# Combining Speed-Up Techniques for Shortest-Path Computations*

Martin Holzer, Frank Schulz, and Thomas Willhalm

Universität Karlsruhe, Fakultät für Informatik, Postfach 6980, 76128 Karlsruhe, Germany.
{mholzer,fschulz,willhalm}@ira.uka.de

**Abstract.** Computing a shortest path from one node to another in a directed graph is a very common task in practice. This problem is classically solved by Dijkstra's algorithm. Many techniques are known to speed up this algorithm heuristically, while optimality of the solution can still be guaranteed. In most studies, such techniques are considered individually. The focus of our work is the *combination* of speed-up techniques for Dijkstra's algorithm. We consider all possible combinations of four known techniques, namely *goal-directed search*, *bi-directed search*, *multi-level approach*, and *shortest-path bounding boxes*, and show how these can be implemented. In an extensive experimental study we compare the performance of different combinations and analyze how the techniques harmonize when applied jointly. Several real-world graphs from road maps and public transport and two types of generated random graphs are taken into account.

## 1 Introduction

We consider the problem of (repetitively) finding single-source single-target shortest paths in large, sparse graphs. Typical applications of this problem include route planning systems for cars, bikes, and hikers [1,2] or scheduled vehicles like trains and buses [3,4], spatial databases [5], and web searching [6]. Besides the classical algorithm by Dijkstra [7], with a worst-case running time of $\mathcal{O}(m + n \log n)$ using Fibonacci heaps [8], there are many recent algorithms that solve variants and special cases of the shortest-path problem with better running time (worst-case or average-case; see [9] for an experimental comparison, [10] for a survey and some more recent work [11,12,13]).

It is common practice to improve the running time of Dijkstra's algorithm heuristically while correctness of the solution is still provable, i.e., it is guaranteed that a shortest path is returned but not that the modified algorithm is faster. In particular, we consider the following four speed-up techniques:

* This work was partially supported by the Human Potential Programme of the European Union under contract no. HPRN-CT-1999-00104 (AMORE) and by the DFG under grant WA 654/12-1.

C.C. Ribeiro and S.L. Martins (Eds.): WEA 2004, LNCS 3059, pp. 269–284, 2004.
© Springer-Verlag Berlin Heidelberg 2004

**Goal-Directed Search** modifies the given edge weights to favor edges leading towards the target node [14,15]. With graphs from timetable information, a speed-up in running time of a factor of roughly 1.5 is reported in [16].

**Bi-Directed Search** starts a second search backwards, from the target to the source (see [17], Section 4.5). Both searches stop when their search horizons meet. Experiments in [18] showed that the search space can be reduced by a factor of 2, and in [19] it was shown that combinations with the goal-directed search can be beneficial.

**Multi-Level Approach** takes advantage of hierarchical coarsenings of the given graph, where additional edges have to be computed. They can be regarded as distributed to multiple levels. Depending on the given query, only a small fraction of these edges has to be considered to find a shortest path. Using this technique, speed-up factors of more than 3.5 have been observed for road map and public transport graphs [20]. Timetable information queries could be improved by a factor of 11 (see [21]), and also in [22] good improvements for road maps are reported.

**Shortest-Path Bounding Boxes** provide a necessary condition for each edge, if it has to be respected in the search. More precisely, the bounding box of all nodes that can be reached on a shortest path using this edge is given. Speed-up factors in the range between 10 and 20 can be achieved [23].

Goal-directed search and shortest-path bounding boxes are only applicable if a layout of the graph is provided. Multi-level approach and shortest-path bounding boxes both require a preprocessing, calculating additional edges and bounding boxes, respectively. All these four techniques are tailored to Dijkstra's algorithm. They crucially depend on the fact that Dijkstra's algorithm is label-setting and that it can be terminated when the destination node is settled.

The focus of this paper is the *combination* of the four speed-up techniques. We first show that, with more or less effort, all $2^4 = 16$ combinations can be implemented. Then, an extensive experimental study of their performance is provided. Benchmarks were run on several real-world and generated graphs, where operation counts as well as CPU time were measured.

The next section contains, after some definitions, a description of the speed-up techniques and shows how to combine them. Section 3 presents the experimental setup and data sets for our statistics, and the belonging results are given in Section 4. Section 5, finally, gives some conclusions.

## 2 Definitions and Problem Description

### 2.1 Definitions

A directed simple *graph* $G$ is a pair $(V, E)$, where $V$ is the set of nodes and $E \subseteq V \times V$ the set of edges in $G$. Throughout this paper, the number of nodes, $|V|$, is denoted by $n$ and the number of edges, $|E|$, by $m$.

A *path* in $G$ is a sequence of nodes $u_1, \dots, u_k$ such that $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. Given non-negative edge lengths $l : E \to \mathbb{R}_0^+$, the *length* of a path

$u_1, \ldots, u_k$ is the sum of weights of its edges, $\sum_{i=1}^{k-1} l(u_i, u_{i+1})$. The *(single-source single-target) shortest-path problem* consists of finding a path of minimum length from a given source $s \in V$ to a target $t \in V$.

A graph *layout* is a mapping $L : V \to \mathbb{R}^2$ of the graph's nodes to the Euclidean plane. For ease of notation, we will identify a node $v \in V$ with its location $L(v)$ in the plane. The Euclidean distance between two nodes $u, v \in V$ is then denoted by $d(u, v)$.

### 2.2 Speed-Up Techniques

Our base algorithm is Dijkstra's algorithm using Fibonacci heaps as priority queue. In this section, we provide a short description of the four speed-up techniques, whose combinations are discussed in the next section.

