Computing Many-to-Many Shortest Paths Using Highway Hierarchies*

Sebastian Knopp† Peter Sanders‡ Dominik Schultes‡ Frank Schulz†
Dorothea Wagner‡

Abstract

We present a fast algorithm for computing all shortest paths between source nodes $s \in S$ and target nodes $t \in T$. This problem is important as an initial step for many operations research problems (e.g., the vehicle routing problem), which require the distances between S and T as input. Our approach is based on highway hierarchies, which are also used for the currently fastest speedup techniques for shortest path queries in road networks. We show how to use highway hierarchies so that for example, a $10\,000 \times 10\,000$ distance table in the European road network can be computed in about one minute. These results are based on a simple basic idea, several refinements, and careful engineering of the approach. We also explain how the approach can be parallelized and how the computation can be restricted to computing only the k closest connections.

1 Introduction

Many logistics problems like tour planning, warehouse location or vehicle routing problems[1] require the knowledge of distances (and sometimes the actual paths) between all pairs of nodes $(s,t) \in S \times T$ for node sets $S, T \subseteq V$ in some road network $G = (V, E)$. For example, $S = T$ might be the nodes to be connected by a traveling salesman tour. In practice, fast heuristics are used for the logistics problems, and often the initial step to compute the distances takes more time than to solve the 'real' problem.

The underlying road networks are huge, e.g. currently around 20 million nodes for Western Europe or North America. Let us consider two obvious approaches to solve the problem using $|S| = |T| = 10\,000$ as an exemplary input size. We can find all required paths by doing $|S|$ single source shortest path computations. Using an efficient implementation of Dijkstra's algorithm, this approach (DIJKSTRA) would take about 10 000 times 10 s, i.e., about one day. For most applications, this is too slow. We also could use the query algorithm from [16] which can answer an s–t query in about one millisecond. However, making $10\,000 \times 10\,000$ queries would need about the same time—one day—as the naive algorithm for the instance under consideration. However, we will see that the problem can be solved very efficiently using an approach that is based on highway hierarchies. We review the basic concepts needed for this paper in Section 2 (see [16] for more details) and present our algorithm in Section 3. Implementation details are explained in Section 4. Our code solves the $10\,000 \times 10\,000$ problem mentioned above in 67 s—more than 1 000 times faster than either DIJKSTRA or 10 000[2] highway hierarchy queries.[2] This experiment and many more are presented in Section 5. Section 6 presents generalizations, parallelization, and incremental computation of shortest connections. A summary and further directions of research are found in Section 7.

Related Work. There are results that accelerate many-to-many shortest paths for rather dense graphs with $|E| \gg |V|$, e.g., [19]. However, we are not aware of specific results that would be useful for road networks (or any other kind of sparse graphs). In practice, it is very common to speed up the single-pair variant of Dijkstra's algorithm using goal-directed search [10] or bidirectional search. These techniques need no preprocessing and usually yield a speed-up factor of around 2. We adapted these techniques to the many-to-many case and observed speed-up factors up to 2 or 3 (depending on the type of input) compared to DIJKSTRA (cf. [11]). In this paper, however, we aim at speed-up factors that are orders of magnitude larger, allowing a certain amount of preprocessing. Of course, we can use any speedup technique for individual shortest path queries to perform $|S| \times |T|$ queries. But, as the above

*Partially supported by DFG grant SA 933/1-3 and by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP) under contract no. FP6-021235-2 (project ARRIVAL).

†PTV AG, Stumpfstr. 1, 76131 Karlsruhe, Germany, http://www.ptv.de, {Sebastian.Knopp, Frank.Schulz}@ptv.de

‡Universität Karlsruhe (TH), 76128 Karlsruhe, Germany, {sanders,schultes,dwagner}@ira.uka.de

[1] See [5] for information about the vehicle routing problem; Intertour [1] is a software package developed by PTV AG that solves such logistics problems.

[2] If not otherwise mentioned, execution times are given *without* the time for computing the highway hierarchies (about 15 minutes for the case above).

example has shown, we can do much better than this direct approach.

For geometric speedup techniques it is not clear how to adapt them efficiently to the multiple target case: Goal directed search with preprocessed information (e.g., [7, 13]) directs search towards *the* target.[3] Geometric containers [18] and edge flags [14, 12] explore edges only if they lead to *the* target.[4]

Reach based routing [9] prunes edges if they are sufficiently far away from both source *and target*. By pruning edges based on the supposedly closest target, reach based routing can search for several targets. However, this is a very conservative assumption and the involved bookkeeping is likely to be prohibitive unless $|T|$ is very small or highly clustered.

Besides highway hierarchies, there are several speedup techniques based on bidirected, non-goal-directed search. Our approach can be adapted to all of these techniques. The bidirectional 'self bounded' variant of reach based routing [6] only explores nodes or edges which appear in shortest paths "far enough" away from source or target. The *multi-level method* (e.g., [17, 3]) accelerates search by precomputing connections between nodes from graph separators. We have only implemented our approach with highway hierarchies because this technique is currently fastest both with respect to query time and preprocessing time.

