The Eleventh International Symposium on Combinatorial Search
(SoCS 2018)

Forward Search in Contraction Hierarchies

Daniel D. Harabor
Monash University
Melbourne, Australia
daniel.harabor at monash.edu

Peter J. Stuckey
DATA61 and The University of Melbourne
Melbourne, Australia
pstuckey at unimelb.edu.au

Abstract

Contraction hierarchies are graph-based data structures developed to speed up shortest path search in road networks. Built during an offline pre-processing step, contraction hierarchies are always paired with an online query algorithm which is a variation on bi-directional Dijkstra search. Though effective and highly popular this combination can sometimes be difficult to extend, for example in order to leverage goal-directed heuristics or other forward-driven pruning techniques. In this paper we deconstruct the bi-directional query algorithm of contraction hierarchies and derive a new algorithmic schema which is compatible with standard uni-directional or bi-directional search. We then develop a variety of new uni-directional query algorithms to find optimal paths in contraction hierarchies. These are based on the combination of A* search and Geometric Containers, a well known and successful edge-pruning technique. Empirical results show that our approach can improve search times by an order of magnitude vs bi-directional Dijkstra, albeit at the cost of additional memory and pre-processing time.

Introduction

A contraction hierarchy (Geisberger et al. 2008; 2012) can be understood as a type of multi-level graph abstraction. Created as part of a pre-processing step, this auxiliary data structure is used to quickly compute shortest paths, from one node in the graph to another. The intuition is simple: during search one follows edges *up* into the hierarchy to reach nodes that are "far away" and later *down* into the hierarchy to reach nodes that are "nearby" [1] We will refer to this pairing (bi-directional Dijkstra + contraction hierarchies) as BCH.

BCH shares many similarities with abstraction-based search techniques such as HPA* (Botea, Müller, and Schaeffer 2004), MMA (Sturtevant 2007) and, more recently, SUB (Uras, Koenig, and Hernàndez 2013). There are two notable differences:

- BCH is both fast and optimal while abstraction-based algorithms commonly trade optimality for speed.
- BCH always invokes a bi-directional Dijkstra search while abstraction-based algorithms are more general and also compatible with uni-directional search.

In this paper we deconstruct BCH and give a new perspective on search in contraction hierarchies. First, we show that BCH is the combination of a specific graph traversal policy with a particular search technique. A traversal policy in this context refers to a rule which specifies when the search should go "up" in the hierarchy and when it should go "down". We show that BCH uses an implicit traversal policy which we call AUP (short for *always up*). We then give another possible traversal policy, UDP (short for *up-then-down*), and show that this alternative method can be paired with different search strategies, including uni-directional search.

The first advantage of such a generalisation is performance. UDP allows us to easily combine contraction hierarchies with A* search and, by extension, a wide variety of goal-directed pruning techniques such as, for example, Geometric Containers (Wagner, Willhalm, and Zaroliagis 2005). A main result is that such pairings can be over one order of magnitude faster than BCH, at the cost of additional pre-processing time and memory overheads.

The second advantage is an improved theoretical understanding of contraction hierarchies. Previous research has improved on the performance of BCH using goal-directed pruning, e.g. (Batz et al. 2009; Bauer et al. 2010; Storandt 2013), but the resulting techniques are not strict generalisations. In particular all prior works involve some type of online bi-directional search as a basic ingredient. Despite their success, there are nonetheless several good reasons why one might like to apply a strict forward search instead: (i) two-stage algorithms can be complicated to derive and understand and involve more effort to implement; (ii) two-stage search schemas are often not easily and/or not as effectively combined with arbitrary goal-directed pruning techniques, as compared to simple forward search. (iii) two-stage algorithms that rely on bidirectional search can expand the same nodes twice, effectively duplicating the amount of work. This is also the main disadvantage of bidirectional search (interleaved or otherwise).

[1] The intuition draws on empirical observation: we have seen that going higher generally allows one to travel further.

Preliminaries

A *weighted graph* graph $G = (V, E)$ comprises a set of vertices (equiv. nodes) V and a set of directed edges $E : V \times V$. Each edge $(i, j) \in E$ represents a traversable link (e.g. a road) from node i to node j. More, each edge has a real-valued and non-negative weight $c_{i,j}$ that represents the cost (usually in time or distance) of traversing the link. An *in-neighbour* of node j is a node i where edge (i, j) exists, and *out-neighbour* of node i is a node j where edge (i, j) exists.