**Goal-Directed Search.** This technique uses a potential function on the node set. The edge lengths are modified in order to direct the graph search towards the target. Let $\lambda$ be such a potential function and $l(e)$ be the length of $e$. The new length of an edge $(v, w)$ is defined to be $\bar{l}(v, w) := l(v, w) - \lambda(v) + \lambda(w)$. The potential must fulfill the condition that for each edge $e$, its new edge length $\bar{l}(e)$ is non-negative, in order to guarantee optimal solutions.

In case edge lengths are Euclidean distances, the Euclidean distance $d(u, t)$ of a node $u$ to the target $t$ is a valid potential, due to the triangular inequality. Otherwise, a potential function can be defined as follows: let $v_{max}$ denote the maximum "edge-speed" $d(u, v)/l(e)$, over all edges $e = (u, v)$. The potential of a node $u$ can now be defined as $\lambda(u) = d(u, t)/v_{max}$.

**Bi-Directed Search.** The bi-directed search simultaneously applies the "normal", or forward, variant of the algorithm, starting at the source node, and a so-called reverse, or backward, variant of Dijkstra's algorithm, starting at the destination node. With the reverse variant, the algorithm is applied to the reverse graph, i.e., a graph with the same node set $V$ as that of the original graph, and the reverse edge set $\overline{E} = \{(u, v) \mid (v, u) \in E\}$. Let $d_f(u)$ be the distance labels of the forward search and $d_b(u)$ the labels of the backward search, respectively. The algorithm can be terminated when one node has been designated to be permanent by both the forward and the reverse algorithm. Then the shortest path is determined by the node $u$ with minimum value $d_f(u) + d_b(u)$ and can be composed of the one from the start node to $u$, found by the forward search, and the edges reverted again on the path from the destination to $u$, found by the reverse search.

**Multi-Level Approach.** This speed-up technique requires a preprocessing step at which the input graph $G = (V, E)$ is decomposed into $l + 1$ $(l \geq 1)$ levels and enriched with additional edges representing shortest paths between certain nodes. This decomposition depends on subsets $S_i$ of the graph's node set for

each level, called selected nodes at level $i$: $S_0 := V \supseteq S_1 \supseteq \ldots \supseteq S_l$. These node sets can be determined on diverse criteria; with our implementation, they consist of the desired numbers of nodes with highest degree in the graph, which has turned out to be an appropriate criterion [20].

There are three different types of edges being added to the graph: *upward edges*, going from a node that is not selected at one level to a node selected at that level, *downward edges*, going from selected to non-selected nodes, and *level edges*, passing between selected nodes at one level. The weight of such an edge is assigned the length of a shortest path between the end-nodes.

To find a shortest path between two nodes, then, it suffices for Dijkstra's algorithm to consider a relatively small subgraph of the "multi-level graph" (a certain set of upward and of downward edges and a set of level edges passing at a maximal level that has to be taken into account for the given source and target nodes).

**Shortest-Path Bounding Boxes.** This speed-up technique requires a preprocessing computing all shortest path trees. For each edge $e \in E$, we compute the set $S(e)$ of those nodes to which a shortest path starts with edge $e$. Using a given layout, we then store for each edge $e \in E$ the bounding box of $S(e)$ in an associative array $BB$ with index set $E$.

It is then sufficient to perform Dijkstra's algorithm on the subgraph induced by the edges $e \in E$ with the target node included in $BB[e]$. This subgraph can be determined on the fly, by excluding all other edges in the search. (One can think of bounding boxes as traffic signs which characterize the region that they lead to.)

A variation of this technique has been introduced in [16], where as geometric objects angular sectors instead of bounding boxes were used, for application to a timetable information system. An extensive study in [23] showed that bounding boxes are the fastest geometric objects in terms of running time, and competitive with much more complex geometric objects in terms of visited nodes.

### 2.3 Combining the Speed-Up Techniques

In this section, we enlist for every pair of speed-up techniques how we combined them. The extension to a combination of three or four techniques is straight forward, once the problem of combining two of them is solved.

**Goal-Directed Search and Bi-Directed Search.** Combining goal-directed and bi-directed search is not as obvious as it may seem at first glance. [18] provides a counter-example for the fact that simple application of a goal-directed search forward and backward yields a wrong termination condition. However, the alternative condition proposed there has been shown in [19] to be quite inefficient, as the search in each direction almost reaches the source of the other direction. This often results in a slower algorithm.

To overcome these deficiencies, we simply use the very same edge weights $\bar{l}(v,w) := l(v,w) - \lambda(v) + \lambda(w)$ for both the forward and the backward search. With these weights, the forward search is directed to the target $t$ and the backward search has no preferred direction, but favors edges that are directed towards $t$. This should be (and indeed is) faster than each of the two speed-up techniques. This combination computes a shortest path, because a shortest $s$-$t$-path is the same for given edge weights $l$ and edge weights modified according to goal-directed search, $\bar{l}$.

**Goal-Directed Search and Multi-Level Approach.** As described in Section 2.2, the multi-level approach basically determines for each query a subgraph of the multi-level graph, on which Dijkstra's algorithm is run to compute a shortest path. The computation of this subgraph does not involve edge lengths and thus goal-directed search can be simply performed on it.

**Goal-Directed Search and Shortest-Path Bounding Boxes.** Similar to the multi-level approach, the shortest-path bounding boxes approach determines for a given query a subgraph of the original graph. Again, edge lengths are irrelevant for the computation of the subgraph and goal-directed search can be applied offhand.

**Bi-Directed Search and Multi-Level Approach.** Basically, bi-directed search can be applied to the subgraph defined by the multi-level approach. In our implementation, that subgraph is computed on the fly during Dijkstra's algorithm: for each node considered, the set of necessary outgoing edges is determined. If applying bi-directed search to the multi-level subgraph, a symmetric, backward version of the subgraph computation has to be implemented: for each node considered in the backward search, the incoming edges that are part of the subgraph have to be determined.