Very recently, *transit node routing* [2] has accelerated shortest path queries by another two orders of magnitude. However, transit node routing needs considerably more preprocessing time and space. Furthermore *its* preprocessing *uses* our algorithm to precompute a huge distance table. Even if the information for transit node routing is available, our algorithm remains up to one order of magnitude faster for large distance tables.

2 Highway Hierarchies

The basic idea of the highway hierarchies approach is that outside some local areas around the source and the target node, only a subset of 'important' edges has to be considered in order to be able to find the shortest path. The concept of a *local area* is formalized by the definition of a neighborhood node set[5] $\mathcal{N}(v)$ for each node v. We then get a straightforward definition of a *highway network* of a graph $G = (V, E)$ that has the property that all shortest paths are preserved: an edge $(u, v) \in E$ belongs to the highway network iff there are nodes $s, t \in V$ such that the edge (u, v) appears in the canonical shortest path[6] $\langle s, \ldots, u, v, \ldots, t\rangle$ from s to t in G with the property that $v \notin \mathcal{N}(s)$ and $u \notin \mathcal{N}(t)$.

The size of a highway network (in terms of the number of nodes) can be considerably reduced by a contraction procedure: for each node v, we check a *bypassability criterion* that decides whether v should be *bypassed*—an operation that creates shortcut edges (u, w) representing paths of the form $\langle u, v, w\rangle$. The graph that is induced by the remaining nodes and enriched by the shortcut edges forms the *core* of the highway network. The bypassability criterion takes into account the degree of the node v and the number of shortcuts that would be created if v was bypassed. For details, we refer to [16].

A *highway hierarchy* of a graph G consists of several levels $G_0, G_1, G_2, \ldots, G_L$. Level 0 corresponds to the original graph G. Level 1 is obtained by computing the *highway network* of level 0, level 2 by computing the highway network of the core G'_1 of level 1 and so on.

Starting with the source node s as root, DIJKSTRA*'s algorithm* grows a *shortest path tree* that contains shortest paths from s to all other nodes. A node u that already belongs to the tree is said to be '*settled*': a shortest path from s to u has been found and the *shortest path distance* $d(s, u)$ is known. In the beginning, only s is settled. A node u that is adjacent to a settled node v is said to be '*reached*': a path from s to u (via v), which might not be the shortest one, has been found and a *tentative distance* from s is known. As long as there are nodes left that are reached but not settled, DIJKSTRA's algorithm picks the one with the smallest tentative distance and settles it. A *bidirectional* version of DIJKSTRA's algorithm can be used to find a shortest path from a given node s to a given node t. Two DIJKSTRA searches are executed in parallel: one searches from the source node s in the original graph; another searches from the target node t backwards, i.e., it searches in the *reverse graph* $\overleftarrow{G} = (V, \overleftarrow{E})$, $\overleftarrow{E} := \{(v, u) \mid (u, v) \in E\}$. When both search scopes meet, a shortest path from s to t has been found.

The *highway query algorithm* [16] performs a mod-

[3] In [11] we also show how the landmark approach from [7] can be turned—by using some of the query nodes as implicit landmarks—into a technique without preprocessing that achieves speed-up factors of up to 3 or 4 compared to the DIJKSTRA approach.

[4] Although this test could be generalized to test efficiently whether the edge leads to some target, this is not likely to be successful if the target nodes spread sufficiently widely.

[5] In [16], we give more details on the definition of neighborhoods. In particular, we distinguish between a forward and a backward neighborhood. However, in this context, we would like to slightly simplify the notation and concentrate on the concepts that are important to understand the subsequent sections.

[6] For each connected node pair (s, t), we select a unique *canonical shortest path* in such a way that each subpath of a canonical shortest path is canonical as well. For details, we refer to [15].

ified bidirectional DIJKSTRA search. We only describe the forward search. The backward search works analogously. The search is controlled by the current level ℓ of the search which in turn depends on the neighborhood sets of the nodes. A node s_ℓ is called an *entrance point* into level ℓ if it marks the point where the search switches to level ℓ. The next point s'_ℓ on a shortest path from s over s_ℓ that is in the *core* G'_ℓ of level ℓ is called an entrance point into the core of level ℓ (if s_ℓ itself belongs to the core of level ℓ, we have $s_\ell = s'_\ell$). Suppose the current search path has the form $\langle s, \ldots, s'_\ell, \ldots, u\rangle$ and we consider to relax an edge (u, v). If the node v is outside the neighborhood of s'_ℓ in level ℓ, then the search switches to level $\ell + 1$. There are two restrictions that are responsible for the accelerated search: Do not relax edges which are not in the current level of the highway hierarchy. Do not relax edges going from nodes in the core of level ℓ to bypassed nodes. Note that these rules have the disadvantage that the usual stopping criterion for bidirectional search does not work. We have to continue searching until the search radius of both directions reaches the length of the shortest path found so far.

