View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KITopen

Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm*

Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany,
{rbauer,delling,sanders,schief,schultes,wagner}@ira.uka.de

Abstract. In [1], basic speed-up techniques for Dijkstra's algorithm have been combined. The key observation in their work was that it is most promising to combine *hierarchical* and *goal-directed* speed-up techniques. However, since its publication, impressive progress has been made in the field of speed-up techniques for Dijkstra's algorithm and huge data sets have been made available.

Hence, we revisit the systematic combination of speed-up techniques in this work, which leads to the fastest known algorithms for various scenarios. Even for road networks, which have been worked on heavily during the last years, we are able to present an improvement in performance. Moreover, we gain interesting insights into the behavior of speed-up techniques when combining them.

1 Introduction

Computing shortest paths in a graph $G = (V, E)$ is used in many real-world applications like route planning in road networks, timetable information for railways, or scheduling for airplanes. In general, Dijkstra's algorithm [2] finds a shortest path of length $d(s, t)$ between a given source s and target t. Unfortunately, the algorithm is far too slow to be used on huge datasets. Thus, several speed-up techniques have been developed (see [3] for an overview) yielding faster query times for typical instances, e.g., road or railway networks. In [1], basic speed-up techniques have been combined systematically. One key observation of their work was that it is most promising to combine hierarchical and goal-directed techniques. However, since the publication of [1], many powerful hierarchical speed-up techniques have been developed, goal-directed techniques have been improved, and huge data sets have been made available to the community. In this work, we revisit the systematic combination of speed-up techniques.

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

1.1 Related Work

Since there is an abundance of related work, we decided to concentrate on previous combinations of speed-up techniques and on the approaches that our work is directly based on.

Bidirectional Search executes Dijkstra's algorithm simultaneously forwards from the source s and backwards from the target t. Once some node has been visited from both directions, the shortest path can be derived from the information already gathered [4]. Many more advanced speed-up techniques use bidirectional search as an optional or sometimes even mandatory ingredient.

Hierarchical Approaches try to exploit the hierarchical structure of the given network. In a preprocessing step, a hierarchy is extracted, which can be used to accelerate all subsequent queries.

Reach. Let $R(v) := \max R_{st}(v)$ denote the *reach* of node v, where $R_{st}(v) := \min(d(s,v),\, d(v,t))$ for all s-t shortest paths including v. Gutman [5] observed that a shortest-path search can be pruned at nodes with a reach too small to get to either source or target from there. The basic approach was considerably strengthened by Goldberg et al. [6], in particular by a clever integration of *shortcuts* [3], i.e., single edges that represent whole paths in the original graph.

Highway-Node Routing [3] computes for a given sequence of node sets $V =: V_0 \supseteq V_1 \supseteq \ldots \supseteq V_L$ a hierarchy of *overlay graphs* [7, 8]: the level-ℓ overlay graph consists of the node set V_ℓ and an edge set E_ℓ that ensures the property that all distances between nodes in V_ℓ are equal to the corresponding distances in the underlying graph $G_{\ell-1}$. A bidirectional query algorithm takes advantage of the multi-level overlay graph by never moving downwards in the hierarchy—by that means, the search space size is greatly reduced. The most recent variant of HNR [9], Contraction Hierarchies, obtains a node classification by iteratively contracting the 'least important' node, yielding a hierarchy with up to $|V|$ levels. Moreover, the input graph G is transfered to a search graph G' by storing only edges directing from unimportant to important nodes. As a remarkable result, G' is *smaller* than G yielding a *negative* overhead per node. Finally, by this transformation the query is simply a plain bidirectional Dijkstra operating on G'.

Transit-Node Routing [10] is based on a simple observation intuitively used by humans: When you start from a source node s and drive to somewhere 'far away', you will leave your current location via one of only a few 'important' traffic junctions, called (forward) *access nodes* $\overrightarrow{A}(s)$. An analogous argument applies to the target t, i.e., the target is reached from one of only a few backward access nodes $\overleftarrow{A}(t)$. Moreover, the union of all forward and backward access nodes of all nodes, called *transit-node set* $\mathcal{T}$, is rather small. This implies that for each node

the distances to/from its forward/backward access nodes and for each transit-node pair (u, v) the distance between u and v can be stored. For given source and target nodes s and t, the length of the shortest path that passes at least one transit node is given by $d_{\mathcal{T}}(s,t) = \min\{d(s,u) + d(u,v) + d(v,t) \mid u \in \overrightarrow{A}(s), v \in \overleftarrow{A}(t)\}$. As a final ingredient, a *locality filter* $\mathcal{L} : V \times V \to \{\mathsf{true}, \mathsf{false}\}$ is needed that decides whether given nodes s and t are too close to travel via a transit node. $\mathcal{L}$ has to fulfill the property that $\mathcal{L}(s,t) = \mathsf{false}$ implies $d(s,t) = d_{\mathcal{T}}(s,t)$. Then, the following algorithm can be used to compute the shortest-path length $d(s,t)$:

if $\mathcal{L}(s,t) = \mathsf{false}$ **then** compute and return $d_{\mathcal{T}}(s,t)$; **else** use any other routing algorithm.