A *path* $\pi_{s,t} = \langle v_s, \ldots, v_t \rangle$ is a sequence of nodes which connect a start point s to a target point t. Each adjacent pair of nodes on a path is connected by an edge and the cost of a path is equal to the sum of its edge weights. A path is *optimal* if its cost is minimum among all paths between s and t. We distinguish such paths using the standard notation $\pi^*_{s,t}$.

Throughout this paper we will assume that G has an associated and admissible heuristic function h; i.e. each node has a pair of coordinates x and y (equiv. lat and lng when G is a road network) and for any pair of nodes $h(s, t) \leq \pi^*_{s,t}$ (h is always a lower-bound on the optimal distance between s and t).

Contraction Hierarchies

Building a contraction hierarchy is a simple process requiring only the repeated application of an eponymous contraction operation to the nodes of the input graph G. In broad strokes:

1. Apply a total lex order $\mathcal{L}$ to the nodes of G.
2. W.r.t. $\mathcal{L}$, choose the least node v from the graph that has not been previously selected.
3. (Contraction) Add to G a shortcut edge (u, w) between selected pairs of in-neighbour u and out-neighbour w of v. The nodes u and w are selected if both u and w are lexically larger than v and if there exists a subpath $\langle u, v, w \rangle$ in G. The cost of the shortcut edge is always $c_{u,w} = c_{u,v} + c_{v,w}$.
4. Repeat steps 2-3 until every node in the graph has been contracted [2].

Figure 1 shows one possible contraction of a small toy graph. Notice that the added shortcut edges offer an opportunity to connect the start and target node faster than would otherwise be possible. For example, in the original graph, the optimal path between the start and target node requires traversing six edges: $\langle 6, 1, 10, 9, 11, 3, 5 \rangle$. An equivalent-cost path, which uses shortcuts and traverses only three edges, is $\langle 6, 10, 11, 5 \rangle$.

Computing shortest paths in a contraction hierarchy requires a search algorithm which can both exploit shortcuts and also prune edges which are redundant. In the example, there is no point following edge $(10, 9)$ if we also follow $(10, 11)$. The only reason to consider $(10, 9)$ is if the target is one of the intermediate nodes the shortcut is bypassing (i.e. 9, 7, or 8). The main query algorithm used in this context is BCH, which we discuss shortly.

[2] The number of shortcuts can be kept to a minimum by further requiring that the subpath $\langle u, v, w \rangle$ is both unique and optimal. This normally yields smaller graphs and faster query performance but also requires additional pre-processing time.

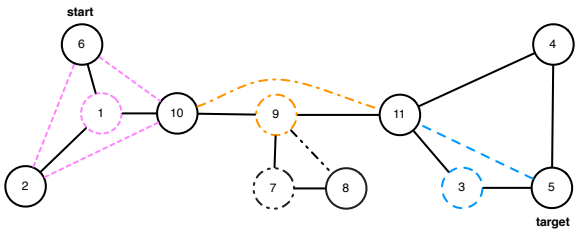

Figure 1: Contraction hierarchies applied to a small (undirected, for simplicity) toy graph. Node labels indicate contraction order. Solid edges appear in the original input graph. Dashed edges represent added shortcuts. We use a variety of dashed styles and colours to indicate which nodes, when contracted, result in which shortcuts. Also indicated is one possible start-target pair.

Practical Considerations

The successful application of contraction hierarchies requires practitioners to carefully balance the efficacy of contraction (as measured on the one hand by total pre-processing time and on the other hand by the total space required to store all shortcut edges) against the efficiency of online search (as measured by total node expansions and total edge relaxation operations). Intuitively, a "good" hierarchy allows one to reach any target node from any start node with only a "small" search effort. There are three practical questions to consider when building a contraction hierarchy:

1. How to order the nodes for contraction?
2. How to decide when to add a new shortcut edge?
3. How to traverse the resulting hierarchy?

Questions (1) and (2) have been intensely studied in the literature. Though outside the immediate scope of our work we will, for added clarity and context, briefly discuss the main challenges and identify some related works. Question (3), at which this paper is aimed, has received much less attention. We will discuss, and subsequently deconstruct, BCH: the main algorithm used to compute shortest paths in contraction hierarchies.

Node ordering heuristics

It is well known that a "good" ordering of the input nodes results in a contraction hierarchy which is more efficient to search. For example, we might consider a hierarchy efficient if the number of edges, from any node to the topmost node, is logarithmic in the height of the hierarchy. Likewise, we might consider inefficient a linear hierarchy as it provides no speedup. A variety of practical and efficient ordering heuristics are discussed in (Geisberger et al. 2008) and (Abraham et al. 2012). Recent and more sophisticated works, based on graph separators, are discussed in (Strasser and Wagner 2015).