3 The Algorithm

Without loss of generality we will assume $|T| \geq |S|$. Otherwise, it is more efficient to apply the algorithm below to the reverse graph. We will first only explain how to compute *distances* $d(s, t)$ for $(s, t) \in S \times T$. In Section 6.1 we outline how this can be generalized to computing actual shortest *paths*. Our starting point is the naive application of $|S| \times |T|$ queries using the highway hierarchy query algorithm from [16] as outlined in Section 2. We transform this approach step by step to a more efficient approach so that the preservation of correctness is evident.

Our first step is to introduce a tuning parameter K by using only levels $0, \ldots, K$ of the highway hierarchy. Since level K can have considerable size, it would be wasteful to search it from both directions. Rather, backward search never looks beyond entrance points to G'_K—the core of level K. This restriction is implemented by not relaxing the outgoing edges of an entrance point to the core of level K. Backward search relies on the forward search to explore enough of level K to meet it.[7]

The next and crucial idea is to execute both forward and backward searches only once. More precisely, we store the (fairly small) search spaces of the backward searches and when we perform a forward search, we access the stored information from the backward searches to emulate the behavior of $|S| \times |T|$ pairwise searches. Figure 1 illustrates our basic approach.

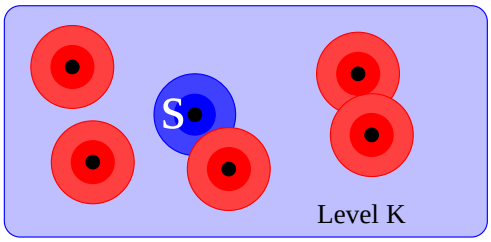

Figure 1: Schematic view of our asymmetric many-to-many algorithm. Forward search explores all of level K whereas the backward searches stop at entrance points to the core of level K. Note that 'well separated' search spaces only interact at these entrance points.

It remains to explain how the information gained during forward and backward searches can be brought together efficiently. We do this by associating a bucket $b(v)$ with each node of the graph. Bucket $b(v)$ stores a set of pairs (t, d) representing paths from v to $t \in T$ with length d encountered during the backward searches. During the forward searches, we maintain a two-dimensional array D of *tentative distances* for each source-destination pair. These distances are initialized to ∞ (or some sufficiently large value). When a forward search settles node v, it scans $b(v)$. For each pair (t, d) stored in $b(v)$, it sets $D[s, t]$ to $\min\{D[s, t], d(s, v) + d\}$.

3.1 Refinements.

Fewer Bucket Entries. We can speed up bucket scanning by reducing the number of bucket entries that are made during the backward searches. Our algorithm finds a shortest path P from $s \in S$ to $t \in T$ if there is at least one intermediate node $v \in P$ such that t is entered into bucket $b(v)$ during backward search from t and v is scanned during forward search from s. Every additional bucket entry with this property costs unnecessary extra scanning time. We can save such scans based on the observation that during a highway search the current search level can differ from the actual level of a node. Due to this fact, bucket entries at nodes in the core of level K are made while the search is still in a level $\ell < K$. Because every forward search settles all nodes in G'_K, a bucket entry $(t, d) \in b(v)$ can be omitted if it corresponds to a path of the form $(v, v', \ldots, t)$ where both v' and v are in the core of level K.

Accurate Backward Search. The reduction of bucket scans described in the previous paragraph

[7] An optimized version of the algorithm in [16] does something similar—*both* search directions stop searching at level K and the remaining distance is covered by a precomputed distance table between nodes in G'_K.

can be strengthened by performing accurate backward searches. The current version of backward search is not accurate because we break the search at entrance points to the core of the topmost level. To make them exact, backward searches are enlarged: We do not prune the search at core entrance points and continue until all nodes in the priority queue are in the core of level K. This method leads to fewer bucket entries, because the restriction of the previous paragraph applies more often.

3.2 Analysis. Since highway hierarchies do not give worst case performance guarantees that hold for arbitrary graphs, our analysis will be based on parameterizations and assumptions that still have to be checked experimentally. We nevertheless believe such an analysis to be valuable because it explains the behavior of the algorithm and helps choosing the tuning parameters.

Let $F(K)$ denote the average size of (forward and backward) search spaces below the core of level K. until the entrance points into the core of level K are found and let $f(K)$ denote the average number of bucket entries made by a backward search in the core of level K. If the value of K is clear from the context, we simply write F and f. Let Dijkstra(k) denote the cost of Dijkstra-search when exploring k nodes in a road network.

The backward searches have cost $|T| \cdot \text{Dijkstra}(F)$. Building buckets costs time $O(|T| \cdot F)$. The forward searches have cost about $|S| \cdot \text{Dijkstra}(F + |G'_K|)$ for the search itself where G'_K is the core of level K. To estimate the cost of scanning buckets during forward search, we first make the simplifying assumption that the search spaces of forward searches and backward searches only meet in the core of level K. Then only the buckets in entrance points to the core of level K need to be scanned and the total scanning cost is $O(|S| \cdot |T| \cdot f)$. We get a total cost of

$$|S| \cdot \text{Dijkstra}(F + |G'_K|) + |T| \cdot \text{Dijkstra}(F) + O(|S| \cdot |T| \cdot f).$$

If both sets S and T are large, the dominating term is $|S| \cdot |T| \cdot f$. From this we can learn several things. First, since the constant behind this term is very small, we can expect very good performance for large problems. Second, since $f(K)$ can be expected to grow[8] with the maximum search level K, we can actually save time by choosing K smaller than the maximum possible level.