Note that for a given source-target pair (s,t), let $a := \max(|\overrightarrow{A}(s)|, |\overleftarrow{A}(t)|)$. For a global query (i.e., $\mathcal{L}(s,t) = \mathsf{false}$), we need $O(a)$ time to lookup all access nodes, $O(a^2)$ to perform the table lookups, and $O(1)$ to check the locality filter.

Goal-Directed Approaches direct the search towards the target t by preferring edges that shorten the distance to t and by excluding edges that cannot possibly belong to a shortest path to t—such decisions are usually made by relying on preprocessed data.

ALT [11] is based on $\underline{A}^*$ search, *Landmarks*, and the Triangle inequality. After selecting a small number of nodes, called landmarks, for all nodes v, the distances $d(v,\lambda)$ and $d(\lambda,v)$ to and from each landmark λ are precomputed. For nodes v and t, the triangle inequality yields for each landmark λ two lower bounds $d(\lambda,t) - d(\lambda,v) \le d(v,t)$ and $d(v,\lambda) - d(t,\lambda) \le d(v,t)$. The maximum of these lower bounds is used during an A^* search. The original ALT approach has fast preprocessing times and provides reasonable speed-ups, but consumes too much space for very large networks. In the subsequent paragraph on "Previous Combinations", we will see that there is a way to reduce the memory consumption by storing landmark distances only for a subset of the nodes.

Arc-Flags. The arc-flag approach, introduced in [12], first computes a partition $\mathcal{C}$ of the graph. A *partition* of V is a family $\mathcal{C} = \{C_0, C_1, \ldots, C_k\}$ of sets $C_i \subseteq V$ such that each node $v \in V$ is contained in exactly one set C_i. An element of a partition is called a *cell*. Next, a *label* is attached to each edge e. A label contains, for each cell $C_i \in \mathcal{C}$, a flag $AF_{C_i}(e)$ which is true if a shortest path to a node in C_i starts with e. A modified Dijkstra then only considers those edges for which the flag of the target node's cell is true. The big advantage of this approach is its easy and fast query algorithm. However, preprocessing is very expensive, either regarding preprocessing time or memory consumption [13].

Previous Combinations. Many speed-up techniques can be combined. In [7], a combination of a special kind of geometric container [14], the separator-based multi-level method [8], and A^* search yields a speed-up of 62 for a railway

transportation problem. In [1], combinations of A^* search, bidirectional search, the separator-based multi-level method, and geometric containers are studied: Depending on the graph type, different combinations turn out to be best.

REAL. Goldberg et al. [6] have successfully combined their advanced version of REach with landmark-based A^* search (the ALt algorithm), obtaining the REAL algorithm. In the most recent version, they introduce a variant where landmark distances are stored only with the more important nodes, i.e., nodes with high reach values. By this means, the memory consumption can be reduced.

*HH** [15] combines highway hierarchies [16] (HH) with landmark-based A* search. Similar to [6], the landmarks are not chosen from the original graph, but for some level k of the highway hierarchy, which reduces the preprocessing time and memory consumption. As a result, the query works in two phases: in an initial phase, a non-goal-directed highway query is performed until all entrance points to level k have been discovered; for the remaining search, the landmark distances are available so that the combined algorithm can be used.

SHARC [17] extends and combines ideas from highway hierarchies (namely, the contraction phase, which produces SHortcuts) with the ARC flag approach. The result is a fast *unidirectional* query algorithm, which is advantageous in scenarios where bidirectional search is prohibitive. In particular, using an approximative variant allows dealing with time-dependent networks efficiently. Even faster query times can be obtained when a bidirectional variant is applied.

1.2 Our Contributions

In this work, we study a systematic combination of speed-up techniques for Dijkstra's algorithm. However, we observed in [18] that some combinations are more promising than others. Hence, we focus on the most promising ones: adding goal-direction to hierarchical speed-up techniques. By evaluating different inputs and scenarios, we gain interesting insights into the behavior of speed-up techniques when combining them. As a result, we are able to present the fastest known techniques for several scenarios. For sparse graphs, a combination of Highway-Node Routing and Arc-Flags yields excellent speed-ups with low preprocessing effort. The combination is only overtaken by Transit-Node Routing in road networks with travel times, but the gap is almost closed. However, even Transit-Node Routing can be further accelerated by adding goal-direction. Moreover, we introduce a hierarchical ALT algorithm, called CALT, that yields a good performance on denser graphs. Finally, we reveal interesting observations when combining Arc-Flags with Reach.

We start our work on combinations in Section 2 by presenting a generic approach how to improve the performance of basic speed-up techniques in general. The key observation is that we extract an important subgraph, called the *core*, of the input graph and use only the core as input for the preprocessing-routine of the applied speed-up technique. As a result, we derive a two-phase query

algorithm, similar to partial landmark REAL or HH*. During phase 1 we use plain Dijkstra to reach the core, while during phase 2, we use a speed-up technique in order to accelerate the search within the core. The full power of this *core-based routing* approach can be unleashed by using a goal-directed technique during phase 2. Our experimental study in Section 5 shows that when using ALT during phase 2, we end in a very robust technique that is superior to plain ALT.