Contraction approaches

A simple but naive approach to contraction is to introduce a new shortcut edge for every pair of in-neighbour and out-neighbour both lexically larger than the current node. This procedure has a small overhead in terms of pre-processing time but a potentially large overhead in terms of memory. Additionally, many of the added edges are redundant (i.e. they never appear on any shortest path) and their presence only serves to slow down pathfinding search in the hierarchy.

A variety of alternative contraction techniques, with different time/memory tradeoffs, have appeared in the literature; e.g. in (Geisberger et al. 2008) and (Geisberger et al. 2012). These approaches have been shown to work well in practice. A theoretical analysis on the advantages of different contraction approaches is given in (Bauer et al. 2013).

BCH

A core idea of contraction hierarchies is that shortcut edges can be used to bypass one or more intermediate nodes from the graph in a single step. However, for each shortcut edge (u, w) and each intermediate node v we have $f(v) \leq f(w)$; i.e. given a monotonically increasing cost function a simple best-first search will usually expand v before w in order to compute an optimal path. To achieve a speedup the authors of (Geisberger et al. 2008) divide the set of edges E into two as follows:

- $E_{\uparrow} = \{(u, v) \in E \mid u <_{\mathcal{L}} v\}$
  (i.e. the set of all "up" edges); and
- $E_{\downarrow} = \{(u, v) \in E \mid u >_{\mathcal{L}} v\}$
  (i.e. the set of all "down" edges).

The following results, paraphrased here, are due to (Geisberger et al. 2008).

Lemma 1 ***(ch-path)****: For every optimal path $\pi^*_{s,t}$ in E there is a cost equivalent alternative $\pi'_{s,t}$ whose prefix $\langle s, \ldots k\rangle$ is found in $E_{\uparrow}$ and whose suffix $\langle k \ldots t\rangle$ is found in $E_{\downarrow}$.* □

Corollary 1 ***(apex node)****: Every ch-path has a node k which is lexically largest among all nodes on the path.* □

Following Lemma 1, a natural decomposition of the shortest path problem in a contraction hierarchy is the following: first compute a subpath $\langle s, \ldots, k\rangle$ in $E_{\uparrow}$; next, compute a second subpath $\langle k, \ldots, t\rangle$ in $E_{\downarrow}$. All that remains is to identify a suitable node k which minimises the total distance. BCH is a variation on bi-directional Dijkstra search that was developed specifically for solving such problems.

In the forward direction, BCH considers only the outgoing edges in $E_{\uparrow}$. In the reverse direction BCH considers only the incoming edges in $E_{\downarrow}$[3]. Each meeting point of the two search frontiers corresponds to a tentative shortest path. Unlike standard bi-directional Dijkstra search however, which can be terminated as soon as both directions expand the same node, BCH continues until it can prove the meeting point k minimises the total distance between s and t. That means BCH stops when the minimum f-value on either open list is at least as large as the best candidate path found so far (or when both lists are empty, if there is no such path). Though simple, BCH is highly effective. It remains a state of the art method for pathfinding on static road networks with many millions of nodes (Geisberger et al. 2012).

[3]Other edges from E, such as incoming up-edges and outgoing down-edges are safely discarded by BCH to save space.

Deconstructing BCH

In this section we deconstruct BCH in order to develop a more general search framework. To begin, we derive a straightforward variant of the Single Pair Shortest Path problem for contraction hierarchies:

Definition 1 ***(CH-SPSP)*** *Given a contracted graph $G = (V, E, \mathcal{L})$, and a pair of points $s, t \in V$, find a path $\langle s = v_1, \ldots, v_k, \ldots v_n = t\rangle$ where*

$\min \sum_{i=1}^{(n-1)} c_{v_i, v_{i+1}}$

Subject to:

1. $v_i <_{\mathcal{L}} v_{i+1}$ *for all* $1 \leq i < k$
2. $v_i >_{\mathcal{L}} v_{i+1}$ *for all* $k \leq i < n$

The CH-SPSP problem asks for a minimum cost path between a start and target node. Moreover, the path must be divisible into two segments. In the first segment, each node on the path is lexically larger than the node that precedes it (we call this the *up path*). In the second segment, each node on the path is lexically smaller than the node that precedes it (we call this the *down path*). The following result is immediate from the definition of CH-SPSP:

Lemma 2 *Any solution to the CH-SPSP problem is a* ch-path *and has an associated* apex node *which is lexically largest among all nodes on the path.* □