It is also interesting to look at extreme cases. When $|S| = |T| = 1$, it is best to choose K as the highest level and we essentially get the ordinary highway hierarchy query algorithm (except that we do not stop the backward search early when source and target are close together). When $T = V$, it is best to choose $K = 0$ and we get the ordinary Dijkstra algorithm for (repeated) single source shortest path. In other words, our algorithm smoothly interpolates between the best algorithms for these extreme cases and promises considerable speedups in the middle where none of these other algorithms works very well.

Reducing K also reduces the bucket scans below level K that we have so far ignored. The experimental section will show that bucket scans only dominate cost for rather large inputs so that we could use random sampling to estimate the amount of overlap present in the input. This approach yields a more accurate cost model that has the potential to determine (near) optimal values for K.

4 Implementation

We have implemented our many-to-many query algorithm in C++. It uses the highway hierarchies computed with the code from [16] but the query is a largely independent implementation. During backward search, our implementation records the search spaces by just collecting triples (intermediate node, target index, distance) in a resizeable array. The target index is a number between 0 and $T - 1$ that can be used for direct addressing of the tentative distance array D. After all backward searches are completed, we use a variant of counting sort to group the triples by intermediate node. Afterwards, the now redundant intermediate node information is discarded and we end up with a representation of all buckets concatenated into a single array B holding pairs (target index, distance). Each node stores its bucket size and an offset into B indicating the beginning of its bucket.[9]

5 Experiments

We performed the experiments on two 64-bit machines with 8 GB and 16 GB of main memory, respectively, 1 MB L2 cache using one out of two AMD Opteron processors clocked at 2.6 GHz, running SUSE Linux 10.1. Our programs were compiled with the GNU C++ compiler version 3.4 using optimisation level 3.

We used commercial data for Western Europe with 18 029 721 nodes and 42 199 587 directed edges. Edge weights correspond to travel time estimates based on 13

[8] When K is so large that most backward searches explore most of level K this is no longer true. However, in that case we can also expect considerable overlaps in the search spaces below level K.

[9] This implementation brought a speedup of more than two compared to an initial attempt that dynamically filled a C++ `vector` attached to each node.

road categories. The conventions are the same as in [16]. Constructing the highway hierarchy for this instance takes about 15 minutes. For choosing S and T we use random instances of two types. Symmetric instances with $S = T$ and asymmetric ones with $|S| \cdot |T| = 3\,240\,000$. Node sets are chosen uniformly at random without replacement. We also have nine symmetric real world instances with $|S| = |T|$ in the range of 173–2 892 nodes stemming from vehicle routing problems. We also tried random symmetric instances on a commercial graph of North America with 18 741 705 nodes and 47 244 849 directed edges. The results are quite analogous to those for Europe. We refer to [11] for more details.

Figures 2 and 3 gives running times for different variants of our algorithm using a $20\,000 \times 20\,000$ symmetric instance and an asymmetric instance, respectively. We can see that reducing the number of bucket entries without changing the backward search is always helpful. Investing more into backwards search pays (only) for large symmetric instances and is highly counterproductive for asymmetric instances. From now on we stick to the variant *with* reducing the number of bucket entries but *without* more accurate backward search. This seems to be a good compromise that always improves on the basic variant.

Performance for random symmetric instances with $|S| = |T|$ between 100 and 20 000 with maximum level K between 5 and 9 is given in Figure 4. We see that $K = 7$ is always a good value. Only for very large inputs, $K = 6$ is somewhat better. The break even point is near $|S| = |T| = 6\,000$. Since the inputs from our applications are usually symmetric and not so big, we have decided not to implement an automatic algorithm for selecting the best value of K. It is interesting to compare this with the running time of alternative algorithms given in Figure 5. Over the entire range of input sizes, our algorithm outclasses both Dijkstra's algorithm and a naive highway hierarchy algorithm that performs $|S| \times |T|$ individual queries (Highway Hierarchies[2]).

Figure 6 shows the performance for asymmetric inputs with fixed $|S| \cdot |T|$. As to be expected, forward search and bucket scanning dominates for near symmetric instances while backward search dominates for large $|T|$. With decreasing $|S|$, the optimal level for K goes all the way down from seven to one. But even for $|S| = 5$ our algorithm with level 1 outperforms Dijkstra's algorithm. Figure 7 gives the total execution time for an even larger spectrum of size ratios and also includes the competing algorithms.