**Bi-Directed Search and Shortest-Path Bounding Boxes.** In order to take advantage of shortest-path bounding boxes in both directions of a bi-directional search, a second set of bounding boxes is needed. For each edge $e \in E$, we compute the set $S_b(e)$ of those nodes from which a shortest path ending with $e$ exists. We store for each edge $e \in E$ the bounding box of $S_b(e)$ in an associative array $BB_b$ with index set $E$. The forward search checks whether the target is contained $BB(e)$, the backward search, whether the source is in $BB_b(e)$.

**Multi-Level Approach and Shortest-Path Bounding Boxes.** The multi-level approach enriches a given graph with additional edges. Each new edge $(u_1, u_k)$ represents a shortest path $(u_1, u_2, \ldots, u_k)$ in $G$. We annotate such a new edge $(u_1, u_k)$ with $BB(u_1, u_2)$, the associated bounding box of the first edge on this path.

**Table 1.** Number of nodes and edges for all test graphs

| | street | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| n | 1444 | 3045 | 16471 | 20466 | 25982 | 38823 | 45852 | 45073 | 51510 | 79456 |
| m | 3060 | 7310 | 34530 | 42288 | 57620 | 79988 | 98098 | 91314 | 110676 | 172374 |
| | public transport | | | | | | | | | |
| n | 409 | 705 | 1660 | 2279 | 2399 | 4598 | 6884 | 10815 | 12070 | 14335 |
| m | 1215 | 1681 | 4327 | 6015 | 8008 | 14937 | 18601 | 29351 | 33728 | 39887 |
| | planar | | | | | | | | | |
| n | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
| m | 5000 | 10000 | 15000 | 20000 | 25000 | 30000 | 35000 | 40000 | 45000 | 50000 |
| | waxman | | | | | | | | | |
| n | 938 | 1974 | 2951 | 3938 | 4949 | 5946 | 6943 | 7917 | 8882 | 9906 |
| m | 4070 | 9504 | 14506 | 19658 | 24474 | 29648 | 34764 | 39138 | 44208 | 48730 |

## 3 Experimental Setup

In this section, we provide details on the input data used, consisting of real-world and randomly generated graphs, and on the execution of the experiments.

### 3.1 Data

**Real-World Graphs.** In our experiments we included a set of graphs that stem from real applications. As in other experimental work, it turned out that using realistic data is quite important as the performance of the algorithms strongly depends on the characteristics of the data.

**Street Graphs.** Our street graphs are street networks of US cities and their surroundings. These graphs are bi-directed, and edge lengths are Euclidean distances. The graphs are fairly large and very sparse because bends are represented by polygonal lines. (With such a representation of a street network, it is possible to efficiently find the nearest point in a street by a point-to-point search.)

**Public Transport Graphs.** A public transport graph represents a network of trains, buses, and other scheduled vehicles. The nodes of such a graph correspond to stations or stops, and there exists an edge between two nodes if there is a non-stop connection between the respective stations. The weight of an edge is the average travel time of all vehicles that contribute to this edge. In particular, the edge lengths are not Euclidean distances in this set of graphs.

**Random Graphs.** We generated two sets of random graphs that have an estimated average out-degree of 2.5 (which corresponds to the average degree in the real-world graphs). Each set consists of ten connected, bi-directed graphs with (approximately) $1000 \cdot i$ nodes ($i = 1, \dots, 10$).

**Random Planar Graphs.** For the construction of random planar graphs, we used a generator provided by LEDA [24]. A given number of $n$ nodes are uniformly distributed in a square with a lateral length of 1, and a triangulation of the nodes is computed. This yields a complete undirected planar graph. Finally, edges are deleted at random until the graph contains $2.5 \cdot n$ edges, and each of these is replaced by two directed edges, one in either direction.

**Random Waxman Graphs.** The construction of these graphs is based on a random graph model introduced by Waxman [25]. Input parameters are the number of nodes $n$ and two positive rational numbers $\alpha$ and $\beta$. The nodes are again uniformly distributed in a square of a lateral length of 1, and the probability that an edge $(u, v)$ exists is $\beta \cdot \exp(-d(u, v)/(\sqrt{2}\alpha))$. Higher $\beta$ values increase the edge density, while smaller $\alpha$ values increase the density of short edges in relation to long edges. To ensure connectedness and bi-directedness of the graphs, all nodes that do not belong to the largest connected component are deleted (thus, slightly less than $n$ nodes remain) and the graph is bi-directed by insertion of missing reverse edges. We set $\alpha = 0.01$ and empirically determined that setting $\beta = 2.5 \cdot 1620/n$ yields an average degree of 2.5, as wished.

### 3.2 Experiments

We have implemented all combinations of speed-up techniques as described in Sections 2.2 and 2.3 in C++, using the graph and Fibonacci heap data structures of the LEDA library [24] (version 4.4). The code was compiled with the GNU compiler (version 3.3), and experiments were run on an Intel Xeon machine with 2.6 GHz and 2 GB of memory, running Linux (kernel version 2.4).

For each graph and combination, we computed for a set of queries shortest paths, measuring two types of *performance*: the mean values of the *running times* (CPU time in seconds) and the *number of nodes* inserted in the priority queue. The queries were chosen at random and the amount of them was determined such that statistical relevance can be guaranteed (see also [23]).

## 4 Experimental Results

The outcome of the experimental study is shown in Figures 1–4. Further diagrams that we used for our analysis are depicted in Figures 5–10. Each combination is referred to by a 4-tuple of shortcuts: `go` (goal-directed), `bi` (bi-directed), `ml` (multi-level), `bb` (bounding box), and `xx` if the respective technique is not used (e.g., `go-bi-xx-bb`). In all figures, the graphs are ordered by size, as listed in Table 1.

