Speed-Up Techniques for Shortest-Path Computations*

Dorothea Wagner and Thomas Willhalm

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Theoretische Informatik
D-76128 Karlsruhe
{wagner,willhalm}@ira.uka.de
http://i11www.informatik.uni-karlsruhe.de/

Abstract. During the last years, several speed-up techniques for Dijkstra's algorithm have been published that maintain the correctness of the algorithm but reduce its running time for typical instances. They are usually based on a preprocessing that annotates the graph with additional information which can be used to prune or guide the search. Timetable information in public transport is a traditional application domain for such techniques. In this paper, we provide a condensed overview of new developments and extensions of classic results. Furthermore, we discuss how combinations of speed-up techniques can be realized to take advantage from different strategies.

1 Introduction

Computing shortest paths is a base operation for many problems in traffic applications. The most prominent are certainly route planning systems for cars, bikes and hikers, or timetable information systems for scheduled vehicles like trains and busses. If such a system is realized as a central server, it has to answer a huge number of customer queries asking for their best itineraries. Users of such a system continuously enter their requests for finding their "best" connections. Furthermore, similar queries appear as sub-problems in line planning, timetable generation, tour planning, logistics, and traffic simulations.

The algorithmic core problem that underlies the above scenario is a special case of the single-source shortest-path problem on a given directed graph with non-negative edge lengths. While this is obvious for route planning in street networks, different models and approaches have been presented to solve timetable information by finding shortest paths in an appropriately defined graph. The typical problem to be solved in timetable information is "given a departure and an arrival station as well as a departure time, which is the connection that arrives as early as possible at the arrival station?". There are two main approaches for

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

modeling timetable information as shortest path problem, the time-expanded and the time-dependent approach. For an overview of models and algorithms for optimally solving timetable information we refer to [28].

In any case the particular graphs considered are huge, especially if the model used for timetable information expands time by modelling each event by a single vertex in the graph. Moreover, the number of queries to be processed within very short time is huge as well. This motivates the use of speed-up techniques for shortest-path computations. The main focus is to reduce the response time for on-line queries. In this sense, a speed-up technique is considered as a technique to reduce the search space of DIJKSTRA'S ALGORITHM e.g. by using precomputed information or inherent information contained in the data. Actually, often the underlying data contain geographic information, that is a layout of the graph is provided. Furthermore, in many applications the graph can be assumed to be static, which allows a preprocessing. Due to the size of the graphs considered in route planning or timetable information and the fact that those graphs are typically sparse, preprocessing space requirements are only acceptable to be linear in the number of nodes.

In this paper, we provide a systematic classification of common speed-up techniques and combinations of those. Our main intention is to give a concise overview of the current state of research. We restrict our attention to speed-up techniques where the correctness of the algorithms is guaranteed, i.e., that provably return a shortest path. However, most of them are heuristic with respect to the running time. More precisely, in the worst case, the algorithm *with* speed-up technique can be slower than the algorithm *without* speed-up technique. But experimental studies showed–sometimes impressive–improvements concerning the search front and consequently the running time. For most of these techniques, experimental results for different real-world graphs as well as generated graphs have been reported. However, as the effectiveness of certain speed-up techniques strongly depends on the graph data considered, we do not give a comparison of the speed-ups obtained. But we want to refer to the *9th DIMACS Implementation Challenge - Shortest Paths* where also experiments on common data sets were presented [7].

In the next section, we will provide some formal definitions and a description of DIJKSTRA'S ALGORITHM. Section 3 presents a classification of speed-up techniques for DIJKSTRA'S ALGORITHM and discusses how they can be combined.

2 Preliminaries

2.1 Definitions

A *(directed) graph* G is a pair (V, E), where V is a finite set of *nodes* and E is a set of *edges*, where an edge is an ordered pair (u, v) of nodes $u, v \in V$. Throughout this paper, the number of nodes $|V|$ is denoted by n and the number of edges $|E|$ is denoted by m. For a node $u \in V$, the number of outgoing edges $|\{(u, v) \in E\}|$ is called the *degree* of the node. A *path* in G is a sequence of nodes $(u_1, \ldots, u_k)$

```
 1 for all nodes u ∈ V set dist(u) := ∞
 2 initialize priority queue Q with source s and set dist(s) := 0
 3 while priority queue Q is not empty
 4     get node u with smallest tentative distance dist(u) in Q
 5     for all neighbor nodes v of u
 7         set new-dist := dist(u) + w(u, v)
 8         if new-dist < dist(v)
 9             if dist(v) = ∞
10                 insert neighbor node v in Q with priority new-dist
11             else
12                 set priority of neighbor node v in Q to new-dist
13             set dist(v) := new-dist
```