Of course, it is interesting whether our measurements with random data have any relation to the performance on real world inputs. Figure 8 compares our real world instances with random instances of the same size. An important difference in the inputs is that the real world data is clustered (mostly in some area the size of the Netherlands). Overall, the running times are fairly close together and never more than a factor 1.7 apart. Hence, using random data for measurements is not completely unreasonable. The main difference is that real world instances need considerably more time for bucket scanning. This is easily explained by the clustering since search spaces will more often overlap on lower levels of the search. Real world instances need noticeably less time for backward search and sorting although the search spaces are very similar in size (data not shown here). A possible explanation are cache effects. Subsequent backward searches from clustered nodes are more likely to find the graph data they need in cache.

The time for forward search is about the same for both, real world and random instance families. The reason is that our current implementation does not break forward search when all entrance points have been covered. Thus, it cannot profit from the clustering of the inputs. We plan to investigate the following modification of our query algorithm that improves the behavior exactly for such clustered inputs: A forward search from s can stop when its remaining search space is completely in the core of level K and when all nonempty buckets in the core of level K have been scanned. This condition can be checked efficiently and might give significant speedup if sources and targets are concentrated in a small part of the road network.

6 Generalizations

6.1 Outputting Paths. So far we have only described how to compute distances. We now describe how the algorithm can be modified so that it computes a data structure that allows output of an (s, t)-shortest path P (for $(s, t) \in S \times T$) in time $O(|P|)$.

First note, that any path in the highway hierarchy can be efficiently converted to a path in the input: Store the constituent edges (from the same level in the hierarchy) of each shortcut in a separate list. This leads to a linear increase in space consumption and allows efficient recursive conversion of a highway hierarchy edge into a path in the input graph. A more detailed explanation can be found in [4].

We explicitly store the search spaces of forward and backward searches in the highway hierarchy in the form of rooted trees.[10] For each query pair (s, t), the shortest

[10] Alternatively we could use buckets associated with each node in the highway hierarchy. But this is at least conceptually more complicated.

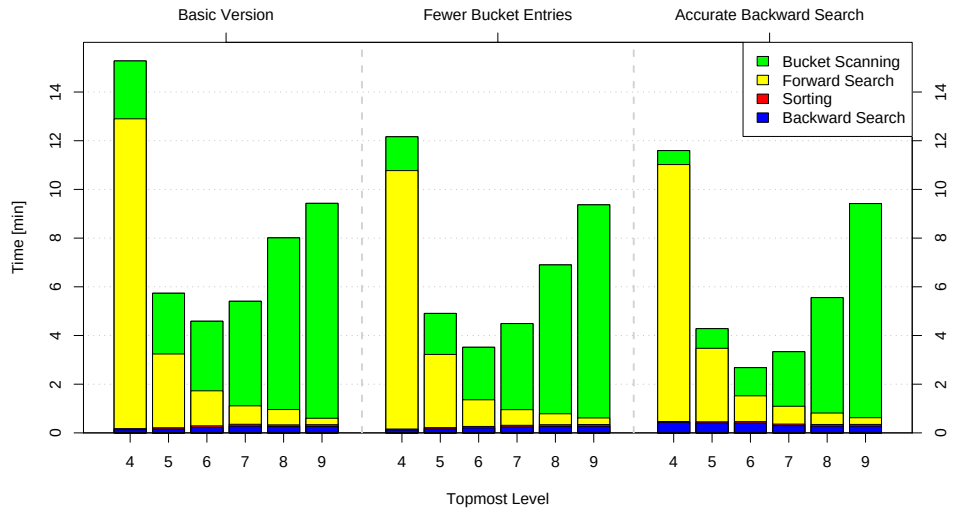

Figure 2: Performance of different algorithm variants for a large symmetric instance ($20\,000 \times 20\,000$). The numbers on the abscissa denote the highest level K that was used in the query algorithm.

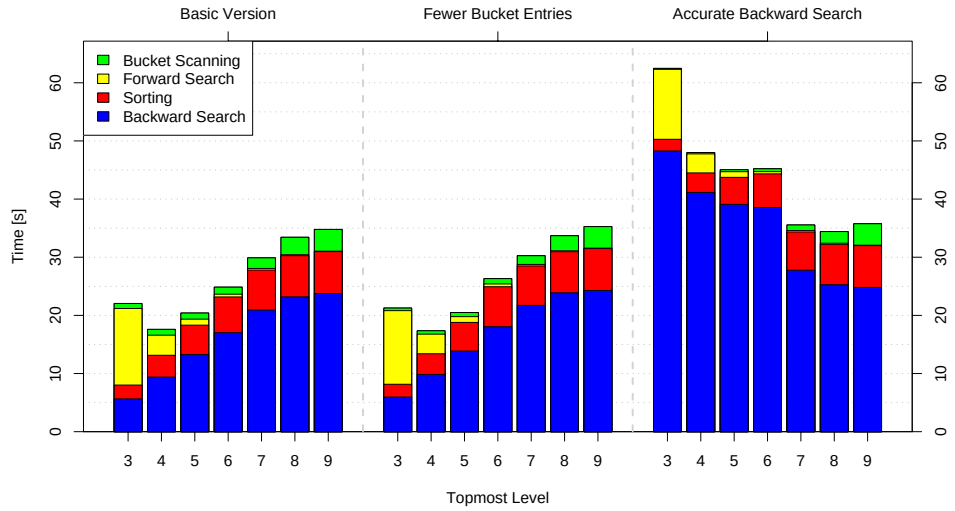

Figure 3: Performance of different algorithm variants for an asymmetric instance ($100 \times 32\,400$). The numbers on the abscissa denote the highest level K that was used in the query algorithm.

