

Candidate Sets for Alternative Routes in Road Networks^{*}

Dennis Luxen and Dennis Schieferdecker {luxen,schieferdecker}@kit.edu

Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract. We present a fast algorithm with preprocessing for computing multiple good alternative routes in road networks. Our approach is based on single via node routing on top of Contraction Hierarchies and achieves superior quality and efficiency compared to previous methods. The algorithm has neglectable memory overhead.

^{*} Partially supported by the German Research Foundation (DFG) within the Research Training Group GRK 1194 "Self-organizing Sensor-Actuator-Networks" and by DFG grant SA 933/5-1.

1 Introduction and Related Work

Today’s requirements for routing services, be it in-car or as a web-service, ask for more than just computing the shortest or quickest paths. Thus it is desirable to not only present a single path to a user, but instead a set of paths which are perceived as reasonable alternatives.

We show how to engineer previous algorithms to provide reasonable alternative paths with better efficiency. Then, we build on the results and introduce the notion of *candidate via nodes* to further speed up the computation by an order of magnitude. We show how to perform query variants and how to conduct the preprocessing efficiently. Finally, we conduct an experimental study on the performance and quality of our method.

The shortest path problem can be solved by Dijkstra’s seminal algorithm [1]. Unfortunately, it does not scale to large-scale instances. Heuristics to prune the search space like *that provide goal direction* [2, 3] ease the problem. An early optimal and performant technique that provides substantial speedups is *arc flags*; originally conceived by Lauther [4, 5]; later by Möhring et al. [6] and Köhler et al. [7]. The road network is partitioned into regions and each edge stores a flag to indicate if there is a shortest path into a region. Techniques exploiting the *hierarchy of a road network* follow the notion that sufficiently long routes will enter the arterial network at some point, e.g. enter a highway or national road. *Contraction Hierarchies (CH)* [8] have a convenient trade-off between preprocessing and query time. Road networks of continental size can be preprocessed within minutes and queries run in the order of about one hundred microseconds. CH heuristically order the nodes by some measure of importance and shortcut them in this order. This means that a node is removed from the graph and as few edges as possible are inserted to preserve shortest path distances. The original edges are augmented by the shortcut edges to build the search data structure. A query (a bidirected Dijkstra) only needs to follow edges that lead to more important nodes. Hence, the data structure forms a directed acyclic graph. Albeit the length of any shortest path is optimal, it may consist of shortcut edges that need to be recursively unpacked. The fastest CH variant is *CHASE* [9] that combines CH with arc flags. Its queries run in the order of ten microseconds.

Recently, Abraham et al. [10, 11] give analyses of the performance of speedup techniques to Dijkstra’s algorithm. Also, Abraham et al. [12] give an efficient implementation of the theoretical algorithm, which achieves distance query times below a single microsecond. Please note that we refer to various papers when speaking of Abraham et al. [10–13].

Alternative paths that combine two shortest paths over a *via node* are used by *Choice Routing* [14], also referred to as *plateau method*. The road network is modelled as a graph $G = (V, E)$ and shortest path trees are grown from nodes s and t . *Plateaus* $\langle u, \dots, v \rangle$ running from node u to v are maximal paths that appear in both trees. They give candidates for natural alternative paths, i.e. follow the forward tree from s to u , then the plateau, and then the reverse tree from v to t . Although not entirely published, the plateau method provides alternatives of good quality in practice. Further discussion on this can be found in [13].

2 The Baseline Algorithm

Abraham et al. [13] define a class of *admissible* alternative paths. For a given s - t -pair and via node v the (via) path P_v is a concatenation of the two shortest paths s - v and v - t . The shortest path between s and t is called P_{opt} and the length of a path P_v is denoted by $l(P_v)$. Via path P_v has to be reasonable to be considered as a viable alternative and thus must obey three heuristic, but natural properties:

First, P_v has to be significantly different from P_{opt} . This states that the total length of the edges both paths share must only be a fraction of the length of the optimal path. Second, P_v has to be *T -locally optimal (T -LO)*, which means that every sufficiently short subpath P' of P_v must be a shortest path. In other words, every local decision along the alternative path must be reasonable. This is formalized by two properties. Every sufficiently short subpath $P' \subseteq P_v$ with $l(P') \leq T$ has to be a shortest path. If P' is a subpath of P_v and P'' is obtained by removing endpoints of P' then P' must also be a shortest path if $l(P') > T \wedge l(P'') < T$ holds. Third, the alternative path needs to have limited stretch. A path P_v is said to have $(1+\varepsilon)$ *uniformly bound stretch (UBS)* if every subpath $P' \subseteq P_v$ has stretch of at most $(1+\varepsilon)$. As such, every alternative should only be a fraction longer than a shortest path.

