The Eleventh International Symposium on Combinatorial Search
(SoCS 2018)

Fast Near-Optimal Path Planning on State Lattices with Subgoal Graphs

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

Abstract

Subgoal graphs can be constructed on top of graphs during a preprocessing phase to speed-up shortest path queries. They have an undominated query-time/memory trade-off in the Grid-Based Path Planning Competitions. While grids are useful for path planning, other kinds of graphs, such as state lattices, have to be used for motion planning. While state lattices are regular graphs like grids, subgoal graphs improve query times by a much smaller factor on state lattices than on grids. In this paper, we present a new version of subgoal graphs that forfeits its optimality guarantee for smaller query times. It guarantees completeness, and our experimental results on state lattices suggest that it can find paths that are close to optimal.

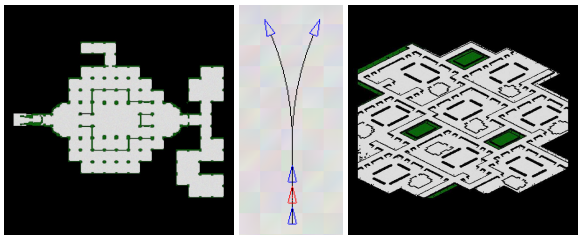
Introduction

Preprocessing-based path-planning algorithms analyze a given graph in a preprocessing phase to generate auxiliary information, which can then be used to significantly speed-up online path queries. The 9th DIMACS implementation challenge (Demetrescu, Goldberg, and Johnson 2009) featured a competition on preprocessing the USA road network, which resulted in several new preprocessing-based path-planning algorithms such as contraction hierarchies (Geisberger et al. 2008), transit routing (Bast, Funke, and Matijević 2006; Arz, Luxen, and Sanders 2013), and hub-labeling (Abraham et al. 2011). More recently, a similar competition was held on grids (Sturtevant et al. 2015), where an entry based on subgoal graphs (SGs) (Uras, Koenig, and Hernán dez 2013) was undominated with respect to its query-time/memory trade-off. SGs aim to exploit structure in maps by capturing it with a reachability relation R and then constructing an overlay graph (whose nodes are called subgoals) that has only R-reachable edges. SGs can be used to find shortest paths by first connecting the start and goal nodes to the SG to form a query SG, searching the query SG for a shortest path, and replacing its R-reachable edges with the corresponding shortest paths on the original graph.

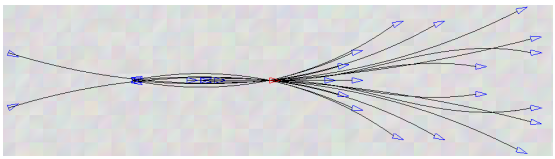
State lattices can be considered as extensions of grids where the state space is extended to take into account discretized poses for an agent. The edges of a state lattice are determined by a set of motion primitives that model kinematically feasible actions for the agent (Pivtoraiko and Kelly 2005; Likhachev and Ferguson 2009; Kushleyev and Likhachev 2009). As opposed to sampling-based motion planning algorithms (Kavraki et al. 1996; Kuffner and LaValle 2000; Karaman and Frazzoli 2011), state lattices can be used to systematically discretize environments into graphs which can then be searched with heuristic search algorithms to find paths that are optimal or bounded suboptimal (with respect to the state lattice). SGs have been applied to state lattices (Uras and Koenig 2017) but only achieved small speed-ups (2-4 times faster than searching the state lattices directly) since SGs tend to have more edges than the state lattices and have a high number of subgoals ($\sim 30\%$ of the nodes of state lattices).

In this paper, we explore different methods for improving the query times of SGs on state lattices. We first try constructing SGs using a preprocessing method which guarantees that the set of subgoals is minimal. We then try pruning some of the edges of SGs while maintaining a suboptimality bound. Both modifications fail to achieve a significant speed-up, which motivates us to give up the optimality guarantee of SGs to try to achieve smaller query times.

Our main contribution in this paper is a new version of SGs, called strongly-connected SGs (SC-SGs), which guarantees completeness by making sure that any pair of start and goal nodes can be connected to the SC-SG and that the SC-SG itself is strongly connected. We believe that this is a good first step in the direction of developing a bounded suboptimal version of SGs that not only modifies the edges of SGs, but also the placement of subgoals, since any bounded suboptimal SG would have to guarantee completeness as well. We observe that SC-SGs can be used to answer queries 1-2 orders of magnitude faster than A* on state lattices and find paths that are not much longer than optimal. Our second contribution is a new reachability relation on state lattices, called canonical freespace reachability, that aims to reduce the time to connect the start and goal nodes to SC-SGs, which can take longer than searching the resulting query SC-SGs. Our experiments show that, using canonical freespace reachability, SC-SGs can be used to find paths 200 times faster than A* on a state lattice with 15 million nodes and 333 million edges. Preprocessing takes 295 seconds, and the lengths of the paths found are on average 9.2% longer than those found by A*.