In Section 3, we show how to remedy the crucial drawback of Arc-Flags: its preprocessing effort. Instead of computing arc-flags on the full graph, we use a purely hierarchical method until a specific point during the query. As soon as we have reached an 'important' subgraph, i.e., a high level within the hierarchy, we turn on arc-flags. As a result, we significantely accelerate hierarchical methods like Highway-Node Routing. Our aggressive variant moderately increases preprocessing effort but query performance is almost as good as Transit-Node Routing in road networks: On average, we settle only 45 nodes for computing the distance between two random nodes in a continental road network. The advantage of this combination over Transit-Node Routing is its very low space consumption.

However, we are also able to improve the performance of Transit-Node Routing. In Section 4, we present how to add goal-direction to this approach. As a result, the number of required table lookups can be reduced by a factor of 13, resulting in average query times of less than 2 μs—more than three million times faster than Dijkstra's algorithm.

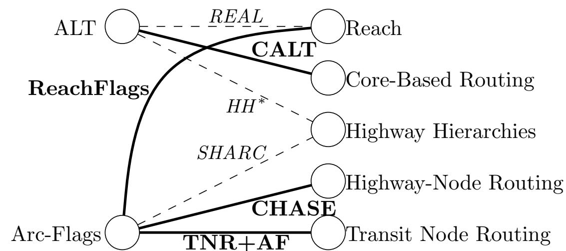

Fig. 1. Overview of combinations of speed-up techniques. Speed-up techniques are drawn as nodes (goal-directed techniques on the left, hierarchical on the right). A dashed edge indicates an existing combination, whereas thick edges indicate combinations presented in this work.

As already mentioned, a few combinations like HH*, REAL, and SHARC have already been published. Hence, Figure 1 provides an overview over existing combinations already published and those which are presented in this work. Note that all techniques in this work use bidirectional search. Also note that due to space limitations all proofs of correctness are skipped but will be included in the full paper.

2 Core-Based Routing

In this section, we introduce a very easy and powerful approach to generally reduce the preprocessing of the speed-up techniques introduced in Section 1. The central idea is to use contraction [9] to extract an important subgraph and preprocess only this subgraph instead of the full graph.

Preprocessing. At first, the input graph $G = (V, E)$ is contracted to a graph $G_C = (V_C, E_C)$, called the *core*. Note that we could use any contraction routine, that removes nodes from the graph and inserts edges to preserve distances between core nodes. Examples are those from [16, 6, 17] or the most advanced one from [9]. The key idea of core-based routing is not to use G as input for preprocessing but to use G_C instead. As a result, preprocessing of most techniques can be accelerated as the input can be shrunk. However, sophisticated methods like Highway Hierarchies, REAL, or SHARC already use contraction during preprocessing. Hence, this advantage especially holds for goal-directed techniques like ALT or Arc-Flags. After preprocessing the core, we store the preprocessed data and merge the core and the normal graph to a full graph $G_F = (V, E_F = E \cup E_C)$. Moreover, we mark the core-nodes with a flag.

Query. The s-t query is a modified bidirectional Dijkstra, consisting of two phases and performed on G_F. During phase 1, we search the graph search until all *entrance points* of s and t are found (cf. [15] for details). We identify a superset of those nodes by the following approach. We run a bidirectional Dijkstra rooted at s and t *not* relaxing edges belonging to the core. We add each core node settled by the forward search to a set S (T for the backward search). The first phase terminates if one of the following two conditions hold: (1) either both priority queues are empty or (2) the distance to the closest entry points of s and t is larger than the length of the tentative shortest path. If case (2), the whole query terminates. The second phase is initialized by refilling the queues with the nodes belonging to S and T. As key we use the distances computed during phase 1. Afterwards, we execute the query-algorithm of the applied speed-up technique which terminates according to its stopping condition.

CALT. Although we could use *any* of the speed-up techniques to instantiate our core-based approach we focus on a variant based on ALT due to the following reasons. First of all, ALT works well in *dynamic* scenarios. As contraction seems easy to dynamize, we are optimistic that CALT (Core-ALT) also works well in dynamic scenarios. Second, pure ALT is a very robust technique with respect to the input. Finally, ALT suffers from the critical drawback of high memory consumption—we have to store two distances per node and landmark—which can be reduced by switching to CALT.

On top of the preprocessing of the generic approach, we compute landmarks on the core and store the distances to and from the landmarks for all core nodes. However, ALT needs lower bounds for *all* nodes to the source and target. As we do not store distances from all nodes to the landmarks, we need *proxy nodes*, which were introduced for the partial REAL algorithm in [6]. The method developed there can directly be applied to CALT: The proxy s' of a node s is the core node closest to s. We compute these proxy nodes for a given s–t query during the initialization phase of the first phase of the query. During the second phase we use the landmark information for the core in order to speed-up the query within the core.

3 Hierarchy-Aware Arc-Flags

Two goal-directed techniques have been established during the last years: ALT and Arc-Flags. The advantages of ALT are fast preprocessing and easy adaption to dynamic scenarios, while the latter is superior with respect to query-performance and space consumption. However, preprocessing of Arc-Flags is expensive. The central idea of *Hierarchy-Aware Arc-Flags* is to combine—similar to REAL or HH*—a hierarchical method with Arc-Flags. By computing arc-flags only for a subgraph containing all nodes in high levels of the hierarchy, we are able to reduce preprocessing times. In general, we could use any hierarchical approach but as Contraction Hierarchies (CH) is the hierarchical method with lowest space consumption, we focus on the combination of Contraction Hierarchies and Arc-Flags. However, we also present a combination of Reach and Arc-Flags.