We calculated two different values denoting relative *speed-up*: on the one hand, Figures 1–4 show the speed-up that we achieved compared to plain Dijkstra, i.e., for each combination of techniques the ratio of the performance of plain Dijkstra and the performance of Dijkstra with the specific combination of

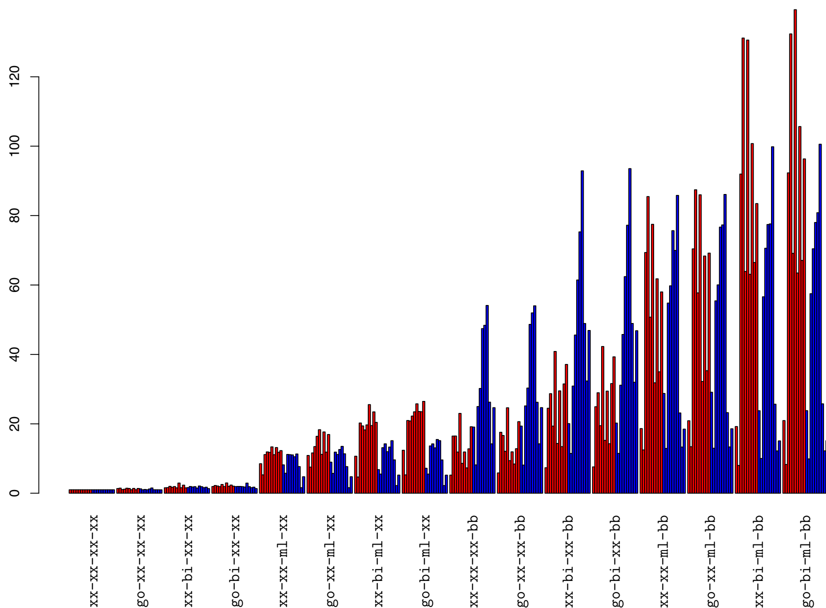


**Fig. 1.** Speed-up relative to Dijkstra's algorithm in terms of visited nodes for real-world graphs (in this order: street graphs in red and public transport graphs in blue)

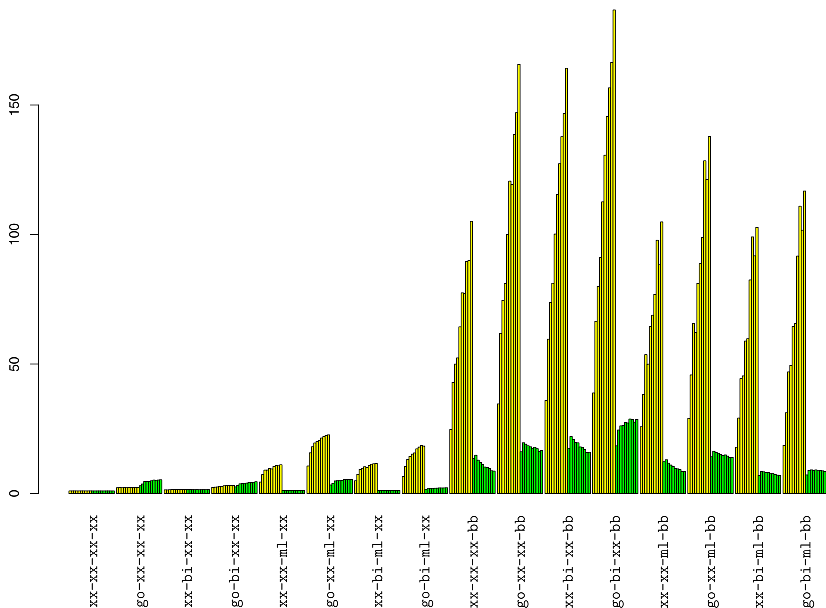


**Fig. 2.** Speed-up relative to Dijkstra's algorithm in terms of visited nodes for generated graphs (in this order: random planar graphs in yellow and random Waxman graphs in green)

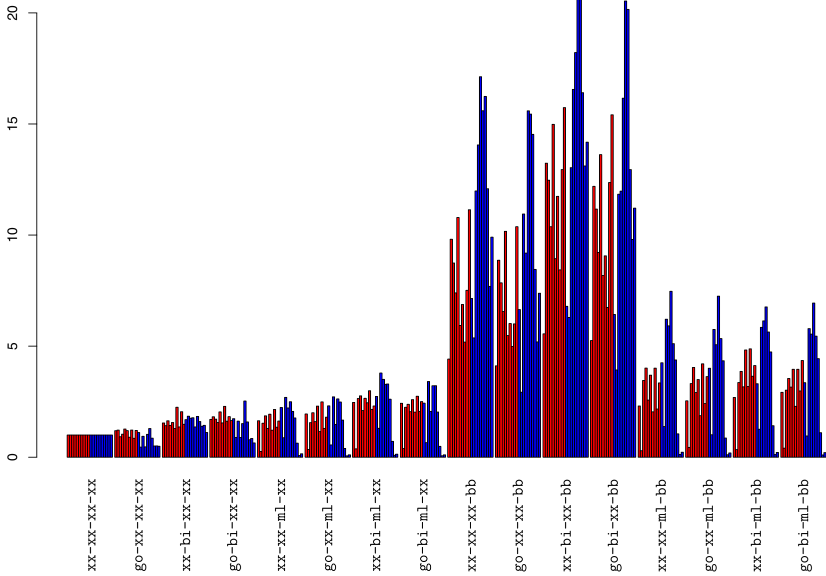


**Fig. 3.** Speed-up relative to Dijkstra's algorithm in terms of running time for real-world graphs (in this order: street graphs in red and public transport graphs in blue)

