Heuristic Contraction Hierarchies with Approximation Guarantee

Robert Geisberger. Dennis Schieferdecker
Karlsruhe Institute of Technology,
Institute for Theoretical Computer Science,
76128 Karlsruhe, Germany
{geisberger,schieferdecker}@kit.edu

October 25, 2010

Abstract

We present a new heuristic point-to-point shortest path algorithm based on contraction hierarchies (CH). Given an $\varepsilon \geq 0$, we can prove that the length of the path computed by our algorithm is at most $(1+\varepsilon)$ times the length of the optimal (shortest) path. Exact CH is based on *node contraction*: removing nodes from a network and adding shortcuts to preserve shortest path distances. Our heuristic CH tries to avoid adding shortcuts even when a replacement path is $(1+\varepsilon)$ times longer. However, we cannot avoid all such shortcuts, as we need to ensure that errors do not stack. Combinations with goal-directed techniques bring further speed-ups.

1 Introduction

The point-to-point shortest path problem in static road networks is essentially solved. There exist fast algorithms that are exact [**?**]. However, for other graph classes, these algorithms do not work very well. Also, when several objective functions should be supported within a road network current algorithms face some problems since the inherent 'hierarchy' of the graph changes with the used edge weights, e. g. time and distance. One possibility to alleviate these problems is to drop the exactness of the algorithms and allow some error. We show how to adapt contraction hierarchies (*CH*) [**?**] so that we can guarantee a multiplicative error of ε and extend it to use in combination with goal-directed techniques [**?**]. CH adds shortcuts to the graph to reduce the query search space. But when too many shortcuts are needed, as on some graph classes, the positive effect of them significantly decreases. Thus, our idea is to avoid some shortcuts by allowing a small error. It is straightforward to change the node contraction so that shortcuts are only added when a potential replacement path (*witness*) is more than a factor $(1+\varepsilon)$ longer. Our non-trivial contribution is how to ensure that errors do not stack during the contraction, and how to change the query algorithm so that it is still efficient.

Related Work

The classic shortest path algorithm for nonnegative edge weights is Dijkstra's algorithm that computes from one source node the shortest paths to all other nodes. During the execution of it, a node is either: *unreached*, *reached* (= open) or *settled* (= closed). It iteratively *settles* the reached node with the smallest tentative distance and updates the tentative distances of its neighbors by relaxing the edges of the settled node (= expanding the node).

However, on large graphs it is rather slow, so more sophisticated speed-up techniques have been developed. There has been extensive work on speed-up techniques for road networks [**?**]. All these techniques have in common that they perform precomputation to speed up shortest paths queries. We can classify current algorithms into three categories: *hierarchical* algorithms, *goal-directed* approaches and *combinations* of both.

Our algorithm is based on CH, a very efficient hierarchical algorithm. A CH orders the nodes by 'importance' and contracts the nodes in this order. A node is contracted by removing it from the network and adding *shortcuts* to preserve shortest paths distances. The original graph augmented by all shortcuts is the result of the preprocessing. A slightly modified bidirectional Dijkstra shortest path search then answers a query request, touching only a few hundred nodes. For our algorithm, we modify the node contraction, i.e. the decision which shortcuts we have to add, and the query.

Transit node routing [**?**] is the only faster hierarchical algorithm than CH. The most successful goal-directed algorithms are *ALT* [**?**] based on A* and landmarks, and Arc-Flags (*AF*) [**?**]. For AF, the graph is partitioned into cells, and each edge stores one flag (bit) per cell indicating whether this edge lies on a shortest path to this cell. Combinations of goal-direction and hierarchy are extensively studied by [**?**], including CHASE, a combination of CH and AF, and CALT, a combination of simple node contraction and ALT. We show how to extend CHASE to our heuristic scenario and introduce CHALT, a combination of CH and ALT with faster query times than CALT.

Weighted A* [**?**] is a heuristic variant of A*, where the heuristic function is weighted with $(1 + \varepsilon)$ and guarantees an error of ε. [**?**] gives an overview of further heuristic variants.

2 Heuristic Node Contraction

CH performs precomputation on a directed graph $G = (V, E)$, with edge weight function $c : E \to \mathbb{R}_+$. Each node is assigned an one-to-one importance level, i.e. $I(u) = 1..n$. Then, the CH is constructed by *contracting* the nodes in the above order. Contracting a node u means removing u from the graph without changing shortest path distances between the remaining (more important) nodes.

In the exact scenario, we want to preserve all shortest path distances. When we contract u, this is ensured by preserving the shortest path distances between the neighbors of u. So, given two neighbors v and w with edges (v, u) and (u, w), we should find the shortest path P between v and w avoiding u. When the length of P is longer than the length of the path $\langle v, u, w\rangle$, a shortcut edge between v and w is necessary with weight $c(v, u) + c(u, w)$. Otherwise, P is *witness* that no shortcut is necessary.