Algorithm 1: DIJKSTRA'S ALGORITHM.

such that $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. A path with $u_1 = u_k$ is called a *cycle*. Given edge weights $l : E \to \mathbb{R}$ ("lengths"), the *length of a path* $P = (u_1, \ldots, u_k)$ is the sum of the lengths of its edges $l(P) := \sum_{1 \leq i < k} l(u_i, u_{i+1})$. For two nodes $s, t \in V$, a *shortest s-t path* is a path of minimal length with $u_1 = s$ and $u_k = t$. The *(graph-theoretic) distance* $d(s, t)$ of s and t is the length of a shortest s-t path. A *layout* of a graph $G = (V, E)$ is a function $L : V \to \mathbb{R}^2$ that assigns each node a position in $\mathbb{R}^2$. The Euclidean distance between two nodes $u, v \in V$ is then denoted by $\|L(u) - L(v)\|$. A graph (without multiple edges) can have up to $O(n^2)$ edges. We call a graph *sparse*, if $m = O(n)$. In the following we assume that the graphs we are dealing with are large and one can only afford a memory consumption linear in the size of the graph. In particular, for large sparse graphs $O(n^2)$ space is not affordable.

2.2 Shortest Path Problem

Let $G = (V, E)$ be a directed graph whose edges are *weighted* by a function $l : E \to \mathbb{R}$. The *(single-source single-target) shortest-path problem* consists in finding shortest s-t path from a given source $s \in V$ to a given target $t \in V$. Note that the problem is only well defined for all pairs, if G does not contain negative cycles (cycles with negative length). In the presence of negative weights but not negative cycles, it is possible, using Johnson's algorithm [19], to convert in $O(nm + n^2 \log n)$ time the original edge weights $l : E \to \mathbb{R}$ to non-negative edge weights $l' : E \to \mathbb{R}_0^+$ that result in the same shortest paths. Hence, we can safely assume in the rest of this paper that edge weights are non-negative. We also assume throughout the paper that for all pairs $(s, t) \in V \times V$, the shortest path from s to t is unique. (This can be achieved by adding a small fraction to the edge weights, if necessary.)

The classical algorithm for computing shortest paths in a directed graph with non-negative edge weights is that of Dijkstra [6], independently discovered by Dantzig [2] (Algorithm 1). The algorithm maintains, for each node $v \in V$, a

label $\texttt{dist}(v)$ with the current tentative distance. The algorithm uses a priority queue Q containing the nodes that build the current search horizon around s. Nodes are either *unvisited* (i.e. $\texttt{dist}(u) = \infty$), in the priority queue, or *finished* (already removed from the priority queue). It is easy to verify that nodes are never reinserted in the priority queue if the extracted node u in line 4 is the node with the smallest tentative distance in the priority queue and all edge weights are non-negative. Thus, the labels are updated while the algorithm visits the nodes of the graph with non-decreasing distance from the source s.

In order to compute a shortest path tree, one has to remember that u is the predecessor of v if a shorter path to v has been found (i.e. between line 8 and 9). DIJKSTRA'S ALGORITHM computes the shortest paths to all nodes in the Graph. If only one shortest path is needed to a target node $t \in V$, the algorithm can stop if the target t is removed from the priority queue in line 4. If DIJKSTRA'S ALGORITHM is executed more than once, the initialization of `dist` in line 1 for each run can be omitted by introducing a global integer variable `time` and replacing the test $\texttt{dist}(v) = \infty$ by a comparison of the `time` with a time stamp for every node. See e.g., [33] for a detailed description.

The asymptotic time complexity of DIJKSTRA'S ALGORITHM depends on the choice of the priority queue. For general graphs, Fibonacci heaps [8] still provide the best theoretical worst-case time of $O(m + n \log n)$. For sparse graphs, binary heaps result in the same asymptotic time complexity. Even more, binary heaps are (1) easier to implement and (2) perform better for many instances in practice [25]. For special cases of edge weights, better algorithms are known. If edge weight are integral and bounded by a small constant, Dial's implementation [5] with an array of lists ("buckets") provides a priority queue where all operations take constant time. An extension with average linear complexity for uniformly distributed edge weights is presented in [9, 26]. One might argue however, that the better a speed-up techniques works, the smaller the search front is, and the less important the priority queue is.

3 Speed-up Techniques

In this section, we present *speed-up techniques* for DIJKSTRA'S ALGORITHM, i.e. modifications of the algorithm or graph that do not change the worst-case behavior but usually reduce considerably the number of visited nodes in practice. We shortly describe two classical speed-up techniques, *bidirectional search* and *goal-directed search*. Moreover, we give a classification of more recently presented techniques.