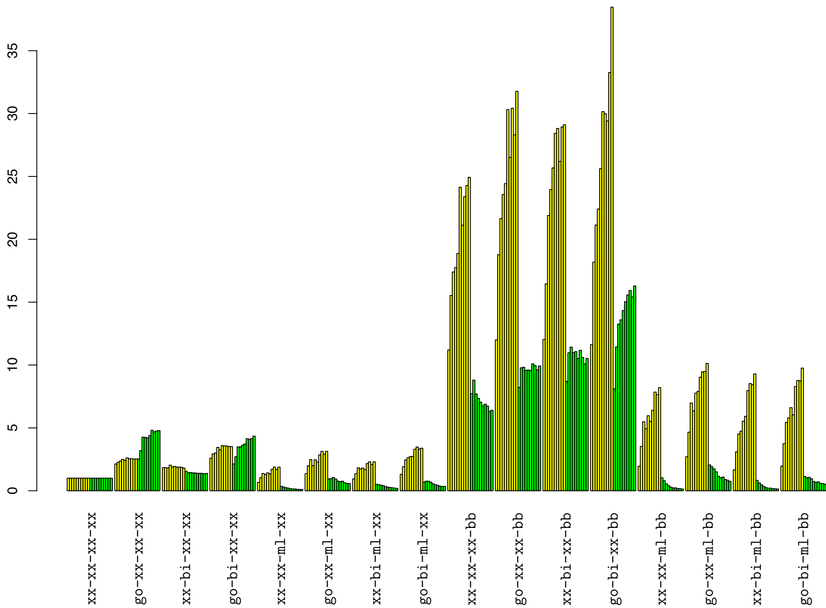


**Fig. 4.** Speed-up relative to Dijkstra's algorithm in terms of running time for generated graphs (in this order: random planar graphs in yellow and random Waxman graphs in green)

techniques applied. There are separate figures for real-world and random graphs, for the number of nodes and running time, respectively.

On the other hand, for each of the Figures 5–8, we focus on one technique $\mathcal{T}$ and show for each combination containing $\mathcal{T}$ the speed-up that can be achieved compared to the combination without $\mathcal{T}$. (Because of lack of space only figures dealing with the number of visited nodes are depicted.) For example, when focusing on bi-directed search and considering the combination `go-bi-xx-bb`, say, we investigate by which factor the performance gets better when the combination `go-bi-xx-bb` is used instead of `go-xx-xx-bb` only.

In the following, we discuss, for each technique separately, how combinations with the specific technique behave, and then turn to the relation of the two performance parameters measured, the number of visited nodes and running time: we define the *overhead* of a combination of techniques to be the ratio of running time and the number of visited nodes. In other words, the overhead reflects the time spent per node.

### 4.1 Speed-Up of the Combinations

**Goal-Directed Search.** Individually comparing goal-directed search with plain Dijkstra (Figure 5), speed-up varies a lot between the different types of graphs: Considering the random graphs, we get a speed-up of about 2 for planar graphs but of up to 5 for the Waxman graphs, which is quite surprising. Only little speed-up, of less than 2, can be observed for the real-world graphs.

Concerning the number of visited nodes, adding goal-directed search to the multi-level approach is slightly worse than adding it to plain Dijkstra and with bi-directed search, we get another slight deterioration. Adding it to bounding boxes (and combinations including bounding boxes) is hardly beneficial.

For real-world graphs, adding goal-directed search to any combination does not improve the running time. For generated graphs, however, running time decreases. In particular, it is advantageous to add it to a combination containing multi-level approach. We conclude that combining goal-directed search with the multi-level approach generally seems to be a good idea.

**Bi-Directed Search.** Bi-directed search individually gives a speed-up of about 1.5 for the number of visited nodes (see Figure 6) and for the running time, for all types of graphs. For combinations of bi-directed search with other speed-up techniques, the situation is different: For the generated graphs, neither the number of visited nodes nor the running time improves when bi-directed search is applied additionally to goal-directed search. However, running time improves with the combination containing the multi-level approach, and also combining bi-directed search with bounding boxes works very well. In the latter case, the speed-up is about 1.5 (as good as the speed-up of individual bi-directed search) for all types of graphs.

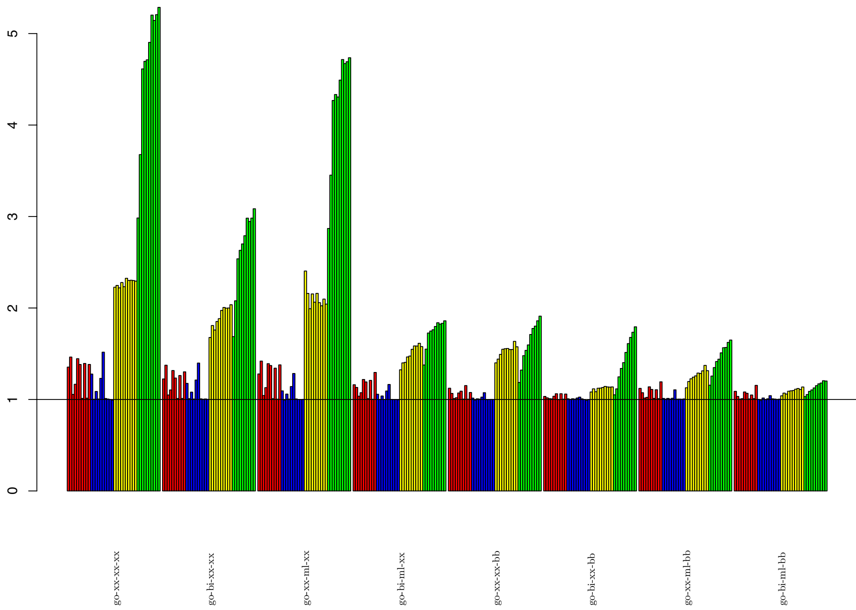


**Fig. 5.** Speed-up relative to the combination without goal-directed search in terms of visited nodes (in this order: street graphs in red, public transport graphs in blue, random planar graphs in yellow, and random Waxman graphs in green)

