Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

Understanding Subgoal Graphs by Augmenting Contraction Hierarchies

Tansel Uras and **Sven Koenig**
Department of Computer Science, University of Southern California, Los Angeles, USA
{turas, skoenig}@usc.edu

Abstract

Contraction hierarchies and (N-level) subgoal graphs are two preprocessing-based path-planning algorithms that have so far only been compared experimentally through the grid-based path-planning competitions, where both algorithms had undominated runtime/memory trade-offs. Subgoal graphs can be considered as a framework that can be customized to different domains through the choice of a reachability relation R that identifies pairs of nodes on a graph between which it is easy to find shortest paths. Subgoal graphs can exploit R in various ways to speed-up query times and reduce memory requirements. In this paper, we break down the differences between N-level subgoal graphs and contraction hierarchies, and augment contraction hierarchies with ideas from subgoal graphs to exploit R. We propose three different modifications, analyze their runtime/memory trade-offs, and provide experimental results on grids using canonical-freespace-reachability as R, which show that both N-level subgoal graphs and contraction hierarchies are dominated in terms of the runtime/memory trade-off by some of our new variants.

1 Introduction

Path planning on graphs has many applications in video games, robotics, and navigation systems. Preprocessing-based path-planning algorithms first analyze a given graph in a preprocessing phase to generate auxiliary information which can then be used to significantly speed-up online shortest-path queries. The performance of these algorithms can be characterized by their runtime/memory trade-off, as it is often possible to achieve shorter runtimes by using more memory.

The 9th DIMACS Implementation Challenge [Demetrescu *et al.*, 2009] featured a competition on preprocessing the USA road network, resulting in many new algorithms such as contraction hierarchies [Geisberger *et al.*, 2008; Dibbelt *et al.*, 2014], transit routing [Bast *et al.*, 2006; Arz *et al.*, 2013], highway hierarchies [Sanders and Schultes, 2005], reach [Gutman, 2004; Goldberg *et al.*, 2009], SHARC [Bauer and Delling, 2008], arc flags [Lauther, 2004; Hilger *et al.*, 2009], and hub-labeling [Abraham *et al.*, 2011]. Most of these algorithms are applicable to any graph and provide excellent experimental results on road networks by exploiting their hierarchical structure, speeding up shortest-path queries by several orders of magnitude. Contraction hierarchies (CHs) have been central to proving that these algorithms are provably efficient on graphs with low highway dimensions [Abraham *et al.*, 2010] and have the smallest memory overhead [Delling *et al.*, 2009]. They use the elegant idea of node contractions, which remove nodes from the graph and add shortcuts to preserve shortest paths between the remaining nodes.

A relatively new preprocessing-based path-planning algorithm uses subgoal graphs (SGs) to exploit the freespace structure in grids to achieve fast query times with only a small memory overhead [Uras *et al.*, 2013]. SGs have been generalized into a framework that can be applied to different types of graphs by choosing a reachability relation R that distinguishes some pairs of nodes as R-reachable [Uras and Koenig, 2017]. A SG can be constructed in a preprocessing phase as an overlay graph that has only R-reachable edges. During queries, start and goal nodes can be connected to the SG through R-reachable edges to form a query SG, which can then be searched for a shortest path that consists of only R-reachable edges. These edges can then be quickly refined into a shortest path on the original graph. SGs can be layered on top of each other to create hierarchies, called N-level subgoal graphs (N-SGs) [Uras and Koenig, 2014], which are similar to CHs as they can be constructed using a modified form of node contractions. Essentially, the SG framework can take advantage of structure in a domain by first capturing it with R and then exploiting it with specialized connection and refinement methods. We believe that some of the other preprocessing-based path-planning algorithms can be augmented with ideas from SGs to exploit structure as well.

Our first contribution in the paper is to break down the different methods of exploiting R and outline the possible trade-offs associated with them for a non-specific R. We identify three such methods of exploiting R in the context of CHs, and analyze their runtime/memory trade-offs. Two of these modifications correspond to the differences between the contraction methods used to construct CHs and (N-)SGs: (1) Allowing only R-reachable shortcuts results in hierarchies that are slower to search but allow for faster refinement and re-

quire less memory per edge. (2) Using *heavier* contractions collapses parts of the hierarchies, introducing more edges to the hierarchies which often result in slower queries and higher memory requirements. However, SGs can discard these edges and reconstruct them during queries by methods that exploit R, which can ultimately result in faster queries and lower memory requirements. We propose a new method of exploiting R that does not modify the contractions but simply marks shortcuts that are R-reachable and refines them with algorithms that exploit R.

Our second contribution is an experimental study to understand how these trade-offs manifest themselves on a specific domain for a specific reachability relation, namely, canonical-freespace-reachability on grids. Our results show that both CHs and N-SGs, which have undominated runtime/memory trade-offs in the Grid-Based Path-Planning Competition (GPPC) [Sturtevant *et al.*, 2015], are dominated by some of our new variants. In our experiments, our new variant CH-SG-R is 3.6 times faster than the CH entry in GPPC while requiring 4.6 times less memory. As a comparison, the fastest optimal entry in the GPPC (whose benchmarks are similar to ours but not exactly the same), Single Row Compression [Strasser *et al.*, 2015], is 2.5 times faster than the CH entry but requires 21.6 times more memory.