Given parameters $0 < \alpha < 1$, $0 \leq \gamma \leq 1$, and $\varepsilon \geq 0$ as well as the above properties, we formalize

Definition 1 (Admissible path). *A path P_v between s and t is an admissible alternative if*

- a) $l(P_{opt} \cap P_v) \leq \gamma \cdot l(P_{opt})$ (*limited sharing*),
- b) P_v is T -locally optimal for $T = \alpha \cdot l(P_{opt})$ (*local optimality*), and
- c) P_v has $(1 + \varepsilon)$ -UBS (*uniformly bounded stretch*).

These measures require a quadratic number of shortest path queries to be verified, which is not feasible for a real-time setting. Thus, more practical algorithms are needed that have a narrower focus on easy computability. There exists a quick 2-approximation (*T -test*) for T -local optimality. Given a via path P_v and a parameter T , let x be the closest node on s - v that is at least T away from v or s . Likewise, y is the closest node on v - t that is also at least T away or s . A path P_v is said to *pass the T -test* if the portion of P_v between x and y is a shortest path.

Abraham et al. [13] give a practical solution based on a bidirectional Dijkstra (BD), called *X -BDV*, to compute single via paths that are reasonable and good alternatives. The algorithm incorporates ideas from the plateau method. An *Exploration Dijkstra* identifies potential alternative paths: A (forward) shortest path tree is grown from s , and another (backward) tree from t , until all nodes are settled that are not farther than $(1 + \varepsilon) \cdot l(P_{opt})$ away from the root of their respective tree. Note that no admissible path can be any longer. Each node v that is settled in both search trees becomes a via node candidate and three measurements are computed in linear time: $l(P_v)$, the length of via path P_v , $\sigma(P_v)$, the amount of sharing of P_v with the optimal route, and $pl(P_v)$, the length of a longest plateau containing v . Note that if $pl(P_v) > T$, the T -test is always successful. These more practical measures are used to sort all candidates in non-decreasing order according to the priority

function $f(P_v) = 2 \cdot l(P_v) + \sigma(P_v) - pl(P_v)$. The first path P_v is returned that is approximately admissible as described below.

Definition 2 (Approximately Admissible). *A path P_v between s and t is approximately admissible if the following three conditions hold*

1. $\sigma(P_v) < \gamma \cdot l(P_{opt})$ (limited sharing),
2. successful T -test for $T = \alpha \cdot l(P_v \setminus P_{opt})$ (local optimality), and
3. $l(P_v \setminus P_{opt}) < (1 + \varepsilon) \cdot l(P_{opt} \setminus P_v)$ (small stretch).

Local optimality and stretch are defined with respect to the detour of the alternative. The above method yields the algorithm *X-CHV* [13] when combined with Contraction Hierarchies. The forward and backward (CH) search spaces of nodes s and t are explored. Nodes v in the forward search space are reached with a *forward distance* $l^\uparrow(P_{sv})$ and nodes in the backward search space with a *backward distance* $l^\downarrow(P_{vt})$. For nodes v that occur in both search spaces a preselection is run. Nodes are discarded, if the sum of forward and backward distance is longer than a certain fraction of the length of the shortest path: $l^\uparrow(P_{sv}) + l^\downarrow(P_{vt}) < (1 + \varepsilon) \cdot l(P_{opt})$. Note that these distances are not necessarily correct but upper bounds. It is tested if the *approximated overlap* $\sigma^{apx}(P_v)$ is no longer than a certain fraction of the length of the shortest path: $\sigma^{apx}(P_v) < (1 + \varepsilon) \cdot l(P_{opt})$. Additionally, the following condition concerning the stretch must hold: $l^\uparrow(P_{sv}) + l^\downarrow(P_{vt}) - \sigma^{apx}(P_v) < (1 + \varepsilon) \cdot (l(P_{opt}) - \sigma^{apx}(P_v))$. Remaining candidates are ranked according to the priority function of X-BDV. The exact path $\langle s..v..t \rangle$ is computed for nodes v in that order. The first node for which the properties of Definition 2 hold is selected as via node.