Traversal Policies

In order to compute a solution to the CH-SPSP problem we need to define a graph traversal policy: a set of pruning rules which restrict the set of solutions to those which are also ch-paths. Suppose we expand a node n with parent $p \neq \emptyset$. Each successor n' of n falls into one of the following four cases:

Type 1. (up-up successor): $p <_{\mathcal{L}} n$ and $n <_{\mathcal{L}} n'$

Type 2. (up-down successor): $p <_{\mathcal{L}} n$ and $n >_{\mathcal{L}} n'$

Type 3. (down-down successor): $p >_{\mathcal{L}} n$ and $n >_{\mathcal{L}} n'$

Type 4. (down-up successor): $p >_{\mathcal{L}} n$ and $n <_{\mathcal{L}} n'$

Suppose the subpath $\langle s, \ldots, n\rangle$ is a ch-path. It is easy to see that Types 1 and 3 maintain the ch-path property since we keep the same trajectory through the hierarchy, continuing to move either up or down. Type 2 also maintains the ch-path property as we need to switch directions (from up to down) when the search reaches the apex of the path. Type 4 does not maintain the ch-path property however since we never switch directions after the apex. Thus we can safely prune any successors matching Type 4. We will refer to this traversal policy as as *up-then-down*, or UDP for short.

BCH implicitly specifies a similar traversal policy, albeit one which prunes the search space more aggressively. By dividing E into two disjoint sets ($E_{\uparrow}$ and $E_{\downarrow}$) BCH implicitly prunes all Type 2 and Type 4 successors (these edges never

appear in either collection). Moreover, the backward search of BCH only relaxes the incoming edges of $E_{\downarrow}$, meaning that each backward successor is lexically larger than its parent. Since both directions of BCH move upwards in the hierarchy we will call this traversal policy *always up*, or AUP for short.

Further Pruning

Both AUP and UDP prune the search space by maintaining an invariant property that applies to any ch-path. Other traversal policies also exist which can further reduce the search space. For example, the following nodes are all redundant with respect to a given pair of start and target nodes:

1. Any down successor whose contraction order is smaller than that of the target node.
2. Any successor whose contraction order is larger than that of the apex node.
3. Any successor which does not appear on any optimal ch-path to the target.

Notice that while Item 1 seems applicable only to UDP Items 2 and 3 are strategies that could be employed to further enhance AUP or UDP. In the next section we propose a variety of such pruning techniques to further speed up shortest path search in contraction hierarchies. We also discuss the efficacy of these approaches by comparing their applicability in the context of different search strategies, specifically uni-directional vs bi-directional best-first search.

Forward Search In Contraction Hierarchies

In this section we develop a uni-directional framework for computing shortest paths in contraction hierarchies. Perhaps the simplest algorithm of this type is the combination of A* search with UDP. We refer to this pairing (forward search + contraction hierarchies) as FCH. Though correct and optimal FCH is unlikely to yield a significant search time speedup. It is easy to see why: UDP prunes all Type 4 successors but never any successors of Type 2 (cf. AUP which prunes both). Put another way, each time FCH takes a step up in the hierarchy it assumes the node just reached is an apex and immediately tries to descend.

In the literature a variety of methods have been developed to improve the efficiency of forward best-first search. Often these techniques exploit information about the position of the target relative to the current node. There are two ways to achieve this: (i) compute more accurate cost-to-go estimates and drive the search toward the target; (ii) prune nodes that cannot appear on any optimal path to the target. Both approaches trade memory for speed and both depend on auxiliary data. We will enhance FCH using a variety of similar ideas, all of which instantiate Geometric Containers, a well known and popular edge-pruning technique.

Geometric Containers

Geometric Containers (Wagner, Willhalm, and Zaroliagis 2005) is an edge labelling technique used to prune successors during (a usually forward) search. The idea is to associate with every outgoing edge (s, n) a bounding volume

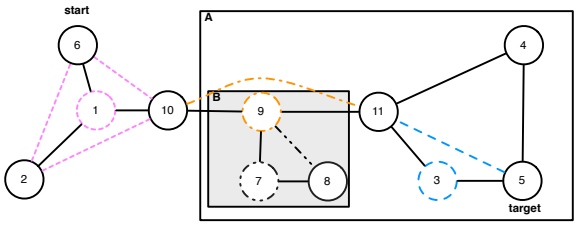

Figure 2: We apply geometric containers to directed edge (10,9). Container A is the bounding box of all nodes that can be reached optimally via edge (10, 9) in the original (uncontracted) graph. Container B includes only those nodes reached by an ch-path via "down" edge (10,9) as computed by DFS labelling.