The grid setting is arguably a best-case scenario for exploiting R since it allows for very efficient refinement and connection operations that require little memory overhead. Our current way to apply SGs to state-lattices uses a version of freespace-reachability that, compared to grids, has less efficient refinement and connection operations, uses more memory, and results in SGs that are larger and thus more expensive to search [Uras and Koenig, 2017]. SGs have never been applied to road networks, as it seems difficult to come up with an R that allows for both fast refinement and connection operations, but our new understanding shows that one might be able to focus only on the refinement operations for faster refinement times with little or no memory overhead. Faster refinement times can have a noticable effect on the overall query times for road networks. For instance, refinement operations that store midpoints for each shortcut edge can still take up to 60% of CH query times [Geisberger, 2008].

2 Background

In this section, we introduce our notation and give an overview of contraction hierarchies (CHs) and subgoal graphs (SGs).

2.1 Notation

$G = (V, E, w)$ denotes the original graph that we preprocess, where V is a set of nodes, $E \subseteq V \times V$ is a set of edges, and $w : E \to \mathbb{R}_{>0}$ is a function that assigns a weight to each edge. For simplicity, we assume that G is undirected and connected, and its edges are stored as directed edges. The algorithms described in this paper are applicable to directed graphs as well. $d(s, t)$ is the distance between s and t on G.

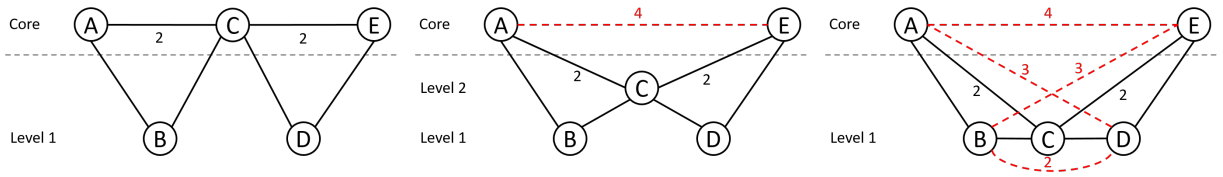
Given a function $l : V \to \mathbb{N}^+$ that assigns levels to each node and a maximum level l_{max}, the set of nodes n with $l(n) = l_{max}$ are the *core nodes*, edges between core nodes are *core edges*, and the set of all core nodes and edges form the *core* of the graph. An edge (u, v) is *upward* iff $l(u) < l(v)$, *downward* iff $l(u) > l(v)$, and *same-level* iff $l(u) = l(v)$. A path $\pi = (v_0, \ldots, v_n)$ is upward iff all its edges are upward, downward iff all its edges are downward, and same-level iff all its edges are same-level. $\pi = (v_0, \ldots, v_n)$ is an *up-down* path iff, for some v_k (*apex node*), $(v_0, \ldots, v_k)$ is upward and $(v_k, \ldots, v_n)$ is downward. The upward-closure of a node is the set of nodes reachable from it by following upward edges.

2.2 Contraction Hierarchies

A CH on G can be computed by contracting the nodes of G one by one. Contracting a node n removes n and its incident edges from the graph, which can increase the lengths of shortest paths between some of its neighbors. To preserve the shortest path between any such pair of neighbors u and v, a shortcut edge (*shortcut*) (u, v) is added (if necessary) with $w(u, v) = w(u, n) + w(n, v)$ when contracting n. The contracted node is assigned a level of one plus the highest level node that has an upward edge to C, or a level of one if no such node exists. After all nodes are contracted, the resulting CH has the property that there is an up-down path with length $d(s, t)$ between any two nodes s and t. CHs do not have same-level edges, and any downward edges are discarded. It is therefore possible for a CH to have fewer directed edges than G. Figure 1(a) shows a graph where nodes B and D are already contracted, and nodes A, C, and E are not yet contracted and remain in the core. Figure 1(b) shows the contraction of node C: When C is removed from the core, the undirected shortcut AE is added to preserve the shortest path between A and E. C is assigned a level of 2, one higher than the highest level node (either B or D) that has an upward edge to C. Edges AC and CE are only stored in the upward direction. The contraction order plays an important role for the query times and memory requirements of the resulting CH, and is usually determined with heuristics that take into account the number of shortcuts added when a node is contracted and the number of edges that can be discarded as they become downward [Geisberger *et al.*, 2008].

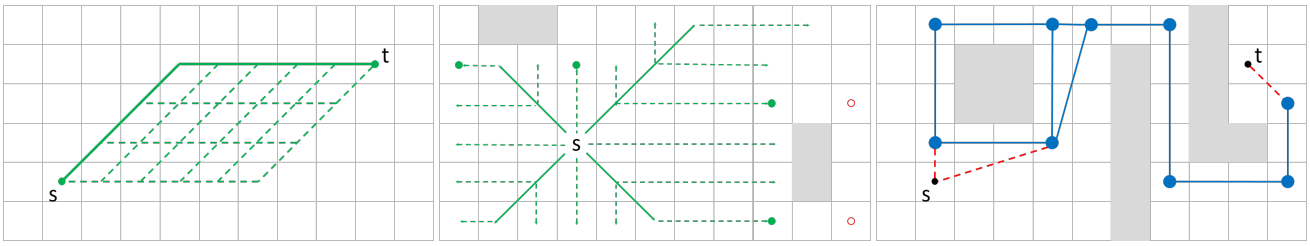
Shortest up-down-paths on CHs can be found with a modified bidirectional Dijkstra search, where the forward search constructs the *up* part and the backward search constructs the *down* part of a path. An expansion in either frontier only generates successors with levels higher than the expanded node, allowing searches over CHs to ignore many nodes and thus resulting in very fast queries. The bidirectional search keeps track of the length of the shortest path found so far, B, which is updated every time a node is expanded in one frontier that has already been generated in the other frontier. Unlike a regular bidirectional Dijkstra search which terminates when the *sum* of the radii of the two frontiers is greater than or equal to B, the search over CHs terminates when the radii of *both* frontiers are greater than or equal to B.[1]

[1]This is so because (1) ignoring successors of lower levels forfeits the guarantee that the g-values of nodes expanded by Dijkstra searches are correct, and (2) when the two frontiers meet at a node n, n is not necessarily the apex node of a shortest up-down path. A technique called *stall-on-demand* can be used to prune some of the



(a) Partially contracted graph. (b) (R-)contracting C. (c) Heavy-R-contracting C.

