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,schiefer,schultes,wagner}@ira.uka.de

Abstract. In recent years, highly effective *hierarchical* and *goal-directed* speedup techniques for routing in large road networks have been developed. This paper makes a systematic study of combinations of such techniques. These combinations turn out to give the best results in many scenarios, including graphs for unit disk graphs, grid networks, and time-expanded timetables. Besides these quantitative results, we obtain general insights for successful combinations.

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 [16] finds an exact 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 [14] for an overview) yielding faster query times for typical instances, e.g., road or railway networks. In [26, 25], 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 [25], 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.

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 [8]. 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.

* 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 grants SA 933/1-3, SA 933/5-1, WA 654/16-1.

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 with $\min\{\} := \infty$. Gutman [22] 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. [19, 20], in particular by a clever integration of *shortcuts* [31, 32], i.e., single edges that represent whole paths in the original graph.

Highway Hierarchies (HH). In [31, 32], the idea to automatically compute a hierarchy of highway networks is introduced. The basic approach is to define a neighborhood for each node to consist of its H closest neighbors. Now an edge (u, v) is a highway edge if there is some shortest path $\langle s, \ldots, u, v, \ldots t\rangle$ such that neither u is in the neighborhood of t nor v is in the neighborhood of s. After contracting the resulting network to remove low degree nodes, the same procedure is applied recursively. We obtain a hierarchy. The query algorithm is bidirectional Dijkstra with restrictions on relaxing certain edges. Roughly, far away from source or target, only high-level edges need to be considered.

Highway-Node Routing (HNR) [35, 34] computes for a given sequence of node sets $V =: V_0 \supseteq V_1 \supseteq \ldots \supseteq V_L$ a hierarchy of *overlay graphs* [36, 37, 24]: 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 [17], Contraction Hierarchies (CH), 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 transferred 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 search operating on G'.

Transit-Node Routing (TNR) [2, 1] 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 $d_{\mathcal{T}}(s,t) = \min\{d(s,u) + d(u,v) + d(v,t) \mid u \in \overrightarrow{A}(s), v \in \overleftarrow{A}(t)\}$. Note that all involved distances $d(s,u)$, $d(u,v)$, and $d(v,t)$ can be directly looked up in the precomputed data structures. 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.

For a given source-target pair (s, t), let $a := \max(|\overrightarrow{\mathrm{A}}(s)|, |\overleftarrow{\mathrm{A}}(t)|)$. Note that 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. In [18, 21], the ALT algorithm is presented that is based on $\underline{A}^*$ search, *Landmarks*, and the Triangle inequality. After selecting a small number of nodes, called landmarks, the distances $d(v, \lambda)$ and $d(\lambda, v)$ to and from each landmark λ are precomputed for all nodes v. For nodes v and t, the triangle inequality yields for each landmark λ two lower bounds $d(\lambda, t) - d(\lambda, v) \leq d(v, t)$ and $d(v, \lambda) - d(t, \lambda) \leq d(v, t)$. 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 (AF). The arc-flag approach, introduced in [28, 27, 29], 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 iff 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 [23]: in the latter case, a centralized tree is built from each cell keeping the distances to *all* boundary nodes of this cell in memory.

Previous Combinations. Many speed-up techniques can be combined. In [36], a combination of a special kind of geometric container [39], the separator-based multi-level method [37], and A^* search yields a speed-up of 62 for a railway transportation problem. In [26], 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. For real-world graphs, a combination of bidirectional search and geometric containers leads to the best running times. For public transportation however, a combination of Arc-Flags and ALT harmonizes well [11].

REAL. Goldberg et al. [19, 20] 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 [20], 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 significantly.

*HH** [12, 13] combines highway hierarchies [32] (HH) with landmark-based A* search. Similar to [20], 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 [4] 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 [33] that—at least in road networks—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 Contraction Hierarchies 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 more dense graphs. Finally, we make 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. Note that this approach is similar to Section 2. As soon as we have reached an 'important' subgraph (or core), 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.

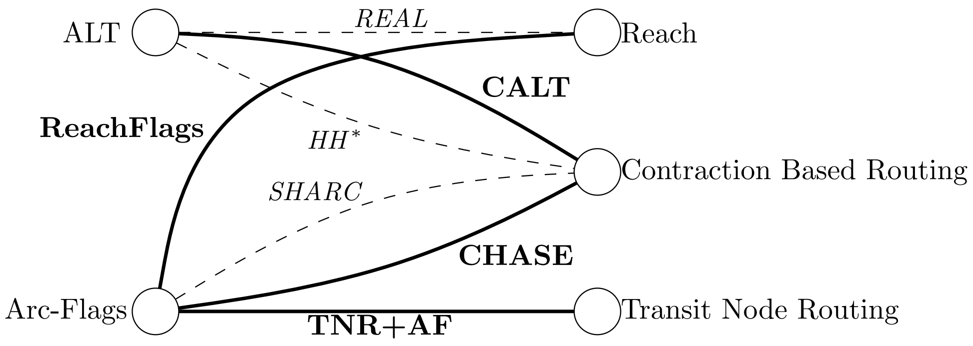

Fig. 1. Overview of combinations of speed-up techniques. Speed-up techniques are drawn as nodes (goal-directed techniques on the left, hierarchical on the right). A dashed edge indicates an existing combination, whereas thick edges indicate combinations presented in this work. Contraction based routing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

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.

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.

An extended abstract of this work has been published in [5]. Here, we additionally give proofs of correctness and introduce new combinations tailored to denser graphs. Moreover, we present a largely extended experimental study.

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 [17] 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 [32, 20, 4] or the most advanced one from [17]. 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.

Contraction. Our contraction routine is inspired by [32, 20] and has been developed for SHARC [4]. We perform two steps during contraction, node- and edge-reduction.

The number of nodes is reduced by iteratively *bypassing* nodes until no node is *bypassable* any more. To bypass a node x we first remove x, its incoming edges I and its outgoing edges O from the graph. Then, for each tail u of I and head v of O we introduce a new edge of the length $len(u, x) + len(x, v)$, where len gives the length of an edge. If there already is an edge connecting u and v in the graph, we only keep the one with smaller length. When a shortcut S represents a path P in the input graph, the *hop number* of S is the number of edges in P. To check whether a node is bypassable we first determine the number $\#shortcut$ of *new* edges that would be inserted into the graph if x was bypassed. Then we say a node is bypassable if our *bypass criterions* are fulfilled First, $\#shortcut \leq c \cdot (|I| + |O|)$ must hold, where c is a tunable *contraction parameter.* Second, bypassing x must not yield a shortcut with a hop number greater than h.

A node being bypassed influences the degree of their neighbors and thus, their bypassability. Therefore, the order in which nodes are bypassed changes the resulting contracted graph. We use a heap to determine the next bypassable node. The key of a node x within the heap is $H \cdot \#shortcut/(|I| + |O|)$ where H is the hop number of the hop-maximal shortcut that would be added if x was bypassed, smaller keys have higher priority. We say that the nodes that have been bypassed belong to the *component*, while the remaining nodes are called *core-nodes.*

Next, we perform an edge-reduction step, similar to [35]. We grow a shortest-path tree from each node u of the core. We stop the growth as soon as all neighbors t of u have been settled. Then we check for all neighbors t whether u is the predecessor of t in the grown partial shortest path tree. If u is not the predecessor, we can remove (u, t) from the graph because the shortest path from u to t does not include (u, t).

Theorem 1. *Contraction preserves distances between core nodes.*

Proof. Correctness follows directly from our rules of adding shortcuts during node-reduction and removal of unneeded edges during edge-reduction. □

Query. The s-t query is a modified bidirectional Dijkstra, consisting of two phases and performed on full graph G_F. During phase 1, 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 (respectively T for the backward search). The first phase terminates if one of the following two conditions holds: either (1) both priority queues are empty or (2) the distance to the closest entry points of s and t is larger than the length of a tentative shortest path possibly found during phase 1. If case (2) holds, the whole query terminates. The second phase is initialized by refilling the queues with the nodes belonging to S and T. As keys 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. Note that when not using any speed-up technique for the second phase, we obtain a special variant of Highway-Node Routing.