(a) arena2.map (209 × 281). (b) Unicycle primitives. (c) Aurora.map (768 × 1024).



(d) Urban challenge primitives.

Figure 1: Maps and primitives used in our experiments. In (b) and (d), all states shown as blue triangles can be reached with a single primitive from the state shown as a red triangle.

Preliminaries and Notation

We operate on (x, y, θ) state lattices, which are implicitly defined by a 2D grid, a set of discrete poses $\mathcal{P}$ (directions that the agent can face towards), and a set of motion primitives (primitives, for short) for each pose. Each primitive p is defined by a tuple $(\theta_p^s, \theta_p^e, x_p, y_p, l_p, C_p)$, where θ_p^s is the start pose of p, θ_p^e is the end pose of p, (x_p, y_p) is the end cell of p relative to the start cell, l_p is the length of p, and C_p is a set of cells, relative to the start cell, that need to be unblocked in order to execute p successfully.

We use $G = (V, E, c)$ to denote the (directed, weighted) graph that corresponds to the state lattice, where V is the set of nodes, E is the set of edges, and $c : E \to (0, \infty)$ is a function that assigns a non-negative length to each edge. Every node of G corresponds to a state (x, y, θ) of the state lattice, where $\theta \in \mathcal{P}$ and (x, y) is a grid cell whose center coincides with a reference point on the agent. For any two nodes $s = (x_s, y_s, \theta_s)$ and $e = (x_e, y_e, \theta_e)$, a primitive p induces an edge (s, e) with length l_m iff: (1) $\theta_s = \theta_p^s$, $\theta_e = \theta_p^e$, (2) $x_s + x_p = x_e$, $y_s + y_p = y_e$, and (3) for all $(x, y) \in C_p$, the cell $(x_s + x, y_s + y)$ is unblocked.

We use $d(s, t)$ to denote the s-t-distance (length of a shortest s-t-path) on G. For simplicity, we assume that G is strongly connected (that is, for any $u, v \in V$, $d(u, v) < \infty$). If the state lattice is not strongly connected, we assume that G is its largest strongly connected component. We describe later how SC-SGs can be extended to graphs that are not strongly connected.

We report experimental results in various sections of the paper, using two different setups: The *Unicycle* setup uses a small grid of size 209 × 281, 16 different discrete poses, and 5 primitives for each pose.[1] The *Urban* setup uses a large grid of size 768 × 1024, 32 different discrete poses, and 36 primitives for each pose.[2] We use the Unicycle setup to evaluate our various modifications to optimal SGs and use both the Unicycle and Urban setups to evaluate SC-SGs, using 1000 randomly generated instances for each setup. We use a PC with a 3.6GHz Intel Core i7-7700 CPU and 32GB of RAM to run our experiments.

Rather than using an implicit state lattice where each primitive's swept cells need to be checked to see if the primitive is executable, we store a bitfield for every state where each bit indicates whether a specific primitive is executable from that state. This implementation does not use as much memory as storing G explicitly and avoids checking during searches if primitives are executable, which can significantly reduce the search times over G and the preprocessing and query times of our SG and SC-SG variants. We remove redundant primitives (those that can be replaced with a sequence of other primitives), bringing down the number of primitives for each pose in the Unicycle setup to 4 and the Urban setup to between 27 to 32, which allows us to use 32-bit integers as bitfields. Figure 1 shows the maps and the primitives (for a single pose).

We normalize the edge lengths so that executing a primitive of length l can change both the x and y coordinates of the agent by at most $\lfloor l \rfloor$. For Urban primitives, the lengths of primitives that move the agent backward (with respect to its current pose) are multiplied by 3. For Unicycle primitives, backward moves are multiplied by 5 and turns are multiplied by 2. The largest strongly connected component of the resulting Unicycle setup has 375,391 nodes and 1,262,812 edges, and the largest strongly connected component of the resulting Urban setup has 15,255,281 nodes and 333,465,922 edges. We use an A* search with the Euclidean Distance heuristic (the straight-line distance between two nodes' corresponding cells in the underlying grid), A*-Euc, as the baseline algorithm when reporting the speed-up achieved by SGs and SC-SGs. All searches over SGs and SC-SGs also use the Euclidean Distance heuristic.