Figure 1: Different types of contractions. Dashed red edges show the shortcuts added after contracting C. Edge weights are 1 unless specified otherwise.



(a) Freespace paths. (b) R-connect. (c) Query subgoal graph.

Figure 2: Subgoal graphs on grids. In (a), solid lines show the canonical freespace path. In (b), R-connect uses precomputed cardinal clearances to avoid traversing the dashed lines. In (c), edges incident to s and t are added by R-connect.

Once a shortest up-down path is found, any shortcut (u, v) on the up-down path can be recursively *unpacked* (refined) into a shortest path on G by replacing it with the *bridged* edges (u, n) and (n, v), where n is the *midpoint* whose contraction introduced (u, v). To perform refinements efficiently, either the midpoint or two pointers to the bridged edges can be stored with every shortcut edge. The former method uses less memory per shortcut, but requires a scan of the incident edges of the midpoint to find the bridged edges.

2.3 Subgoal Graphs

Subgoal graphs (SGs) can be specialized to different domains (types of graphs) through the choice of a reachability relation $R \subseteq V \times V$. A pair of nodes (u, v) and the corresponding edge are called *R-reachable* iff $(u, v) \in R$. A SG can be constructed on G with respect to R by identifying a set of subgoals and adding R-reachable edges between them. The set of subgoals S satisfies the property that, between any two nodes s and t, there exists a *subgoal path* $(v_0 = s, \ldots, v_n = t)$ with length $d(s, t)$, where $v_1, \ldots, v_{n-1} \in S$ and, $\forall i\ (v_i, v_{i+1}) \in R$ (that is, a shortest s-t-path can be partitioned by subgoals into subpaths between R-reachable nodes). SGs can be used to answer shortest path queries in three steps: (1) *Connecting* the start and goal nodes to the SG (and to each other, if possible) through R-reachable edges by an *R-connect* operation specialized for R. (2) *Searching* the resulting query SG for a shortest subgoal path π_S that consists of only R-reachable edges. (3) *Refining* π_S into a shortest path on G by replacing its edges with the corresponding shortest paths by an *R-refine* method specialized for R. Searches over SGs are typically faster than searches over G because they ignore any non-subgoal nodes. When constructing a SG or connecting query points to a SG, only *direct-R-reachable* edges are used, which are R-reachable edges that cannot be refined into paths that pass through subgoals.

Specializing the SG framework to a domain involves choosing R and developing specialized R-connect and R-refine operations that exploit R, all of which play an important role for the runtime/memory trade-off of SGs. For example, one can specialize the SG framework to grids by using *freespace-reachability* as R [Uras *et al.*, 2013]. Figure 2(a) shows all shortest paths between two cells on an 8-neighbor grid with no blocked cells. These paths are called *freespace paths*, and they consist of moves in at most one cardinal and at most one diagonal direction. Two cells on a grid are called freespace-reachable iff at least one freespace path between them is unblocked. SGs on grids can exploit freespace-reachability as follows: (1) Subgoals can be identified very quickly by placing them at the convex corners of blocked cells, as shown in Figure 2(c). (2) R-connect can be implemented efficiently by using precomputed clearances (distances to blocked cells and subgoals) from every unblocked cell in all four cardinal directions, as illustrated in Figure 2(b). (3) R-refine can be implemented efficiently by calculating the numbers and directions of cardinal and diagonal moves on a freespace path between two cells (from their relative positions) and then using a depth-first search (DFS) to find an ordering of moves that avoids blocked cells. (4) The edge weights are Octile distances and thus do not need to be stored.

The edges of SGs on grids are actually *safe-freespace-reachable* (all freespace paths between two cells are unblocked) when subgoals are placed at the convex corners of blocked cells, which means that R-refine's DFS finds an unblocked path without backtracking when refining SGs' edges. This is not necessarily the case when R-refining freespace-reachable shortcuts in the context of our modifica-

nodes with suboptimal g-values from the search [Geisberger *et al.*, 2008].

tions to CHs. In our experiments, we use *canonical-freespace reachability* as R, where R-refine can be implemented without a DFS, resulting in faster refinement times. Among all freespace paths between two cells, we fix one as the canonical freespace path. Our definition follows that of Jump Point Search [Harabor and Grastien, 2011] with a small tweak to make the canonical paths symmetric: When moving from a source cell s to a target cell t, if t has a larger x-coordinate than s, the canonical freespace path is the one where diagonal moves come before cardinal moves. Otherwise, the canonical freespace path is the one where cardinal moves come before diagonal moves. Two cells are canonical-freespace-reachable iff the canonical freespace path between them is unblocked.

2.4 N-Level Subgoal Graphs

A SG can be understood as a 2-level hierarchy, where a node $u \in V$ is in the core of the hierarchy iff u is a subgoal. An edge exists in this hierarchy iff it is direct-R-reachable. The edges in the core of this hierarchy are stored as a SG. The remaining edges are not stored and constructed by R-connect during queries.