Algorithm 1: SimplifiedHeuristicConstructionProcedure($G = (V, E)$,I,ε)

```
1  $\tilde{c}$:= $c$;                              // store second weight per edge
2  foreach $u \in V$ ordered by $I(u)$ ascending do        // contract all
   nodes in order
3      foreach $(v, u) \in E$ with $I(v) > I(u)$ do
4          foreach $(u, w) \in E$ with $I(w) > I(u)$ do
5              find shortest path $P = \langle v, \ldots, w\rangle$ using only nodes $x$ with
               $I(x) > I(u)$;
6              if $c(P) > (1 + \varepsilon)(\tilde{c}(v, u) + \tilde{c}(u, w))$ then
7                  $E$:= $E \cup \{(v, w)\}$ (use weight $c(v, w)$:= $c(v, u) + c(u, w)$,
                   $\tilde{c}(v, w)$:= $\tilde{c}(v, u) + \tilde{c}(u, w)$);
8              else
9                  $\gamma$:= $\frac{c(P)}{\tilde{c}(v,u)+\tilde{c}(u,w)}$ - 1; //  $c(P) = (1 + \gamma)(\tilde{c}(v, u) + \tilde{c}(u, w))$
10                 foreach $(x, y) \in P$ do
11                     $\tilde{c}(x, y)$:= $\min\left\{\tilde{c}(x, y), \frac{c(x,y)}{1+\gamma}\right\}$;
```

In the heuristic scenario, we will not preserve the shortest path distances, but we still want to guarantee an error bound. Intuitively, we also want to avoid a shortcut between v and w, when the path P is just a bit longer than $\langle v, u, w\rangle$. To guarantee a maximum relative error of ε, we need to ensure that the errors do not stack when a node on P is contracted later. We call this algorithm approximate CH (*apxCH*) (Algorithm 1). When a witness P prevents a shortcut, even though in the exact scenario the shortcut would be necessary, the witness must remember this. We let the edges (x, y) of the witness P remember this by storing a second edge weight $\tilde{c}(x, y)$, so that Lemma 1 is fulfilled, and $\tilde{c}(P) \leq \tilde{c}(v, u)+\tilde{c}(u, w)$ (Lines 9–11). Intuitively, $\tilde{c}(P)$ stores the minimal length of a shortcut that P prevented as witness.

Lemma 1 *For each edge (v, w) holds*

$$\frac{c(v, w)}{1 + \varepsilon} \leq \tilde{c}(v, w) \leq c(v, w) \ .$$

A simple way to implement $\tilde{c}$ is to proportionally distribute the difference between $c(P)$ and $c(v, u) + c(u, w)$ among all edges of the witness. Example: Path $\langle u, v, w\rangle$ with $c(u, v) = 8,\ c(v, w) = 4$ prevents a shortcut of length 11. Thus,

$\tilde{c}(u,v) = 8/12 \cdot 11$, $\tilde{c}(v,w) = 4/12 \cdot 11$ unless $\tilde{c}(u,v)$ or $\tilde{c}(v,w)$ are already smaller. However, we could distribute it differently or even try to find other potential witnesses. Also, avoiding a shortcut can lead to more shortcuts later, as every shortcut is a potential witness later.

3 Heuristic Query

The basic apxCH query algorithm is the same as for CH. It is a symmetric Dijkstra-like bidirectional procedure performed on the original graph plus all shortcuts added during the preprocessing. However, it does not relax edges leading to nodes less important than the current node. This property is reflected in the *upward graph* $G_\uparrow := (V, E_\uparrow)$ with $E_\uparrow := \{(u,v) \in E \mid I(u) < I(v)\}$ and, analogously, the *downward graph* $G_\downarrow := (V, E_\downarrow)$ with $E_\downarrow := \{(u,v) \in E \mid I(u) > I(v)\})$.

We perform a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$. Forward and backward search are interleaved, we keep track of a tentative shortest-path length and abort the forward/backward search process when all keys in the respective priority queue are greater than the tentative shortest-path length (abort-on-success criterion).

Both search graphs $G_\uparrow$ and $G_\downarrow$ can be represented in a single, space-efficient data structure: an adjacency array. Each node has its own edge group of incident edges. Since we perform a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$, we only need to store an edge in the edge group of the less important incident node. This formally results in a search graph $G^* = (V, E^*)$ with $\overline{E_\downarrow} := \{(v,u) \mid (u,v) \in E_\downarrow\}$ and $E^* := E_\uparrow \cup \overline{E_\downarrow}$. Finally, we introduce a forward and a backward flag such that for any edge $e \in E^*$, $\uparrow(e) = \text{true}$ iff $e \in E_\uparrow$ and $\downarrow(e) = \text{true}$ iff $e \in \overline{E_\downarrow}$. Note that G^* is a directed acyclic graph (DAG).

In Lemma 2 we construct from an arbitrary path, a replacement path that can be found by our query algorithm.

Lemma 2 *Let $G = (V, E)$ be the graph after apxCH preprocessing with I and ε. Let P be an s-t-path in G. Then there exists an s-t-path P' in G of the form $\langle s = u_0, u_1, \ldots, u_p, \ldots, u_q = t\rangle$ with $p, q \in \mathbb{N}$, $I(u_i) < I(u_{i+1})$ for $i \in \mathbb{N}, i < p$ and $I(u_j) > I(u_{j+1})$ for $j \in \mathbb{N}, p \leq j < q$, called **path form** (PF). For P' holds $\tilde{c}(P') \leq \tilde{c}(P)$.*

Proof. Given a shortest s-t-path $P = \langle s = u_0, u_1, \ldots, u_p, \ldots, u_q = t\rangle$ with $p, q \in \mathbb{N}$ and $I(u_p) = \max I(P)$, that is not of the form (PF). Then there exists a $k \in \mathbb{N}, k < q$ with $I(u_k) < I(u_{k-1}), I(u_k) < I(u_{k+1})$. We will recursively construct a path of the form (PF).