3.1 Bidirectional Search

Bidirectional search simultaneously performs two searches: a "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 backward algorithm. Then, the shortest path is determined by the node u with minimum value $d_f(u) + d_b(u)$ and it can be composed of the shortest path from the start node s to u, (found by the forward search), and the shortest path from u to the destination t (found by the reverse search). Note that the node u itself is not necessarily marked as permanent by both searches.

One degree of freedom in bidirectional search is the choice whether a forward or backward step is executed. Common strategies are to choose the direction with the smaller priority queue, to select the direction with the smaller minimal distance in the priority queue, or simply alternate the directions. For a theoretical discussion of bidirectional search, see [24].

3.2 Goal-Directed Search or *A**

This technique, originating from AI [15], modifies the priority of active nodes to change the order in which the nodes are processed. More precisely, a goal-directed search adds to the priority $\texttt{dist}(u)$ a *potential* $p_t : V \to \mathbb{R}_0^+$ (often called *heuristic*) depending on the target t of the search. The modified priority of a node $v \in V$ is therefore $\texttt{dist}(v) + p_t(v)$. With a suited potential, the search can be pushed towards the target thereby reducing the running time while the algorithm still returns a shortest path. Intuitively speaking, one can compare a path in traffic network with a walk in a landscape. If you add a potential, the affected region is raised. If the added potential is small next to the target, you create a valley around the target. As walking downhill is easier than uphill, you are likely to hit the target sooner than without the potential added.

We will now use an alternative formulation of goal-directed search to discuss its correctness. Equivalently to modifying the priority, one can change the edge lengths such that the search is driven towards the target t. In this case, the weight of an edge $(u, v) \in E$ is replaced by $l'(u, v) := l(u, v) - p_t(u) + p_t(v)$. The length of a s-v path $P = (s = v_1, v_2, \ldots, v_{k+1} = v)$ is then

$$\begin{aligned} l'(P) = \sum_{i=1}^{k} l'(v_i, v_{i+1}) &= \sum_{i=1}^{k} l(v_i, v_{i+1}) - p_t(v_i) + p_t(v_{i+1}) \\ &= -p_t(s) + p_t(v) + \sum_{i=1}^{k} l(v_i, v_{i+1}) \\ &= -p_t(s) + p_t(v) + l(P). \end{aligned}$$

In particular, the length of an s-t path with modified edge lengths is the same up to the constant $-p_t(s) + p_t(t)$. Therefore, a path from s to t is a shortest s-t path according to l', if and only if it is a shortest s-t path according to l.

If all modified edge lengths $l'(u,v)$ are non-negative, we can apply DIJKSTRA'S ALGORITHMto the graph with modified edge lengths l' and get a shortest s-t path according to l. This leads to the following definition:

Definition 1. *Given a weighted graph $G = (V,E), l : V \to \mathbb{R}_0^+$, a potential $p : V \to \mathbb{R}$ is called* feasible*, if $l(u,v) - p(u) + p(v) \geq 0$ for all edges $e \in E$.*

Usually, potentials are used that estimate the distance to the target. In fact, it can be shown that a feasible potential p is a lower bound of the distance to the target t if $p(t) \leq 0$. Note that every feasible potential p can be transposed into an equivalent potential $p'(v) = p(v) - p(t)$ which is a lower bound of the distance to the target. We can therefore assume without loss of generality that the potential is indeed a lower bound. The tighter the bound is, the more the search is attracted to the target. In particular, a goal-directed search visits only nodes on the shortest path, if the potential is the distance to the target.

In an actual implementation of goal-directed search, you will most probably use the first formulation, namely to modify the priority with which nodes are inserted in the priority queue. This has the advantage that p is called (at most) once per edge instead of two calls. Furthermore, the distance labels of the nodes are unmodified. This improves the numerical stability and simplifies the handling of the labels (in particular in combinations with other speed-up techniques).

We will now present three scenarios and how to obtain feasible potentials in these cases:

Euclidean Distances Assume a layout $L : V \to \mathbb{R}^2$ of the graph is available where the length of an edge is somehow correlated with the Euclidean distance of its end nodes. Then a feasible potential for a node v can be obtained using the Euclidean distance (the "flight distance") $\|L(v) - L(t)\|$ to the target t.