Theorem 2. *Core-Based Routing is correct.*

Proof. Let $P = (s, u_1, \ldots, u_k, t)$ be an arbitrary shortest path in G. If no node on P is part of the core in G_F, core-based routing is correct since we find the path during phase 1. If only one node on P is part of the core, we also find the path during phase 1. Now, let more than one node on P be part of the core. Let u_i be the first and u_j be the last core node on P. During phase one, we obtain paths from s to u_i and from u_j to t not longer than the corresponding subpaths in G. Due to the fact that distances within the core are preserved by our contraction routine (Theorem 1), we know that a path between u_i and u_j in G_C exists with the same length as the corresponding subpath of P. Hence, there exists a path in G_F with equal length that is found by core-based routing. $\square$

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 [15]. Hence, we expected that CALT (Core-ALT) also works well in dynamic and time-dependent scenarios, which recently has been shown in [10]. Second, we showed in [6] that 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 drastically 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. The second phase of core-based routing is replaced by ALT.

Proxy Nodes. Note that the ALT query requires lower bounds to s and t from every node within the core but both s and t need not be part of the core. In order to perform correct queries anyway, we adapt the ideas from [20, 13] to overcome this problem. Let t'—called the *proxy node* of t—be the core node with minimum $d(t, t')$ and let l_1 and l_2 be two arbitrary landmarks $L \subset V_C$. Then the following equations hold for all $u \in V_C$. See Figure 2 for illustration.

$$\begin{aligned} d(u,t') &\leq d(u,t) + d(t,t') \\ d(u,l_2) &\leq d(u,t') + d(t',l_2) \\ d(l_1,t') &\leq d(l_1,u) + d(u,t') \end{aligned}$$

Hence,

$$\underline{d}(u,t) := \max_{l \in L} \max\{d(u,l) - d(t',l) - d(t,t'), d(l,t') - d(l,u) - d(t,t')\} \qquad (1)$$

provides a feasible lower bound for the distance $d(u,t)$. Analogously, we obtain $\underline{d}(s,u) := \max_{l \in L} \max\{d(s',l) - d(u,l) - d(s',s), d(l,u) - d(l,s') - d(s',s)\}$ as feasible lower bound for the distance $d(s,u)$ with $s' \in V_C$ being the node with minimum $d(s',s)$.

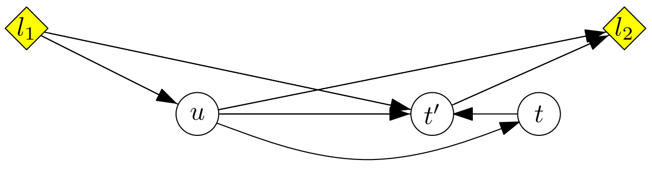

Fig. 2. Proxy nodes for CALT. When computing lower bounds for $d(u,t)$ with $u \in V_C$, $t \notin V_C$ using landmark distances in the core, $t' \in V_C$ acts as proxy node for t.

We compute these proxy nodes of s and t for a given s–t query during the initialization phase of the query by running Dijkstra-queries. Note for CALT, the quality of the lower bounds not only depends on the quality of the selected landmarks but also on $d(t, t')$ and $d(s', s)$.

Improved Locality. In order to improve query performance, we increase—similar to [20]—cache efficiency of G_F by reordering nodes. As most of the query is performed on the core, we store the core nodes followed by the non-core nodes. As a consequence, the number of cache misses is reduced yielding lower query times. Furthermore, this eases accessing landmark distances since we can simply use an array with $|L| \cdot |V_C|$ 64-bit entries for storing these distances (see [15] for details).

Theorem 3. CALT *is correct.*

Proof. According to Theorem 2, plain core-based routing is correct. Moreover, potentials as obtained from Equation (1) are feasible. Hence, applying ALT during phase 2 does not violate Theorem 2. □

3 Hierarchy-Aware Arc-Flags

Two important 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 and best query performance, 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 basically uses 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 (C̲ontraction-H̲ierarchy + A̲rc-flagS̲ + highway-nodE̲ 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 that Contraction 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 [23] 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.

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.

Stall-On-Demand. Pure Contraction Hierarchies benefit from an optimization technique called stall-on-demand. During the query, a very local breadth-first search *stalls* nodes that cannot be part of the shortest path (cf. [34] for details). However, during our experimental study, it turned out that this optimization technique does not pay off for CHASE. The search space decreases only slightly which cannot compensate the computational overhead of stall-on-demand. So, the resulting query of CHASE is a plain bidirectional Dijkstra operating on G' with arc-flags activated on high levels of the hierarchy.

Theorem 4. CHASE *is correct.*

Proof. The correctness of CH is known. If the query terminates during phase 1, then the correctness of the combinations directly follows from the correctness of CH. Otherwise, we know that a shortest s-t path must contain at least two entrance points $\hat{s} \in S$ and $\hat{t} \in T$. As the query relaxes all edges with true arc-flags for at least one cell in C_T and C_S, it is certain that the shortest path is found. $\square$

Partial CHASE (pCHASE). Contraction Hierarchies yield excellent preprocessing and query times in road networks. The main reason is that the average degree of nodes with respect to the search graph G' stays low. However, for other inputs, the average degree may grow rapidly yielding bad preprocessing and query times. Our Partial CHASE algorithm is motivated from such inputs. Instead of computing a complete contraction hierarchy, we *stop* the contraction at a certain point. This yields a CH-core H with size $|V_H|$. We use the subgraph induced by V_H as input for arc-flags preprocessing. The idea is that the lacking hierarchy in the core is compensated by goal-direction.

3.2 Reach + Arc-Flags (ReachFlags)

Similar to CHASE, we can also combine Reach and Arc-Flags, called *ReachFlags*. We first run a complete Reach-preprocessing as described in [20] and assemble the output graph. Next, we extract a subgraph H from the output graph containing all nodes with a reach value $\geq \ell$. Again, we compute arc-flags in H according to Section 1. Note we do not favor one path over another if both paths have the same length. The ReachFlags-query can easily by adapted from the CHASE-query in a straightforward 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.

Theorem 5. *ReachFlags is correct.*

Proof. We know that pure reach-based routing is correct. With the same observations from the proof of Theorem 4, the correctness of ReachFlags follows. $\square$

Partial ReachFlags (pReachFlags). Analogously to Partial CHASE, we can also define a partial variant of ReachFlags. Therefore, we slightly alter the reach preprocessing: Reach-computation according to [20] is a process that iteratively contracts and prunes the input. After each iteration step, all nodes with final reach value assigned are removed from the graph. Starting from this observation, we are able to preprocess Partial ReachFlags. We first run ℓ iteration steps of Reach-preprocessing as described in [20]. All nodes that do not have their final reach value set, get a reach value of ∞ assigned. Next, we assemble the output graph and extract a subgraph H from it containing all nodes with reach ∞. Again, we compute arc-flags in H according to [23]. Note that for Partial ReachFlags the input parameter ℓ adjusts the size of V_H.

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

Recall that the most time-consuming part of a TNR-query are the table lookups. Hence, when 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 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 $r(u) \in \{1,\dots,k\}$. For each node s and each access node $u \in \overrightarrow{\mathrm{A}}(s)$, we manage a flag vector $f_{s,u}^{\rightarrow} : \{1,\dots,k\} \to \{\mathsf{true}, \mathsf{false}\}$ 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 $\min\{d(s,u') + d(u',v) \mid u' \in \overrightarrow{\mathrm{A}}(s)\}$. 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 $u \in \overrightarrow{\mathrm{A}}(s)$ that minimize $d(s,u) + d(u,b)$; we set $f_{\overrightarrow{s,u}}(r(b))$ to `true`. In addition, $f_{\overrightarrow{s,u}}(r(u))$ is set to `true` for each $s \in V$ and each access node $u \in \overrightarrow{\mathrm{A}}(s)$ since each access node obviously minimizes the distance to itself. An analogous preprocessing step has to be done for the backward direction.