Let $M_P := \{I(u_k) \mid I(u_k) < I(u_{k-1}), I(u_k) < I(u_{k+1})\}$ denote the set of local minima excluding nodes s, t. We show that there exists an s-t-path P' with $M_{P'} = \emptyset$ or $\min M_P < \min M_{P'}$.

Let $I(u_k) := \min M_P$ and consider the two edges $(u_{k-1}, u_k), (u_k, u_{k+1}) \in E$. Both edges already exist at the beginning of the contraction of node u_k. So there is either a witness path $Q = \langle u_{k-1}, \ldots, u_{k+1} \rangle$ consisting of nodes more important than u_k with $c(Q) \leq (1+\varepsilon)(\tilde{c}(u_{k-1}, u_k) + \tilde{c}(u_k, u_{k+1}))$ or a shortcut (u_{k-1}, u_{k+1}) of the same weight is added. So the subpath $P|_{u_{k-1} \to u_{k+1}}$ can either be replaced by Q or by the shortcut (u_{k-1}, u_{k+1}). If we replace the subpath by Q, our construction ensures that $\tilde{c}(Q) \leq \tilde{c}(P|_{u_{k-1} \to u_{k+1}})$. Also if we added a shortcut, $\tilde{c}(u_{k-1}, u_{k+1}) \leq \tilde{c}(P|_{u_{k-1} \to u_{k+1}})$ holds. So the resulting path P' consists of nodes more important than u_k and has the property $\tilde{c}(P') \leq \tilde{c}(P)$. Since $n < \infty$, there must exist an s-t-path P'' with $M_{P''} = \emptyset$, is therefore of the form described in (PF), and $\tilde{c}(P'') \leq \tilde{c}(P)$. □

Theorem 1 proves the correctness of our basic query algorithm by guaranteeing an error bound.

Theorem 1 *Given a source node s and a target node t. Let $\tilde{d}(s,t)$ be the distance computed by the apxCH algorithm with $\varepsilon \geq 0$ and let $d(s,t)$ be the optimal (shortest) distance in the original graph. Then $d(s,t) \leq \tilde{d}(s,t) \leq (1+\varepsilon)d(s,t)$.*

Proof. Let $(G = (V, E), I, \varepsilon)$ be an apxCH with $\varepsilon \geq 0$. Let $s, t \in V$ be the *source/target* pair of a query. It follows from the definition of a shortcut, that the shortest path distance between s and t in the apxCH is the same as in the original graph. So we will never find a shorter path in the shortcut-enriched graph, thus $d(s,t) \leq \tilde{d}(s,t)$ holds. Every shortest s-t-path in the original graph still exists in the apxCH but there may be additional s-t-paths. However since we use a modified Dijkstra algorithm that does not relax all incident edges of a settled node, our query algorithm does only find particular ones. In detail, exactly the shortest paths of the form (PF) are found by our query algorithm. From Lemma 2, we know that if there exists a shortest s-t-path P then there also exists an s-t-path P' of the form (PF) with $\tilde{c}(P') \leq \tilde{c}(P)$. Because of Lemma 1, we know that $\frac{c(P')}{1+\varepsilon} \leq \tilde{c}(P')$ and $\tilde{c}(P) \leq c(P)$ so that $c(P') \leq (1+\varepsilon)c(P)$. So our query algorithm will either find P' or another path, that is not longer than P'. □

Although we use $\tilde{c}$ in the proof, the query algorithm does not use it at all. So we only require $\tilde{c}$ during precomputation but we do not need to store it for the query. Also note that the correctness does not depend on the importance level $I(\cdot)$. However, in practice, the choice of $I(\cdot)$ has a big impact on the performance, see [**?**].

4 Heuristic Stall-on-Demand

In the previous section, we proved that the basic heuristic query algorithm does not need any changes compared to the exact scenario. However, there are changes necessary for the *stall-on-demand* technique, an important ingredient of a practically efficient implementation of CH. This single improvement brings additional speed-up of factor two or more. We will first explain how exact stall-on-demand *stalls* nodes that are reached with suboptimal distance. While a regular Dijkstra search would never do that, it can happen during a CH query since we do not relax

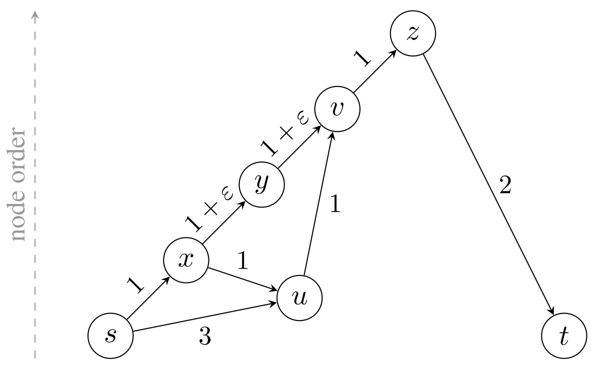

Figure 1: The stalling condition of the exact query fails, as node z is never reached in the forward search from s, since the path $\langle s, x, y, v\rangle$ to node v is stalled by the path via u.