In case the edge lengths are in fact the Euclidean distances, the Euclidean distance $\|L(v) - L(t)\|$ itself is already a feasible potential, due to the triangular inequality. Using this potential, an edge that points directly towards the destination has a modified edge length of zero, while the modified length of an edge that points in the opposite direction is twice the distance. A theoretical analysis for various random graphs can be found in [35].

If the edge lengths are *not* the Euclidean distances of the end nodes, a feasible potential can be defined as follows: let v_{max} denote the maximum "edge-speed" $\|L(u) - L(v)\|/l(u,v)$, over all edges $(u,v) \in E$. The potential of a node u can now be defined as $p(u) = \|L(u), L(t)\|/v_{max}$. The maximum velocity can be computed in a preprocessing step by a linear scan over all edges. Numerical problems can be reduced if the maximum velocity is multiplied by $1 + \varepsilon$ for a small $\varepsilon > 0$. [37] presents how graph-drawing algorithms help in the case where a layout of the graph is not given beforehand.

This approach can be extended in a straight forward manner to other metric spaces than $(\mathbb{R}^2, \|\cdot\|)$. In particular, it is possible to use more than two dimensions or other metrics like the Manhattan metric. Finally, the expensive square root function to compute the Euclidean distance can be replaced by an approximation.

Landmarks With preprocessing, it is possible to gather information about the graph that can be used to obtain improved lower bounds. In [10], a small fixed-sized subset $L \subset V$ of "landmarks" is chosen. Then, for all nodes $v \in V$, the distance $d(v, l)$ to all nodes $l \in L$ is precomputed and stored. These distances can be used to determine a feasible potential. For each landmark $l \in L$, we define the potential $p_t^{(l)}(v) := d(v, l) - d(t, l)$. Due to the triangle inequality $d(v, l) \leq d(v, t) + d(t, v)$, the potential $p_t^{(l)}$ is feasible and indeed a lower bound for the distance to t. The potential is then defined as the maximum over all potentials: $p_t(v) := \max\{p_t^{(l)}(v); l \in L\}$. It is easy to show that the maximum of feasible potentials is again a feasible potential.

For landmarks that are situated next to or "behind" the target t, the lower bound $p_t^{(l)}(u)$ should be fairly tight, as shortest paths to t and l most probably share a common sub-path. Landmarks in other regions of the graph however, may attract the search to themselves. This insight justifies to consider, in a specific search from s to t, only those landmarks with the highest potential $p_t^{(l)}(u)$. The restriction of the landmarks in use has the advantage that the calculation of the potential is faster while its quality is improved.

An interesting observation is that using k landmarks is in fact very similar to using the maximum norm in a k-dimensional space. Each landmark corresponds to one dimension and, for a node, the distance to a landmark is the coordinate in the corresponding dimension. Such high-dimensional drawings have been used in [14], where they are projected to 2D using principal component analysis (PCA). This graph-drawing techniques has also been successfully used in [37] for goal-directed search and other geometric speed-up techniques.

Distances from Graph Condensation For restricted shortest-path problems, performing a single run of an unrestricted DIJKSTRA'S ALGORITHM is a relatively cheap operation. Examples are travel planning systems for scheduled vehicles like busses or trains. The complexity of the problem is much higher if you take connections, vehicle types, transfer times, or traffic days into account. It is therefore feasible to perform a shortest-path computation to find tighter lower bounds [29]. More precisely, you run DIJKSTRA'S ALGORITHM on a condensed graph: The nodes of this graph are the stations (or stops) and an edge between two stations exists iff there is a non-stop connection. The edges are weighted by the minimal travel time. The distances of all v to the target t can be obtained by a single run of DIJKSTRA'S ALGORITHM from the target t with reversed edges. These distances provide a feasible potential for the time-expanded graph, since the distances are a feasible potential in the condensed graph and an edge between two stations in the time-expanded graph is at least as long as the corresponding edge in the condensed graph.

3.3 Hierarchical Methods

This speed-up technique requires a preprocessing step at which the input graph $G = (V, E)$ is enriched with additional edges representing shortest paths between

certain nodes. The additional edges can be seen as "bridges" or "short-cuts" for DIJKSTRA'S ALGORITHM. These additional edges thereby realize new levels that step-by-step coarsen the graph. To find a shortest path between two nodes s and t using a hierarchy, it suffices for DIJKSTRA'S ALGORITHM to consider a relatively small subgraph of the "hierarchical graph". The hierarchical structure entails that a shortest path from s to t can be represented by 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. Mainly two methods have been developed to create such a hierarchy, the *multi-level approach* [33, 34, 18, 4] and *highway hierarchies* [31, 32]. These hierarchical methods are already close to the idea of using precomputed shortest paths tables for a small number of very frequently used "transit nodes". Recently, this idea has been explored for the computation of shortest paths in road networks with respect to travel time [1].