Theorem 6. *The preprocessing algorithm is correct.*

Proof. We consider an arbitrary node s, an access node $u \in \overrightarrow{\mathrm{A}}(s)$, and a region x. If $x = r(u)$, then $d(s,u) = \min\{d(s,u') + d(u',u) \mid u' \in \overrightarrow{\mathrm{A}}(s)\}$ implies that $f_{\overrightarrow{s,u}}(r(u))$ has to be `true`, which is explicitly ensured by the preprocessing algorithm. Otherwise ($x \neq r(u)$), we distinguish between two cases:
Case 1: there is a node $v \in \mathcal{T}$ with $r(v) = x$ such that $d(s,u) + d(u,v) = \min\{d(s,u') + d(u',v) \mid u' \in \overrightarrow{\mathrm{A}}(s)\}$, which implies that $f_{\overrightarrow{s,u}}(x)$ has to be `true`. Consider a shortest path P from u to v in $G_{\mathcal{T}}$ and the node b on P with $r(b) = x$ that is closest to u. From this definition and the fact that $r(u) \neq x$, it follows that the predecessor of b on P belongs to a different region, i.e., b is a border node. Furthermore, we can conclude that $d(s,u) + d(u,b) = \min\{d(s,u') + d(u',b) \mid u' \in \overrightarrow{\mathrm{A}}(s)\}$. Hence, $f_{\overrightarrow{s,u}}(x)$ is set to `true`.
Case 2: there is no node $v \in \mathcal{T}$ with $r(v) = x$ such that $d(s,u) + d(u,v) = \min\{d(s,u') + d(u',v) \mid u' \in \overrightarrow{\mathrm{A}}(s)\}$, which implies that $f_{\overrightarrow{s,u}}(x)$ has to be `false`. Assume that $f_{\overrightarrow{s,u}}(x)$ was `true`. Since $x \neq r(u)$, this would imply that there is a border node $b \in B$ with $r(b) = x$ such that u minimises $d(s,u) + d(u,b)$. Because of $B \subseteq \mathcal{T}$, this is a contradiction. □

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 $f_t(r(u)) =$ `true` for each $u \in \overleftarrow{\mathrm{A}}(t)$. Second, we consider only forward access nodes u of s with the property that the bitwise AND of $f_{\overrightarrow{s,u}}$ and f_t is not zero; we denote this set by $\overrightarrow{\mathrm{A}}'(s)$; during this step, we also build the vector f_s such that $f_s(r(u)) =$ `true` for each $u \in \overrightarrow{\mathrm{A}}'(s)$. Third, we use f_s to determine the subset $\overleftarrow{\mathrm{A}}'(t) \subseteq \overleftarrow{\mathrm{A}}(t)$ analogously to the second step. Now, it is sufficient to perform only $|\overrightarrow{\mathrm{A}}'(s)| \times |\overleftarrow{\mathrm{A}}'(t)|$ table lookups. An example is given in Fig. 3. Note that determining $\overrightarrow{\mathrm{A}}'(s)$ and $\overleftarrow{\mathrm{A}}'(t)$ is in $O(a)$, in particular operations on the flag vectors can be considered as quite cheap because we can use bit parallelism.

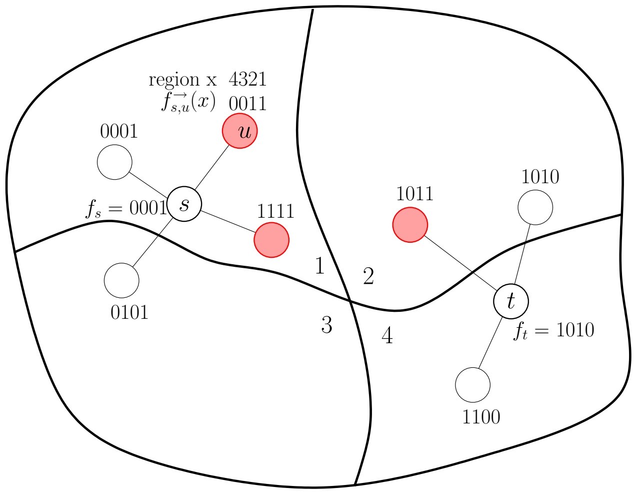

Fig. 3. An example of a goal-directed transit-node query. Nodes selected for $\overrightarrow{\mathrm{A}}'(s)$ and $\overleftarrow{\mathrm{A}}'(t)$ are shaded. The number of required table lookups is reduced from 12 to 2.

Theorem 7. *The query algorithm is correct.*

Proof. We have to show that $\min\{d(s,u) + d(u,v) + d(v,t) \mid u \in \overrightarrow{\mathrm{A}}(s), v \in \overleftarrow{\mathrm{A}}(t)\} = \min\{d(s,u) + d(u,v) + d(v,t) \mid u \in \overrightarrow{\mathrm{A}}'(s), v \in \overleftarrow{\mathrm{A}}'(t)\}$. Consider the nodes $u \in \overrightarrow{\mathrm{A}}(s)$ and $v \in \overleftarrow{\mathrm{A}}(t)$ that minimise the distance $d(s,u) + d(u,v) + d(v,t)$. In particular, we have $d(s,u)+d(u,v) = \min\{d(s,u')+d(u',v) \mid u' \in \overrightarrow{\mathrm{A}}(s)\}$, which implies that $f_{\overrightarrow{s,u}}(r(v)) = \mathsf{true}$. Furthermore, we have $f_t(r(v)) = \mathsf{true}$ since $v \in \overleftarrow{\mathrm{A}}(t)$. Hence, the bitwise AND of $f_{\overrightarrow{s,u}}$ and f_t is not zero and, consequently, $u \in \overrightarrow{\mathrm{A}}'(s)$. Analogously, we can show that $v \in \overleftarrow{\mathrm{A}}'(t)$. □

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 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. In the following, we report preprocessing effort and query performance of all speed-up techniques. We measure the preprocessing effort in time to compute the additional data and the *additional* space per node this data occupies. For query performance, we report the average number of settled nodes, i.e. the number of nodes taken from the priority queues, and resulting query times. 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 [13, 1, 17].

5.1 Road Networks

As inputs for our test on road networks 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. 9.

CALT. For CALT, we first evaluate the impact of contraction on preprocessing effort and query performance. Table 1 reports the performance of CALT with 64 *avoid* land-

[1] The machine runs SUSE Linux 10.3, 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.

[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.

Table 1. Performance of CALT for varying contraction parameters c (contraction quotient) and h (hop-limit). *Core nodes* depicts the percentage of core nodes in G_F, *#add edges* reports the number of additional edges in G_F, and the resulting overhead (including landmark distances) is given in Bytes per node. The preprocessing time is given in minutes. For queries, we report the size of the search space in number of settled nodes, the number of entry nodes and the resulting average query times in milliseconds.

		Europe							USA						
		PREPRO.				QUERY			PREPRO.				QUERY		
		core	$\|E\|$	time	space	#settled	#entry	time	core	$\|E\|$	time	space	#settled	#entry	time
c	h	nodes	incr.	[min]	[B/n]	nodes	nodes	[ms]	nodes	incr.	[min]	[B/n]	nodes	nodes	[ms]
0.0	0	100.00%	0.00%	68	512.0	25 324	1.0	19.61	100.00%	0.00%	93	512.0	68 861	1.0	48.87
0.5	10	35.48%	10.23%	21	187.7	10 925	3.2	8.02	28.90%	11.40%	20	154.2	21 544	3.4	16.61
1.0	20	6.32%	14.24%	7	38.4	2 233	8.2	2.16	8.29%	12.68%	11	48.9	7 662	7.1	6.96
2.0	30	3.04%	11.41%	9	21.8	1 382	13.3	1.55	3.21%	10.53%	12	22.5	3 338	12.6	4.11
2.5	50	1.88%	9.16%	11	15.4	1 394	18.6	1.34	2.06%	8.00%	13	16.1	2 697	17.1	3.01
3.0	75	1.29%	7.80%	12	12.2	1 963	24.2	1.43	1.45%	6.39%	14	12.6	2 863	22.0	2.85
5.0	100	0.86%	6.94%	18	9.8	3 126	34.0	1.67	0.86%	4.50%	21	9.3	3 416	30.2	2.35

marks [4] with varying contraction rate. Note that for $c = 0.0$ and $h = 0$, we end up in a plain ALT-setup.