edges leading to less important nodes. We call a path leading only upwards being an *upward* path. Our query algorithm can only find upward paths. There is a simple trick that allows us to check whether the currently settled node u is reached via a suboptimal upward s-u-path $P = \langle s = v_1, \dots, v_k = u\rangle$: for each more important neighbor v of u with edge (v, u) that was already reached by an upward path $\langle s, \dots, v\rangle$, we inspect the s-u-path $P' = \langle s, \dots, v, u\rangle$. As P' is no upward path, P' could be shorter than P. In this case, if $c(P) > c(P')$, we stall node u, i.e. we do not relax its incident edges. This is correct, as our exact CH query is correct and the suboptimal path P would never be part of an optimal path. We further try to even stall the reached neighbors w of u, if the path via v is shorter than their current tentative distance. For correctness, unstalling such a reached node w can be necessary when the search later finds a shorter upward path than the path via v.

However, in the heuristic scenario with $\varepsilon > 0$, we would destroy the correctness of our algorithm when we would apply the same rule, as our query algorithm no longer computes optimal paths. Consider as example the graph in Figure 1. During the contraction of u, no shortcut for the path $\langle x, u, v\rangle$ is added since the path $\langle x, y, v\rangle$ is a witness that is just a factor $(1 + \varepsilon)$ larger. The forward search starting at s should settle the nodes in the order s, x, y, u, v, z. However, if we would not change the stalling condition, we would stall u while settling it because the path $\langle s, u\rangle$ is longer than the path $\langle s, x, u\rangle$, which is not an upward path. Furthermore, we would propagate the stalling information to v, so node v reached via upward path $P = \langle s, x, y, v\rangle$ gets stalled by the shorter path $P' = \langle s, x, u, v\rangle$. Thus, node v is stalled and we would never reach node z with the forward search and therefore could never meet with the backward search there.

To ensure the correctness, we change the stalling condition. We split the path s-u-path P' in paths P'_1 and P'_2 so that P'_1 is the maximal upward subpath starting at s. Let x be the node that splits P' in these two parts, i.e. $P'_1 = P'|_{s\to x}$ and

Algorithm 2: HeuristicQuerySOD(s,t)

```
1  $d_\uparrow$:= $\langle\infty,\dots,\infty\rangle$; $d_\uparrow[s]$:= 0; $d_\downarrow$:= $\langle\infty,\dots,\infty\rangle$; $d_\downarrow[t]$:= 0, $d$:= $\infty$;
   // tentative distances
2  $Q_\uparrow = \{(0,s)\}$; $Q_\downarrow = \{(0,t)\}$; $r$:= $\uparrow$;          // priority queues
3  while ($Q_\uparrow \neq \emptyset$ or $Q_\downarrow \neq \emptyset$) and ($d > \min\{\min Q_\uparrow, \min Q_\downarrow\}$) do
4      if $Q_{\neg r} \neq \emptyset$ then $r$:= $\neg r$;   // interleave direction, $\neg\uparrow=\downarrow$
       and $\neg\downarrow=\uparrow$
5      $(\cdot,u)$:= $Q_r$.deleteMin(); $d$:= $\min\{d, d_\uparrow[u] + d_\downarrow[u]\}$;        // $u$ is
       settled and new candidate
6      if isStalled($r$, $u$) then continue; // do not relax edges of a
       stalled node
7      foreach $e = (u,v) \in E^*$ do                    // relax edges of $u$
8          if $r(e)$ and $(d_r[u] + c(e) < d_r[v])$ then       // shorter path
           found
9              $d_r[v]$:= $d_r[u] + c(e)$; // update tentative distance
10             $Q_r$.update($d_r[v]$,$v$);          // update priority queue
11             if isStalled($r$, $v$) then unstall($r$, $v$);
12         if $(\neg r)(e) \wedge d_r[v] + (1+\varepsilon)c(e) < d_r[u]$ then   // path via $v$
           is shorter
13             stall($r$, $u$, $d_r[v] + (1+\varepsilon)c(e)$);          // stall $u$ with
               stalling distance $d_r[v] + (1+\varepsilon)c(e)$
14             break; // stop relaxing edges of stalled node
               $u$
15 return $d$;
```

$P'_2 = P'|_{x\to u}$. Then we stall u only if node x is reached by the forward search and

$$c(P'_1) + (1+\varepsilon)c(P'_2) < c(P) \tag{1}$$

The symmetric condition applies to the backward search, see Algorithm 2 for pseudo-code. Note that for $\varepsilon = 0$, this algorithm corresponds the the exact query algorithm with stall-on-demand.

To prove that stall-on-demand with (1) is correct, we will iteratively construct in Lemma 3 a new path from a stalled one.