Multi-Level Approach The decomposition of the graph can be realized using separators $S_i \subset V$ 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. In a simple, but practical implementation, they consist of the desired numbers of nodes with highest degree in the graph. However, with domain-specific knowledge about the central nodes in the graph, better separators can be found. Alternatively, the planar separator theorem or betweenness centrality can be used to find small separators [18]. 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.

In [4] a further enhancement of the multi-level approach is presented, which uses a precomputed auxiliary graph with additional information. Instead of a single multi-level graph, a large number of small partial graphs is precomputed, which are optimized individually. This approach results in even smaller query times than achieved by the original multi-level approach. On the other hand, however, a comparably heavy preprocessing is required.

Highway Hierarchies A different approach presented by [31, 32] is also based on the idea that only a "highway network" needs to be searched outside a the neighborhood of the source and the target node. Shortest path trees are used to determine a hierarchy. This has the advantage that no additional information like a separator is needed. Moreover, the use of highway hierarchies requires a less extensive preprocessing. The construction relies on a slight modification of DIJKSTRA'S ALGORITHM that ensures that a sub-path $u_i, \ldots, u_j$ of a shortest path $u_1, \ldots, u_i, \ldots, u_j, \ldots, u_k$ is always returned as the shortest path from u_i to u_j. These shortest paths are called *canonical*. Consider the sub-graph of G that consists of all edges in canonical shortest paths. The next level of the hierarchy is then induced by all nodes with degree at least two (i.e. the 2-core of the union of canonical shortest paths). Finally, nodes of degree 2 are then iteratively replaced by edges for a further contraction of the new level of the hierarchy.

3.4 Node and Edge Labels

Approaches based on node or edge labels use precomputed information as an indicator if a node or an edge has to be considered during an execution of DIJKSTRA'S ALGORITHM for a certain target node t.

Reach-Based Routing Reach-based routing prunes the search space based on a centrality measure called "reach" [13]. Intuitively, a node in the graph is important for shortest paths, if it is situated in the middle of long shortest paths. Nodes that are only at the beginning or the end of long shortest paths are less central. This leads to the following formal definition:

Definition 2 (Reach). *Given a weighted graph $G = (V, E), l : E \to \mathbb{R}_0^+$ and a shortest s-t path P, the* reach on the path P *of a node $v \in P$ is defined as $r(v, P) := \min\{l(P_{sv}), l(P_{vt})\}$ where P_{sv} and P_{vt} denote the sub-paths of P from s to v and from v to t, respectively. The* reach $r(v)$ *of $v \in V$ is defined as the maximum reach for all shortest s-t paths in G containing v.*

In a search for a shortest s-t path P_{st}, a node $v \in V$ can be ignored, if (1) the distance $l(P_{sv})$ from s to v is larger than the reach of v and (2) the distance $l(P_{vt})$ from v to t is larger than the reach of v. While performing DIJKSTRA'S ALGORITHM, the first condition is easy to check, since $l(P_{sv})$ is already known. The second condition is fulfilled if the reach is smaller than a lower bound of the distance from v to t. (Suited lower bounds for the distance of a node to the target are already described for goal-directed search in Sect. 3.2.) Lines 7-13 of Algorithm 1 are therefore not performed if conditions (1) and (2) are surely fulfilled.

To compute the reach for all nodes, we perform a single-source all-target shortest-path computation for every node. With a modified depth first search on the shortest-path trees, it is easy to compute the reach of all nodes using the following insight: For two shortest paths P_{sx} and P_{sy} with a common node $v \in P_{sx}$ and $v \in P_{sy}$, we have

$$\max\{r(v, P_{sx}), r(v, P_{sy})\} = \min\{l(P_{sv}), \max\{l(P_{vx}), l(P_{vy})\}\}.$$

The preprocessing for sparse graphs needs therefore $O(n^2 \log n)$ time and $O(n)$ space. In case such a heavy preprocessing is not acceptable, [13] also describes how to compute upper bounds for the reach. As mentioned in [11], the reach criterion can be extended to edges, which even improves its effectiveness but also increases the preprocessing time.