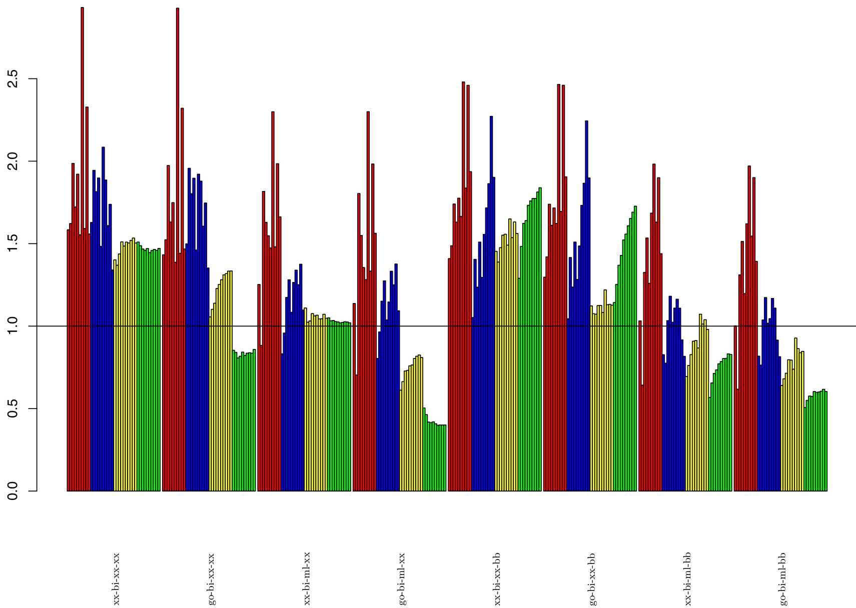


**Fig. 6.** Speed-up relative to the combination without bi-directed search in terms of visited nodes (in this order: street graphs in red, public transport graphs in blue, random planar graphs in yellow, and random Waxman graphs in green)

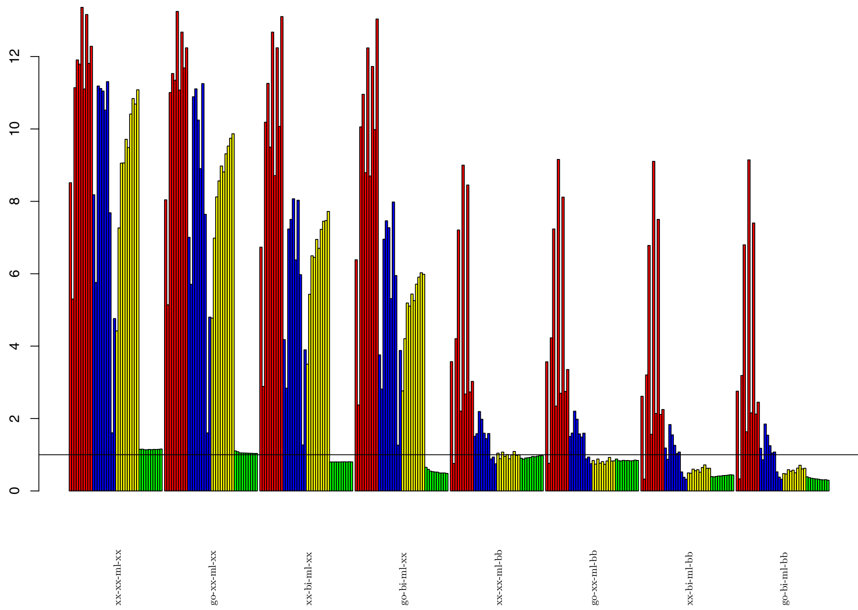


**Fig. 7.** Speed-up relative to the combination without multi-level approach in terms of visited nodes (in this order: street graphs in red, public transport graphs in blue, random planar graphs in yellow, and random Waxman graphs in green)

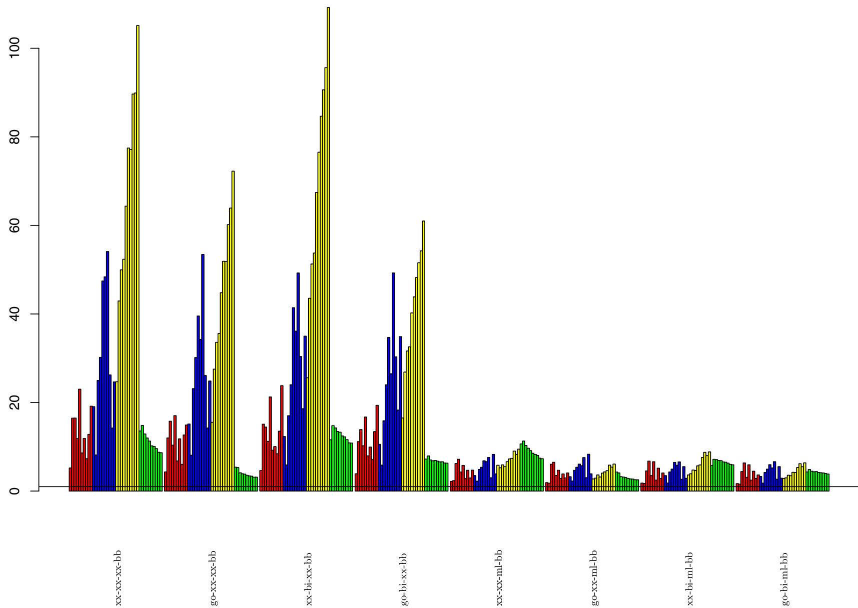


**Fig. 8.** Speed-up relative to the combination without shortest-path bounding boxes in terms of visited nodes (in this order: street graphs in red, public transport graphs in blue, random planar graphs in yellow, and random Waxman graphs in green)

**Multi-Level Approach.** The multi-level approach crucially depends on the decomposition of the graph. The Waxman graphs could not be decomposed properly by the multi-level approach, and therefore all combinations containing the latter yield speed-up factors of less than 1, which means a slowing down. Thus we consider only the remaining graph classes.