Lemma 3 *Let (P, v, w) be a **s**tall **s**tate **t**riple (SST): P being an s-t-path of the form (PF), node v being reached by the forward search by $P|_{s\to v}$ and not stalled and node w being reached by the backward search by $P|_{w\to t}$ and not stalled. Define a function g on an SST:*

$$g(P,v,w) := c(P|_{s\to v}) + (1+\varepsilon)\tilde{c}(P|_{v\to w}) + c(P|_{w\to t}).$$

If one of the nodes in $P|_{v\to w}$ *becomes stalled, then there exists an SST* (Q,x,y) *with*

$$g(Q,x,y) < g(P,v,w).$$

Proof. Let $u \in P|_{v\to w}$ be the node that becomes stalled. W.l.o.g. we assume that $P|_{v\to u}$ is an upward path, i.e. the stalling happens during the forward search. Then there exists an s-u-path P' that is split in P_1' and P_2' as defined in (1) so that $c(P_1') + (1+\varepsilon)c(P_2') < c(P|_{s\to u})$. Let x be the node that splits P' into these two subpaths. Let R be the path of form (PF) that is constructed following Lemma 2 from the concatenation of P_2' and $P|_{u\to w}$. Let Q be the concatenation of P_1', R and $P|_{w\to t}$ and $y{:=}\ w$. By construction, (Q,x,y) is a SST and we will prove that it is the one that we are looking for:

$$\begin{aligned}
&g(Q,x,y)\\
&= c(Q|_{s\to x}) + (1+\varepsilon)\tilde{c}(Q|_{x\to y}) + c(Q|_{y\to t})\\
&\overset{\text{def.}}{=} c(P_1') + (1+\varepsilon)\tilde{c}(R) + c(P|_{w\to t})\\
&\overset{\text{L.2}}{\leq} c(P_1') + (1+\varepsilon)(\tilde{c}(P_2') + \tilde{c}(P|_{u\to w})) + c(P|_{w\to t})\\
&\overset{\text{L.1}}{\leq} c(P_1') + (1+\varepsilon)c(P_2') + (1+\varepsilon)\tilde{c}(P|_{u\to w})\\
&\quad + c(P|_{w\to t})\\
&\overset{(1)}{<} c(P|_{s\to u}) + (1+\varepsilon)\tilde{c}(P|_{u\to w}) + c(P|_{w\to t})\\
&= c(P|_{s\to v}) + c(P|_{v\to u}) + (1+\varepsilon)\tilde{c}(P|_{u\to w})\\
&\quad + c(P|_{w\to t})\\
&\overset{\text{L.1}}{\leq} c(P|_{s\to v}) + (1+\varepsilon)\tilde{c}(P|_{v\to u}) + (1+\varepsilon)\tilde{c}(P|_{u\to w})\\
&\quad + c(P|_{w\to t})\\
&= g(P,v,w)
\end{aligned}$$

□

With Lemma 3 we are able to prove the correctness of heuristic stall-on-demand (1) in Theorem 2.

Theorem 2 *Theorem 1 still holds when we use heuristic stall-on-demand (1).*

Proof. The proof will iteratively construct SSTs with Lemma 3 starting with the path P found in the proof of Theorem 1/Lemma 2 and the nodes s and t. Obviously, at the beginning of the query, both nodes s and t are reached and not stalled, so (P,s,t) is an SST and

$$\begin{aligned}
&g(P,s,t)\\
&= c(P|_{s\to s}) + (1+\varepsilon)\tilde{c}(P|_{s\to t}) + c(P|_{t\to t})\\
&= (1+\varepsilon)\tilde{c}(P)\\
&\leq (1+\varepsilon)d(s,t)\ .
\end{aligned}$$

We will prove that after a finite number of applications of Lemma 3, we obtain an SST (Q,x,y) so that Q is found by our query with stalling. For this path Q holds:

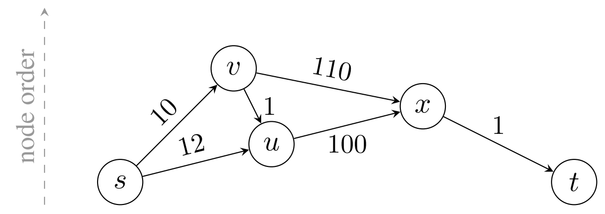

Figure 2: Stalling may increase the observed error ($\varepsilon = 10\%$). Node u gets stalled while being on the shortest path of form (PF).

$$\begin{aligned} c(Q) &= c(Q|_{s\to x}) + c(Q|_{x\to y}) + c(Q|_{y\to t}) \\ &\overset{\text{L.1}}{\leq} c(Q|_{s\to x}) + (1+\varepsilon)\tilde{c}(Q|_{x\to y}) + c(Q|_{y\to t}) \\ &= g(Q,x,y) \\ &\overset{\text{L.3}}{\leq} g(P,s,t) \\ &\leq (1+\varepsilon)d(s,t) \end{aligned}$$

Since our graph is finite, and due to the "<" in Lemma 3, we can apply Lemma 3 only finitely many times. The final SST (Q, x, y) will be found by our query algorithm since x is reached in the forward search and not stalled, y is reached in the backward search and not stalled. And since this is the final SST, no node on the path $Q|_{x\to y}$ will be stalled. Thus, our query will find the path Q or a shorter path. $\square$

Note that stall-on-demand can still increase the observed error of the query, we just proved that it will never be larger than ε. Look at the example in Figure 2. The shortest s-t-path $\langle s, v, u, x, t\rangle$ has length 112. During the contraction of node u, no shortcut (v, x) was added as the existing edge is less than 10% longer. During the query from s to t, path $\langle s, v, u\rangle$ of length 11 stalls node u reached with length 12. So the query does not find the path $\langle s, u, x, t\rangle$ of length 113 but instead the path $\langle s, v, x, t\rangle$ of length 121 having an error of 8%.