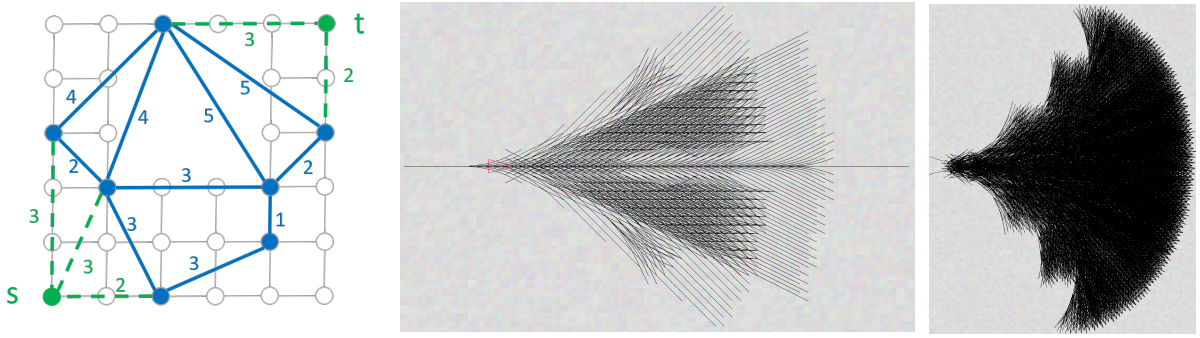
Background: Subgoal Graphs on State Lattices

A subgoal graph (SG) can be constructed as an overlay graph on G that only contains edges that satisfy a given reachability relation $R \subseteq V \times V$. Intuitively, R identifies pairs of nodes (u, v) such that a shortest u-v-path can be quickly found by exploiting structure in G. We say that a node t is *R-reachable* from a node s (or an edge (s, t) is R-reachable) iff $(s, t) \in R$. We use $R_s^{\rightarrow}$ to denote the set of nodes that are R-reachable from s and $R_s^{\leftarrow}$ to denote the set of nodes from which s is R-reachable. We assume that $\forall n \in V\ (n, n) \in R$ and $\forall (u, v) \in E\ (u, v) \in R$.

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 *cover* property: For any $(u, v) \notin R$, at least one shortest u-v-path

[1] https://github.com/sbpl/sbpl/blob/master/matlab/mprim/unicycle_noturninplace.mprim.

[2] We thank Maxim Likhachev for making the primitives used in their Urban Challenge entry available to us and for helpful discussions.



(a) Query SG on an undirected graph with unit-length edges.

(b) Freespace paths up to length 50 (Unicycle).

(c) Freespace paths up to length 50 (Urban).

Figure 2: SGs on state lattices. In (a), $(u, v) \in R \Leftrightarrow d(u, v) \leq 5$. In both (b) and (c), corresponding lookup tables for freespace distances contain $|\mathcal{P}|$ entries for each cell in the figure, one for each end pose. (b) is scaled up by a factor of 2.15.

passes through (is covered by) a subgoal. SGs can be used to answer shortest path queries in three steps: (1) *Connecting* the start node s to subgoals $u \in R_s^{\rightarrow}$ with edges (s, u) and the goal node t to subgoals $v \in R_t^{\leftarrow}$ with edges (v, t), and adding an edge (s, t) iff $(s, t) \in R$. (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. 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, it is sufficient to use only *direct-R*-reachable edges, which are R-reachable edges that cannot be refined into paths that pass through subgoals. Figure 2(a) shows a query SG where $(u, v) \in R \Leftrightarrow d(u, v) \leq 5$. Essentially, SGs can exploit structure in a domain by capturing it with R and developing specialized connection and refinement operations that utilize R. We call these operations *R-connect* and *R-refine*, respectively, and also distinguish between connecting the start ($R^{\rightarrow}$-connect) and goal ($R^{\leftarrow}$-connect) nodes to SGs.

Similarly to SGs on grids, SGs on state lattices use *freespace reachability* (FR) as R. Shortest paths between nodes on a state lattice where the underlying grid has no blocked cells are called *freespace paths*. The length of a freespace s-t-path is called the *freespace s-t-distance* $d_f(s, t)$. A node t is freespace reachable from a node s iff at least one freespace s-t-path is unblocked (or, equivalently, iff $d_f(s, t) = d(s, t)$). SGs on state lattices can exploit FR as follows: (1) FR$^{\rightarrow}$-connect can be implemented as a breadth-first search that maintains g-values for nodes and only generates nodes u with $g(u) = d_f(s, u)$. FR$^{\leftarrow}$-connect can be implemented similarly. (2) FR-refine can be implemented as a depth-first search that only generates nodes u with $d_f(s, u) + d_f(u, t) = d_f(s, t)$. (3) Edge lengths are freespace distances and therefore do not need to be stored.