that contains every target t optimally reachable from s via n. During search any edge whose label does not contain the target t can be safely pruned. There are many possible instantiations of Geometric Containers. As a general rule, smaller and more accurate bounding volumes yield stronger pruning during search (there are fewer false positives; i.e. cases where we relax an edge (s, n) whose bounding volume contains the target but where (s, n) does not appear on any optimal path from s to t). In this work we store a rectangular bounding box with each edge. We choose this approach on account of its simplicity and demonstrated effectiveness in pathfinding search (e.g. see (Rabin and Sturtevant 2016)). Figure 2 shows an example of rectangular bounding boxes applied to one of the edges in our running example.

Labelling

Geometric Containers are usually created by running a single Dijkstra search for each node in the graph. We propose some further alternatives with lower preprocessing requirements.

DFS: We apply a post-order DFS traversal to label the edges of the graph as follows:

Starting from the root of the hierarchy, and following only outgoing down arcs, we label each edge along the way with the bounding box that contains all nodes reached through that edge; i.e. its down-closure. Once all edges of a node have been followed we mark it as finished. This prevents DFS from processing the same node twice. This process is $O(|E_{\downarrow}|)$.

Next, we label each outgoing up arc using a similar procedure. Bounding boxes in this case contain all nodes n found in the up-closure of each up arc, plus all nodes found in the down closure of each n. As before, we employ additional bookkeeping to avoid processing a node more than once. This process is $O(|E_{\uparrow}|)$. Note that when computing up closures DFS will often produce the universal bounding box (and hence provide no pruning). We improve this below.

Dijkstra: We can compute stronger edge labellings, for both up and down arcs, by running a single-source Dijkstra search (employing UDP traversal rules) from selected nodes

Graph	$\|V\|$	$\|E\|$		
		#Input	#Shortcuts	Total
NY-d	264346	733846	920078	1653924
BAY-d	321270	800172	808952	1609124
COL-d	435666	1057066	1062850	2119916
FLA-d	1070376	2712798	2697836	5410634

Table 1: Benchmark graphs. We report total nodes and edges for the original input graphs and for contracted variants with shortcuts.

$n \in V$. For each outgoing edge of n we store a bounding box that contains the set of nodes optimally reached through that edge. The nodes at the top of a contraction hierarchy typically have very high degree and thus are most likely to benefit from such improved edge labels. Any remaining nodes, not selected for processing using Dijkstra search, have their edges labelled using the DFS scheme. Note that we perform the Dijkstra labelling before computing any DFS labels. In this case the up closures computed by DFS may not be the universal bounding box.

Experimental Setup

We compute contraction hierarchies for a subset of road networks drawn from the 9th DIMACS Challenge (Demetrescu, Goldberg, and Johnson 2009). The input graphs, and their contracted variants (which we compute), are described in Table 1. Each graph is associated with a set of 1000 instances (start-target pairs), also taken from the 9th DIMACS Challenge. All graphs and instances are publicly available[4]. To measure query performance we solve each instance to optimality five times and return the average (this approach reduces measurement error for small runtimes).

The nodes of each input graph are contracted during a pre-processing step. The order of contraction is determined using a "lazy" scheme first described in (Geisberger et al. 2008). In particular, we use the linear combination of ordering heuristics known as EDSL (ED = Edge Difference, S = Search-space size, L = Limit on the number of number of nodes expanded by each witness search (a parameter which we set to 1000)). EDSL is simple to understand, simple to implement and it is known to provide a good balance between fast query times and reasonable pre-processing times.

In our experiments we implement and test plain **FCH** against plain **BCH**. Our implementation of FCH uses the UDP traversal policy and pre-sorts the list of outgoing edges so that all "up" and "down" successors appear together. Our implementation of BCH is based on the description in (Geisberger et al. 2008) and uses *stall-on-demand*, a common optimisation which avoids some redundant node expansions caused by AUP traversal.

We further develop two variant algorithms: **FCH+BB** and **BCH+BB**. These combine FCH and BCH with geometric containers (Wagner, Willhalm, and Zaroliagis 2005) implemented as rectangular bounding boxes. Our implementation records the origin of each rectangle and its dimensions. We thus store 4×4 = 16 bytes per (outgoing) edge.