5 Improved Node Ordering

Node ordering is the process to compute the importance levels $I(\cdot)$. The node ordering is done heuristically, as the computation of an optimal node ordering (i.e. shortcut minimal or query search space minimal) is NP-hard [**?**]. We assign each remaining node a priority on how attractive it is to contract this node. The priority is a linear combination of several terms [**?**]. There are terms to keep the number of shortcuts low, e.g. the edge difference between the number of necessary shortcuts and the number of incident edges of the node, and to keep the search spaces small, e.g. the number of contracted neighbors. We iteratively contract the node with lowest priority and update the priorities of the remaining nodes. Updating these priorities takes the most time during precomputation, as computing the number of necessary shortcuts takes as much time as computing the set of necessary shortcuts. Therefore, [**?**] already update only (a) the priorities of the neighbors of the contracted node. As this does not catch all nodes with affected priority, they (b) repeatedly update the priority of the node on top of the priority queue (*lazy update*) and reinsert it until it does not change anymore. So nodes with increased priority become updated in time. They further (c) update the priorities of all remaining nodes when too many of these reinserts happened. This works very well for road networks, but we observed in our experiments on other graph classes that we can significantly reduce precomputation time by skipping (a) and (c), and only rely on (b). We call this optimization *OLU* (only lazy updates).

6 Combination with Goal-directed Techniques

The CHASE algorithm combines CH and AF. Its preprocessing computes a CH and then computes AF on a small core, consisting of the most important nodes. AF preprocessing is usually very time- and space-consuming on a large graph, much larger than a CH preprocessing. By applying AF only to a small core, we can get faster queries than CH or AF alone at only slightly increased preprocessing costs compared to CH. The CHASE query is performed in two phases, first a CH query that does not relax edges within the core, and second a CH query within the core guided by arc flags. The AF computation partitions the core into k cells. To set the arc flags, we could compute the backward shortest path DAG (not a tree due to paths of same length) from each node and set the arc flag for the cell of the node for exactly each edge in this DAG. But it is sufficient to only do this from boundary nodes that have a neighbor in another cell [**?**]. As we perform a bidirectional query, we also compute symmetric arc flags using forward shortest path DAGs.

CHASE can be adapted to our heuristic scenario. Our apxCHASE preprocessing uses an apxCH, and determines arc flags by a modified backward search that only considers paths of the form (PF). For our apxCHASE query, we need to employ the changes to the stall-on-demand technique. Additionally, a path P' can only stall an upward path P if the target arc-flags are set on all edges of P'.

As on some graphs, ALT is superior to AF, we also propose to combine apxCH with ALT to obtain the apxCHALT algorithm. The pattern is the same, we first compute a apxCH and then apply ALT on a small core. ALT preprocessing selects landmarks L heuristically and then computes the shortest path distances from/to all nodes u in the core using Dijkstra's algorithm. If the target node t is in the core, the minimum of the distances $d(L,t) - d(L,u)$ and $d(u,L) - d(t,L)$ is a lower bound on the distance $d(u,t)$ used for the heuristic function. If not, proxy nodes in the core are introduced [**?**]. It is symmetric to obtain a lower bound from the source node. The apxCHALT query algorithm uses ALT instead of AF, but is still performed in two phases.

	preproc.		query			preproc.		query		
	[s]	[B/n]	#settled	[ms]	error	[s]	[B/n]	#settled	[ms]	error
sensor	*average degree 10*					*average degree 20*				
bidir. Dijkstra	0	0	326 597	127.1	-	0	0	327 626	181.2	-
CALT	62	165	954	1.4	-	188	432	2 616	4.4	-
bidir. AF	8 753	322	7 002	2.6	-	48 055	641	10 838	5.2	-
bidir. ALT-a64	194	512	3 173	2.8	-	240	512	3 852	4.6	-
bidir. WALT-a64-10%	194	512	687	1.3	0.99%	240	512	437	1.5	1.17%
bidir. WALT-a64-21%	194	512	636	1.3	1.86%	240	512	404	1.5	1.86%
unidir. ALT-a64	97	256	8 248	4.9	-	120	256	6 782	5.5	-
unidir. WALT-a64-10%	97	256	845	0.9	2.59%	120	256	372	0.8	1.62%
unidir. WALT-a64-21%	97	256	692	0.8	4.24%	120	256	327	0.8	2.30%
unidir. A*	0	16	57 385	36.6	-	0	16	31 928	31.2	-
unidir. WA*-10%	0	16	1 234	1.0	1.25%	0	16	308	0.61	1.16%
unidir. WA*-21%	0	16	724	0.7	2.87%	0	16	272	0.58	1.86%
CH	20 578	-2	2 816	2.9	-	$>$ 2 days	-	-	-	-
CH OLU	1 887	0	2 969	4.0	-	82 243	31	9 232	37.8	-
apxCH-1%	993	-4	2 742	2.7	0.16%	14 025	-2	7 657	17.6	0.19%
apxCH-10%	474	-18	2 584	1.9	2.17%	2 767	-48	5 496	6.6	1.75%
CHALT	20 597	22	257	0.5	-	$>$ 2 days	-	-	-	-
CHALT OLU	1 907	26	251	0.6	-	82 296	57	924	4.1	-
apxCHALT-1%	1 011	21	243	0.5	0.15%	14 057	22	784	2.2	0.19%
apxCHALT-10%	489	7	215	0.3	2.16%	2 786	-23	475	1.0	1.75%
apxCHALT-10% W-10%	489	7	102	0.2	3.56%	2 786	-23	269	0.45	3.20%

Table 1: Performance of our approximate algorithms on sensor networks.