Both FR-connect and FR-refine operations require freespace distances, which we precompute and store up to a certain bound B, called the reachability bound. We refer to FR with a reachability bound B as FRB (throughout the paper, we simply use FR instead of FRB if the specific B is not important). We exploit the translation invariance of freespace distances (that is, changing the x- or the y-coordinate of two states by the same value does not change the freespace distance between them) to store freespace distances more compactly: For each start pose θ, we store a freespace distance lookup table T_θ with up to $(2B - 1) \times (2B - 1) \times |\mathcal{P}|$ entries, where each entry (x_e, y_e, θ_e) in T_θ stores the freespace distance from $(0, 0, \theta)$ to (x_e, y_e, θ_e) (equivalently, from any (x, y, θ) to $(x + x_e, y + y_e, \theta_e)$). Note that, by exploiting symmetries in the state lattice, we can avoid storing a lookup table for each start pose (for Unicycle primitives, we can store only 3 tables rather than 16; for Urban primitives, we can store only 5 tables rather than 32). In our experiments, we do not do this compression and report results assuming each freespace distance is stored using 4 Bytes. Figures 2(b) and 2(c) show all the freespace paths (for a single start pose) up to length 50 for the Unicycle and Urban primitives.

We also use *bounded distance reachability* (DR) as a baseline reachability relation in our experiments, to see how much we gain from exploiting freespace structure with FR. A node t is bounded distance reachable (with reachability bound B) from a node s iff $d(u, v) \leq B$. We implement DR$^{\rightarrow}$- and DR$^{\leftarrow}$-connect as Dijkstra searches that do not generate nodes n with $g(n) > B$. We implement DR-refine as A*-Euc.

Subgoal Graphs on State Lattices Revisited

The current method of constructing SGs on state lattices starts with an empty set of subgoals S and then *grows* S by iterating over every node $s \in V$ and adding subgoals to S to cover at least one shortest path to every node t with $(s, t) \notin R$ (Uras and Koenig 2017). In this section, we try a different method of constructing SGs on state lattices based on *pruning* S rather than growing it, and also experiment with bounded suboptimal SGs.

	Prep. time (s)	Mem. (MB) F	Mem. (MB) SG	Size vs G Nodes	Size vs G Edges	Sp. up
G-DR50	28	-	12.1	55%	126%	1.50
G-DR75	81	-	15.5	42%	160%	1.77
G-DR100	191	-	19.9	34%	206%	1.90
G-DR125	312	-	24.5	29%	254%	1.94
G-DR150	533	-	29.5	25%	307%	1.95
G-FR50	34	10.0	6.0	55%	125%	1.50
G-FR75	90	22.3	7.2	44%	149%	1.75
G-FR100	173	39.5	8.2	40%	170%	1.77
G-FR125	253	61.5	9.0	38%	187%	1.81
G-FR150	305	88.5	9.7	38%	202%	1.83
P-BD50	676	-	13.9	46%	144%	1.65
P-BD75	13646	-	19.0	33%	198%	1.96
P-FR50	691	10.0	6.9	46%	143%	1.65
P-FR75	10547	22.3	8.8	35%	183%	1.99

Table 1: Comparison of SGs constructed by growing (G) and pruning (P) the set of subgoals, using DR and FR with different reachability bounds, on the Unicycle setup. F: Freespace distances.

Constructing SGs by Pruning

The pruning-based algorithm of constructing SGs is similar to constructing N-level SGs (Uras and Koenig 2014) and constructing k-hop covers on road networks (Funke, Nusser, and Storandt 2014). We only give an overview of this algorithm since it is not a main contribution of our paper. It starts by initializing $S = V$ and then tries to remove nodes from S one by one, while maintaining that S satisfies the cover property (as discussed in the previous section). To determine whether a node s can be removed from S without violating the cover property, the algorithm first generates the list of nodes u such that s is direct-R-reachable from u (using a modified version of $R^{\leftarrow}$-connect). Then, for each such u, it checks if there exists a node v such that $(u, v) \notin R$ and no shortest u-v-path is covered by a subgoal (using a modified version of $R^{\rightarrow}$-connect). If no such pair of nodes (u, v) is found, then s is removed from S. Otherwise, s remains a subgoal. As noted in Funke, Nusser, and Storandt's paper, this method of construction guarantees that the resulting set of subgoals is minimal (that is, no subgoal can be removed from S without violating the cover property) and the order that the nodes are processed in can effect the resulting set of subgoals. As suggested in their paper, we use a depth-first search from a random node to determine the order in which we try to prune nodes (where nodes whose children are processed first by the depth-first search appear earlier in the order). Our preliminary results suggest that this ordering works well for constructing SGs as well (although, it is not significantly better than a random ordering).

