random.

In our first experiment, reported in Table 1, we vary p, the desired number of alternatives to be computed by the algorithms. There is a clear trade-off between success rates and query times. As expected, X-BDV is successful more often, while

Table 1. Performance of various algorithms on the European road network as the number of desired alternatives (p) changes. Column *success rate* reports how often the algorithm achieves this goal. For the successful cases, we report the (average and worst-case) quality of the p-th alternative in terms of UBS, sharing, and detour-based local optimality. Finally, we report the average number of scanned vertices and query times (both in milliseconds and as a multiple of the corresponding point-to-point variant).

		PATH QUALITY							PERFORMANCE		
		success	UBS[%]		sharing[%]		locality[%]		#scanned	time	slow-
p	algo	rate[%]	avg	max	avg	max	avg	min	vertices	[ms]	down
1	X-BDV	94.5	9.4	35.8	47.2	79.9	73.1	30.3	16 963 507	26 352.0	6.0
	X-REV	91.3	9.9	41.8	46.9	79.9	71.8	30.7	16 111	20.4	5.6
	X-CHV	58.2	10.8	42.4	42.9	79.9	72.3	29.8	1 510	3.1	4.6
2	X-BDV	81.1	11.8	38.5	62.4	80.0	71.8	29.6	16 963 507	29 795.0	6.8
	X-REV	70.3	12.2	38.1	60.3	80.0	71.3	29.6	25 322	33.6	9.2
	X-CHV	28.6	10.8	45.4	55.3	79.6	77.6	30.3	1 685	3.6	5.3
3	X-BDV	61.6	13.2	41.2	68.9	80.0	68.7	30.6	16 963 507	33 443.0	7.7
	X-REV	43.0	12.8	41.2	66.6	80.0	74.9	33.3	30 736	42.6	11.7
	X-CHV	10.9	12.0	41.4	59.3	80.0	79.0	36.1	1 748	3.9	5.8

X-CHV is fastest. Unfortunately, X-BDV takes more than half a minute for each query and X-CHV finds an alternative in only 58.2% of the cases (the numbers are even worse with two or three alternatives). The reach-based algorithm, X-REV, seems to be a good compromise between these extremes. Queries are still fast enough to be practical, and it is almost as successful as X-BDV. In only 20.4 ms, it finds a good alternative in 91.3% of the cases.

Alternative paths, when found, tend to have similar quality, regardless of the algorithm. This may be because the number of admissible alternatives is small: the algorithms are much more successful at finding a single alternative than at finding three (see Table 1). On average, the first alternative has 10% stretch, is 72% locally optimal, and shares around 47% with the optimum. Depending on the success rate and p, the alternative query algorithm is 4 to 12 times slower than a simple P2P query with the same algorithm. This is acceptable, considering how much work is required to identify good alternatives.

As observed in Section 5, we can increase the success rate of REV by multiplying reach values by a constant $\delta > 1$. Our second experiment evaluates how this multiplier affects X-REV, the practical variant of REV. Table 2 reports the success rate and query times of δ-X-REV for several values of δ when $p = 1$. As predicted, higher reach bounds do improve the success rate, eventually matching that of X-BDV on average. The worst-case UBS is also reduced from almost 42% to around 30% when δ increases. Furthermore, most of the quality gains are already obtained with $\delta = 2$, when queries are still fast enough (less than 10 times slower than a comparable P2P query).

The original reach values used by X-REV are not exact: they are upper bounds computed by a particular preprocessing algorithm [11]. On a smaller graph (of the Netherlands, with $n \approx 0.9$M and $m \approx 2.2$M), we could actually afford to

Table 2. Performance of X-REV when varying the multiplier (δ) for reach values.

		PATH QUALITY							PERFORMANCE		
		success	UBS[%]		sharing[%]		locality[%]		#scanned	time	slow-
algo	δ	rate[%]	avg	max	avg	max	avg	min	vertices	[ms]	down
X-REV	1	91.3	9.9	41.8	46.9	79.9	71.8	30.7	16 111	20.4	5.6
	2	94.2	9.7	31.6	46.6	79.9	71.3	27.6	31 263	34.3	9.4
	3	94.2	9.5	29.2	46.7	79.9	71.9	31.2	53 464	55.3	15.2
	4	94.3	9.5	29.3	46.7	79.9	71.8	31.2	80 593	83.2	22.8
	5	94.4	9.5	29.3	46.7	79.9	71.8	31.4	111 444	116.6	31.9
	10	94.6	9.5	30.2	46.8	79.9	71.7	31.4	289 965	344.3	94.3
X-BDV	–	94.5	9.4	35.8	47.2	79.9	73.1	30.3	16 963 507	26 352.0	6.0

compute exact reaches on the shortcut-enriched graph output by the standard RE preprocessing. On this graph, the success rate drops from 83.4% with the original upper bounds to 81.7% with exact reaches. Multiplying the exact reaches by $\delta = 2$ increases the success rate again to 83.6%. With $\delta = 5$, we get 84.7%, very close to the 84.9% obtained by X-BDV. Note that the Dutch subgraph tends to have fewer alternatives than Europe as a whole.