3.1 Contraction Hierarchies + Arc-Flags (CHASE)

As already mentioned in Section 1.1, Contraction Hierarchies is basically a plain bidirected Dijkstra on a search graph constructed during preprocessing. We are able to combine Arc-Flags and Contraction Hierarchies in a very natural way and name it the CHASE-algorithm (Contraction-Hierarchy + Arc-flagS + highway-nodE routing).

Preprocessing. First, we run a complete Contraction Hierarchies preprocessing which assembles the search graph G'. Next, we extract the subgraph H of G' containing the $|V_H|$ nodes of highest levels. The size of V_H is a tuning parameter. Recall thatContraction Hierarchies uses $|V|$ levels with the most important node in level $|V|-1$. We partition H into k cells and compute arc-flags according to [13] for all edges in H. Summarizing, the preprocessing consists of constructing the search graph and computing arc-flags for H.

Query. Basically, the query is a two-phase algorithm. The first phase is a bidirected Dijkstra on G' with the following modification: When settling a node v belonging to H, we do *not* relax any outgoing edge from v. Instead, if v is settled by the forward search, we add v to a node set S, otherwise to T. Phase 1 ends if the search in both directions stops. The search stops in one direction, if either the respective priority queue is empty or if the minimum of the key values in that queue and the distance to the closest entrance point in that direction is equal or larger than the length of the tentative shortest path. The whole search can be stopped after the first phase, if either no entrance points have been found in one direction or if the tentative shortest-path distance is smaller than minimum over all distances to the entrance points and all key values remaining in the queues. Otherwise we switch to phase 2 of the query which we initialize by refilling the queues with the nodes from S and T. As keys we use the distances computed during phase 1. In phase 2, we use a bidirectional Arc-Flags Dijkstra.

We identify the set C_S (C_T) of all cells that contain at least one node $u \in S$ ($u \in T$). The forward search only relaxes edges having a true arc-flag for any of the cells C_T. The backward search proceeds analogously. Moreover, we use the CH stopping criterion and the strict alternating strategy for forward and backward search. However, during our experimental study, it turned out that *stall-on-demand* [3], which accelerates pure CH, does not pay off for CHASE. The computational overhead is too high which is not compensated by the slight decrease in search space. So, the resulting query is a plain bidirectional Dijkstra operating on G' with the CH stopping criterion and arc-flags activated on high levels of the hierarchy.

Note that we have a trade-off between performance and preprocessing. If we use bigger subgraphs as input for preprocessing arc-flags, query-performance is better as arc-flags can be used earlier. However, preprocessing time increases as more arc-flags have to be computed.

3.2 Reach + Arc-Flags (ReachFlags)

Similar to CHASE, we can also combine Reach and Arc-Flags, called *Reach-Flags*. However, we slightly alter the preprocessing: Reach-computation according to [6] is a process that iteratively contracts and prunes the input. This iteration can be interpreted as levels of a hierarchy: A node u belongs to level i if u is still part of the graph during iteration step i. With this notion of hierarchy, we are able to preprocess ReachFlags. We first run a complete Reach-preprocessing as described in [6] and assemble the output graph. Next, we extract a subgraph H from the output graph containing all nodes of level $\geq \ell$. Again, we compute arc-flags in H according to [13]. The ReachFlags-query can easily by adapted from the CHASE-query in straight-forward manner. Note that the input parameter ℓ adjusts the size of V_H. Thus, a similar trade-off in performance/preprocessing effort like for CHASE is given.

4 Transit-Node Routing + Arc-Flags (TNR+AF)

Recall that the most time-consuming part of a TNR-query are the table lookups. Hence, we want to further improve the average query times, the first attempt should be to reduce the number of those lookups. This can be done by excluding certain access nodes at the outset, using an idea very similar to the arc-flag approach. We consider the minimal overlay graph $G_{\mathcal{T}} = (\mathcal{T}, E_{\mathcal{T}})$ of G, i.e., the graph with (transit) node set $\mathcal{T}$ and an edge set $E_{\mathcal{T}}$ such that $|E_{\mathcal{T}}|$ is minimal and for each node pair $(s,t) \in \mathcal{T} \times \mathcal{T}$, the distance from s to t in G corresponds to the distance from s to t in $G_{\mathcal{T}}$. We partition this graph $G_{\mathcal{T}}$ into k regions and store for each node $u \in \mathcal{T}$ its region $r(u) \in \{1, \ldots, k\}$. For each node s and each access node $u \in \overrightarrow{A}(s)$, we manage a flag vector $f_{s,u}^{\rightarrow} : \{1, \ldots, k\} \rightarrow \{\mathsf{true}, \mathsf{false}\}$ such that $f_{s,u}^{\rightarrow}(x)$ is true iff there is a node $v \in \mathcal{T}$ with $r(v) = x$ such that $d(s,u) + d(u,v)$ is equal to $\min\{d(s,u') + d(u',v) \mid u' \in \overrightarrow{A}(s)\}$. In other words, a flag of an access node u for a particular region x is set to true iff u is useful to get

to some transit node in the region x when starting from the node s. Analogous flag vectors $f_{t,u}^{\leftarrow}$ are kept for the backward direction.