The success rate of X-CHV is inferior to X-BDV since search spaces are much narrower. To cope with the smaller success rate, Abraham et al. [13] introduce a relaxed exploration phase: The exploration query is allowed to search more nodes than the plain CH query. Let $p_i(u)$ be the i -th ancestor of u in the search tree. The *x -relaxed X-CHV query* prunes an edge (u, v) if and only if v precedes all vertices $u, p_1(u), \dots, p_x(u)$ in the order of the CH. Note, the x -relaxed variant of *X-CHV*, with $x \in \{0, 3\}$, is the baseline of our work. This section ends the recap of previous work.

3 Engineering the Baseline Algorithm

Recall that the baseline is a two step approach. A bidirectional Exploration (CH) Dijkstra searches for via node candidates that are then tested for admissibility using a number of point-to-point (p2p) shortest path queries, which we call *Target (CH) Dijkstras*. The obvious approach to apply engineering is to handle the Target Dijkstras by faster methods than the normal Contraction Hierarchies query algorithm. For instance, we apply *CHASE* that computes these queries by exploiting additional arc flags [9]. This does not apply to Exploration Dijkstras, because search spaces would be too narrow. Storing all shortcuts pre-unpacked speeds up path computation as well. Both optimization have equal impact and result in an algorithm with query times of less than half of plain X-CHV. We refer to this straight-forward engineered baseline algorithm by *X-CHASEV*.

The analyses of Abraham et al. [10] show that speedup-techniques to Dijkstra’s algorithm work especially well on certain classes of graphs in which all shortest paths out of a region are *covered* by a small node set. This theoretical analysis leads to the following assumption:

Assumption 1 (limited number of alternative paths) *If the number of shortest paths between any two sufficiently far away regions of a road network is small [10]. Likewise the number of admissible paths of the algorithm of Abraham et al. [13] is also small and can be covered by a small number of nodes.*

4 Single-Level Via Node Candidates

We partition the graph and apply bootstrapping to generate *via node candidate sets* for pairs of partitions. Here, bootstrapping means that the query algorithm which is used later on to actually compute an alternative path is used during preprocessing as well.

Assume that for each pair of non-neighboring partitions, we have computed a set of via node candidates. Note that since candidates are already present, we do not need to identify them during an exploration step. Computing an alternative path for a given s - t -query now becomes straight-forward. We loop over all nodes v in the via node candidate set of the pair of partitions of s and t . For each v we check whether P_v is approximately admissible using the properties of Definition 2. The first approximately admissible path found is returned as the result. If no candidate is viable or if the size of the candidate set is zero, no alternative path is returned.

In a s - t query between neighboring partitions or within a single partition we perform X-CHASEV as fallback instead. The reason for this is that the number of candidates between those pairs of partitions and within a single one can be numerous. It is faster to use the fallback algorithm than to check pregenerated node sets in most of these cases.

Precomputating via node candidates starts with a partitioning of the underlying road network. A number of such schemes have been proposed before. We do not focus on that subproblem but refer to [15, 16] instead. A set of via node candidates is generated greedily for each pair of partitions. A tentative via node set that keeps track of the candidates identified so far during preprocessing for each pair. We use the above algorithm with the tentative node set. If no alternative is found, we run X-CHASEV as bootstrapping to identify one. Whenever such a fallback run results in a new via node, it is inserted into the set of tentative via nodes.

4.1 Multi-Level Via Node Candidates

We propose a multi-level partitioning to compute via node candidates for neighboring pairs of partitions or within a single partition. The graph is further partitioned into an order of magnitude more partitions. The finer partitioning respects the coarser one in the sense that the nodes of a fine partition belong to one and only one of the coarse partitions. We do not run full preprocessing for all pairs of fine partitions. This would induce an amount of additional preprocessing steps (quadratic in the number of partitions). Our algorithm

runs fine for most coarse partition pairs and we run the same preprocessing algorithm as before only on a subset of all fine partition pairs. These are the pairs for which origin and destination were too close together, i.e. in the same coarse partition or in neighboring ones. Note, we preprocess each non-neighboring fine partition pair that either belongs to the same or to a pair of neighboring coarse partitions. This implies only a linear amount of additional preprocessing work.