We observe that contraction has a very positive effect on ALT: Preprocessing space and time decreases combined with better query performance. The latter are accelerated by more than one order of magnitude while the high memory consumption of ALT can be reduced to a reasonable amount. Moreover, turning on contraction decreases the variance of the query times. Interestingly, the number of additional edges in the full graph G_F first increases with increasing contractionrates but then decreases again. The reason for this is that the core shrinks rapidly. The few core nodes finally yield a high average degree but with respect to the total number of nodes, the impact of core edges fades. For Europe, we observe that at a certain point, higher contraction values yield worse query performance. It seems as if a good compromise is $c = 2.5$ and $h = 50$. Hence, we use these contraction values as default from now on.

Number of Landmarks. Next, we focus on the impact of number of landmarks. More precisely, we evaluate 8, 16, 32, and 64 landmarks generated on cores obtained from different contraction rates. Note that we use *maxCover*[5] for 8 and 16 landmarks, while avoid was used to select 32 and 64 landmarks. Also note that a contraction of $c = 0.0$ and $h = 0$ again yields a pure ALT setup.

[4] Avoid selects landmarks from the graph by iteratively identifying parts of the graph not well covered by landmarks. For details, see [21].

[5] MaxCover yields slightly better landmarks than avoid. This is achieved by generating more landmarks (with avoid) than necessary and then selecting the best ones among of them by local search. For detail, see [21].

Table 2. Performance of CALT for different number of landmarks applying a low and high contraction.

	no cont. (c=0.0, h=0)				low cont. (c=1.0, h=20)				med. cont. (c=2.5, h=50)				high cont. (c=5.0, h=100)			
	PREPRO.		QUERY		PREPRO.		QUERY		PREPRO.		QUERY		PREPRO.		QUERY	
	time	space	#sett.	time	time	space	#sett.	time	time	space	#sett.	time	time	space	#sett.	time
$\|L\|$	[min]	[B/n]	nodes	[ms]	[min]	[B/n]	nodes	[ms]	[min]	[B/n]	nodes	[ms]	[min]	[B/n]	nodes	[ms]
8	26.1	64.0	163 776	127.8	7.1	10.9	12 529	10.25	10.1	7.0	4 431	3.98	17.8	5.9	4 106	2.51
16	85.2	128.0	74 669	53.6	9.4	14.9	5 672	5.77	11.0	8.2	2 456	2.33	18.3	6.5	3 500	2.23
32	27.1	256.0	40 945	29.4	6.8	23.0	3 268	2.97	10.0	10.6	1 704	1.66	17.7	7.6	3 264	2.01
64	68.2	512.0	25 324	19.6	8.5	36.2	2 233	2.16	10.5	15.4	1 394	1.34	18.0	9.8	3 126	1.67

We observe that with decreasing size of the core, the impact of number of landmarks fades: In a pure ALT setting, doubling the number of landmarks roughly yields an increase of a factor of 2 in query performance. On the contrary, in a high contraction scenario ($c = 5.0, h = 100$), the number of landmarks has nearly no influence on query performance. Using 64 instead of 8 landmarks decreases query times by only $\approx 33\%$. However, as memory consumption is still very low for 64 landmarks, we use this number as default for CALT.

Local Queries In order to gain deeper insights into the impact of contraction on query performance, Fig. 4 reports the query times of CALT for different contraction rates and different measures of the locality of the queries. For ALT, we use 16 maxCover landmarks, for CALT 64 landmarks are selected by avoid.

We observe that pure ALT is faster than CALT for ranks up to 2^8 if low contraction is applied. If the core gets smaller, pure ALT is faster than CALT for ranks up to 2^{10}. This is due to the fact that CALT has a two-phase query yielding a higher overhead. Interestingly, increasing the contraction rate has a negative effect for low-range queries while long-range queries seem to benefit from higher contraction rates. Still, low-range queries are executed in less than 1 ms for both contraction setups. Moreover, space consumption decreases with increasing contraction rates (cf. Tab 1). Hence, our choice of $c = 2.5$, $h = 50$ as default setting seems reasonable.

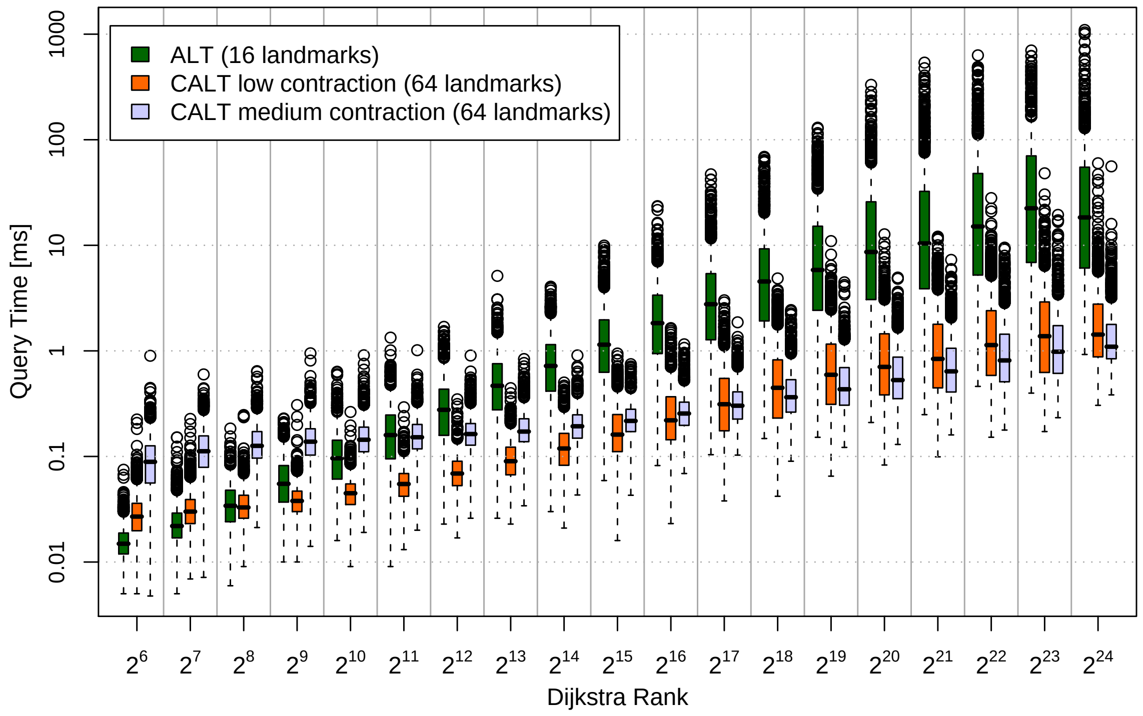

Fig. 4. Comparison of ALT, CALT with low contraction ($c = 1.0$, $h = 20$), and medium contraction ($c = 2.5$, $h = 50$). For each power of two, 1 000 queries with Dijkstra rank 2^i are measured (For an s-t query, the Dijkstra rank of node v is the number of nodes removed from the priority queue by Dijkstra's algorithm before v is removed, see also [31]). The results are represented as box-and-whisker plot [38]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

CHASE. 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 improves. Table 3 reports the performance of CHASE for different sizes of H in percentage of the original graph. We partition H with SCOTCH [30] into 128 cells.

Table 3. Performance of CHASE for Europe with stall-on-demand (s-o-d) turned on and off running 10 000 random queries. The search space is given as #settled nodes during phase 1 and in total. The number of entry points is given as well.