Preprocessing. The flag vectors can be precomputed in the following way, again using ideas similar to those used in the preprocessing of the arc-flag approach: Let $B \subseteq \mathcal{T}$ denote the set of border nodes, i.e., nodes that are adjacent to some node in $G_{\mathcal{T}}$ that belongs to a different region. For each node $s \in V$ and each border node $b \in B$, we determine the access nodes $u \in \overrightarrow{A}(s)$ that minimize $d(s,u) + d(u,b)$; we set $f_{s,u}^{\rightarrow}(r(b))$ to true. In addition, $f_{s,u}^{\rightarrow}(r(u))$ is set to true for each $s \in V$ and each access node $u \in \overrightarrow{A}(s)$ since each access node obviously minimizes the distance to itself. An analogous preprocessing step has to be done for the backward direction.

Query. In a query from s to t, we can take advantage of the precomputed flag vectors. First, we consider all backward access nodes of t and build the flag vector f_t such that $f_t(r(u)) = \mathsf{true}$ for each $u \in \overleftarrow{A}(t)$. Second, we consider only forward access nodes u of s with the property that the bitwise AND of $f_{s,u}^{\rightarrow}$ and f_t is not zero; we denote this set by $\overrightarrow{A}'(s)$; during this step, we also build the vector f_s such that $f_s(r(u)) = \mathsf{true}$ for each $u \in \overrightarrow{A}'(s)$. Third, we use f_s to determine the subset $\overleftarrow{A}'(t) \subseteq \overleftarrow{A}(t)$ analogously to the second step. Now, it is sufficient to perform only $|\overrightarrow{A}'(s)| \times |\overleftarrow{A}'(t)|$ table lookups. Note that determining $\overrightarrow{A}'(s)$ and $\overleftarrow{A}'(t)$ is in $O(a)$, in particular operations on the flag vectors can be considered as quite cheap.

Optimizations. Presumably, it is a good idea to just store the bitwise OR of the forward and backward flag vectors in order to keep the memory consumption within reasonable bounds. The preprocessing of the flag vectors can be accelerated by rearranging the columns of the distance table so that all border nodes are stored consecutively, which reduces the number of cache misses.

5 Experiments

In this section, we present an extensive experimental evaluation of our combined speed-up techniques in various scenarios and inputs. Our implementation is written in C++ (using the STL at some points). As priority queue we use a binary heap. The evaluation was done on two similar machines: An AMD Opteron 2218[1] and an Opteron 270[2]. The second machine is used for the combination

[1] The machine runs SUSE Linux 10.1, is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The DIMACS benchmark on the full US road network with travel time metric takes 6 013.6 s.

[2] SUSE Linux 10.0, 2.0 GHz, 8 GB of RAM, and 2 x 1 MB of L2 cache. The DIMACS benchmark: 5 355.6 s.

of Transit-Node Routing and Arc-Flags, the first one for all other experiments. Note that the second machine is roughly 10% faster than the first one due to faster memory. All figures in this paper are based on 10 000 random s-t queries and refer to the scenario that only the lengths of the shortest paths have to be determined, without outputting a complete description of the paths. Efficient techniques for the latter have been published in [15, 19].

5.1 Road Networks

As inputs we use the largest strongly connected component[3] of the road networks of Western Europe, provided by PTV AG for scientific use, and of the US which is taken from the DIMACS Challenge homepage.The former graph has approximately 18 million nodes and 42.6 million edges. The corresponding figures for the USA are 23.9 million and 58.3 million, respectively. In both cases, edge lengths correspond to travel times. For results on the distance metric, see Tab. 4 in Appendix A.

CALT. In [20], we were able to improve query performance of ALT over [11] by improving the organization of landmark data. However, we do not compress landmark information and use a slightly better heuristic for landmark[4] selection. Hence, we report both results. By adding contraction—we use the one from [17] with $c = 3.0$ and $h = 30$—to ALT, we are able to reduce query time to 2.0 ms for Europe and to 4.9 ms for the US. This better performance is due to two facts. On the one hand, we may use more landmarks (we use 64) and on the other hand, the contraction reduces the number of hops of shortest paths. The latter observation is confirmed by the figures of CALT with 16 landmarks. Moreover, the most crucial drawback of ALT—memory consumption—can be reduced to a reasonable amount, even when using 64 landmarks. Still, CALT cannot compete with REAL or pure hierarchical methods, but the main motivation for CALT is its presumably easy dynamization.

CHASE. We report the figures for two variants of CHASE: the *economical* variant computes arc-flags only for a subgraph of 0.5% size of the input while for the *generous* variant, the subgraph H has a size of 5% of the input (with respect to number of nodes). We partition H with SCOTCH [21] into 128 cells.

For Europe, the economical variant only needs 7 additional minutes of preprocessing over pure CH and the preprocessed data is still smaller than the input. Recall that a negative overhead derives from the fact that the search graph is smaller than the input, see Section 1.1. This economical variant is already roughly 4 times faster than pure CH. However, by increasing the size of

[3] For historical reasons, some quoted results are based on the respective original network that contains a few additional nodes that are not connected to the largest strongly connected component.