Adding multi-levels to goal-directed and bi-directed search and their combination gives a good improvement in the range between 5 and 12 for the number of nodes (see Figure 7). Caused by the big overhead of the multi-level approach, however, we get a considerable improvement in running time only for the real-world graphs. In combination with bounding boxes, the multi-level approach is beneficial only for the number of visited nodes in the case of street graphs.

The multi-level approach allows tuning of several parameters, such as the number of levels and the choice of the selected nodes. The tuning crucially depends on the input graph [20]. Hence, we believe that considerable improvements of the presented results are possible if specific parameters are chosen for every single graph.

**Shortest-Path Bounding Boxes.** Shortest-path bounding boxes work especially well when applied to planar graphs, actually speed-up even increases with the size of the graph (see Figure 8). For Waxman graphs, the situation is completely different: with the graph size the speed-up gets smaller. This can be explained by the fact that large Waxman graphs have, due to construction, more long-distance edges than small ones. Because of this, shortest paths become more tortuous and the bounding boxes contain more “wrong” nodes.

Throughout the different types of graphs, bounding boxes individually as well as in combination with goal-directed and bi-directed search yield exceptionally high speed-ups. Only the combinations that include the multi-level approach cannot be improved that much.

### 4.2 Overhead

For goal-directed and bi-directed search, the overhead (time per visited node) is quite small, while for bounding boxes it is a factor of about 2 compared to plain Dijkstra (see Figures 9 and 10). The overhead caused by the multi-level approach is generally high and quite different, depending on the type of graph. As Waxman graphs do not decompose well, the overhead for the multi-level approach is large and becomes even larger when the size of the graph increases. For very large street graphs, the multi-level approach overhead increases dramatically. We assume that it would be necessary to add a third level for graphs of this size.

It is also interesting to note that the relative overhead of the combination goal-directed, bi-directed, and multi-level is smaller than just multi-level —especially for the generated graphs.

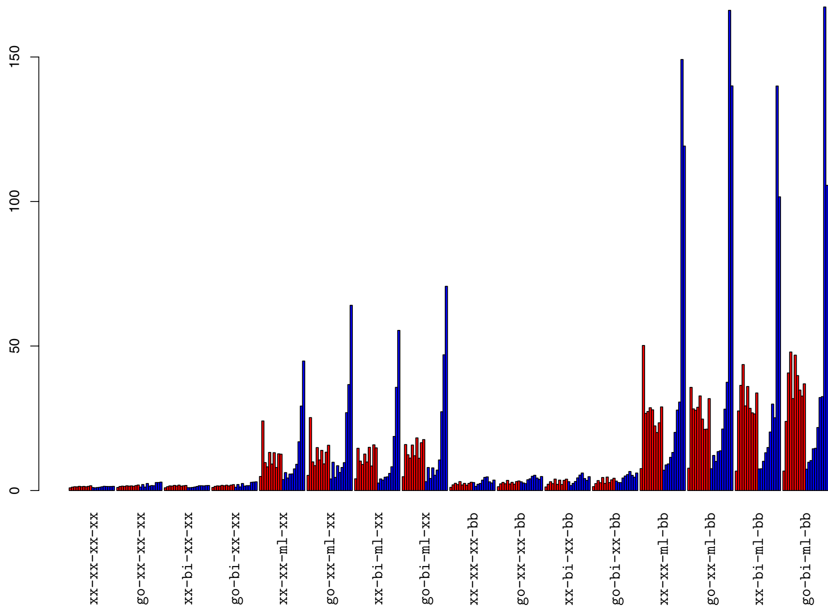


**Fig. 9.** Average running time per visited node in $\mu s$ for real-world graphs (in this order: street graphs in red and public transport graphs in blue)

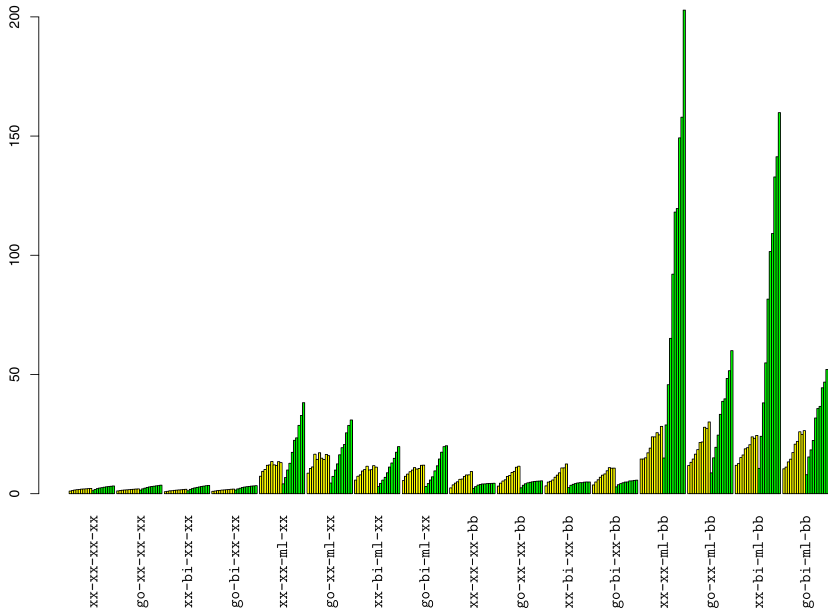


**Fig. 10.** Average running time per visited node in $\mu s$ for generated graphs (in this order: random planar graphs in yellow and random Waxman graphs in green)

## 5 Conclusion and Outlook

To summarize, we conclude that there are speed-up techniques that combine well and others where speed-up does not scale. Our result is that goal-directed search and multi-level approach is a good combination and bi-directed search with shortest-path bounding boxes complement each other.