Table 1 shows a comparison of various SGs on the Unicycle setup with respect to their preprocessing times (which includes the time to calculate freespace distances for FR), query times (which is the summation of the connection, search, and refinement times), the number of subgoals and edges (relative to G), and the memory required to store the SG and freespace distances. We use DR and FR as R with varying reachability bounds $B \in \{50, 75, 100, 125, 150\}$.

w	1	1.5	2	2.5	3
Edge % vs G	182.9	78.0	68.6	68.4	66.2
Speed up	1.99	2.54	2.61	2.61	2.66
Subopt.	1.000	1.023	1.034	1.035	1.037

Table 2: Comparison of graph spanners of P-FR75 with different suboptimality bounds w on the Unicycle setup.

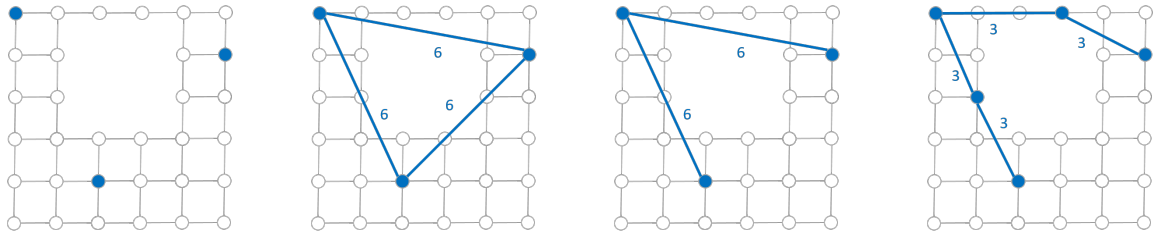
We use P-R to denote a SG constructed with respect to R by pruning and G-R to denote a SG constructed with respect to R by growing. We observe the following trends: 1) The number of subgoals decreases as R includes more pairs of nodes (for both FR and DR, as B increases, or as we switch from FR to DR). However, as the number of subgoals decreases, the number of edges increases. G-DR150 has the lowest number of subgoals (25% of $|V|$) but the highest number of edges (307% of $|E|$). 2) Pruning rather than growing results in fewer subgoals, since pruning guarantees minimality. However, the preprocessing times for pruning become prohibitive as the reachability bound increases. For instance, P-FR75 has 20% fewer subgoals than G-FR75 and can be used to answer queries 14% faster. However, preprocessing takes 117 times longer and it has 23% more edges. 3) For all SGs, the combined R-connect and R-refine times make up less than 10% of the overall query times (not reported in the table) and there is no significant difference between using FR or DR as R. 4) The speed-up achieved by various SGs range from 1.5 to 1.99. These results are similar to earlier ones and indicate that it might be difficult to speed up optimal path planning on state lattices by using SGs.

Graph Spanners of SGs

We now consider a bounded-suboptimal version of SGs by greedily constructing their *graph spanners* (Althöfer et al. 1993). A graph spanner is a subgraph of a graph whose distances are no longer than the distances on the graph, multiplied by a parameter w. The greedy algorithm for constructing a graph spanner iterates over the edges of the graph in order of increasing length, and adds an edge (u, v) with length $d(u, v)$ to the graph spanner iff the current u-v-distance on the spanner is greater than $wd(u, v)$. Table 2 compares the number of edges, query times, and suboptimalities of graph spanners constructed on P-FR75 from Table 1. We observe that, even with $w = 1.5$, constructing the graph spanner eliminates 57% of P-FR75's edges. Using $w = 3$ eliminates 64% of its edges, which results in only 33% faster queries that return paths that are 3.7% suboptimal.

Strongly Connected Subgoal Graphs

Motivated by the results in the previous section, we now describe a variant of SGs that aims to speed up queries by changing the placement of subgoals while forfeiting its optimality guarantee. Ideally, we would like to develop a variant that can guarantee bounded suboptimality. We believe that the variant that we describe in this section is a good first step towards this direction since any bounded-suboptimal version of SGs would have to guarantee completeness as well.



(a) Identify access subgoals so that every node can R-connect to a subgoal.

(b) Construct a graph A of access subgoals.

(c) Construct a spanning tree of A.

(d) Add (connecting) subgoals to split non-R-reachable edges into R-reachable ones.

Figure 3: High level idea of constructing SC-SGs. Example uses an undirected graph with unit-length edges and DR5 as R.

Definition and Theoretical Guarantees

Strongly-connected subgoal graphs (SC-SGs) guarantee completeness by ensuring that any node can be both $R^{\rightarrow}$- and $R^{\leftarrow}$-connected to the SC-SG and ensuring that the SC-SG is strongly connected (which is possible since we assume that G is strongly connected). Definition 1 outlines the minimum requirements for SC-SGs to guarantee completeness.