[4] 16 landmarks are generated by the maxCover algorithm, 64 are generated by avoid [11]

Table 1. Overview of the performance of various speed-up techniques, grouped by (1.) hierarchical methods [Highway Hierarchies (HH), highway-node routing based on HH (HH-HNR) and on Contraction Hierarchies (CH-HNR), Transit-Node Routing (TNR)], (2.) goal-directed methods [landmark-based A^* search (ALT), Arc-Flags (AF)], (3.) previous combinations, and (4.) the new combinations introduced in this paper. The additional overhead is given in bytes per node in comparison to *bidirectional* Dijkstra. Preprocessing times are given in minutes. Query performance is evaluated by the average number of settled nodes and the average running time of 10 000 random queries.

		Europe				USA			
		PREPRO.		QUERY		PREPRO.		QUERY	
method		time [min]	overhead [B/node]	#settled nodes	time [ms]	time [min]	overhead [B/node]	#settled nodes	time [ms]
Reach	[6]	83	17	4 643	3.47	44	20	2 317	1.81
HH	[3]	13	48	709	0.61	15	34	925	0.67
HH-HNR	[3]	15	2.4	981	0.85	16	1.6	784	0.45
CH-HNR	[9]	25	-2.7	355	0.18	27	-2.3	278	0.13
TNR	[19]	164	251	N/A	0.0056	205	244	N/A	0.0049
TNR	[9]	112	204	N/A	0.0034	90	220	N/A	0.0030
ALT-a16	[6]	13	70	82 348	160.3	19	89	187 968	400.5
ALT-m16	[20]	85	128	74 669	53.6	103	128	180 804	129.3
AF	[13]	2 156	25	1 593	1.1	1 419	21	5 522	3.3
REAL	[6]	141	36	679	1.11	121	45	540	1.05
HH*	[3]	14	72	511	0.49	18	56	627	0.55
SHARC	[17]	192	20	145	0.091	158	21	350	0.18
CALT-m16	2	16	8	3 017	3.9	26	8	7 079	8.3
CALT-a64	2	14	20	1 394	2.0	21	19	3 240	4.9
CHASE eco	3.1	32	0.0	111	0.044	36	-0.8	127	0.049
CHASE gen	3.1	99	12	45	0.017	228	11	49	0.019
ReachFlags	3.2	229	30	1 168	0.76	318	25	1 636	1.02
TNR+AF	4	229	321	N/A	0.0019	157	263	N/A	0.0017

the subgraph H used as input for arc-flags, we are able to almost close the gap to pure Transit-Node Routing. CHASE is only 5 times slower than TNR (and is even *faster* than the grid-based approach of TNR [19]). However, the preprocessed data is much smaller for CHASE, which makes it more practical in environments with limited memory. Using the distance metric (cf. Tab. 4 in Appendix A), the gap between CHASE and TNR can be reduced even further. Remarkably, both pure Arc-Flags and CH perform much worse on distances than on travel times, whereas the combination CHASE performs—with respect to queries—very similarly on both metrics.

Size of the Subgraph. The combination of Contraction Hierarchies and Arc-Flags allows a very flexible trade-off between preprocessing and query performance. The bigger the subgraph H used as input for Arc-Flags, the longer preprocessing takes but query performance decreases. Table 2 reports the performance of CHASE for different sizes of H in percentage of the original graph. Recall

Table 2. Performance of CHASE for Europe with stall-on-demand turned on and off running 10 000 random queries.

	size of H	0.0%	0.5%	1.0%	2.0%	5.0%	10.0%	20.0%
Prepro.	time [min]	25	31	41	62	99	244	536
	space [Byte/n]	-2.7	0.0	1.9	4.9	12.1	22.2	39.5
Query	# settled	355	86	67	54	43	37	34
(with s-o-d)	time [μs]	180.0	48.5	36.3	29.2	22.8	19.7	17.2
Query	# settled	931	111	78	59	45	39	35
(without s-o-d)	time [μs]	286.3	43.8	30.8	23.1	17.3	14.9	13.0

that 0.5% equals our economical variant, while 5% corresponds to the generous variant.

Two observations are remarkable: the effect of stall-on-demand (→ Section 3.1) and the size of the subgraphs. While stall-on-demand pays off for pure CH, CHASE does not win from turning on this optimization. The number of settled nodes decreases but due to the overhead query times increase. Another very interesting observation is the influence of the input size for arc-flags. Applying goal-direction on a very high level of the hierarchy speeds up the query significantly. Increasing the size of H to 10% or even 20% yields a much higher preprocessing effort (both space and time) but query performance decreases only slightly, compared to 5%. However, our fastest variant settles only 35 nodes on average having query times of 13 μs. Note that for this input, the average shortest path in its contracted form consists of 22 nodes, so only 13 unnecessary nodes are settled on average.

ReachFlags. We use $l = 2$ to determine the sub-graph for arc-flags preprocessing (cf. Section 3.2). We observe that it does not pay off to combine Arc-Flags—instead of landmarks—with REAL. Although query times are slightly faster than REAL, the search space is higher. One reason might be that our choice of parameters for Reach yield an increase in search space by roughly 20% compared to [6]. Still, it seems as if ReachFlags is inferior to CHASE which is mainly due to the good performance of Contraction Hierarchies.