[4]http://www.diag.uniroma1.it/challenge9/

	NY-d	BAY-d	COL-d	FLA-d
DFS	1	1	1	2
Dijk 1%	102	133	259	1665
Dijk 10%	1009	1259	19704	16117
Dijk 100%	8286	27893	53422	259298

Table 2: Preprocessing time in seconds (smaller is better). We give results for different edge labelling strategies.

	$= k$	$< k$	$> k$
NY-d	0.64	0.35	0.01
BAY-d	0.58	0.41	0.01
COL-d	0.60	0.39	0.01
FLA-d	0.61	0.38	0.01

Table 3: We measure the average proportion of nodes expanded by plain FCH where the apex of the path to each node is smaller than, equal to or larger than the optimal value k. We use 4000 st-pairs on four road networks (1000 instances each) taken from the 9th DIMACS Challenge.

To compute geometric containers we apply the preprocessing strategies from earlier where **DFS** is the depth-first labelling approach and **Dijk** k**%** is the combination of Dijkstra search with DFS (we use Dijkstra for the top $k\%$ of the nodes in the hierarchy and DFS for all the rest).

All algorithms are implemented in C++ and compiled with GCC v6.4.0 using `-O3`. We undertake all experiments on a (single-threaded) machine with an Intel i7-6567U CPU @ 3.30GHz, running macOS 10.12.6 and having 16GB of RAM. Our implementations are made publicly available at https://bitbucket.org/dharabor/pathfinding.

Results

Table 2 shows total time for all labelling strategies. In addition to pure DFS, we also apply Dijkstra to label the outgoing edges of nodes in the top 1%, 10% and 100% (i.e. all) nodes in the hierarchy. Notice that preprocessing time increases roughly linearly with the Dijkstra percentage. Moreover, the time required for both DFS and Dijkstra 1% is usually much less than the time required to construct the hierarchy in the first place (at least, using our implementation).

For a first experiment we ran plain FCH and examined how much of the work (in terms of node expansions) takes place in the upwards phase and the downward phase of the search. Table 3 shows the proportion of node expansions before the apex is reached $< k$, from the apex downwards $= k$, and above the apex $> k$. From this table it is clear that most of the effort is in the downwards search after the apex is reached, or in downwards search before the apex is reached. Hence the most benefit to be gained in improving FCH is in improving the downwards search. Fortunately, forward pruning (e.g. with bounding boxes) is very effective.

In Table 4 (expansions) and Figure 3 (time) we present results from our principal experiment wherein we compare BCH versus FCH (and variants) on some well known road networks. We observe the following:

- Plain bi-directional AUP traversal through the hierarchy (i.e. the strategy employed by BCH) expands many fewer nodes than than uni-directional UDP traversal (i.e. the strategy employed by FCH).

- The combination of FCH with +BB (i.e. geometric containers in the form of bounding boxes) has very large benefits. Our simplest variant, FCH+BB (DFS), expands orders of magnitude fewer nodes than plain FCH and its search time performance is often comparable with BCH.

- FCH+BB can be much improved through the application of selective Dijkstra search during pre-processing. FCH+BB (Dijk 1%) has very modest preprocessing requirements and improves on BCH by up to several factors in most cases. FCH+BB (Dijk 10%) meanwhile achieves most of the benefit derived from pruning with bounding boxes but at a fraction of the full preprocessing cost.

- Pruning with bounding boxes also improves BCH but the gain is more modest. BCH+BB (Dijk 100%) requires a full Dijkstra search for every node in the graph but its performance is usually bettered by variants of FCH+BB with smaller overheads.

- The most effective strategy in our comparison is FCH+BB (Dijk 100%). This algorithm can improve on BCH by over one order of magnitude and computes shortest paths in a small handful of expansions, even on networks with more than one million nodes.

For a final experiment we compare against the sequential two-stage query algorithm described in (Batz et al. 2009) and later appearing in (Storandt 2013). In a standard bidirectional setup expansions are interleaved and the search explores both directions at the same time. In the mentioned works a forward (up) search is preceded by an online *marking phase*, implemented as an exhaustive backward Dijkstra search that follows only incoming down arcs. The idea is to use the results from the marking phase (which correspond to exact down distances for the target) to speed up (i.e. terminate early) the subsequent forward search. Apparently similar to FCH at first glance, this method is clearly bidirectional, though the forward and backward searches are sequential instead of interleaved. In terms of drawbacks: (i) Only the forward search benefits from forward heuristics and forward pruning. (ii) Each of the forward and backward searches can expand many of the same nodes before termination, thus duplicating work. This is a main disadvantage of bidirectional search (interleaved or otherwise).