Definition 1. *A graph $G_S = (S, E_S, c_s)$ is a strongly connected subgoal graph on G (with respect to a reachability relation R) iff the following conditions hold: (1) For any $n \in V$, there exists $u, v \in S$ such that $(u, n) \in R$ and $(n, v) \in R$. (2) G_S is strongly connected. (3) $\forall (u, v) \in E_S$, $(u, v) \in R$ and $c_S(u, v) = d(u, v)$.*

SC-SGs can be used to answer queries in the same way as SGs are used to answer queries, by R-connecting the start and goal nodes to the SC-SG, searching the resulting *query* SC-SG for a path π_S (which is guaranteed to have only R-reachable edges), and finally R-refining the edges on π_S. To guarantee optimality, SGs have edges from every subgoal to every other subgoal that is direct-R-reachable from it. In contrast, to guarantee completeness, SC-SGs can have any set of edges as long as they are all R-reachable and make the SC-SG strongly connected. Similarly, for SGs, R-connect needs to identify all direct-R-reachable edges to connect the start and goal nodes to the SG. For SC-SGs, R-connect needs to identify at least one R-reachable edge to connect the start and the goal nodes, respectively, to the SC-SG.

Theorem 1. *(Completeness) For any start node s and goal node t, an SC-SG can be used to find an s-t-path.*

Proof. By Definition 1, there exists a $u \in S$ with $(s, u) \in R$. Therefore, $R^{\rightarrow}$-connect can find a subgoal $u' \in S$ with $(s, u') \in R$. (We make the distinction between u and u' for generality of the implementation of $R^{\rightarrow}$-connect, as it might connect s to another R-reachable subgoal u' rather than u, since identifying only one such subgoal is sufficient.) Similarly, $R^{\leftarrow}$-connect can find a subgoal $v' \in S$ with $(v', t) \in R$. Since, by Definition 1, a SC-SG is strongly connected, it must contain a path $\pi_{u',v'} = (n_1 = u', \ldots, n_k = v')$. When the edges (s, u') and (v', t) are added to the SC-SG, the resulting query SC-SG then contains a path $\pi_{s,t} = (s, n_1, \ldots, n_k, t)$. All edges of $\pi_{s,t}$ can be R-refined since $(s, u'), (v', t) \in R$ and all edges of the SC-SG are R-reachable by definition. □

Constructing SC-SGs

The subgoals of a SC-SG can be identified in two steps: (1) Identify a set of *access* subgoals which ensure that any node can be $R^{\rightarrow}$- and $R^{\leftarrow}$-connected to a subgoal (Algorithm 1). (2) Identify a set of *connecting* subgoals so that the set of access and connecting subgoals can be strongly connected using only R-reachable edges (Algorithm 2). Figure 3 provides an overview of how SC-SGs can be constructed on undirected graphs. We now explain how Algorithms 1 and 2 operate on directed, strongly connected graphs.

Algorithm 1 Ensuring that all nodes can both $R^{\rightarrow}$- and $R^{\leftarrow}$-connect to a subgoal.

```
1:  function IdentifyAccessSubgoals(G, R)
2:  S ← ∅
3:  for all n ∈ V do
4:      forward[n] ← false, backward[n] ← false
5:  for all n ∈ V do
6:      if ¬forward[n] ∨ ¬backward[n] then
7:          S ← S ∪ {n}
8:          for all t ∈ V : (n, t) ∈ R do
9:              backward[t] ← true
10:         for all s ∈ V : (s, n) ∈ R do
11:             forward[t] ← true
12: return S
```

Algorithm 1 starts with an empty set of subgoals S (line 2) and then, for each $n \in V$, adds n to S iff S does not contain a node u with $(n, u) \in R$ and a node v with $(v, n) \in R$ (lines 5-7). That is, if n cannot be $R^{\rightarrow}$- and $R^{\leftarrow}$-connected to subgoals, it becomes a subgoal. Algorithm 1 maintains two flags for each node n, forward[n] and backward[n], that indicate, respectively, whether there exists a subgoal u with $(n, u) \in R$ and whether there exists a subgoal v with $(v, n) \in R$. These flags are initially set to false for each node (lines 3-4). When a node n becomes a subgoal, for every node $t \in R_n^{\rightarrow}$ (that is, $(n, t) \in R$), backward[t] is set to true and, for every node $s \in R_n^{\leftarrow}$ (that is, $(s, n) \in R$), forward[s] is set to true (lines 8-11). $R_n^{\rightarrow}$ and $R_n^{\leftarrow}$ can be

identified with modified versions of $R^{\rightarrow}$- and $R^{\leftarrow}$-connect, respectively. Using the forward- and backward-flags ensures that the modified versions of $R^{\rightarrow}$- and $R^{\leftarrow}$-connect are executed only for nodes that become subgoals.