	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 (with s-o-d)	#settled total	355	86	67	54	43	37	34
	#settled phase 1	–	60	41	29	18	13	8
	#entry points	–	21	14	10	7	5	4
	time [μs]	180.0	48.5	36.3	29.2	22.8	19.7	17.2
Query (without s-o-d)	#settled total	931	111	78	59	45	39	35
	#settled phase 1	–	76	47	31	19	13	8
	#entry points	–	30	18	12	8	6	4
	time [μs]	286.3	43.8	30.8	23.1	17.3	14.9	13.0

Two observations are remarkable: the effect of stall-on-demand ($\rightarrow$ 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 additional reduction in the number of settled nodes is not large enough to pay for the additional computational overhead incurred by stall-on-demand. Apparently, arc flags already prune most of the nodes from the search space that would otherwise be pruned by stall-on-demand. 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. A core size of 0.5% already yields an additional speed-up of a factor of 4 for an additional preprocessing effort of 6 minutes. Hence, we call this setup our *economical* variant of CHASE. Interestingly, further significant improvements—with respect to query times—are only observable for a core size of up to 5% of the input graph. Here, we achieve query times $\approx$ 10 times faster than plain CH. Still, 99 minutes of preprocessing is reasonable. Hence, we call this setup our *generous* variant of CHASE. Increasing the size of H to 10% or even 20% yields a much higher preprocessing effort (both space and time) but query time decreases only slightly, compared to 5%. However, our fastest variant settles only 35 nodes on the average having query times of 13.0 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.

Local Queries. Like for CALT, Fig. 5 reports the query times of economical and generous CHASE and plain CH with respect to different localities of the queries. We observe that up to a rank of 2^{14}, all three algorithms yield similar query times. This is expected since up to this rank, most of the queries do not touch the upper part of the contraction hierarchy and hence, arc-flags do not contribute to the query. Above this rank, query performance gets better again. This effect has been observed for pure Arc-Flags as well [23]: Long-range queries often relax *only* the shortest path while for low-range queries, the advantage

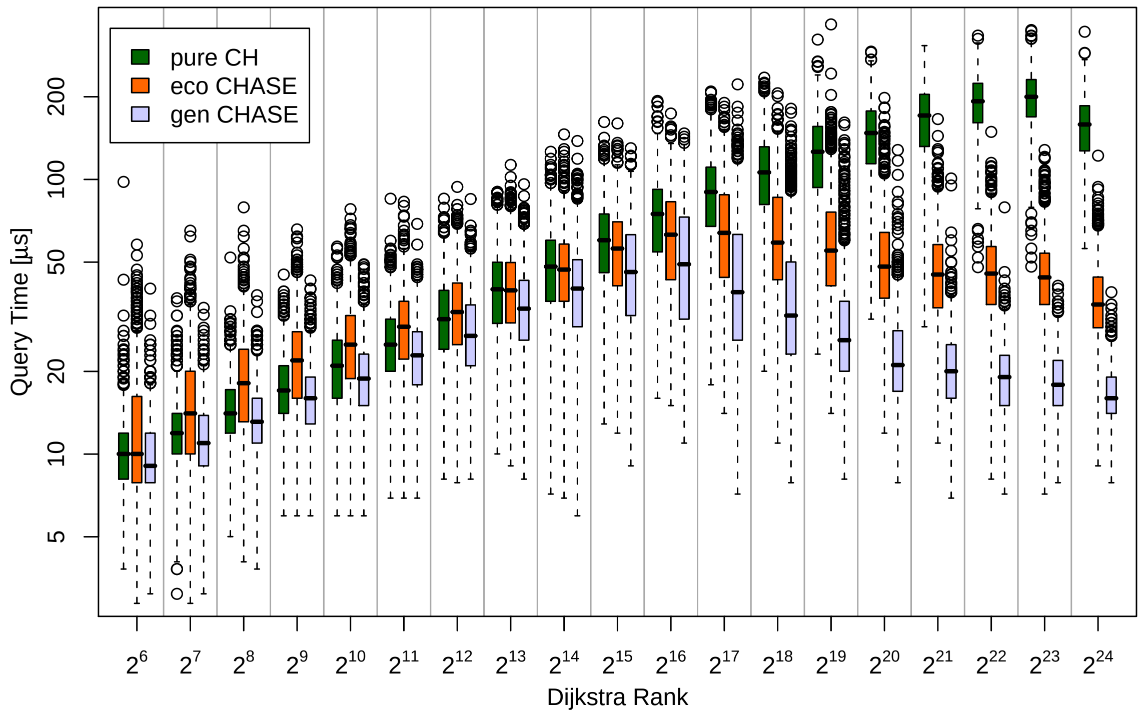

Fig. 5. Comparison of pure CH, economical and generous CHASE using the same setup as in Fig. 4.

of arc-flags fades. Comparing economical and generous CHASE, we observe that above a rank of 2^{17}, the latter is about 2.5 times faster than the former. For very high ranks, generous CHASE is more than an order of magnitude faster than pure CH.

Partial CHASE. Up to now, we evaluated a setup where a complete CH is constructed. However, as discussed in Section 3.1, we may stop the construction at some point and compute arc-flags on a flat core. Table 4 reports the performance of partial CHASE if 0.5% and 5% of the graph is *not* contracted. For comparison, we report the figures of a partial variant of CH, called pCH. Similar to pCHASE, we stop contraction at some point and perform CH-queries in such a partial hierarchy. Moreover, we report the performance of plain CH and economical CHASE.

We observe that for road networks, partial variants of CHASE yield worse results than pure CH: With an uncontracted core of 0.5%, preprocessing is a little bit faster but for the price of a slow-down of a factor of 4.6 in query performance. Higher uncontracted cores seem even more impractical. The reason for this rather bad performance stems from contraction hierarchies. The partial variant of CH yields a very bad query performance which cannot be compensated by arc-flags.

Table 4. Performance of pCHASE and partial CH. The input is Europe. Note that only 1 000 queries were computed for pCH.

	non-contracted	0.0%		0.5%		5.0%	
	algorithm	CH	eco CHASE	pCH	pCHASE	pCH	pCHASE
Prepro.	time [min]	25	31	19	21	15	31
	space [Byte/n]	-2.7	0	-2.8	-1.6	-2.9	3.6
Query	#settled total	355	86	97 913	2 544	965 018	12 782
	time [μs]	180	44	4 281	831	53 627	4,143

ReachFlags. Table 5 gives an overview for ReachFlags, similar to Tab. 3 for CHASE, showing the effects of different sizes of the subgraph H. As expected, query times decrease with an increased subgraph. However, adding goal direction via Arc-Flags yields worse additional speed-ups than for CH. Computing flags on the topmost 0.5% of the graph accelerates queries only by a factor of 2. For CH, the corresponding figure is 4. Moreover, since CH is more than one order of magnitude faster than Reach, CHASE is superior to ReachFlags with respect to all relevant figures. Note however that our implementation of Reach that we also use as base for ReachFlags is roughly a factor of 2 slower than the implementation due to [20]. Thus, a further speed-up of a factor of 2 might be possible.

Table 5. Performance of ReachFlags for Europe and the US. The results of our implementation of Reach correspond to a size of H of 0.0%.

input		size of H	0.0%	0.5%	1.0%	2.0%	5.0%	10.0%
Europe	Prepro.	time [min]	70	82	105	150	348	710
		space [Byte/n]	21.0	22.9	25.1	28.2	37.0	49.3
	Query	#settled total	7 387	5 454	3 754	2 763	1 101	638
		time [ms]	6.24	4.79	3.12	2.22	0.84	0.48
USA	Prepro.	time [min]	62	89	136	272	671	1 279
		space [Byte/n]	21	20	22	26	35	45
	Query	#settled total	4261	2563	1719	1339	693	450
		time [ms]	3.90	2.09	1.33	1.01	0.50	0.33

Partial ReachFlags. Table 6 reports the performance of partial ReachFlags. We stop reach computation after i iterations, set the reach of remaining nodes to infinity and compute arc-flags for the subgraph induced by these nodes.

Table 6. Performance of pReachFlags. The input is Europe. Core nodes indicates how many nodes have a reach value of infinity.

number of iterations		1	2	3	4	all
Prepro.	core-nodes	14.87%	5.32%	1.45%	0.20%	0.00%
	time [min]	400	229	107	69	70
	space [Byte/n]	36	30	25	22	21
Query	#settled total	1 149	1 168	2 797	5 718	7 387
	time [ms]	0.62	0.76	2.24	5.34	6.24

Interestingly, partial ReachFlags provides better results than pure reach: The less reach values we bound, the better the performance of the algorithm gets. This is due to the fact that we compute arc-flags for a bigger part of the graph. Interestingly, for core sizes of $\approx 5\%$, pReachFlags outperforms pCHASE, while for core sizes of $\approx 0.5\%$, pCHASE outperforms pReachFlags. It seems as if the loss in performance for cutting a hierarchy based on reach is less than cutting a contraction hierarchy.