Edge Labels This approach attaches a label to each edge that represents all nodes to which a shortest path starts with this particular edge [22, 23, 27, 33, 36, 38]. More precisely, we first determine, for each edge $(u, v) \in E$, the set $S(u, v)$ of all nodes $t \in V$ to which a shortest u-t path starts with the edge (u, v). The shortest path queries are then answered by DIJKSTRA'S ALGORITHM restricted to those edges (u, v) for which the target node is in $S(u, v)$. Similar to a traffic sign, the edge label shows the algorithm if the target node might

be in the target region of the edge. It is easy to verify that such a pruned shortest-path computation returns a shortest path: If (u, v) is part of a shortest s-t path, then its sub-path from u to t is also a shortest path. Therefore, t must be in $S(u, v)$, because all nodes to which a shortest path starts with (u, v) are located in $S(u, v)$. The restriction of the graph can be realized on-line during the shortest-path computation by excluding those edges whose edge label does not contain the target node (line 5 of algorithm 1).

Geometric Containers As storing all sets $S(u, v)$ would need $O(n^2)$ space, one can use a superset of $S(u, v)$ that can be represented with constant size. Using constant-sized edge labels, the size of the preprocessed data is linear in the size of the graph. Given a layout $L : V \to \mathbb{R}^2$ of the graph, an efficient and easy object type for an edge label associated to (u, v) is an *enclosing geometric object* of $\{L(t) \mid t \in S(u, v)\}$. Actually, the *bounding box*, i.e. the smallest rectangle parallel to the axes that contains $\{L(t) \mid t \in S(u, v)\}$ turns out to be very effective as geometric container [38]. The bounding boxes can be computed beforehand by running a single-source all-target shortest-path computation for every node. The preprocessing for sparse graphs needs therefore $O(n^2 \log n)$ time and $O(n)$ space.

Arc Flags If you drop the condition that the edge labels must have constant size, you can get much better however. An approach that performs very well in practice [22, 23, 27], is to partition the node set in p regions with a function $r : V \longrightarrow \{1, \ldots, p\}$. Then an arc flag, i.e. a *p-bit-vector* where each bit represents one region is used as edge label. For an edge e, a region is marked in the p-bit-vector of e if it contains a node v with $v \in S(e)$.) Then the overall space requirement for the preprocessed data is $\Theta(p \cdot m)$. But an advantage of bit-vectors as edge labels is the insight that the preprocessing does not need to compute *all*-pairs shortest paths. Every shortest path from any node s outside a region R to a node inside a region R has to enter the region R at some point. As s is not a member of region R, there exists an edge $e = (u, v)$ such that $r(u) \neq r(v)$. It is therefore sufficient, if the preprocessing algorithm regards only the shortest paths to nodes v that are on the boundary of a region. These paths can be determined efficiently by a backward search starting at the boundary nodes. Usually, the number of boundary nodes is by orders of magnitude smaller than n. A crucial point for this type of edge labels is an appropriate partitioning of the node set. Using a layout of the graph, e.g. a *grid*, *quad-trees* or *kd-trees* can be used. In a general setup, a separator according to [21] is the best choice we are aware of [27].

3.5 Combining Speed-Up Techniques

It has been shown in various publications [3, 11, 12, 16, 17, 30–33, 37] that the full power of speed-up techniques is unleashed, if various speed-up techniques are combined. In [16, 17] combinations of *bidirectional search, goal-directed search, multi-level approach* and *geometric container* are examined. For an experimental evaluation we refer to these papers. In this section, we concentrate on cases,

where an effective combination of two speed-up techniques is not obvious. The extension to a combination of three or four techniques is straight forward, once the problem of combining two of them is solved. However, not every combination is useful, as the search space may not be decreased (much) by adding a third or fourth speed-up techniques.

Bidirectional Search and Goal-Directed Search Combining goal-directed and bidirectional search is not as obvious as it may seem at first glance. [30] provides a counter-example to show that simple application of a goal-directed search forward and a "source-directed" search backward yields a wrong termination condition. However, the alternative condition proposed there has been shown in [20] to be quite inefficient, as the search in each direction almost reaches the source of the other direction. An alternative is to use the *same* potential in both directions. With a potential from Sect. 3.2, you already get a speed-up (compared to using either goal-directed or bidirectional search). But one can do better using a combination of potentials: if $p_s(v)$ is a feasible potential for the backward search, then $p_s(t) - p_s(v)$ is a feasible potential for the forward search (although not necessarily a good one). In order to balance the forward and the backward search, the average $\frac{1}{2}(p_t(v) + p_s(t) - p_s(v))$ is a good compromise [10].

Bidirectional Search and Hierarchical Methods Basically, bidirectional search can be applied to the subgraph defined by the multi-level approach. In an actual 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 a bidirectional search is applied 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. See [16, 17] for an experimental evaluation. Actually, [31, 32] takes this combination even further in that it fully integrates the two approaches. The conditions for the pruning of the search space are interweaved with the fact that the search is performed in two directions at the same time.