7 Experiments

Environment. Experiments have been done on one core of a dual Xeon 5345 processor clocked at 2.33 GHz with 16 GB main memory and $2 \times 2\times$ 4 MB of cache, running SuSE Linux 11.1 (kernel 2.6.27). The program was compiled by the GNU C++ compiler 4.3.2 using optimization level 3.

Test Instances. We use the largest strongly connected component of the road network of Western Europe, provided by PTV AG for scientific use, with 18 million nodes and 42.2 million edges. The second class of instances are unit disk graphs with 1 000 000 nodes and with an average degree of 10 and 20, modelling sensor networks with limited connection range (*sensor*). We also use grid graphs of 2 and 3 dimensions having 250 000 nodes, with edge weights picked uniformly at random between 1 and 1 000.

Setup. We report results in Tables 1–3. Graphs are stored explicitly in main memory as adjacency array. We compare the algorithms in the three-dimensional space of preprocessing time, preprocessing space and query time. Usually, there is not a single best algorithm, but there are several ones providing different tradeoffs between these three dimensions. The preprocessing space is the *space overhead* compared to the space a bidirectional Dijkstra needs. We state it as Bytes per node [B/n], as for our graph classes, the number of edges is roughly linear in the number of nodes. The number of settled nodes, runtime, and error are average over 10 000 shortest path distance queries, selected uniformly at random. Although the number of settled nodes gives a rough estimate on the runtime of the query, there can be deviations: Reasons are cache locality (we observed 20% difference in runtime by just choosing different node ids), and more shortcuts on most important nodes, so that settling those is more expensive, and the cost for stall-on-demand. For

	preprocessing		query			preprocessing		query		
	[s]	[B/n]	#settled	[ms]	error	[s]	[B/n]	#settled	[ms]	error
Europe			*travel time*					*distance*		
bidir. Dijkstra	0	0	4.714 M	1 991	-	0	0	5.309 M	1 547	-
TNR[1]	6 720	204	N/A	0.0034	-	9 720	301	N/A	0.038	-
TNR+AF[1]	13 740	321	N/A	0.0019	-	-	-	-	-	-
CH	1 510	-3	353	0.125	-	4 433	0	1 628	1.293	-
CH OLU	1 050	-1	430	0.206	-	1 258	0	1 333	1.198	-
apxCH-10%	1 099	-2	430	0.199	0.40%	950	0	1 248	0.873	1.32%
CHASE	8 699	4	44	0.023	-	72 278	12	73	0.062	-
CHASE OLU	13 421	7	42	0.028	-	84 759	15	59	0.058	-
apxCHASE-10%	11 977	5	42	0.026	0.40%	33 147	10	62	0.048	1.32%
CHALT	1 703	22	146	0.114	-	4 658	25	192	0.318	-
CHALT OLU	1 257	23	149	0.155	-	1 491	26	159	0.300	-
apxCHALT-10%	1 300	23	153	0.147	0.40%	1 163	24	232	0.322	1.32%
apxCHALT-10% W-10%	1 300	23	106	0.111	0.65%	1 163	24	70	0.116	2.14%

Table 2: Performance of our approximate algorithms on road networks.

CH node ordering, we use the aggressive variant from [?] to determine the node priorities. CHASE uses $k = 128$ cells for AF, CHALT uses 64 avoid landmarks, both on a core of the 5% highest ordered nodes.

Improved node ordering. Adding the OLU optimization to CH reduces preprocessing time on sensor networks and the 3-dimensional grid network by one order of magnitude. We cancelled the normal CH preprocessing of *sensor20*, it would probably have taken 10 days. On the road network, we see a more differentiated picture. For CH, the preprocessing for travel time metric is almost 3 times faster than for distance metric, both on the same graph. With OLU we are able to decrease the difference to a factor of 1.2. The difference is due to the travel time metric featuring a hierarchy with fast highways and slower roads, so that most of the long shortest paths use the highways. In contrast, the distance metric (also used on the sensor networks) does not necessarily prefer the highways so that more shortcuts are needed and a larger number of nodes has to be explored during a query. But OLU can also decrease the performance, e.g. the query time with travel time metric increases, and also the preprocessing time for CHASE, this is because there are more boundary nodes.

Approximate CH. ApxCH uses OLU, and further decreases preprocessing time and space by allowing some error, although the observed error is much smaller than the error bound. On *sensor20*, apxCH-10% ($\varepsilon = 10\%$) reduces preprocessing time by a factor of 30 and has negative space overhead. The negative space overhead is possible due to the adjacency array representation, as a bidirectional edge in a CH is only stored with the less important endpoint. So, when we add fewer shortcuts than there are input edges, we achieve a negative space overhead.