TNR+AF. The fastest variant of Transit-node Routing *without* using flag vectors is presented in [17]; the corresponding figures are quoted in Tab. 7. 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 [1]) and a speed-up compared to Dijkstra's algorithm of more than factor 3 *million*. The results for the US are even better.

Table 7. Overview of the performance of Transit-Node Routing with and without additional arc-flags. For pure TNR, we report two figures. The first is due to [1] and based on Highway Hierarchies, while numbers for a TNR-implementation based on Contraction Hierarchies are given in [17].

	Europe			USA		
	PREPRO.		QUERY	PREPRO.		QUERY
	time	overhead	time	time	overhead	time
method	[min]	[B/node]	[μs]	[min]	[B/node]	[μs]
HH-TNR	164	251	5.6	205	244	4.9
CH-TNR	112	204	3.4	90	220	3.0
CH-TNR+AF	229	321	1.9	157	263	1.7

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 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.

Comparison. Table 8 reports the performance of our new combinations in comparison to existing speed-up techniques.

CALT. In [15], we were able to improve query performance of ALT over [21] by improving the organization of landmark data. However, we do not compress landmark information and use a slightly better heuristic for landmark selection. Hence, we report both results. By adding contraction to ALT, we are able to reduce query times to 1.3 ms for Europe and to 3.0 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. 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 easy adaptability to dynamic and time-dependent scenarios [10].

CHASE. We report the figures for *economical* and *generous* CHASE. 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 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 [1]). However, the preprocessed data is much smaller for CHASE, which makes it more practical in environments with limited memory. Moreover, it seems as if CHASE can be adapted to time-dependent scenarios easier than TNR [3].

Table 8. Overview of the performance of various speed-up techniques, grouped by (1.) hierarchical methods [Highway Hierarchies (HH), highway-node routing (HNR), Contraction Hierarchies (CH), 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 (1 000 for pCH) random queries. Each column highlights the best result in bold. In addition, *Pareto-optimal* speed-up techniques are also printed in bold. Note that the Pareto-optima are the same for both road networks.

		Europe				USA			
		PREPRO.		QUERY		PREPRO.		QUERY	
		time	overhead	#settled	time	time	overhead	#settled	time
method	source	[min]	[B/node]	nodes	[ms]	[min]	[B/node]	nodes	[ms]
Reach	[20]	83	17	4 643	3.47	44	20	2 317	1.81
Reach	3.2	70	21	7 387	6.24	62	21	4 261	3.90
HH	[34]	13	48	709	0.61	15	34	925	0.67
HNR	[34]	15	2.4	981	0.85	16	1.6	784	0.45
CH	[17]	25	-2.7	355	0.18	27	-2.3	278	0.13
grid TNR	[1]	-	-	-	-	1 200	21	N/A	0.063
TNR	[1]	164	251	N/A	0.0056	205	244	N/A	0.0049
TNR	[17]	112	204	N/A	0.0034	90	220	N/A	0.0030
ALT-a16	[20]	13	70	82 348	160.3	19	89	187 968	400.5
ALT-m16	[15]	85	128	74 669	53.6	103	128	180 804	129.3
ALT-a64	[15]	68	512	25 234	19.6	93	512	68 861	48.9
AF	[23]	2 156	25	1 593	1.1	1 419	21	5 522	3.3
REAL	[20]	141	36	679	1.11	121	45	540	1.05
HH*	[34]	14	72	511	0.49	18	56	627	0.55
SHARC	[4]	81	14.5	654	0.29	58	18.1	865	0.38
SHARC bidir.	[4]	158	21.0	125	0.065	158	21	350	0.18
CALT	2	**11**	15.4	1 394	1.34	**13**	16.1	2 697	3.01
pCH-0.5%	3.1	19	-2.8	97 913	4.28	21	-2.3	121 636	50.57
pCH-5.0%	3.1	15	**-2.9**	965 018	53.63	15	**-2.3**	1 209 290	667.80
eco CHASE	3.1	32	0.0	111	0.044	36	-0.8	127	0.049
gen CHASE	3.1	99	12	**45**	0.017	228	11	**49**	0.019
pCHASE-0.5%	3.1	21	-1.6	2 544	0.83	25	-1.5	4 693	1.40
pCHASE-5.0%	3.1	31	3.6	12 782	4.14	96	4.3	22 436	7.21
ReachFlags	3.2	348	37	1 101	0.84	671	35	693	0.50
pReachFlags	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**

Comparing SHARC and CHASE, one may notice that both combinations are based on contraction and arc-flags. Since CHASE uses a better contraction routine than bidirectional SHARC, the former outperforms the latter. However, the main motivation for SHARC was a unidirectional query algorithm allowing easy adaptions to augmented scenarios [9].

ReachFlags. As already mentioned, our reach implementation yields worse results than the numbers reported in [20]. Hence, we report both results. By adding arc-flags to reach we obtain query times comparable to REAL. However, preprocessing takes a little bit longer. Still, it seems as if ReachFlags is inferior to CHASE which is mainly due to the good performance of Contraction Hierarchies.

Summary. We observe that the best results for each measured performance criterion is obtained by one of our newly introduced speed-up techniques. In addition, we see that

Table 9. Overview on the performance of prominent speed-up techniques and combinations analogous to Tab. 8 but with travel distances as metric.

		Europe				USA			
		PREPRO.		QUERY		PREPRO.		QUERY	
		time	overhead	#settled	time	time	overhead	#settled	time
method		[min]	[B/node]	nodes	[ms]	[min]	[B/node]	nodes	[ms]
Reach	[20]	49	15	7 045	5.53	70	22	7 104	5.97
HH	[34]	32	36	3 261	3.53	38	66	3 512	3.73
CH	3.1	89	**-0.1**	1 650	4.19	57	**-1.2**	953	1.50
TNR	[34]	162	301	N/A	**0.038**	217	281	N/A	**0.086**
ALT-a16	[20]	**10**	70	240 750	430.0	**15**	89	276 195	530.4
ALT-m16	2	70	128	218 420	127.7	102	128	278 055	166.9
AF	[23]	1 874	33	7 139	5.0	1 311	37	12 209	8.8
REAL	[20]	90	37	583	1.16	138	44	628	1.48
HH*	[34]	33	92	1 449	1.51	40	89	1 372	1.37
SHARC	[4]	64	19	3 014	1.34	75	20	3 871	1.78
CALT	2	14	19	2 958	4.2	15	19	4 015	5.6
eco CHASE	3.1	224	7.0	175	0.156	185	2.5	148	0.103
gen CHASE	3.1	1 022	27	**67**	0.064	1 132	18	**63**	0.043
pCHASE-0.5%	3.1	40	1.9	5 957	2.61	43	0.3	8 276	3.17
pReachFlags	3.2	516	31	5 224	4.05	1 897	27	6 849	4.69

almost all of our techniques are *Pareto-optimal*[6]. Thus, each technique is the optimal choice for a specific task with regards to the analyzed algorithms. Only our variants of ReachFlags and pCHASE with a larger core size fall short in this aspect.

Travel Distances. Up to now, we concentrated on travel times as metric. Table 9 reports the performance of our new combinations compared to existing techniques if travel distances are used as metric.

We observe that both pure hierarchical and goal-directed approaches work worse on travel distances than on travel times. Interestingly, this does not hold for combinations. Most of the combinations yield similar performance on both metrics. For example, both pure Arc-Flags and CH are 5 and 20 times slower on travel distance whereas the combination of both approaches, CHASE, yields similar query times. The reason for this is the following. For pure CH, edge reduction works worse on travel distances yielding higher degrees for high-level nodes. By applying Arc-Flags on this rather dense core, a lot of edge relaxations can be avoided. This also explains the highly increased preprocessing times of CHASE: The core is denser making arc-flags computation more time-consuming. Hence, partial CHASE is more promising on this input than on travel times as metric. We observe that preprocessing times are faster than for pure CH combined—at least for Europe—with better query times.