A SG can be constructed on G by first initializing the core of a 2-level hierarchy to G and then using a modified form of contraction, which we call a *heavy-R-contraction* [Uras and Koenig, 2014]. Heavy-R-contractions differ from regular contractions in two ways: (1) Heavy-R-contractions add shortcuts to ensure that, between any two nodes s and t, there is a shortest path with length $d(s,t)$ that only passes through nodes that are subgoals. (2) Heavy-R-contractions can only add R-reachable shortcuts. If heavy-R-contracting a node would introduce a non-R-reachable shortcut, the node is not contracted and remains in the core as a subgoal. This method of constructing SGs is equivalent to starting with the set of subgoals $S = V$ and then removing (contracting) nodes from S one by one, while maintaining that a subgoal path with length $d(s,t)$ exists between any two nodes $s, t \in V$, and that the 2-level hierarchy contains exactly the direct-R-reachable edges between nodes. Heavy-R-contractions preserve shortest paths in the core, similar to regular contractions (otherwise, if all shortest path between two nodes s and t in the core are lost, then there cannot be an s-t-path with length $d(s,t)$ that only passes through subgoals). Thus, heavy-R-contractions add a superset of the shortcuts added by regular contractions, which is why we think the term *heavy* is suitable. Consider the example in Figure 1. When contracting C, a regular contraction adds the shortcut AE to preserve the shortest path between A and E. Heavy-R-contracting C adds the shortcut AE as well, but also adds the shortcuts BD (otherwise, the only shortest B-D-path, B-C-D, passes through the non-subgoal node C), BE (otherwise, both shortest B-E-paths, B-C-E and B-C-D-E, pass through C), and DA.

The 2-level hierarchy can be extended to multiple levels by iteratively constructing a SG on top of the previous one, until one that is empty or equivalent to the previous one is constructed. The resulting hierarchy is called an N-level SG (N-SG), where the last constructed SG forms its core. We explore how the differences in the contraction methods used to construct CHs and N-SGs affect the query times and memory requirements in the next two sections.

3 Augmenting CHs with R

In this section, we introduce three different methods of exploiting R in the context of CHs: R-refining R-reachable shortcuts of CHs, using only R-reachable shortcuts in CHs, and constructing CHs on SGs. Figure 3 provides an overview of the different methods and their trade-offs. When discussing trade-offs, we distinguish between connection, search, and refine times, which, together, add up to the overall query time.

3.1 CHs with R-Refine (CH-R)

The first method replaces the unpacking operation of CHs with R-refine for R-reachable shortcuts. Depending on the efficiency of R-refine, this method can result in shorter refinement times and does not affect the search times. R-reachable shortcuts can be marked during preprocessing without using extra memory by replacing their unpacking information with a unique identifier. Extra memory might be required to implement efficient R-refine operations. Several optimizations for this method are possible which we outline below but do not use in our experiments. (1) Rather than marking each R-reachable shortcut, mark only the ones whose R-refinement is faster than unpacking. (2) Use a 1-bit flag to mark a shortcut as R-reachable and use the remaining bits of the unpacking information to store any information that can be used to speed-up R-refinements. (3) Use a number of reachability relations and/or refinement methods and refine each edge with the fastest refinement method. More bits per edge can be used to signal which refinement method to use.

3.2 R-Contraction Hierarchies (RCH)

Depending on R, SGs can avoid storing refinement information and/or weights for each edge. It is not straightforward to implement a version of CHs where the same is done only for R-reachable shortcuts, given that any node can have a mix of shortcuts to R-reachable and non-R-reachable neighbors. We can instead introduce a new method of contraction, called an R-contraction, that is only allowed to add R-reachable shortcuts. However, this constraint implies that nodes whose contraction requires shortcuts that are not R-reachable cannot be contracted. We call a hierarchy constructed by only R-contractions an R-contraction hierarchy (RCH), which is guaranteed to have only R-reachable shortcuts but can also have a core of non-contracted nodes.

RCHs can be searched with a version of CHs' bidirectional Dijkstra search that also generates same-level neighbors of core nodes. Either the forward or the backward frontier can be disallowed to expand nodes in the core to prevent redundant expansions.[2] Even with this modification, searches over RCHs are likely to be slower than searches over CHs since the non-contracted nodes can always be generated/expanded during searches (whereas, in CHs, they are unreachable by the search if they are not in the upward closures of the start and goal nodes). Whether this is worth the reduced memory requirements per shortcut and potentially faster refinement times largely depends on the domain and R. For instance,

[2] This is an issue only because the search of CHs does not necessarily stop when the two frontiers meet, as discussed earlier.

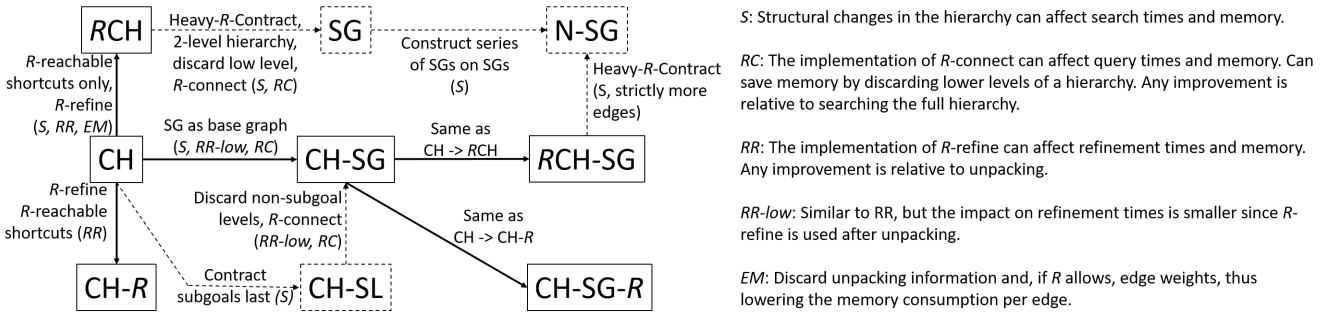


Figure 3: Summary of the differences between various algorithms and their trade-offs. Solid lines correspond to the three modifications discussed in this paper.

if R only excludes a small percentage of pairs of nodes, then it might not interfere too much with contractions.