Once the access subgoals have been identified (Figure 3(a)), we can strongly connect them by adding edges between every pair of access subgoals u and v. Let A denote the resulting graph (Figure 3(b)), which might contain more edges than necessary to make it strongly connected. If A is undirected (because G is undirected), we can instead use the edges of a spanning tree of A to make the access subgoals strongly connected. On directed graphs, we can use the edges of two directed spanning trees of A, an out-tree (all edges point away from the root) and an in-tree (all edges point toward the root) rooted at the same (arbitrarily chosen) node s to make the access subgoals strongly connected: For any two access subgoals u and v, the in-tree contains a u-s-path, the out-tree contains an s-v-path, and, therefore, their combination contains a u-v-path. Note that, the edges used for strongly connecting access subgoals are not necessarily R-reachable. We can replace every non-R-reachable edge (u, v) with a sequence of R-reachable edges $(u, n_1), (n_1, n_2), \ldots, (n_k, v)$ and add the nodes $n_1, \ldots, n_k$ to the set of subgoals (Figure 3(d)) to finalize the construction of an SC-SG. Algorithm 2 outlines a method of constructing SC-SGs that interleaves the construction of A, construction of the spanning trees on A, and splitting of non-R-reachable edges into R-reachable ones.

Algorithm 2 Ensuring that the SG-SC is strongly connected with only R-reachable edges.

```
1: function StronglyConnectSubgoals(G, R, S)
2: E_S ← ∅
3: randomly select s ∈ S
4: for both the forward and backward directions do
5:     for all n ∈ V do
6:         g[n] ← ∞, parent[n] ← undefined
7:     g[s] ← 0, OPEN ← {s}
8:     while OPEN ≠ ∅ do
9:         v ← OPEN.PopMinGValNode()
10:        if v ∈ S \ {s} then
11:            π ← follow parents from v to a u ∈ S
12:            Reverse π if searching backward
13:            S', E'_S ← SplitIntoRReachable(π)
14:            S ← S ∪ S', E_S ← E_S ∪ E'_S
15:            g[v] ← 0
16:            for all n ∈ S' do
17:                g[n] ← 0, insert/update n in OPEN
18:        Expand(v)
19: return S, E_S
```

For both the forward and backward directions (we describe only the forward direction), Algorithm 2 performs a Dijkstra search[3] from a randomly selected subgoal s (lines 3-9, 18), with the following modifications: (1) When the search selects a subgoal $v \neq s$ for expansion, it follows the parent pointers to a subgoal u to extract a path π, which corresponds to an edge (u, v) in the spanning (out-)tree, and then introduces subgoals to split (u, v) into R-reachable edges using the procedure SplitInto$R$Reachable[4] (lines 11-14). (2) The g-values of v and all new connecting subgoals are set to 0 (lines 15-17), which ensures that the next subgoal to be reached is the one that is closest to all the subgoals that have been reached so far (which mimics Prim's algorithm (Prim 1957) for constructing minimum spanning trees on undirected graphs).

[3]The parent pointer of a node u is updated to v iff the g-value of u is lowered when v is expanded.

SC-SGs for Graphs that are not Strongly Connected

If G is not strongly connected, we can construct its strongly connected component graph $\mathcal{C}$ in linear time (Tarjan 1972). Each node c_i of $\mathcal{C}$ corresponds to a strongly connected component (subgraph) C_i of G, and each edge $(n_i, n_j) \in E$ where n_i belongs to C_i and n_j belongs to $C_j \neq C_i$ induces an edge (c_i, c_j) in $\mathcal{C}$. We can apply Algorithms 1 and 2 to each subgraph C_i to construct a SC-SG S_i. To guarantee completeness, we also need to connect each S_i to S_j for every edge (c_i, c_j) of $\mathcal{C}$. We can do this by randomly picking an edge $(n_i, n_j) \in E$ for every edge (c_i, c_j) of $\mathcal{C}$, where n_i belongs to C_i and n_j belongs to C_j, then add n_i to the access subgoals of S_i and n_j to the access subgoals of S_j before running Algorithm 2 on C_i and C_j, which allows us to connect the resulting S_i to the resulting S_j with the R-reachable edge (n_i, n_j).

Experimental Results

For our experiments, similar to SGs, we use all direct-R-reachable edges both when constructing SC-SGs and when connecting start and goal nodes to them, in order to find paths that are close to optimal. We therefore use the FR-connect and DR-connect operations described earlier. We process nodes in a random order in Algorithm 1.