Concerning preprocessing times, CALT outperforms any other technique combined with reasonable query times. We conclude that CALT indeed is almost as robust as pure ALT with respect to metric changes.

5.2 Robustness of Combinations

General Results. In the last section we focused on the performance of our combinations on road networks. However, existing combinations of goal-directed and hierarchical methods

[6] A speed-up technique is called *Pareto-optimal* if there is no other technique that is better with respect to the four variables preprocessing time and space, search space, and query time.

Table 10. Performance of bidirectional Dijkstra, ALT, CALT, ArcFlags, CH, economical CHASE and partial variants of CH and CHASE on grid graphs with different number of dimensions and time-expanded timetables of different railway networks. Note that the we use the *aggressive* variant of contraction hierarchies, better results may be achieved by better input parameters. For each column and each family of graphs the best measurement value is highlighted in bold. Note that bidirectional Dijkstra was not considered for this comparison. In addition checkmarks indicate Pareto-optimal algorithms.

	PREPRO.		QUERY			PREPRO.		QUERY			PREPRO.		QUERY		
	[s]	[B/n]	#settled	[ms]		[s]	[B/n]	#settled	[ms]		[s]	[B/n]	#settled	[ms]	
grid	*2-dimensional*					*3-dimensional*					*4-dimensional*				
bidir. Dijkstra	0	0	79 962	24.2	-	0	0	45 269	28.2	-	0	0	21 763	20.3	-
ALT-m16	65	128	2 362	1.5	-	100	128	1 759	2.1	✓	133	128	1 335	2.6	✓
CALT	60	211	458	1.1	✓	**101**	386	**557**	1.9	✓	**129**	487	**774**	2.2	✓
AF	5 340	130	1 340	0.4	-	49 800	191	1 685	**0.6**	✓	187 020	191	2 799	**1.4**	✓
CH	70	0	418	0.3	✓	13 567	14	2 177	6.6	✓	133 734	29	14 501	60.0	✓
pCH-10%	37	-1	25 486	9.6	✓	1 038	9	27 001	28.7	✓	-	-	-	-	-
pCH-20%	30	-2	50 720	19.0	✓	606	4	52 671	52.2	✓	11 798	34	53 354	160.8	✓
pCH-50%	**19**	**-3**	126 304	47.8	✓	273	**-3**	129 060	118.9	✓	2 035	**6**	129 475	249.4	✓
CHASE	73	2	**274**	**0.2**	✓	13 585	22	2 836	10.1	-	133 741	32	30 848	131.0	-
pCHASE-10%	64	15	1 967	0.5	✓	1 512	57	10 788	7.5	✓	-	-	-	-	-
pCHASE-20%	91	26	3 063	0.8	-	1 850	69	10 052	5.2	✓	28 898	208	31 384	52.1	-
pCHASE-50%	212	55	5 964	1.5	-	2 984	95	13 402	5.7	-	26 470	279	36 473	33.0	-
railways	*Berlin/Brandenburg*					*Ruhrgebiet*					*long distance*				
bidir.Dijkstra	0	0	1 299 830	406.2	-	0	0	1 134 420	389.2	-	0	0	609 352	221.2	-
ALT-m16	604	128	56 404	27.3	-	556	128	60 004	30.9	-	291	128	30 021	14.4	-
CALT	**123**	45	2 830	6.3	✓	**191**	68	4 247	11.3	✓	**87**	63	2 088	5.3	✓
AF	268 740	98	24 004	5.9	-	278 760	98	28448	7.0	-	120 960	98	10 560	2.4	-
CH	1 636	0	416	0.4	✓	2584	4	546	0.6	✓	486	3	376	0.3	✓
pCH-0.5%	1 115	0	15 433	9.4	✓	1 723	4	14 149	9.6	✓	320	3	6 767	3.3	✓
pCH-5.0%	647	-1	132 992	90.0	✓	923	3	123 277	85.1	✓	157	2	66 457	42.2	✓
pCH-10.0%	489	**-1**	257 778	170.6	✓	662	**3**	236 050	168.4	✓	113	**2**	129 144	90.4	✓
CHASE	2 008	2	**125**	**0.1**	✓	2863	7	**244**	**0.2**	✓	536	5	**229**	**0.2**	✓
pCHASE-0.5%	1 313	2	4 492	2.1	✓	1 800	6	8 209	4.5	✓	343	4	14 482	1.6	✓
pCHASE-5.0%	15 572	7	18 698	7.2	-	10 159	13	20 224	8.0	-	1 087	11	8 985	3.2	-
pCHASE-10.0%	45 992	12	36 828	13.7	-	37 429	19	34 056	13.0	-	2 506	17	3 695	5.1	-

like REAL or SHARC are very robust to the input. Here, we evaluate our most promising combinations—CALT, CHASE, and partial CHASE—on various other inputs. We use time-expanded timetable networks[7] and synthetic grid graphs (2–4 dimensions with 250 000 nodes, edge weights picked uniformly at random between 1 and 1000.). The results can be found in Table 10.

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.

As expected, cutting the hierarchy pays off only for denser inputs, i.e. 3- and 4-dimensional grids. Preprocessing of pCHASE is much lower than for pure CH or CHASE combined with *better* query times. This advantage comes at the price of a much larger space overhead.

[7] 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)

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 for very sparse graphs while CALT yields better performance on our denser inputs. Interestingly, even pCHASE cannot compete with CALT on these inputs. We assume that this derives from the general mediocre performance of CH on denser graphs combined with too many entry points into the core. Hence, many regions are activated during the arc-flag query yielding a small speed-up within the core. We conclude that for such inputs it is better to combine ALT with a hierarchical method.

Overall, we observe again, that almost all of our new combinations of speed-up techniques are Pareto-optimal with regards to the examined algorithms and the regarded types of graphs. Thus, we can conclude that there is a task where each of our combinations excels.

Sensor Networks. In addition to synthetic grid graphs and time-expanded timetable networks we also focused our analyses on unit disk graphs[8] (1 000 000 nodes with an average degree of 7, 10, and 20). These graphs provide a simple model to describe the connectivity in radio networks such as large-scale sensor networks. A common cost measure is some power of the Euclidean distance. We use 1, 2, and 4. Power 1 for example models signal latency. Power 2 models energy requirement for free space communications and the area in which a communication would cause significant interference. Powers from $(2, 5]$ are commonly used for modeling energy requirement in presence of signal absorption etc. The results of our algorithms are shown in Table 11.

Several interesting things can be seen. At first, we concentrate on the case of a pure Euclidean distance measure (power 1) for unit disk graphs of a varying density. We observe that most speed-up techniques have problems when the average node degree becomes too large. Pure hierarchical techniques seem to suffer more than pure goal-directed ones because they have problems to identify a clear hierarchy between the nodes. As already mentioned in the last paragraph, cutting the hierarchy seems to be the appropriate strategy for the denser graphs, i.e. average node degree 10 and 20. In case of pCHASE both the preprocessing time and the query time can be cut by a considerable amount but at the cost of a much higher space overhead.

Now, we compare the same graphs but apply different power laws to compute the edge weights. We observe that only changing the weight functions but not the topological structure of the graphs has little influence on goal-directed techniques. On the other hand, the impact on hierarchical techniques is huge. Using higher powers, the weight differences between short and long edges become more distinct and thus it becomes easier to identify hierarchical structures in the graphs. Speed-ups of up to a factor of 6 can be observed. We also see a large decrease in required space overhead for those techniques based on Contraction Hierarchies. This effect results from CH removing all superfluous edges—up to two thirds in case of unit disk graphs with an average node degree of 20 using a power

[8] 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 Euclidean distance to the power 1, 2 and 4.

Table 11. Overview on the performance of prominent speed-up techniques and combinations analogous to Tab. 10 but for unit disk graphs with different average node degrees and varying power laws. Note that the preprocessing of CH, CHASE, all variants of pCHASE have each been canceled after 20 hours for unit disk graphs with an average node degree of 20 and an Euclidean distance metric (lower left cell). Therefore, marking Pareto-optimal algorithms is of limited informative value.