To examine this issue in detail, Figure 3 reports the success rate of our algorithms for $p = 1$ and various Dijkstra ranks on Europe. The *Dijkstra rank* of v with respect to s is i if v is the ith vertex taken from the priority queue when running a Dijkstra query from s. The results are based on 1 000 queries for each rank. We observe that the success rate is lower for local queries, as expected. Still, for mid-range queries we find an alternative in 60% to 80% of the cases, which seems reasonable. (Recall that an admissible alternative may not exist.) Multiplying reach values helps all types of queries: 2-X-REV has almost the same success rate as X-BDV for all ranks and number of alternatives examined.

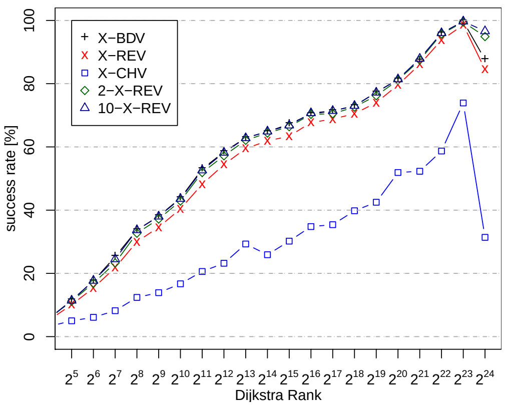

Fig. 3. Success rate for X-BDV, X-REV, 2-X-REV, 10-X-REV, and X-CHV when varying the Dijkstra rank of queries for $p = 1$.

8 Conclusion

By introducing the notion of admissibility, we have given the first formal treatment to the problem of finding alternative paths. The natural concept of local optimality allows us to prove properties of such paths. Moreover, we have given theoretically efficient algorithms for an important subclass, that of single via paths. We concentrated on making the approach efficient for real-time applications by designing approximate admissibility tests and an optimization function biased towards admissible paths. Our experiments have shown that these simplified versions are practical for real, continental-sized road networks.

More generally, however, our techniques allow us to do optimization constrained to the (polynomially computable) set of admissible single via paths. We could optimize other functions over this set, such as fuel consumption, time in traffic, or tolls. This gives a (heuristic) alternative to multi-criteria optimization.

Our work leads to natural open questions. In particular, are there efficient exact tests for local optimality and uniformly bounded stretch? Furthermore, can one find admissible paths with multiple via vertices efficiently? This is especially interesting because it helps computing arbitrary admissible paths, since any admissible alternative with stretch $1+\epsilon$ and local optimality $\alpha \cdot \ell(Opt)$ is defined by at most $\lceil (1+\epsilon)/\alpha \rceil - 1$ via points.

References

1. I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *SODA*, pages 782–793, 2010.
2. Cambridge Vehicle Information Technology Ltd. Choice Routing, 2005. Available at http://www.camvit.com.
3. G. B. Dantzig. *Linear Programming and Extensions*. Princeton Univ. Press, Princeton, NJ, 1962.
4. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 117–139. Springer, 2009.
5. D. Delling and D. Wagner. Pareto Paths with SHARC. In J. Vahrenhold, editor, *SEA'09*, volume 5526 of *LNCS*, pages 125–136. Springer, June 2009.
6. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *9th DIMACS Implementation Challenge - Shortest Paths*, 2006.
7. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
8. D. Eppstein. Finding the k shortest paths. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS'94)*, pages 154–165, 1994.
9. R. Geisberger, M. Kobitzsch, and P. Sanders. Route Planning with Flexible Objective Functions. In *ALENEX*, pages 124–137. SIAM, 2010.
10. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In C. C. McGeoch, editor, *WEA'08*, volume 5038 of *LNCS*, pages 319–333. Springer, June 2008.
11. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 93–139. American Mathematical Society, 2009.
12. R. J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *ALENEX*, pages 100–111. SIAM, 2004.
13. P. Hansen. Bricriteria Path Problems. In G. Fandel and T. Gal, editors, *Multiple Criteria Decision Making: Theory and Application*, pages 109–127. Springer, 1979.
14. E. Q. Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 26(3):236–245, 1984.