3.3 Constructing CHs on SGs (CH-SG)

CHs can be constructed on top of SGs rather than G, resulting in CH-SGs. During queries, start and goal nodes are connected to CH-SGs by R-connect, similar to how they are connected to SGs. Then, CHs' bidirectional search is used to find an up-down path, which is first refined into a path on the SG by CHs' unpacking and then refined into a path on G by R-refine.

To understand the trade-offs associated with this modification, suppose we first contract only the non-subgoal nodes of G so that the remaining core of non-contracted nodes form a SG.[3] If we replace the remaining core with a CH-SG, we get a CH where the levels of subgoals (subgoal levels) are higher than the levels of non-subgoals (non-subgoal levels). We call this hierarchy CH-SL since it can be constructed by contracting subgoals last. The difference between a CH and a CH-SL is the interference with the contraction order, which might negatively effect query times and memory requirements. The difference between a CH-SL and a CH-SG is that the non-subgoal levels of the CH-SL are discarded and replaced by R-connect and R-refine. Whether this improves query times and memory requirements depends on the implementations of R-connect and R-refine: On a CH-SL, searches traverse the non-subgoal levels before reaching the subgoal levels, whereas, on a CH-SG, R-connect directly connects the start and goal nodes to the subgoal levels. On a CH-SL, refinement is done by unpacking alone, whereas, on a CH-SG, unpacking delegates the task to R-refine when it needs to unpack an edge of the SG. A CH-SG does not need to store the non-subgoal levels of the hierarchy, but might require extra memory to implement R-connect and R-refine.

4 N-SG Revisited

As discussed in Section 2.4, SGs can be constructed by using heavy-R-contractions, and N-SGs can be constructed by repeatedly constructing SGs on top of previous ones. Similar to CH-SGs (and SGs), N-SGs discard the lowest level of their hierarchies and reconstruct it during queries via R-connect. Similar to RCHs, N-SGs only have R-reachable shortcuts. As discussed in Sections 3.2 and 3.3, both modifications can result in smaller query times and lower memory requirements depending on R, R-connect, and R-refine.

These trade-offs can also be achieved by constructing RCHs on SGs (RCH-SGs). The only difference between N-SGs and RCH-SGs is that N-SGs are constructed by heavy-R-contractions, which, as discussed in Section 2.4, introduce a superset of the shortcuts introduced by R-contractions. Therefore, for any N-SG, we can use the same contraction order to create a RCH-SG that has no more edges than the N-SG. N-SGs guarantee that, between any two nodes $s, t \in V$, there is an *arching* path of length $d(s,t)$ [Uras and Koenig, 2014]. A path $\pi = (v_0, \ldots, v_n)$ is an arching path iff, for some v_k and v_m with $0 \leq k \leq m \leq n$, $(v_0, \ldots, v_k)$ is upward, $(v_m, \ldots, v_n)$ is downward, and $(v_k, \ldots v_m)$ only has same-level core edges or at most one same-level non-core edge. We can search N-SGs with R-CHs' bidirectional search with the following modification: The backward search never generates non-core same-level successors, and the forward search only generates them to check if the two frontiers have met (but does not put them in the open list). We compare the query times of N-SGs and RCH-SGs experimentally in the next section.

5 Experimental Results

We present experimental results on grids to see how the trade-offs outlined in Figure 3 manifest themselves on 8-neighbor grids when we use canonical-freespace-reachability as R.

We implemented all our algorithms from scratch in a single framework that uses the same implementation of graphs, search algorithms and data structures.[4] All searches use the Octile distance heuristic and a binary heap as priority queue. In all hierarchies, downward edges are discarded. Our implementation follows the implementation of the CH entry in the GPPC with respect to the online node ordering for contractions and using stall-on-demand for searches. We have replaced unpacking using midpoints with unpacking using pointers to the bridged edges and replaced the bidirectional Dijkstra searches with bidirectional A* searches. Each edge

[3] This is so because: (1) A node remains in the core iff it is a subgoal. (2) Contractions preserve shortest paths in the core and do not introduce redundant edges. (3) The set of direct-R-reachable edges is the minimum set of edges necessary to preserve shortest paths between subgoals.

[4] The source code is available at http://idm-lab.org/sg-ch.

	All			Game			Maze			Random			Room		
	PT	M	SU	PT	M	SU	PT	M	SU	PT	M	SU	PT	M	SU
SG	0	1.42	12.95	0	1.24	36.63	0	1.11	29.91	0	2.27	2.64	0	1.06	78.49
CH-DM	72	12.80	74.43	135	11.64	27.64	34	12.89	167.33	77	9.73	33.43	43	16.94	105.33
CH-M	72	12.80	95.24	135	11.64	39.08	34	12.89	158.82	77	9.73	60.91	43	16.94	106.95
CH	72	17.07	127.15	135	15.51	46.00	34	17.19	255.29	77	12.98	77.94	43	22.58	137.87
CH-R	72	17.07	141.80	135	15.51	51.50	34	17.19	297.20	77	**12.98**	**81.29**	43	22.58	156.11
RCH	54	4.19	22.02	135	3.88	47.33	34	4.27	269.54	4	2.98	3.83	43	5.63	56.93
CH-SL	98	17.44	105.96	226	16.61	27.76	33	17.17	254.88	90	13.43	76.43	42	22.53	138.14
CH-SG	29	2.77	256.87	32	1.43	155.96	0	1.34	683.69	82	7.06	75.11	0	1.27	414.43
CH-SG-R	29	**2.77**	**268.63**	32	**1.43**	**160.54**	0	**1.34**	**704.86**	82	**7.06**	**80.09**	0	**1.27**	**424.54**
RCH-SG	5	**1.38**	**26.97**	21	**1.17**	**124.53**	0	**1.07**	**534.56**	1	**2.20**	**4.36**	0	**1.06**	**88.24**
N-SG	8	1.41	25.30	25	1.24	108.45	1	1.08	525.74	5	2.27	4.09	0	1.06	84.19