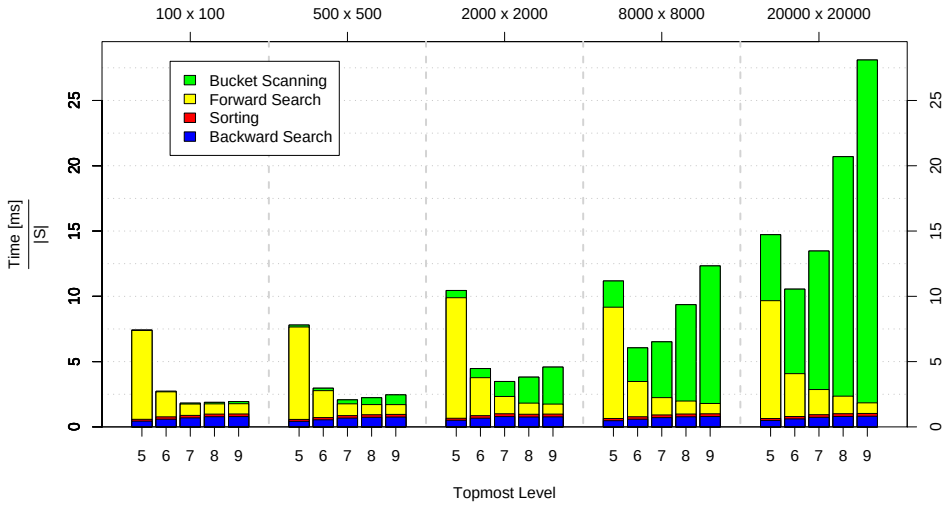

Figure 4: Different choices of maximum level K for symmetric instances.

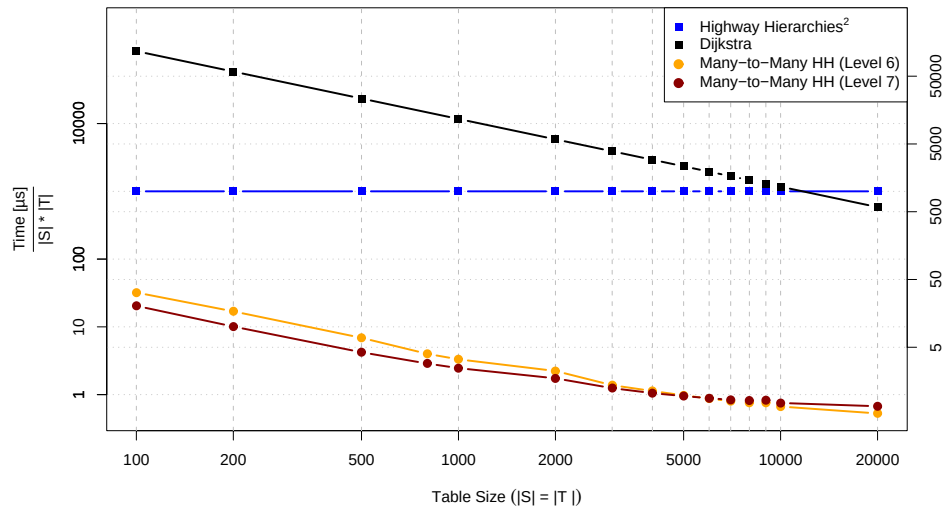

Figure 5: Comparison of our algorithm with alternative algorithms. The time per entry of the distance table for the Highway Hierarchies^2 algorithm is assumed to be 1 ms for all instances.

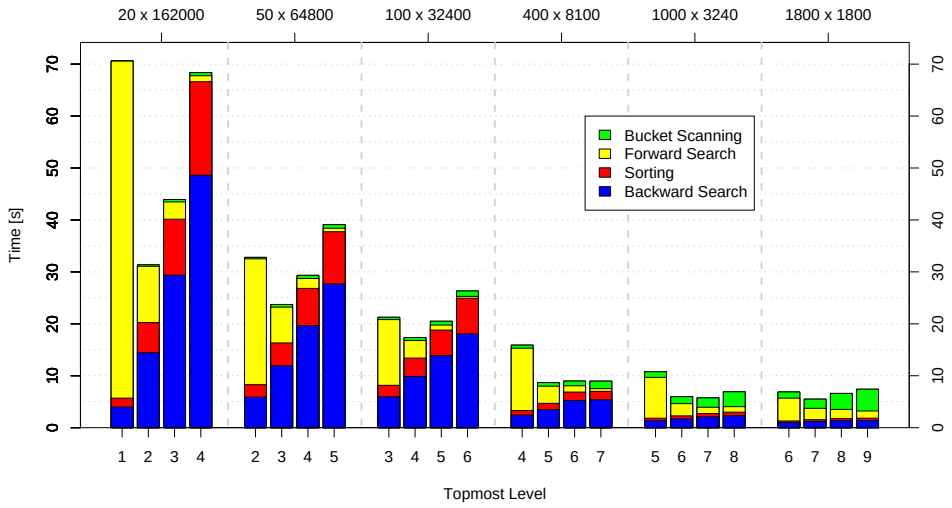

Figure 6: Performance for asymmetric instances at different choices of the maximum level K.