TNR+AF. The fastest variant of Transit-node Routing *without* using flag vectors is presented in [9]; the corresponding figures are quoted in Tab. 1. For this variant, we computed flag vectors according to Section 4 using $k = 48$ regions. This takes, in the case of Europe, about two additional hours and requires 117 additional bytes per node. Then, the average query time is reduced to as little as 1.9 μs, which is an improvement of almost factor 1.8 (factor 2.9 compared to our first publication in [19]) and a speed-up compared to Dijkstra's algorithm of more than factor 3 *million*. The results for the US are even better.

The improved running times result from the reduced number of table accesses: in the case of Europe, on average only 3.1 entries have to be looked up instead of 40.9 when no flag vectors are used. Note that the runtime improvement is

Table 3. Performance of bidirectional Dijkstra, ALT, CALT, CH, and economical CHASE on unit disk graphs with different average degree and grid graphs with different number of dimensions. Note that the we use the *aggressive* variant of Contraction Hierarchies, better results may be achieved by better input parameters.

	Prepro		Query	Prepro		Query	Prepro		Query
	time [s]	space [B/n]	#settled nodes	time [s]	space [B/n]	#settled nodes	time [s]	space [B/n]	#settled nodes
unit disk	average degree 5			average degree 7			average degree 10		
bidir. Dijkstra	0	0	299 077	0	0	340 801	0	0	325 803
ALT-m16	490	128	10 051	514	128	10 327	566	128	11 704
CALT-m16	34	2	726	166	13	927	658	62	2 523
CALT-a64	32	7	689	135	29	670	511	137	992
CH-HNR	94	-13	236	1 249	-11	1 089	34 274	-4	2 475
CHASE	103	-12	66	1 368	-7	424	34 847	6	1 457
grid	2-dimensional			3-dimensional			4-dimensional		
bidir. Dijkstra	0	0	79 962	0	0	45 269	0	0	21 763
ALT-m16	65	128	2 362	100	128	1 759	133	128	1 335
CALT-m16	113	98	798	202	165	1 057	171	142	1 275
CALT-a64	60	211	458	101	386	557	129	487	774
CH-HNR	70	0	418	13 567	14	2 177	133 734	29	14 501
CHASE	73	2	274	13 585	22	2 836	133 741	32	30 848
railways	Berlin/Brandenburg			Ruhrgebiet			long distance		
bidir. Dijkstra	0	0	1 299 830	0	0	1 134 420	0	0	609 352
ALT-m16	604	128	56 404	556	128	60 004	291	128	30 021
CALT-m16	174	18	4 622	377	32	7 107	158	29	3 335
CALT-a64	123	45	2 830	191	68	4 247	87	63	2 088
CH-HNR	1 636	0	416	2584	4	546	486	3	376
CHASE	2 008	2	125	2863	7	244	536	5	229

considerably less than a factor of 40.9 / 3.1 = 13.2 though. This is due to the fact that the average runtime also includes looking up the access nodes and dealing with local queries.

5.2 Robustness of Combinations

In the last section we focused on the performance of our combinations on road networks. However, existing combinations of goal-directed and hierarchical methods like REAL or SHARC are very robust to the input. Here, we evaluate our most promising combinations—CALT and CHASE—on various other inputs. We use time-expanded timetable networks[5], synthetic unit disk graphs[6]

[5] 3 networks: local traffic of Berlin/Brandenburg (2 599 953 nodes and 3 899 807 edges), local traffic of the Ruhrgebiet (2 277 812 nodes, 3 416 552 edges), long distance connections of Europe (1 192 736 nodes,1 789 088 edges)

[6] We obtain such graphs by arranging nodes uniformly at random on the plane and connecting nodes with a distance below a given threshold. As metric we use the distance according to the embedding.

(1 000 000 nodes with an average degree of 5, 7, and 10), and grid graphs (2–4 dimensions with each having 250 000 nodes, edge weights picked uniformly at random between 1 and 1000.). The results can be found in Tab. 3.

For almost all inputs it pays off to combine goal-directed and hierarchical techniques. Moreover, CHASE works very well as long as the graph stays somehow sparse, only on denser graphs like 3- and 4-dimensional grids, preprocessing times increase significantly, which is mainly due to the contraction routine. Especially the last 20% of the graph take a long time to contract.

Concerning CALT, we observe that turning on contraction pays off—in most cases—very well: Preprocessing effort gets less with respect to time and space while query performance improves. However, as soon as the graph gets too dense, e.g. 4-dimensional grids, the gain in performance is achieved by a higher amount of preprocessed data. The reason for this is that contraction works worse on dense graphs, thus the core is bigger. Comparing CALT and CHASE, we observe that CHASE works better or very sparse graphs while CALT yields better performance on denser graphs. So, it seems as if for dense graphs, it is better to stop contraction at some point and use a goal-directed technique on the core of the graph.

6 Conclusion

In this work, we systematically combine hierarchical and goal-directed speed-up techniques. As a result we are able to present the fastest algorithms for several scenarios and inputs. For sparse graphs, CHASE yields excellent speed-ups with low preprocessing effort. The algorithm is only overtaken by Transit-Node Routing in road networks with travel times, but the gap is almost closed. However, even Transit-Node Routing can be further accelerated by adding goal-direction. Finally, we introduce CALT yielding a good performance on denser graphs.