Table 1: Preprocessing times and runtime/memory trade-offs of various algorithms. PT: Preprocessing time (s). M: Memory required to store the graph/hierarchy and clearance values (MB). SU: Speed-up over A*. Bold values show undominated runtime/memory trade-offs.

of CHs requires 16 bytes (destination, weight, and two pointers to the bridged edges). Our implementation uses the same R-connect method as the N-SG entry in the GPPC (as shown in Figure 2(b)), which requires 4 bytes per cell. We store only the destination for R-reachable edges, which requires 4 bytes per edge, as their weight can be calculated as the Octile distance.

We ran our experiments on a PC with a 3.6GHz Intel Core i7-7700 CPU and 32GB of RAM. We used Nathan Sturtevant's publicly available 8-neighbor grid benchmarks [Sturtevant, 2012], which has four types of maps, each with several subtypes: Game maps with maps from different games, maze maps with varying corridor widths, room maps with varying room dimensions, and random maps with varying percentages of blocked cells. Game map sizes vary from 22×28 to 1260×1104. All other maps have size 512×512. When averaging statistics for an algorithm over all maps of the same type, we first average statistics over all instances (start and goal pairs) on a map, then average statistics over all maps of the same subtype, and finally average statistics over all subtypes, in order to avoid biasing the results toward maps with higher numbers of instances and toward map (sub)types with higher numbers of maps. Statistics that relate two algorithms (for instance, speed-ups) are calculated from these averages.[5]

Table 1 shows a comparison of the algorithms listed in Figure 3 plus the algorithms CH-DM (CHs with bidirectional Dijkstra search and midpoint unpacking, similar to the GPPC entry) and CH-M (CHs with midpoint unpacking) with respect to their preprocessing times and runtime/memory trade-offs. We observe the following trends: CH-SG-Rs are overall the fastest algorithm across all benchmarks except on random maps where only CH-Rs are 1.5% faster. CHs (CH-SGs) are consistently dominated (with respect to the runtime/memory trade-off) across all benchmarks by CH-Rs (CH-SG-Rs) which have the same memory requirements but faster refinement times. Similarly, N-SGs are dominated by RCH-SGs, which are consistently faster than N-SGs and consistently use the least amount of memory among all algorithms, since they use SGs as smaller base graphs, do not store unpacking information and weights for edges, and have fewer edges than N-SGs. On game and maze maps, CHs are dominated by all variants (except SGs, CH-DMs, CH-Ms, and CH-SLs). Our modifications are least effective on random maps, where SGs, RCHs, RCH-SGs, and N-SGs are less than 5 times faster than A*.

We now analyze these results in more depth. Table 2 shows more detailed statistics about the trade-offs associated with our three modifications. We distinguish between distance queries that include connection and search times, and path queries that also include refinement times.

CH $\rightarrow$ CH-R: We observe that most of CHs' shortcuts are canonical-freespace-reachable (R%). This percentage is smallest on random maps at 54.97% and highest on game maps at 99.77%. As a result, most of CH-Rs' shortcuts are R-refined rather than unpacked, improving refinement times (Rf) by a factor of 1.52 and path query times (PQ) by a factor of 1.11. As the corridor width decreases for maze maps, the blocked cell percentage increases for random maps, and the room size decreases for room maps, both the percentage (R%) and lengths (not reported) of canonical-freespace-reachable shortcuts decrease, which negatively effect refinement times. As mentioned earlier, R-refining canonical-freespace-reachable shortcuts involves an overhead of calculating the number of cardinal and diagonal moves to make, and this overhead becomes more pronounced as the shortcut lengths decrease, to the point where R-refine is slower than unpacking on maze01, maze02, random35, and random40 maps. This can be addressed by R-refining only those shortcuts that can be R-refined faster than they can be unpacked, as suggested in Section 3.1.

CH $\rightarrow$ RCH: RCHs improve the refinement times (Rf) by a factor of 2.98 over CHs, even more so than CH-Rs, since all their shortcuts are canonical-freespace-reachable. Compared to CHs, searches on RCHs are only 8% slower on game maps, whereas they are 25 times slower on random maps. The percentage of CHs' shortcuts that are canonical-freespace-reachable (R%) seems to be a stronger indicator of whether the search times on RCHs are close to the search times on CHs, with a correlation coefficient of 0.83, as opposed to the percentage of contracted nodes in RCHs (C%), which has a correlation coefficient of 0.71. For instance, on mazes, although all the nodes are contracted, searches are 33% slower, which indicates that the structure of the hierarchies is still negatively effected.

[5]That is, if A* has an average runtime of 10ms on map1 with 10 instances and 20ms on map2 with 15 instances, we report the average as 15ms. If SGs have an average runtime of 1.5ms over both maps, we report their speed-up over A* as 10.