A query recurses to the multi-level partitioning for nodes of two neighboring coarse partitions or between nodes within the same coarse partition. When origin and destination are within the same or in neighboring fine partitions, plain X-CHASEV is run as fallback. Fine partitions are much smaller, and origin and destination are generally close to each other.

128 partitions are used for the arc flags of X-CHASEV. The number of explored nodes during a CHASE query with 128 partitions that do not belong to the shortest path is tiny [9]. Hence, we do not see any benefit of investing time into the generation of arc flags for 1024 partitions.

4.2 Further Engineering

The preprocessing is easily adaptable to shared-memory parallelism by preprocessing all pairs of partitions independently. This parallelization scales almost linearly with the number of processors until the memory bandwidth is reached. Most preprocessing runs verify the existence of a via node, but do not result in a new one. Sampling effectively decreases the preprocessing time when the sample is of reasonable size. E.g. running such a preprocessing on 1/16 of all of the pairs of boundary nodes for each partition pair results in only slightly inferior query performance.

Much effort during preprocessing is spent in search space exploration. The search space of each boundary node is required repeatedly. This can be hastened by about a factor of three by storing the search spaces of boundary nodes. Another tuning parameter is the order in which the nodes are stored in the tentative sets. We order by the number of how often a node occurs as a via node during preprocessing. This order is not necessarily the best of all orders. It depends on the order in which the pairs of boundary nodes are visited. Computing a best among all possible sorting orders, independent of the visiting order, is feasible and leads to slightly superior query times, but is computationally expensive. Note that selecting a via node greedily is of course faster since the first viable node is used, while selecting the via node that yields a best quality alternative is more expensive. Queries can be further accelerated by storing (forward and backward) search spaces of the via node candidate sets and also by storing the shortcuts pre-unpacked, as mentioned before.

5 Experiments

We implement the above algorithms in C++ using GCC's compiler with full optimizations. A binary heap is used as priority queue data structure. The experiments are conducted on two separate machines. Queries run on one core of an Intel Core i7-920 CPU (4 cores), clocked at 2.66 GHz with 12 GiB main memory. It is running Linux (kernel 2.6.34, gcc version 4.5.0).

Parallel preprocessing is done on 4 AMD Opteron 6168 CPUs (12 cores each), clocked at 1.90 Ghz with 256 GiB main memory. This machine is running Linux (kernel 2.6.38, GCC version 4.5.2) and has roughly half the single-core performance compared to the Core i7 machine. Timings are done using the clock cycle counter available in 64 bit x86 CPUs.

5.1 Methodology

We test our approach on a road network of Western Europe provided by PTV AG for the 9th Dimacs Challenge [17]. It consists of 18 million nodes and 42 million edges and uses the travel time metric as edge weights. We partition the graph into 128 partitions using the algorithm of Sanders and Schulz [16], yielding an average edge cut of 6 360 and 91.8 boundary nodes per partition. Note that their partitioner does not necessarily yield connected partitions. On average each partition is adjacent to 5.2 neighboring ones. Our finer partitioning into 1 024 partitions has an edge cut of 25 715 with an average of 46.5 boundary nodes and 5.3 neighbors. All figures are based on 10 000 random but fixed queries, unless otherwise stated. To compare against the results of [13], we use the same quality parameter values. Minimum (detour based) local-optimality is set to $\alpha = 0.25$, maximum sharing to $\gamma = 0.8$, and maximum stretch to $\varepsilon = 0.25$.

We test the performance of our algorithm in terms of both efficiency and quality according to Definition 1.