Bidirectional Search and Reach-Based Routing The reach criterion $l(P_{sv}) \leq r(v) \vee l(P_{vt}) \leq r(v)$ can be used directly in the backward direction of the bidirectional search, too. In the backward search, $l(P_{vt})$ is already known whereas we have to use a lower bound instead of $l(P_{sv})$ to replace the first condition $l(P_{sv}) \leq r(v)$. However, even without using a geometric lower bound but only the known distances for pruning, [11] reports good results.

Bidirectional Search and Edge Labels In order to take advantage of edge labels in both directions of a bidirectional search, a second set of edge labels is needed. For each edge $e \in E$, we compute the set $S(e)$ and the set $S_{\text{rev}}(e)$ of those nodes from which a shortest path ending with e exists. Then we store for each edge $e \in E$ appropriate edge labels for $S(e)$ and $S_{\text{rev}}(e)$. The forward search checks whether the target is contained $S(e)$, the backward search, whether the source is in $S_{\text{rev}}(e)$. See [16, 17].

Goal-Directed Search and Highway Hierarchies Already the original highway algorithm [31, 32] accomplishes a bidirectional search. In [3] the highway hierarchies are further enhanced with goal-directed capabilities using potentials for forward and backward search based on landmarks. Unfortunately, the highway algorithm cannot abort the search as soon as an s-t path is found. However, another aspect of goal-directed search can be exploited, the pruning. As soon as an s-t path is found it yields an upper bound for the length of the shortest s-t path. Comparing upper and lower bound can then be used to prune the search. Altogether, the combination of highway hierarchies and landmarks brings less improvement than one might hope. On the other hand, using stopping the search as soon as an s-t path is found at the cost of losing correctness of the result (the s-t path found is not always the shortest s-t path) leads to an impressive speed-up. Moreover, almost all paths found are also shortest and, in the rare other cases the approximation error is extremely small.

Goal-Directed Search and Reach-Based Routing Goal-directed search can also be applied to the subgraph that is defined by the reach criterion. However, some care is needed if the subgraph is determined on-line (which is the common way to implement it) with the restriction by the reach. In particular, one should choose an implementation of goal-directed search that doesn't change the distance labels of the nodes, as they are used to check the reach criterion. A detailed analysis of this combination can be found in [11]. Finally, in [12] the study of reach-based routing in combination with goal-directed search based on landmarks is continued.

4 Conclusion

We have summarized various techniques to speed-up DIJKSTRA'S ALGORITHM. All of them guarantee to return a shortest path but run considerably faster. After all, the "best" choice of a speed-up technique heavily depends on the availability of a layout, the size of the main memory, the amount of preprocessing time you are willing to spend, and last but not least on the graph data considered.

References

1. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Shortest-Path Queries in Road Networks. In Proc. Algorithm Engineering and Experiments (ALENEX'07), SIAM (2007) to appear.
2. Dantzig, G.: On the shortest route through a network. Mgnt. Sci. **6** (1960) 187–190
3. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway Hierarchies Star. 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/ challenge9/papers.shtml
4. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-Performance Multi-Level Graphs. 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/ challenge9/papers.shtml

5. Dial, R.: Algorithm 360: Shortest path forest with topological ordering. Communications of ACM **12** (1969) 632–633
6. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
7. 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/ challenge9/papers.shtml
8. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (JACM) **34** (1987) 596–615
9. Goldberg, A.V.: Shortest path algorithms: Engineering aspects. In Eades, P., Takaoka, T., eds.: Proc. International Symposium on Algorithms and Computation (ISAAC 2001). Volume 2223 of LNCS., Springer (2001) 502–513
10. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* search meets graph theory. In: Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05), 156–165
11. Goldberg, A.V., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In Raman, R., Stallmann, M., eds.: Proc. Algorithm Engineering and Experiments (ALENEX'06), SIAM (2006) 129–143
12. Goldberg, A.V., Kaplan, H., Werneck, R.: Better Landmarks within Reach. In 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/ challenge9/papers.shtml
13. Gutman, R.: Reach-based routing: A new approach to shortest path algortihms optimized for road networks. In Arge, L., Italiano, G.F., Sedgewick, R., eds.: Proc. Algorithm Engineering and Experiments (ALENEX'04), SIAM (2004) 100–111
14. Harel, D., Koren, Y.: A fast multi-scale method for drawing large graphs. Journal of graph algorithms and applications **6** (2002) 179–202
15. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on systems science and cybernetics **4** (1968) 100–107
16. Holzer, M., Schulz, F., Willhalm, T.: Combining speed-up techniques for shortest-path computations. In Ribeiro, C.C., Martins, S.L., eds.: Experimental and Efficient Algorithms: Third International Workshop, (WEA 2004). Volume 3059 of LNCS., Springer (2004) 269–284
17. Holzer, M., Schulz, F., Wagner, D., Willhalm, T.: Combining speed-up techniques for shortest-path computations. ACM Journal of Experimental Algorithmics (JEA) **10**, (2005-2006) Article No. 2.05
18. Holzer, M., Schulz, F., Wagner, D.: Engineering multi-level overlay graphs for shortest-path queries. In Raman, R., Stallmann, M., eds.: Proc. Algorithm Engineering and Experiments (ALENEX'06), SIAM (2006) 156–170
19. Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. Journal of the ACM (JACM) **24** (1977) 1–13
20. Kaindl, H., Kainz, G.: Bidirectional heuristic search reconsidered. Journal of Artificial Intelligence Research **7** (1997) 283–317
21. Karypis, G.: METIS: Family of multilevel partitioning algorithms. http://www-users.cs.umn.edu/ karypis/metis/ (1995)
22. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path computation. In Nikoletseas, S.E., ed.: Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005. Volume 3503 of LNCS., Springer (2005) 126–138
23. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In Raubal, M., Sliwinski, A., Kuhn, W.,