	CH				CH → CH-R		CH → RCH					CH → CH-SL	CH → CH-SG						
	Sr	Rf	Sc%	R%	Rf	PQ	C%	Sr	Rf	PQ	M	PQ	DN%	DE%	Sr	Rf	DQ	PQ	M
all	45	16	45.29	90.10	1.52	1.11	93.19	0.13	2.98	0.17	4.07	0.83	87.70	89.76	2.76	1.42	2.35	2.01	6.16
game	62	9	54.38	99.77	3.79	1.12	99.86	0.92	4.55	1.03	4.00	0.60	97.98	97.75	5.18	2.22	3.67	3.39	10.85
bg512	50	8	54.19	99.85	3.75	1.12	99.91	0.96	3.95	1.07	4.00	0.80	99.10	99.24	9.00	2.50	4.90	4.35	8.86
dao	19	4	49.05	99.31	2.18	1.11	99.54	0.82	2.47	0.92	3.98	0.66	95.80	96.17	2.97	1.12	2.10	1.84	3.92
sc1	118	16	54.80	99.78	4.74	1.12	99.87	0.93	6.19	1.03	3.99	0.54	97.85	97.44	4.90	2.60	3.73	3.54	13.25
maze	32	30	40.59	97.07	1.35	1.16	100.00	0.66	3.02	1.06	4.03	1.00	94.44	98.10	10.39	1.68	6.19	2.69	12.83
maze01	5	37	41.77	50.40	0.76	0.78	100.00	0.09	1.80	0.58	4.45	1.00	72.40	72.67	1.48	0.99	1.07	1.00	1.76
maze02	7	33	19.36	79.39	0.92	0.94	100.00	0.17	2.24	0.70	4.12	1.01	87.42	91.98	1.79	1.19	1.30	1.21	4.93
maze04	10	30	23.70	96.86	1.51	1.37	100.00	0.45	3.41	1.28	4.03	1.01	96.22	98.09	2.76	1.71	1.85	1.74	10.69
maze08	20	30	37.71	99.37	2.10	1.50	100.00	0.76	4.29	1.51	4.01	1.00	98.97	99.62	6.09	2.47	3.68	2.84	18.97
maze16	44	28	46.18	99.80	2.96	1.38	100.00	0.96	5.30	1.42	4.00	1.00	99.71	99.91	16.30	3.35	8.46	5.30	25.77
maze32	108	24	50.73	99.94	4.12	1.18	100.00	1.01	5.83	1.19	4.00	0.99	99.93	99.98	49.18	4.19	18.66	11.49	30.37
random	43	13	44.92	54.97	1.15	1.04	74.57	0.04	2.14	0.05	4.36	0.98	60.40	53.37	1.07	0.79	1.03	0.96	1.84
random10	54	15	52.27	65.45	1.71	1.11	75.43	0.08	3.63	0.10	4.37	0.95	71.70	59.36	1.08	1.20	1.03	1.06	2.23
random15	64	14	49.46	55.36	1.41	1.07	70.97	0.06	3.00	0.07	4.47	0.96	63.96	52.21	1.06	0.99	1.03	1.02	1.89
random20	64	14	46.01	49.22	1.23	1.05	69.87	0.05	2.60	0.06	4.46	0.98	59.08	49.19	1.06	0.88	1.03	1.00	1.74
random25	56	14	41.55	46.12	1.11	1.03	71.47	0.04	2.40	0.04	4.36	1.00	56.34	49.00	1.07	0.81	1.04	0.98	1.68
random30	37	11	36.26	45.29	1.01	1.01	74.97	0.03	1.90	0.03	4.23	1.00	55.03	50.69	1.07	0.71	1.02	0.93	1.66
random35	21	10	30.07	46.86	0.93	0.98	80.42	0.02	1.52	0.03	4.10	1.01	54.86	54.02	1.08	0.62	1.01	0.84	1.64
random40	7	12	24.94	49.62	0.85	0.90	86.74	0.01	1.19	0.03	4.06	1.00	54.95	57.15	1.17	0.54	0.97	0.65	1.38
room	41	10	42.84	97.79	2.04	1.13	98.54	0.34	3.33	0.41	4.01	1.00	98.18	98.83	4.61	1.82	3.59	3.01	17.78
room08	26	10	28.55	85.14	1.32	1.09	95.09	0.09	2.44	0.13	4.06	1.02	93.82	94.20	1.42	1.22	1.28	1.26	7.67
room16	23	9	37.96	97.58	1.82	1.18	98.84	0.31	3.10	0.42	4.01	1.00	98.60	99.03	2.49	1.63	2.04	1.90	16.91
room32	37	10	45.67	99.50	2.53	1.17	99.73	0.76	4.00	0.91	4.00	1.00	99.68	99.83	7.38	2.23	4.92	3.94	24.81
room64	76	11	50.35	99.88	3.41	1.11	99.95	0.98	4.74	1.09	4.00	1.00	99.93	99.97	28.26	3.03	12.51	8.99	29.33

Table 2: Analysis of the three methods of augmenting CHs. All values (except under the CH column) are relative to CHs where higher values are better (values without percentages are improvement factors). Improvements compared to CHs are shown in green (darker is better) and deterioration compared to CHs are shown in red (darker is worse). Times are reported in μs. Sr: Search time. Rf: Refinement time. DQ: Distance query time. PQ: Path query time. Sc%: Percentage of edges that are shortcuts. R%: Percentage of shortcuts that are canonical-freespace-reachable. C%: Percentage of contracted nodes. DN%: Percentage of discarded nodes. DE%: Percentage of discarded edges. M: Memory.