5.2 Engineered Baseline Algorithm

We compare our engineered baseline algorithm, X-CHASEV, against X-BDV and X-CHV. The results of Table 1 report on the query performance and path quality of the engineered baseline algorithm. As described in Section 3 the engineered baseline algorithm is faster by a factor of two than the other algorithms. We reimplemented both X-BDV and X-CHV algorithms. A direct comparison against the numbers of Abraham et al. [13] is unfair, since the heuristics of the underlying CH are different. X-BDV has the highest success rate and, of course, the highest query times by several orders of magnitude. This makes X-BDV unsuitable for any practical setting in which speed is a factor. The success rates of all three algorithms

Table 1. Query performance of algorithms for alternatives $p = 1, 2, 3$.

p	algorithm	performance		path quality					
		time [ms]	success rate[%]	UBS[%]		sharing[%]		locality[%]	
				avg	max	avg	max	avg	min
1	X-BDV	11 451.5	94.5	9.4	52.5	42.7	79.9	77.0	26.2
	X-CHV	1.2	75.5	9.2	48.1	44.7	80.0	74.8	26.3
	X-CHASEV	0.5	75.5	9.2	48.1	44.7	80.0	74.8	26.3
2	X-BDV	12225.8	80.6	11.5	43.0	60.0	80.0	78.6	27.0
	X-CHV	1.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0
	X-CHASEV	0.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0
3	X-BDV	13330.9	59.5	13.2	52.9	68.1	80.0	76.2	25.9
	X-CHV	2.3	14.2	10.0	33.4	65.0	79.9	84.3	30.9
	X-CHASEV	1.0	14.2	10.0	33.4	65.0	79.9	84.3	30.9

drop with the number of alternatives. The average path quality measures are very similar for all algorithms and identical for X-CHV and X-CHASEV by design. This is expected behavior.

5.3 Preprocessed Candidate Sets

Table 2 reports on the performance of the preprocessing required for the single- and multi-level algorithms. Preprocessing is run in parallel for up to three alternatives with relaxation either off ($x = 0$) or set to $x = 3$. Row *multi-level* denotes the results of adding a finer partitioning compared to just the single-level approach. Numbers are listed for alternative $p = 1, 2, 3$ and only pertain to candidate sets of non-neighborings, non-equal pairs of partitions. We note that preprocessing can be done on server hardware in a few hours for all of the experiments. The relative speedup on 48 cores is only about 28 due to the memory-bandwidth bottleneck, which is about 60% of the perfect linear speedup. The space overhead is more or less neglectable. Even for relaxation with $x = 3$ and multi-level partitioning the amount of additionally data is less than 9 MiB. Multi-level preprocessing shows a higher average number of candidates per partition pair as only partition pairs close to each other are processed. Fewer candidate sets remain empty using the relaxed algorithm.

X-CHASEV without candidate sets is compared to single- and multi-level candidate sets. Table 3 gives basic performance numbers. Algorithms with preprocessed candidate sets have query times well below 0.5 ms on average even for the third alternative, which is more than practical. We see that the multi-level optimization even improves the success rate, while the path quality remains at high level. Fallback rates to the baseline are generally low, 95% of the queries are covered by preprocessed via node candidates. We tested on omitting the fallback entirely for this setting and observe that results do not degrade noticeably. A third partitioning level would not give any further improvements to the performance of the query.

Results of the 3-relaxed variant of the query are given in Table 4. Numbers for X-BDV and X-CHV are shown for reference. We omit path quality since it is virtually unaffected and remains high.

The success rate further improves especially for the second and third alternative. Using precomputed candidate sets is faster by an order of magnitude than X-CHASEV and naturally much faster than the original method. We identify two reasons. A) an expensive (relaxed) Exploration Dijkstra has to be done only in the rare case when a fallback is needed.

Table 2. Preprocessing results for normal ($x = 0$) and 3-relaxed ($x = 3$) algorithms.

	time	size	candidate sets					
			p=1		p=2		p=3	
x preprocessing	[h]	[kiB]	empty [%]	avg. size	empty [%]	avg. size	empty [%]	avg. size
0 single-level	1.1	859	2.6	4.4	12.7	5.1	30.5	4.4
multi-level	1.7	3 669	6.2	6.1	17.4	5.9	36.9	4.2
3 single-level	2.3	1 742	1.4	6.7	3.0	10.2	10.8	11.5
multi-level	4.3	8 909	1.1	12.2	4.9	15.0	11.6	14.2

Table 3. Query performance with preprocessed candidate sets.