eds.: Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung. Volume 22 of IfGI prints., Institut für Geoinformatik, Münster (2004) 219–230
24. Luby, M., Ragde, P.: A bidirectional shortest-path algorithm with good average-case behavior. Algorithmica **4** (1989) 551–567
25. Mehlhorn, K., Näher, S.: LEDA, A platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
26. Meyer, U.: Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds. Journal of Algorithms **48** (2003) 91–134
27. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graph to speed up dijkstra's algorithm. In Nikoletseas, S.E., ed.: Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005. Volume 3503 of LNCS., Springer (2005) 189–202; Journal version to appear in ACM Journal on Experimental Algorithmics (JEA), **12** (2006).
28. Müller-Hannemann, M., Schulz, F., Wagner, D., Zaroliagis, C.: Timetable information: Models and algorithms. In: Geraets, F., Kroon, L., Schöbel, A., Wagner, D., Zaroliagis, C.: Algorithmic Methods for Railway Optimization, LNCS, to appear.
29. Müller-Hannemann, M., Weihe, K.: Pareto shortest paths is often feasible in practice. In Brodal, G., Frigioni, D., Marchetti-Spaccamela, A., eds.: Proc. 5th Workshop on Algorithm Engineering (WAE'01). Volume 2141 of LNCS., Springer (2001) 185–197
30. Pohl, I.: Bi-directional and heuristic search in path problems. Technical Report 104, Stanford Linear Accelerator Center, Stanford, California (1969)
31. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In Brodal, G.S., Leonardi, S., eds.: Proc. Algorithms ESA 2005: 13th Annual European Symposium. Volume 3669 of LNCS., Springer (2005) 568–579
32. Sanders, P., Schultes, D.: Engineering Highway hierarchies. In Assar, Y., Erlebach, T., eds.: Proc. Algorithms ESA 2006: 14th Annual European Symposium. Volume 4168 of LNCS., Springer (2006)
33. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. ACM Journal of Experimental Algorithmics (JEA) **5**, (2000) Article No. 12.
34. Schulz, F., Wagner, D., Zaroliagis, C.: Using Multi-Level Graphs for Timetable Information. In: Mount, D. M., Stein, C. eds.: Proc. 4th Workshop Algorithm Engineering and Experiments (ALENEX'02). Volume 2409 of LNCS., Springer (2002) 43–59
35. Sedgewick, R., Vitter, J.S.: Shortest paths in Euclidean space. Algorithmica **1** (1986) 31–48
36. Wagner, D., Willhalm, T.: Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In Di Battista, G., Zwick, U., eds.: Proc. Algorithms ESA 2003: 11th Annual European Symposium on Algorithms. Volume 2832 of LNCS., Springer (2003), 776–787
37. Wagner, D., Willhalm, T.: Drawing graphs to speed up shortest-path computations. In: Joint Proc. 7th Workshop Algorithm Engineering and Experiments (ALENEX 2005) and 2nd Workshop Analytic Algorithmics and Combinatorics (ANALCO 2005) 15–22
38. Wagner, D., Willhalm, T., Zaroliagis, C.: Geometric shortest path containers. ACM Journal on Experimental Algorithmics (JEA) **10**, (2005-2006) Article No. 1.03