CH (→ CH-SL) → CH-SG: Contracting subgoals last seems to have a noticeable downside only on game maps (CH-SL, PQ), which can be explained by many game maps having diagonal walls that introduce a lot of subgoals that are not important for long-distance queries and contracted very early on by CHs, but not by CH-SLs. Using SGs as base graphs discards $97-98\%$ of the nodes and edges (DN%, DE%) on non-random maps and 60.4% of nodes and 53.37% edges on random maps. This improves search times (Sr) by as much as a factor of 49 on maze32 maps and as little as 7% on random maps. Distance query times (DQ) are improved overall, indicating that, on most maps, the specific R-connect method on grids is faster than traversing the lower-levels of the hierarchy through node expansions. Refinement times (Rf) are also improved except on random maps, but the improvement is smaller than that of CH-Rs over CHs, as refinements on CH-SGs first unpack shortcuts.

RCH-SG vs. N-SG: Not shown in Table 2, the numbers of nodes expanded by searches on RCH-SGs and N-SGs differ by at most 0.2% on maze, random, and room maps. On game maps, searches on N-SGs expand 2.2% fewer nodes but generate 11.4% more nodes. We conjecture that heavy-R-contracting a node removes it from the upward closures of more nodes than R-contracting it, which can result in the node being excluded from more searches. However, since N-SGs have more edges (74.82% more on game maps and 10.21% overall), more successors are generated per expansion, which seems to result in slower searches overall.

We conclude with a note on memory requirements. In domains where the graph is implicitly defined, such as grids or state lattices, CHs are constructed on an explicit copy of the graph and therefore incur a memory overhead. In domains where the graph is explicit, such as road networks, CHs can be constructed on the provided copy of the graph and can have a negative memory overhead (that is, require less memory to store than G) since downward edges are discarded. In such cases, using a SG as base graph might not allow one to discard the lowest level of the hierarchy since doing so would discard information about the original graph that is likely required by R-connect and R-refine.

6 Conclusions

We have discussed three methods of augmenting CHs to exploit domain structure by using reachability relations, discussed their trade-offs, and performed experiments to see how these trade-offs manifest themselves on grids. We reiterate that grids seem to be a best-case scenario for using subgoal graphs as base graphs, but we believe that efficient R-refine operations can be implemented for different domains that can be exploited by our methods. As future work, we plan to devise different reachability relations on road networks and also try out the different optimizations we have outlined in Section 3.1.

Acknowledgements

We thank Daniel Harabor and Ben Strasser for helpful discussions. The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1724392, 1409987, and 1319966. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

References

[Abraham *et al.*, 2010] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 782–793, 2010.

[Abraham *et al.*, 2011] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the International Symposium on Experimental Algorithms*, pages 230–241, 2011.

[Arz *et al.*, 2013] Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. In *Proceedings of the International Symposium on Experimental Algorithms*, pages 55–66, 2013.

[Bast *et al.*, 2006] Holger Bast, Stefan Funke, and Domagoj Matijević. TRANSIT— ultrafast shortest-path queries with linear-time preprocessing. In *Proceedings of the DIMACS Implementation Challenge – Shortest Paths*, 2006.

[Bauer and Delling, 2008] Reinhard Bauer and Daniel Delling. SHARC: Fast and robust unidirectional routing. In *Proceedings of International Workshop on Algorithm Engineering and Experiments*, pages 13–26, 2008.

[Delling *et al.*, 2009] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139. Springer, 2009.

[Demetrescu *et al.*, 2009] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Mathematical Society, 2009.

[Dibbelt *et al.*, 2014] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In *Proceedings of the International Symposium on Experimental Algorithms*, pages 271–282, 2014.

[Geisberger *et al.*, 2008] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the International Workshop on Experimental Algorithms*, pages 319–333, 2008.

[Geisberger, 2008] Robert Geisberger. *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks*. PhD thesis, Institut für Theoretische Informatik Universität Karlsruhe, 2008.

[Goldberg *et al.*, 2009] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Shortest path algorithms with preprocessing. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:93–139, 2009.

[Gutman, 2004] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of the Workshop on Algorithm Engineering and Experiments and the Workshop on Analytic Algorithmics and Combinatorics*, pages 100–111, 2004.

[Harabor and Grastien, 2011] Daniel Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the of AAAI Conference on Artificial Intelligence*, pages 1114–1119, 2011.

[Hilger *et al.*, 2009] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:41–72, 2009.

[Lauther, 2004] Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation und Mobilität-von der Forschung zur praktischen Anwendung*, 22:219–230, 2004.

[Sanders and Schultes, 2005] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings of the European Symposium on Algorithms*, pages 568–579, 2005.

[Strasser *et al.*, 2015] Ben Strasser, Adi Botea, and Daniel Harabor. Compressing optimal paths with run length encoding. *Journal of Artificial Intelligence Research*, 54:593–629, 2015.

[Sturtevant *et al.*, 2015] Nathan Sturtevant, Jason Traish, James Tulip, Tansel Uras, Sven Koenig, Ben Strasser, Adi Botea, Daniel Harabor, and Steve Rabin. The grid-based path planning competition: 2014 entries and results. In *Proceedings of the Symposium on Combinatorial Search*, 2015.

[Sturtevant, 2012] Nathan Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

[Uras and Koenig, 2014] Tansel Uras and Sven Koenig. Identifying hierarchies for fast optimal search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 878–884, 2014.

[Uras and Koenig, 2017] Tansel Uras and Sven Koenig. Feasibility study: Subgoal graphs on state lattices. In *Proceedings of the Symposium on Combinatorial Search*, 2017.

[Uras *et al.*, 2013] Tansel Uras, Sven Koenig, and Carlos Hernández. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 224–232, 2013.