	PREPRO.		QUERY			PREPRO.		QUERY			PREPRO.		QUERY		
	[s]	[B/n]	#settled	[ms]		[s]	[B/n]	#settled	[ms]		[s]	[B/n]	#settled	[ms]	
avg. deg. 7	*power 1*					*power 2*					*power 4*				
bidir. Dijkstra	0	0	340 801	225.1	-	0	0	318 070	210.9	-	0	0	320 187	213.2	-
ALT-m16	514	128	10 327	11.8	-	557	128	10 014	11.1	-	529	128	9 132	10.2	-
CALT	**135**	29	670	1.0	✓	**125**	38	624	1.0	✓	**127**	39	611	1.0	✓
CH	1 249	-11	1 089	1.8	✓	420	-23	749	0.9	✓	334	-29	404	0.3	✓
pCH-10%	238	-13	105 047	77.0	✓	204	-24	100 306	70.7	✓	289	-30	99 501	60.9	✓
pCH-20%	196	**-13**	209 492	151.5	✓	187	-25	200 081	136.3	✓	282	**-30**	198 772	122.1	✓
pCH-50%	137	-13	522 768	405.8	✓	162	**-25**	499 422	339.8	✓	267	-29	496 411	312.9	✓
CHASE	1 368	-7	**424**	**0.6**	✓	475	-18	**332**	**0.2**	✓	355	-26	**150**	**0.1**	✓
pCHASE-10%	731	10	5 677	2.7	✓	472	-8	5 569	2.3	-	418	-18	4 788	1.8	-
pCHASE-20%	1 268	25	9 990	4.8	-	727	2	9 762	4.2	-	557	-10	8 508	3.3	-
pCHASE-50%	3 511	68	21 064	10.9	-	1 671	31	21 016	9.5	-	1 072	16	17 795	7.4	-
avg. deg. 10	*power 1*					*power 2*					*power 4*				
bidir. Dijkstra	0	0	325 803	269.4	-	0	0	320 971	270.0	-	0	0	322 764	278.0	-
ALT-m16	566	128	11 704	15.5	✓	704	128	10 724	14.1	✓	772	128	9 889	13.2	✓
CALT	511	137	**992**	**2.6**	✓	**355**	173	1131	2.9	✓	**363**	173	1 248	3.3	✓
CH	34 274	-4	2 475	11.5	✓	1 001	-38	970	1.3	✓	850	-49	423	0.3	✓
pCH-10%	1 191	-9	100 977	121.1	✓	503	-40	101 529	86.0	✓	798	-50	101 122	64.3	✓
pCH-20%	815	-12	201 304	225.2	✓	458	-41	202 632	161.3	✓	787	-50	201 825	130.2	✓
pCH-50%	**453**	**-15**	502 273	557.3	✓	400	**-43**	506 066	391.5	✓	765	**-50**	503 848	332.2	✓
CHASE	34 847	6	1 457	4.7	✓	1 084	-31	**536**	**0.4**	✓	872	-46	**171**	**0.1**	✓
pCHASE-10%	4 149	35	7 529	2.7	✓	1 090	-18	5 910	2.7	-	962	-37	4 855	1.9	-
pCHASE-20%	6 737	54	11 622	4.8	-	1 674	-7	10 391	4.8	-	1 159	-28	8 319	3.3	-
pCHASE-50%	9 769	113	25 194	10.9	-	3 695	23	23 912	11.4	-	1 812	-4	1 8083	7.6	-
avg. deg. 20	*power 1*					*power 2*					*power 4*				
bidir. Dijkstra	0	0	320 468	427.2	-	0	0	321 619	455.7	-	0	0	321 941	472.6	-
ALT-m16	**964**	128	13 645	25.7	✓	**1 248**	128	9 784	19.1	✓	**1 160**	128	9 526	19.2	✓
CALT	1 331	512	**2 944**	**9.2**	✓	1 316	515	2 503	8.0	✓	1336	516	2 858	9.7	✓
CH	>72 000	-	-	-	-	2 442	-110	1 023	1.5	✓	4181	-124	426	0.3	✓
pCH-10%	25 051	-8	101 514	371.6	✓	1 865	-112	100 723	89.6	✓	4124	-124	100 640	64.8	✓
pCH-20%	13 249	-17	201 571	582.1	✓	1 848	-113	201 019	170.0	✓	4102	-124	201 125	131.5	✓
pCH-50%	5 984	**-29**	501 735	1207.2	✓	1 715	**-116**	501 790	407.2	✓	4081	**-124**	502 422	334.6	✓
CHASE	>72 000	-	-	-	-	2 533	-104	**595**	**0.5**	✓	4202	-121	**172**	**0.1**	✓
pCHASE-10%	>72 000	-	-	-	-	2 593	-89	6 064	2.8	-	4287	-112	4 762	1.8	-
pCHASE-20%	>72 000	-	-	-	-	3 453	-77	10 573	5.0	-	4481	-103	8 170	3.2	-
pCHASE-50%	>72 000	-	-	-	-	6 387	-46	23 886	11.7	-	5167	-78	18 253	7.6	-

law of 4. A possible explanation for this large amount of unnecessary edges can be found in [7]. Here, Chan et al. state that for full graphs using a power law greater than 2, the graph can be reduced to its Delaunay triangulation for finding shortest paths. Note that a Delaunay triangulation is a sparse graph with an average node degree of 6.

Finally, we observe again, that most of the analyzed speed-up techniques are Pareto-optimal. Thus, there exists no best solution but a variety of techniques each with its own advantages and disadvantages.

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 [25]: 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. We also introduced a variant of CHASE working for graphs where hierarchical preprocessing fails. This variant runs only a hierarchical query during the first phase and the second phase is only a goal-directed search, similar to CALT.

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 [17] in our work. He provided his implementation of [17] and some precomputed contraction hierarchies for various networks.

References

1. H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 46–59. SIAM, 2007.
2. H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.
3. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. SIAM, 2009. Accepted for publication, to appear.

4. R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. In I. Munro and D. Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, 2008.
5. R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In C. C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer, June 2008.
6. R. Bauer, D. Delling, and D. Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In C. Liebchen, R. K. Ahuja, and J. A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, pages 209–225. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
7. T. M. Chan, A. Efrat, and S. Har-Peled. Fly Cheaply: On the Minimum Fuel Consumption Problem. *Journal of Algorithms*, 41(2):330–337, 2001.
8. G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962.
9. D. Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, September 2008. Best Student Paper Award - ESA Track B.
10. D. Delling and G. Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.
11. D. Delling, T. Pajor, and D. Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, September 2008.
12. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway Hierarchies Star. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
13. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway Hierarchies Star. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2008. Accepted for publication, to appear.
14. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science. Springer, 2009. To appear.
15. D. Delling and D. Wagner. Landmark-Based Routing in Dynamic Graphs. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, June 2007.
16. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
17. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In C. C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.
18. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.
19. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, pages 129–143. SIAM, 2006.
20. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better Landmarks Within Reach. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 38–51. Springer, June 2007.
21. A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.
22. R. J. Gutman. 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)*, pages 100–111. SIAM, 2004.
23. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2008. To appear.

24. M. Holzer, F. Schulz, and D. Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*. SIAM, 2006.
25. M. Holzer, F. Schulz, D. Wagner, and T. Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. *ACM Journal of Experimental Algorithmics*, 10, 2006.
26. M. Holzer, F. Schulz, and T. Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. In *Proceedings of the 3rd Workshop on Experimental Algorithms (WEA'04)*, volume 3059 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2004.
27. E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science, pages 126–138. Springer, 2005.
28. U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. volume 22, pages 219–230. IfGI prints, 2004.
29. R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science, pages 189–202. Springer, 2005.
30. F. Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.
31. P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.
32. P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.
33. D. Schieferdecker. Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, January 2008.
34. D. Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008.
35. D. Schultes and P. Sanders. Dynamic Highway-Node Routing. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.
36. F. Schulz, D. Wagner, and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5, 2000.
37. F. Schulz, D. Wagner, and C. Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.
38. R. D. Team. R: A Language and Environment for Statistical Computing, 2004.
39. D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, 10:1.3, 2005.