In Table 5 we give average running times on the FLA graph for each variant of BCH and FCH described thus far. We also run (row Bwd Dijkstra) an exhaustive backwards Dijkstra search in the contraction hierarchy and from the target node. This is equivalent to the marking phase in the mentioned works and represents a lower-bound on the performance of the combined sequential algorithm. We observe that while this approach does indeed improve upon BCH, its performance in this case appears dominated by most variants of FCH.

	Min.	Q1	Med	Avg	Q3	Max.
			NY-d			
Dijkstra	33	68K	130K	132K	197K	264K
BCH	15	263	315	311	366	584
BCH+BB (Dijk 100%)	14	89	108	109	129	196
FCH	4	10K	23.8K	32K	44K	202K
FCH+BB (DFS)	3	123	209	220	291	906
FCH+BB (Dijk 1%)	3	57	88	103	127	665
FCH+BB (Dijk 10%)	3	27	37	41	50	167
FCH+BB (Dijk 100%)	3	18	25	27	33	82
			BAY-d			
Dijkstra	428	85.4K	165K	163K	243K	321K
BCH	36	175	212	212	252	406
BCH+BB (Dijk 100%)	21	67	82	81	94	171
FCH	96	17K	39K	53K	75K	23K
FCH+BB (DFS)	9	97	154	174	236	527
FCH+BB (Dijk 1%)	6	40	55	67	79	310
FCH+BB (Dijk 10%)	6	23	29	31	37	109
FCH+BB (Dijk 100%)	6	18	22	24	28	69
			COL-d			
Dijkstra	428	85K	165K	163K	243K	321K
BCH	21	217	269	260	317	450
BCH+BB (Dijk 100%)	16	76	96	96	114	196
FCH	96	17K	39K	53K	75K	232K
FCH+BB (DFS)	6	80	155	201	294	708
FCH+BB (Dijk 1%)	6	33	48	60	73	451
FCH+BB (Dijk 10%)	6	22	29	31	38	140
FCH+BB (Dijk 100%)	6	18	23	25	30	109
			FLA-d			
Dijkstra	167	107K	210K	214K	322K	435K
BCH	24	252	293	289	334	480
BCH+BB (Dijk 100%)	19	83	102	100	116	177
FCH	49	20K	48K	79K	105K	303K
FCH+BB (DFS)	7	138	217	231	300	825
FCH+BB (Dijk 1%)	7	39	53	63	76	282
FCH+BB (Dijk 10%)	7	23	29	32	37	110
FCH+BB (Dijk 100%)	6	19	23	24	28	80

Table 4: Node expansions (smaller is better). We compare BCH and FCH against variants using different preprocessing strategies.

Hybrid Search Techniques

We have shown that for contraction hierarchies on road networks there exist simple and uni-directional search methods that can lead to significant run-time improvements vs. interleaved or sequential bi-directional search. In (Bauer et al. 2010) contraction hierarchies and goal-directed edge pruning also appear as ingredients in the development of the hybrid two-stage (bi-directional first, then forward only) search algorithms CHASE and CALT. As in this work, the authors report order-of-magnitude performance improvements over plain BCH and on road networks similar to the ones tested here. Unlike our work, both CHASE and CALT depend on plain BCH as a basic ingredient and neither directly improves that algorithm. We believe these methods are orthogonal to our current work and can be improved by leveraging the results we have developed here. A theoretical and empirical study of such possibilities is an interesting topic for further work.

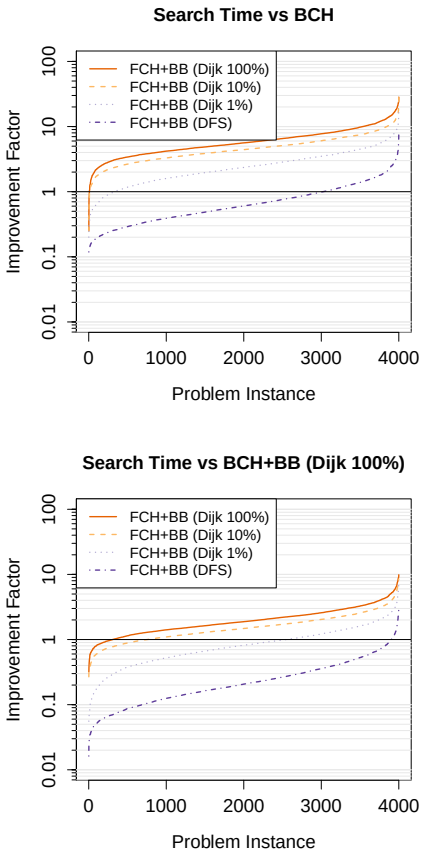

Figure 3: Search time comparison. We measure the relative improvement factor (i.e. speedup) for variants of FCH+BB vs BCH (top diagram) and vs BCH+BB (100% Dijk) (bottom diagram). Each curve in the plot represents a complete distribution; i.e. we order all instances by relative improvement and rank them for lowest to highest. The x-axis can be understood as a quantile; i.e. the 2000th instance corresponds to the 50th percentile. In each plot larger is always better and an improvement factor of > 1 indicates a positive result. Note the log10 scale on the y-axis.