Node contraction is fast on sparse networks that also stay sparse in the remaining graph during contraction. But on sensor networks, slight variations in source and target positions suddenly make another path the shortest one, so that CH works bad as a lot of shortcuts are necessary. ApxCH works very well on these graphs, as there are a lot of similar paths with similar lengths, so we can omit a lot of shortcuts by allowing some error. Road networks have a different structure than sensor networks. Due to the travel time metric, there is hierarchy so that CH works well as we mostly need shortcuts only for the fast highways. Also, there are not a lot of similar paths, as most go through these highways, so that apxCH cannot skip a lot of shortcuts and brings no advantage. The distance metric exhibits less hierarchy, thus also slow roads become important when they represent a short path to the target, and there are more similar paths. So, apxCH shows some improvements over CH there. The grid networks have some hierarchy due to the random edge weights, but it is less structured so that apxCH shows only some improvements, especially in the preprocessing time.

Approximate CHALT. The best query times on the sensor networks are achieved using goal-direction. We report results for ALT, A* (based on coordinates on the disk) and CHALT, and also for their weighted variant (marked with W). The coordinates are stored in two double values ($2\times$ 8 Byte/node). Using ALT is faster than A*, whereas bidirectional ALT is faster than unidirectional ALT. But the weighted variant has just the opposite order, A* based on coordinates is the fastest with 0.58 ms on *sensor20* using $\varepsilon = 21\%$. It seems that having a denser network helps WA* based on coordinates, as an about 3 times smaller search space is explored for *sensor20* in comparison to *sensor10*. So apxCHALT-10% is faster than WA-10% on *sensor10*, but slower on *sensor20*. When we use weighed A* with CHALT, we get the fastest query time of 0.45 on *sensor20*, being 20% faster than WA*-21% and even 9 times faster than CHALT OLU. We compare to WA*-21%, as apxCHALT-10% W-10% has a total error bound of 21% as the errors multiply. You may note that both have almost the same number of settled nodes. But as CHALT has fewer cache misses, due to a node numbering in the adjacency array based on the importance levels and the upward-only query, CHALT is faster in practice. The second advantage of apxCHALT-10% W-10% over WA*-21% is the smaller space overhead in adjacency array representation.

Approximate CHASE. We only report results for CHASE on the road and grid networks, as the preprocessing on the sensor networks took more than 2 days. CHASE has a smaller preprocessing space and query times than CHALT. Furthermore, on road networks it is the fastest speed-up technique except for TNR[1]. However, in comparison to CHALT and CH, the preprocessing time is very large, especially for the distance metric. With apxCHASE-10%, we are able to reduce the preprocessing time by a factor of 2.

8 Applications

We described our heuristic for a single edge weight function and tested it on some graph classes. They should provide comparable performance on other sim-

[1]Experiments done on a 2.0 GHz AMD Opteron running SuSE Linux 10.0 with 8 GB of RAM and 2x1 MB of L2 cache.

	preproc.		query			preproc.		query		
	[s]	[B/n]	#settled	[ms]	error	[s]	[B/n]	#settled	[ms]	error
grid	*2-dimensional*					*3-dimensional*				
bidir. Dijkstra	0	0	80 168	22.59	-	0	0	44 244	19.78	-
CALT	40	226	445	0.82	-	53	409	598	1.30	-
bidir. AF	622	130	1 369	0.34	-	6 287	189	1 718	0.62	-
bidir. ALT-a64	42	512	1 083	0.86	-	55	512	722	1.10	-
CH	59	0	409	0.12	-	9 205	14	2 207	1.82	-
CH OLU	30	1	408	0.14	-	1 088	18	2 236	2.54	-
apxCH-10%	26	-1	388	0.13	0.70%	605	8	2 124	2.00	0.19%
CHASE	105	11	102	0.04	-	10 051	62	807	0.59	-
CHASE OLU	87	14	91	0.04	-	2 344	74	749	0.72	-
apxCHASE-10%	68	9	85	0.04	0.70%	1 418	49	777	0.66	0.19%
CHALT	61	25	90	0.06	-	9 210	39	524	0.65	-
CHALT OLU	33	26	80	0.07	-	1 094	44	494	0.77	-
apxCHALT-10%	28	23	76	0.06	0.70%	610	34	434	0.62	0.19%
apxCHALT-10% W-10%	28	23	55	0.05	1.38%	610	34	361	0.53	0.59%

Table 3: Performance of our approximate algorithms on grid networks.

ilar graph classes, e.g. game networks or communication networks. Other areas may be time-dependent road networks, where not only the travel time functions are approximations, but also the shortcuts. Also, it can help for multi-criteria optimization. It would be simple to extend it to the flexible scenario [**?**] with two edge weight functions. There, a lot more shortcuts than in the single-criteria scenario are added, which significantly increases preprocessing time and space. As our heuristics reduce the number of shortcuts, this can bring a big improvement.

9 Conclusion

We developed an approximate version of contraction hierarchies with guaranteed error bound. In our experimental evaluation, we showed that on certain graph classes, this new version is able to reduce preprocessing time and space, and also query time by an order of magnitude. Query times are further decreased by combination with AF or ALT.

Continuing work should be done on testing our algorithms on other graphs. Further tuning on the algorithms is possible, too. The node ordering priorities are currently optimized for road networks, so there is potential for improvement. Also, the shortcuts are currently avoided in a greedy fashion. Using smarter approaches may further decrease the number of necessary shortcuts.

Acknowledgments. Partially supported by DFG Research Training Group GRK 1194 and DFG grant SA 933/5-1.