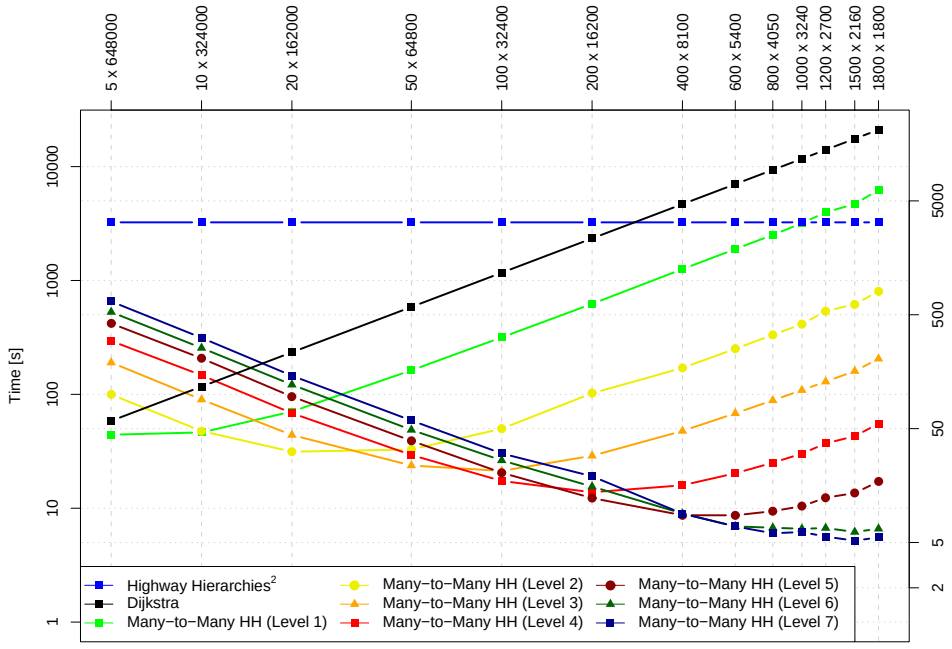

Figure 7: Comparison of algorithms and algorithm variants for asymmetric instances. Here, the time for the Highway Hierarchies^2 algorithm is constant because all distance tables have the same number of entries.

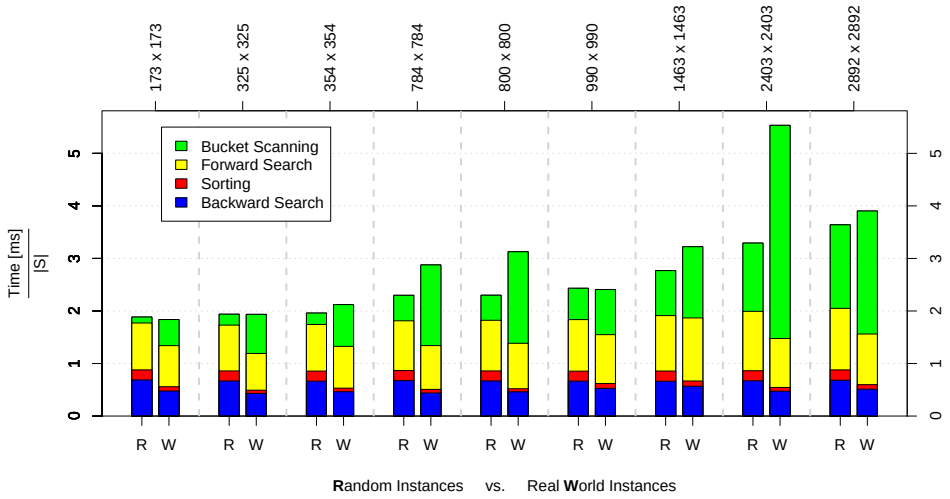

Figure 8: Comparison of **r**andom instances (R) and real **w**orld instances (W) of the same size.

path from s to t consists of a path s–v in the forward search space from s and a path v–t in the backwards search space to t. Hence, all we need to store are pointers to v in the two search spaces. This information is updated during the main computation whenever a better s–t path is encountered.

We can save some space by pruning those parts of the search spaces that are not needed for any shortest connection. At least for the forward search spaces, this pruning can be done incrementally whenever a forward search finishes.