p	algorithm	performance		path quality				candidate sets				
		time [ms]	success rate [%]	UBS[%]	sharing[%]	locality[%]	v.cand. [%]	fallb. [%]	avg. tested			
1	X-CHASEV	0.5	75.5	9.2	48.1	44.7	80.0	74.8	26.3	-	-	-
	single-level	0.1	80.7	9.8	48.1	48.5	80.0	75.8	26.3	92.4	4.9	1.9
	multi-level	0.1	81.2	9.9	48.1	48.6	80.0	75.8	26.3	96.5	0.6	2.0
2	X-CHASEV	0.7	40.2	10.1	39.7	59.1	80.0	79.7	27.0	-	-	-
	single-level	0.3	50.8	10.7	40.4	57.1	80.0	80.3	26.3	91.6	2.6	2.8
	multi-level	0.3	51.2	10.7	40.4	57.0	80.0	80.4	26.3	93.8	0.3	2.9
3	X-CHASEV	1.0	14.2	10.0	33.4	65.0	79.9	84.3	30.9	-	-	-
	single-level	0.4	24.8	10.7	41.0	59.9	79.9	82.5	27.9	88.7	1.1	3.8
	multi-level	0.4	25.0	10.7	41.0	59.8	79.9	82.6	27.9	89.7	0.1	3.8

B) the average number of nodes to be tested as via node candidates is small and always less than half a dozen. Our single- and multi-level approaches deliver consistently higher success rates than the (engineered) baseline with the more speedup the more relaxation is applied.

Figure 1 shows success rates with varying Dijkstra ranks to test performance for local and long range queries alike. Success rates (left) are consistently equal or better for our algorithms than for the baseline. With relaxation (right) the numbers get even closer to the rates of X-BDV. The difference is less than 10%. Success rates are compared to X-BDV as the quality “gold standard” even though its computation is prohibitively high.

6 Conclusion and Future Work

We introduced via node candidate sets. We showed their compact size, their efficient precomputation on large-scale networks and report one order of magnitude faster queries. We also show that success rates are higher than for previous algorithms with neglectable memory overhead. As a result of our extensive experimental evaluation, we conclude that Assumption 1 holds. There are a number of interesting directions for future work. We would like to explore the amount of preprocessing that is necessary to match the success rates of X-BDV. Also, we would like to use our method to generate alternative graphs similar to [18]. A challenging question is to extend alternative path computation to multiple via nodes. Combining the idea of *transit nodes* with via node candidates may be a great opportunity of future research. Instead of characterizing an alternative by a single via node, *via entrance nodes* for

Table 4. Query performance of multiple algorithms with 3-relaxation.

algorithm	p=1			p=2			p=3		
	time [ms]	success rate[%]	avg. tested	time [ms]	success rate[%]	avg. tested	time [ms]	success rate[%]	avg. tested
X-BDV	11 451.5	94.5	-	12 225.8	80.6	-	13 330.9	59.5	-
X-CHV	3.4	88.5	-	4.3	64.7	-	5.3	38.0	-
X-CHASEV	2.7	88.5	-	3.2	64.7	-	3.8	38.0	-
single-level	0.2	90.0	2.22	0.4	70.2	3.8	0.6	44.0	5.6
multi-level	0.1	90.0	2.3	0.3	70.4	4.0	0.5	44.2	5.8