However, our study not only leads to faster algorithms but to interesting insights into the behavior of speed-up techniques in general. By combining goal-directed and hierarchical methods we obtain techniques which are very robust to the input. It seems as if hierarchical approaches work best on sparse graphs but the denser a graph gets, the better goal-directed techniques work. By combining both approaches the influence—with respect to performance—of the type of input fades. Hence, we were able to refine the statement given in [1]: Instead of blindly combining goal-directed and hierarchical techniques, our work suggest that for large networks, it pays off to drop goal-direction on lower levels of the hierarchy. Instead, it is better with respect to preprocessing (and query performance) to use goal-direction *only* on higher levels of the hierarchy.

Regarding future work, it may be interesting how the insight stated above can be used for graphs where hierarchical preprocessing fails. One could think of a technique that runs only a hierarchical query during the first phase and the second phase is only a goal-directed search, similar to CALT. For example, we could stop the construction of a contraction hierarchy at some point and apply Arc-Flags or ALT to the remaining core. We are optimistic that such a technique

would even achieve very good results on dense graphs. Another open problem is the dynamization of CALT. We are confident, that CALT is very helpful in scenarios where edge updates occur very frequentely, e.g. dynamic timetable information systems.

Acknowledgments. We would like to thank Riko Jacob for interesting discussions on the combination of Transit-Node Routing and Arc-Flags. Moreover, we thank Robert Geisberger for helping us to use Contraction Hierarchies [9] in our work. He provided his implementation of [9] and some precomputed contraction hierarchies for various networks.

References

1. Holzer, M., Schulz, F., Wagner, D., Willhalm, T.: Combining Speed-up Techniques for Shortest-Path Computations. ACM J. of Exp. Algorithmics **10** (2006)
2. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik **1** (1959) 269–271
3. Schultes, D.: Route Planning in Road Networks. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik (2008)
4. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press (1962)
5. Gutman, R.J.: Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In: Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04), SIAM (2004) 100–111
6. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Better Landmarks Within Reach. In: Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07). Volume 4525 of Lecture Notes in Computer Science., Springer (2007) 38–51
7. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. ACM J. of Exp. Algorithmics **5** (2000)
8. Holzer, M., Schulz, F., Wagner, D.: Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. In: Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06), SIAM (2006)
9. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08). Lecture Notes in Computer Science, Springer (2008)
10. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast Routing in Road Networks with Transit Nodes. Science **316** (2007) 566
11. Goldberg, A.V., Werneck, R.F.: Computing Point-to-Point Shortest Paths from External Memory. In: Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05), SIAM (2005) 26–40
12. Lauther, U.: An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In: Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung. Volume 22. IfGI prints (2004) 219–230
13. Hilger, M.: Accelerating Point-to-Point Shortest Path Computations in Large Scale Networks. Master's thesis, Technische Universität Berlin (2007)

14. Wagner, D., Willhalm, T., Zaroliagis, C.: Geometric Containers for Efficient Shortest-Path Computation. ACM J. of Exp. Algorithmics **10** (2005) 1.3
15. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway Hierarchies Star. In: 9th DIMACS Implementation Challenge - Shortest Paths. (2006)
16. Sanders, P., Schultes, D.: Engineering Highway Hierarchies. In: Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06). Volume 4168 of Lecture Notes in Computer Science., Springer (2006) 804–816
17. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. In: Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08), SIAM (2008) 13–26
18. Schieferdecker, D.: Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries. Master's thesis, Universität Karlsruhe (TH) (2008)
19. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Shortest-Path Queries in Road Networks. In: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07), SIAM (2007) 46–59
20. Delling, D., Wagner, D.: Landmark-Based Routing in Dynamic Graphs. In: Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07). Volume 4525 of Lecture Notes in Computer Science., Springer (2007) 52–65
21. Pellegrini, F.: SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package (2007)

A Further Experiments

Table 4. Overview on the performance of prominent speed-up techniques and combinations analogous to Tab. 1 but with travel distances as metric.

method		Europe PREPRO. time [min]	Europe PREPRO. overhead [B/node]	Europe QUERY #settled nodes	Europe QUERY time [ms]	USA PREPRO. time [min]	USA PREPRO. overhead [B/node]	USA QUERY #settled nodes	USA QUERY time [ms]
Reach	[6]	49	15	7 045	5.53	70	22	7 104	5.97
HH	[3]	32	36	3 261	3.53	38	66	3 512	3.73
CH-HNR	3.1	89	-0.1	1 650	4.19	57	-1.2	953	1.50
TNR	[3]	162	301	N/A	0.038	217	281	N/A	0.086
ALT-a16	[6]	10	70	276 195	530.4	15	89	240 750	430.0
ALT-m16	2	70	128	218 420	127.7	102	128	278 055	166.9
AF	[13]	1 874	33	7 139	5.0	1 311	37	12 209	8.8
REAL	[6]	90	37	583	1.16	138	44	628	1.48
HH*	[3]	33	92	1 449	1.51	40	89	1 372	1.37
SHARC	[17]	156	26	4 462	2.01	-	-	-	-
CALT-m16	2	17	8	6 453	8.1	20	8	9 034	11.0
CALT-a64	2	14	19	2 958	4.2	15	19	4 015	5.6
CHASE eco	3.1	224	7.0	175	0.156	185	2.5	148	0.103
CHASE gen	3.1	1 022	27	67	0.064	1 132	18	63	0.043
ReachFlags	3.2	516	31	5 224	4.05	1 897	27	6 849	4.69