6.2 Computing Shortest Connections Incrementally. In many applications we are not really interested in a complete distance table. For example, many heuristics for the traveling salesman problem start with the closest connections for each node and only compute additional connections on demand [8]. For such applications, the asymmetry in our search algorithm is again helpful. As before, the (small) backwards search is done for all $t \in T$ until all entrance points to level K are encountered. The (large) forward searches that require heavy scanning of buckets are only progressing incrementally after their search frontier is completely in the core of level K.

To do this we remember the number of entrance points to level K encountered by each backward search. Each forward search is equipped with a copy of this counter array. When the forward search encounters an entrance point to level K and scans a bucket entry (t, d) it decrements the counter for t. When the counter reaches zero, $D[s,t] = d(s,t)$ and we can output the newly found distance.

6.3 Parallelization. Suppose we have a shared memory parallel computer with P processing elements (PEs). Then the problem is easy to parallelize—each PE performs $\lceil |S|/P \rceil$ forward searches and $\lceil |T|/P \rceil$ backward searches. If $P > |S|$ we can achieve further parallelism by partitioning $|T|$ and the corresponding buckets into k groups. Now P/k processors are assigned to each group and perform a forward search from all nodes in $|S|$ considering only the target nodes in their group. The algorithm can even be adapted to a distributed memory machine as long as all of level K and the search space in levels $0..K-1$ for one node at a time fits into the local memory of a processor.

6.4 Precomputed Cluster Distances. Suppose a network has been partitioned into clusters $V_1 \dot\cup \cdots \dot\cup V_k$. In [13] it is shown how the cluster distances $d(V_i, V_j) := \min_{s \in V_i, t \in V_j} d(s,t)$ can be used to make shortest path search goal directed. For large networks this method achieves higher speedups and needs less memory than the successful landmark A^* method [7, 13]. Using a variation of our algorithm, we can now compute cluster distances quickly: add new level 0 nodes s_i and t_j for each cluster that are connected to the border nodes of a cluster by weight zero edges. Solve the many-to-many problem for $S = \{s_1, \ldots, s_k\}$ and $T = \{t_1, \ldots, t_k\}$. There is no need to recompute the highway hierarchy, since the new nodes would be contracted away in the initial contraction phase anyway.

7 Conclusions

We have presented a simple approach that efficiently solves the many-to-many routing problem in road net-

works. The basic idea—storing backwards search spaces in buckets—works with any speedup technique based on non-goal-directed bidirectional search.

Therefore, it is likely that our approach will work well in any other graph where such a speedup technique may prove useful in the future. For example, experiments in [2] indicate that the approach also works if we optimize for travel distances instead of travel times. If the tables are big enough (something like table size $> 100 \times 100$) our algorithm even beats Dijkstra's algorithm if computing the highway hierarchy is considered to be part of our task. If this is necessary, it might be interesting however to find ways to compute a hierarchy specially tailored to S and T—after all we only need to preserve shortest paths between nodes in S and T. The hope would be that this can be done more efficiently than building a complete highway hierarchy.

Acknowledgments. We would like to thank an anonymous referee for numerous constructive comments and suggestions.

References

[1] PTV Intertour. `http://www.english.ptv.de/cgi-bin/logistics/log_it.pl`.

[2] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Workshop on Algorithm Engineering and Experiments*, 2007.

[3] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-performance multi-level graphs. In *9th DIMACS Implementation Challenge – Shortest Paths*, 2006. `http://www.dis.uniroma1.it/~challenge9/`.

[4] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway hierarchies star. In *9th DIMACS Implementation Challenge – Shortest Paths*, 2006. `http://www.dis.uniroma1.it/~challenge9/`.

[5] B. D. Díaz. The VRP Web, 2006. `http://neo.lcc.uma.es/radi-aeb/WebVRP/`.

[6] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A^*: Efficient point-to-point shortest path algorithms. In *Workshop on Algorithm Engineering & Experiments*, Miami, 2006.

[7] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* meets graph theory. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[8] G. Gutin and A. Punnen, editors. *The Traveling Salesman Problem and its Variations*. Kluwer, 2002.

[9] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.

[10] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4(2):100–107, 1968.

[11] S. Knopp. Efficient computation of many-to-many shortest paths, 2006. Diplomarbeit, Universität Karlsruhe.

[12] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Münster GI-Days*, 2004.

[13] J. Maue, P. Sanders, and D. Matijevic. Goal directed shortest path queries using Precomputed Cluster Distances. In *5th Workshop on Experimental Algorithms (WEA)*, number 4007 in LNCS, pages 316–328. Springer, 2006.

[14] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. In *4th International Workshop on Efficient and Experimental Algorithms*, 2005.

[15] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

[16] P. Sanders and D. Schultes. Engineering highway hierarchies. In *14th European Symposium on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 804–816, 2006.

[17] F. Schulz, D. Wagner, and C. D. Zaroliagis. Using multi-level graphs for timetable information. In *4th Workshop on Algorithm Engineering and Experiments*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

[18] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *11th European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.

[19] I-L. Wang. *Shortest Paths and Multicommodity Network Flows*. PhD thesis, Georgia Inst. Tech., 2003. `http://ilin.iim.ncku.edu.tw/ilin/phdthesis/ilinth2_all.pdf`.