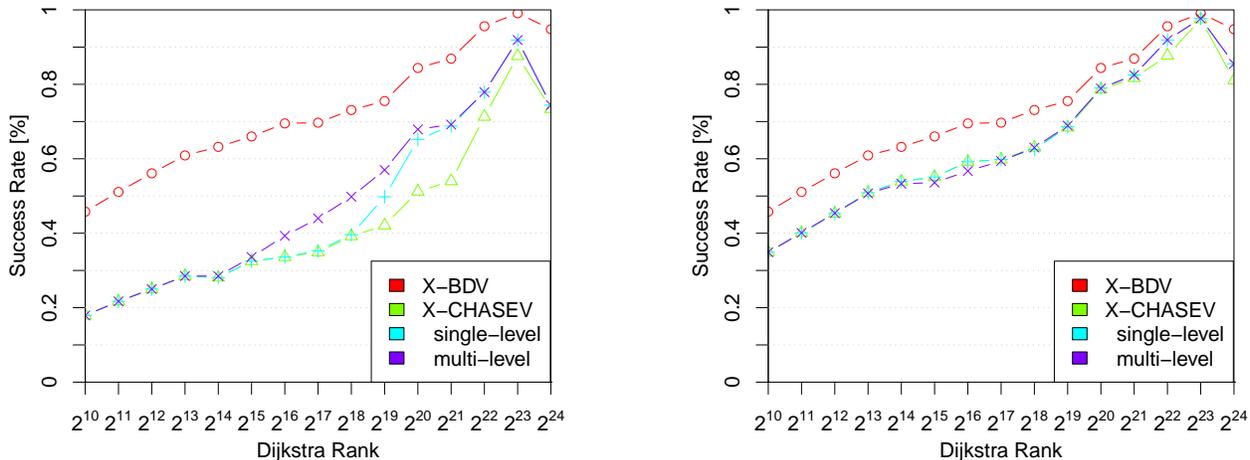


Fig. 1. Success rates according to Dijkstra rank: normal ($x = 0$, left) and 3-relaxed algorithm ($x = 3$, right). The Dijkstra rank of node v with respect to a node s is i if v is the i -th node removed from the priority queue of a unidirectional Dijkstra started at s . Each data point represents 1000 queries.

source and target partitions may provide access to an overlay network with fast lookups of alternatives.

7 Acknowledgments

The authors would like to thank Christian Schulz [16] for providing the partitionings and Daniel Delling [19] for providing arc flags for the partitioning with 128 cells, and Moritz Kobitzsch for great discussions.

References

1. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* **1** (1959) 269–271
2. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Transact. on Syst. Sci. and Cybernetics* **4** (1968)
3. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A* Search Meets Graph Theory. In: *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’05)*, SIAM (2005)
4. Lauther, U.: Slow preprocessing of graphs for extremely fast shortest path calculations. *Workshop on Computational Integer Programming at ZIB* (1997)
5. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation und Mobilität—von der Forschung zur praktischen Anwendung* **22** (2004) 219–230
6. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speedup dijkstra’s algorithm. *J. Exp. Algorithmics* **11** (2007)
7. Köhler, E., Möhring, R., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: *Experimental and Efficient Algorithms*. Volume 3503 of LNCS. Springer (2005)
8. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: *Proc. of the 7th Workshop on Experimental Algorithms (WEA’08)*. (2008)
9. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra’s Algorithm. *ACM Journ. of Exp. Algorithmics* **15** (2010) 1–31
10. Abraham, I., Fiat, A., Goldberg, A.V., Werneck, R.F.: Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In: *Proc. of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’10)*. (2010)
11. Abraham, I., Delling, D., Fiat, A., Goldberg, A.V., Werneck, R.F.: VC-Dimension and Shortest Path Algorithms. In: *Proc. of the 38th International Colloquium on Automata, Languages, and Programming (ICALP’11)*. (2011)
12. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A hub-based labeling algorithm for shortest paths in road networks. In: *SEA*. (2011) 230–241
13. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Alternative Routes in Road Networks. full paper. Online available at <http://88.198.59.15/delling/tmp/alternativesJEA.pdf> (2011)
14. Cambridge Vehicle Information Tech. Ltd: Choice Routing. (<http://camvit.com>)
15. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph Partitioning with Natural Cuts. In: *25th International Parallel and Distributed Processing Symposium (IPDPS’11)*, IEEE Computer Society (2011)
16. Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: *Proc. of the European Symposium on Algorithms*. Volume 6942 of LNCS. (2011)
17. Demetrescu, C., Goldberg, A.V., Johnson, D.S., eds.: *The 9th DIMACS Implementation Challenge – Shortest Paths*. American Mathematical Society (2006)
18. Bader, R., Dees, J., Geisberger, R., Sanders, P.: Alternative route graphs in road networks. In: *Theory and Practice of Algorithms in (Computer) Systems*. (2011)
19. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.: PHAST: Hardware-Accelerated Shortest Path Trees. In: *25th International Parallel and Distributed Processing Symposium (IPDPS’11)*, IEEE (2011)