For real-world graphs, a combination including bi-directed search, multi-level, and bounding boxes is the best choice as to the number of visited nodes. In terms of running time, the winner is bi-directed search in combination with bounding boxes. For generated graphs, the best combination is goal-directed, bi-directed, and bounding boxes for both the number of nodes and running time.

Without an expensive preprocessing, the combination of goal-directed and bi-directed search is generally the fastest algorithm with smallest search space—except for Waxman graphs. For these graphs, pure goal-directed is better than the combination with bi-directed search. Actually, goal-directed search is the only speed-up technique that works comparatively well for Waxman graphs. Because of this different behaviour, we conclude that planar graphs are a better approximation of the real-world graphs than Waxman graphs (although the public transport graphs are not planar).

Except bi-directed search, the speed-up techniques define a modified graph in which a shortest path is searched. From this shortest path one can easily determine a shortest path in the original graph. It is an interesting question whether the techniques can be applied directly, or modified, to improve also the running time of other shortest-path algorithms.

Furthermore, specialized priority queues used in Dijkstra's algorithm have been shown to be fast in practice [26,27]. Using such queues would provide the same results for the number of visited nodes. Running times, however, would be different and therefore interesting to evaluate.

## References

1. Zhan, F.B., Noon, C.E.: A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths. Journal of Geographic Information and Decision Analysis **4** (2000)
2. Barrett, C., Bisset, K., Jacob, R., Konjevod, G., Marathe, M.: Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router. In: Proc. 10th European Symposium on Algorithms (ESA). Volume 2461 of LNCS., Springer (2002) 126–138
3. Nachtigall, K.: Time depending shortest-path problems with applications to railway networks. European Journal of Operational Research **83** (1995) 154–166
4. Preuss, T., Syrbe, J.H.: An integrated traffic information system. In: Proc. 6th Int. Conf. Appl. Computer Networking in Architecture, Construction, Design, Civil Eng., and Urban Planning (europIA '97). (1997)
5. Shekhar, S., Fetterer, A., Goyal, B.: Materialization trade-offs in hierarchical shortest path algorithms. In: Proc. Symp. on Large Spatial Databases. (1997) 94–111
6. Barrett, C., Jacob, R., Marathe, M.: Formal-language-constrained path problems. SIAM Journal on Computing **30** (2000) 809–837

7. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
8. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34** (1987) 596–615
9. Cherkassky, B.V., Goldberg, A.V., Radzik, T.: Shortest paths algorithms: Theory and experimental evaluation. Mathematical Programming **73** (1996) 129–174
10. Zwick, U.: Exact and approximate distances in graphs - a survey. In: Proc. 9th European Symposium on Algorithms (ESA). LNCS, Springer (2001) 33–48
11. Goldberg, A.V.: A simple shortest path algorithm with linear average time. In: Proc. 9th European Symposium on Algorithms (ESA). Volume 2161 of LNCS., Springer (2001) 230–241
12. Meyer, U.: Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In: Proc. 12th Symp. on Discrete Algorithms. (2001) 797–806
13. Pettie, S., Ramachandran, V., Sridhar, S.: Experimental evaluation of a new shortest path algorithm. In: Proc. Algorithm Engineering and Experiments (ALENEX). Volume 2409 of LNCS., Springer (2002) 126–142
14. Hart, P., Nilsson, N.J., Raphael, B.A.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Sys. Sci. Cybernet. **2** (1968)
15. Shekhar, S., Kohli, A., Coyle, M.: Path computation algorithms for advanced traveler information system (ATIS). In: Proc. 9th IEEE Int. Conf. Data Eng. (1993) 31–39
16. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. ACM Journal of Exp. Algorithmics **5** (2000)
17. Ahuja, R., Magnanti, T., Orlin, J.: Network Flows. Prentice–Hall (1993)
18. Pohl, I.: Bi-directional and heuristic search in path problems. Technical Report 104, Stanford Linear Accelerator Center, Stanford, California (1969)
19. Kaindl, H., Kainz, G.: Bidirectional heuristic search reconsidered. Journal of Artificial Intelligence Research **7** (1997) 283–317
20. Holzer, M.: Hierarchical speed-up techniques for shortest-path algorithms. Technical report, Dept. of Informatics, University of Konstanz, Germany (2003) http://www.ub.uni-konstanz.de/kops/volltexte/2003/1038/.
21. Schulz, F., Wagner, D., Zaroliagis, C.: Using multi-level graphs for timetable information in railway systems. In: Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX). Volume 2409 of LNCS., Springer (2002) 43–59
22. Jung, S., Pramanik, S.: An efficient path computation model for hierarchically structured topographical road maps. IEEE Transactions on Knowledge and Data Engineering **14** (2002) 1029–1046
23. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: Proc. 11th European Symposium on Algorithms (ESA). Volume 2832 of LNCS., Springer (2003) 776–787
24. Näher, S., Mehlhorn, K.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press (1999) (http://www.algorithmic-solutions.com).
25. Waxman, B.M.: Routing of multipoint connections. IEEE Journal on Selected Areas in Communications **6** (1988)
26. Dial, R.: Algorithm 360: Shortest path forest with topological ordering. Communications of ACM **12** (1969) 632–633
27. Goldberg, A.V.: Shortest path algorithms: Engineering aspects. In: Proc. International Symposium on Algorithms and Computation (ISAAC). Volume 2223 of LNCS., Springer (2001) 502–513