Conclusion

Contraction Hierarchies is a method for building a multi-level graph abstraction. Until now it has been tied to bi-directional Dijkstra search as an efficient approach to compute shortest paths on large networks. In this paper we show that the benefits of contraction hierarchies can also be applied to (uni-directional) forward search. This means that all the methods and techniques developed to improve forward search can be directly applied to contraction hierarchies. One possible next step involves further improvements to algorithms such as *FCH+BB (Dijk 1%)*, which have strong performance and low preprocessing requirements. Combining these with strong forward-driven pruning techniques, such as Compressed Path Databases (Strasser, Botea, and Harabor 2015), appears promising.

	Min.	Q1	Med	Avg	Q3	Max.
BCH	3	67	82	84	99	171
BCH+BB (Dijk 100%)	4	21	28	28	33	83
FCH+BB(DFS)	11	92	153	169	220	744
FCH+BB(Dijk 1%)	13	25	32	37	42	182
FCH+BB(Dijk 10%)	12	18	21	22	25	76
FCH+BB(Dijk 100%)	12	16	18	19	21	65
Bwd Dijk	19	42	50	51	59	128

Table 5: Average running time, in microseconds, for all 1000 instances from the FLA-d benchmark (smaller is better). We give results for all tested variants of BCH and FCH. We also report (row Bwd Dijk) lower-bound running time results for the sequential bidirectional algorithm appearing in (Batz et al. 2009; Storandt 2013).

References

Abraham, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F. 2012. Hierarchical Hub Labelings for Shortest Paths. In Epstein, L., and Ferragina, P., eds., *Algorithms – ESA 2012*, volume 7501 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 24–35.

Batz, G. V.; Delling, D.; Sanders, P.; and Vetter, C. 2009. Time-dependent contraction hierarchies. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, 97–105. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Bauer, R.; Delling, D.; Sanders, P.; Schieferdecker, D.; Schultes, D.; and Wagner, D. 2010. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *J. Exp. Algorithmics* 15:2.3:2.1–2.3:2.31.

Bauer, R.; Columbus, T.; Rutter, I.; and Wagner, D. 2013. Search-space size in contraction hierarchies. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 7965 of *LNCS*, 93–104. Springer.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near Optimal Hierarchical Path-Finding. *J. Game Dev.* 1(1):7–28.

Demetrescu, C.; Goldberg, A. V.; and Johnson, D. S. 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Mathematical Soc.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Workshop on Experimental Algorithms*, volume 5038 of *LNCS*, 319–333. Springer.

Geisberger, R.; Sanders, P.; Schultes, D.; and Vetter, C. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science* 46(3):388–404.

Rabin, S., and Sturtevant, N. 2016. Combining bounding boxes and JPS to prune grid pathfinding. In *AAAI Conference on Artificial Intelligence*, 746–752.

Storandt, S. 2013. Contraction hierarchies on grid graphs. In Timm, I. J., and Thimm, M., eds., *KI 2013: Advances in Artificial Intelligence*, 236–247. Berlin, Heidelberg: Springer Berlin Heidelberg.

Strasser, B., and Wagner, D. 2015. Graph fill-in, elimination ordering, nested dissection and contraction hierarchies. In *Gems of Combinatorial Optimization and Graph Algorithms*. Springer. 69–82.

Strasser, B.; Botea, A.; and Harabor, D. 2015. Compressing optimal paths with run length encoding. *J. Artif. Intell. Res.* 54:593–629.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *Proceedings of the Third AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 31–36. AAAI Press.

Uras, T.; Koenig, S.; and Hernàndez, C. 2013. Subgoal graphs in for optimal pathfinding in eight-neighbour grids. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, 224–232. AAAI Press.

Wagner, D.; Willhalm, T.; and Zaroliagis, C. 2005. Geometric containers for efficient shortest-path computation. *J. Exp. Algorithmics* 10:article 1.3.