Table 3 shows a comparison of various SC-SGs (denoted as C-R) on the Unicycle setup using DR and FR (and canonical freespace reachability (CR) that we introduce and discuss in the next section). In addition to the metrics we use to evaluate SGs in Table 1, we also report the number of access subgoals (relative to $|V|$), the average and maximum suboptimalities of the returned paths, and a breakdown of query times into connection, search, and refinement times. We observe the following trends: 1) Constructing SC-SGs requires significantly less preprocessing time than constructing SGs. 2) SC-SGs have significantly fewer subgoals and edges than SGs. For instance, C-DR150 has 97.87% fewer nodes and 98.90% fewer edges than G-DR150. Similar to SGs, the number of subgoals decreases as R includes more pairs of nodes. Contrary to SGs, the number of edges decreases as the number of subgoals decreases. 3) Queries on SC-SGs

[4]SplitInto$R$Reachable can be implemented recursively. Our implementation finds the highest index i on $\pi = (n_0, \ldots, n_k)$ such that $(n_0, n_i) \in R$ and, if $i \neq k$, adds n_i to the set of connecting subgoals and repeats for $(n_i, \ldots, n_k)$.

	Prep.	Memory (MB)		Size vs G			Query time (ms)			Speed	Suboptimality	
	time (s)	F	SG	Access	Nodes	Edges	Cn	Sr	Rf	up	Avg	Max
C-DR50	6.12	0.00	6.27	8.36%	13.93%	65.08%	0.158	4.188	0.303	7.08	1.329	4.675
C-DR75	11.75	0.00	5.93	3.90%	6.48%	61.53%	0.611	1.861	0.742	10.23	1.287	3.781
C-DR100	13.95	0.00	3.48	1.91%	2.96%	36.13%	1.424	0.730	1.186	9.85	1.314	4.476
C-DR125	7.74	0.00	0.61	0.60%	0.91%	6.29%	2.772	0.210	2.051	6.53	1.467	4.220
C-DR150	7.77	0.00	0.33	0.35%	0.52%	3.37%	4.823	0.119	2.736	4.28	1.495	3.934
C-FR50	5.83	9.96	3.10	8.39%	13.93%	64.43%	0.104	4.108	0.041	7.73	1.309	4.675
C-FR75	10.12	22.27	2.96	4.08%	6.74%	61.37%	0.329	1.926	0.038	14.34	1.213	4.931
C-FR100	9.94	39.45	1.61	2.13%	3.32%	33.50%	0.624	0.841	0.041	21.85	1.181	3.292
C-FR125	5.95	61.52	0.45	0.91%	1.48%	9.30%	0.972	0.357	0.064	23.60	1.169	2.788
C-FR150	5.96	88.48	0.33	0.67%	1.16%	6.91%	1.267	0.283	0.054	20.51	1.154	2.566
C-CR50	1.91	7.47	6.11	8.40%	13.82%	63.44%	0.039	3.951	0.008	8.22	1.296	4.675
C-CR75	2.89	16.69	5.77	4.17%	6.70%	59.93%	0.119	1.799	0.008	17.08	1.188	4.914
C-CR100	3.08	29.58	3.49	2.31%	3.49%	36.19%	0.223	0.885	0.007	29.47	1.165	3.109
C-CR125	2.80	46.12	1.50	1.29%	1.97%	15.57%	0.330	0.524	0.007	38.20	1.153	2.430
C-CR150	3.63	66.33	1.16	0.99%	1.58%	12.02%	0.454	0.427	0.008	37.05	1.144	2.370

Table 3: Comparison of SC-SGs (C) using DR, FR, and CR with different reachabilty bounds on the Unicycle setup. The query times are split into connection (Cn), search (Sr), and refinement (Rf) times. F: Freespace information.

are faster than queries on SGs. For instance, queries on C-FR125 are 13 times faster than queries on G-FR125, but return paths that are 18% longer. 4) As B increases for FR and DR, the search times decrease (since the resulting SC-SGs get smaller), but the connection times increase and become the dominant factor for the query times. For instance, for C-FR150, the connection times are 4.48 times longer than the search times, and, for C-DR150, the connection times are 40.5 times longer than the search times. As a result, SC-SGs constructed with respect to FR are consistently faster than SC-SGs constructed with respect to DR, due to significantly faster R-connect and R-refine operations. Contrary to SGs, increasing the reachability bound does not result in faster queries for SC-SGs. For instance, C-FR150 is slower than C-FR125 due to its longer connection times.

Canonical Freespace Reachability

Canonical orderings break symmetries between shortest paths by fixing, among multiple symmetric shortest paths, one as the canonical path. A successor v of a node u, with respect to a start node s, is called a *canonical successor* of u iff v extends the canonical s-u-path to the canonical s-v-path. Search algorithms can exploit canonical orderings to avoid generating duplicate nodes during searches by only generating the canonical successors of expanded nodes, which can significantly reduce the number of node expansions if duplicate detection is not possible due to memory limitations (Taylor and Korf 1993; Holte and Burch 2014), or reduce the average node expansion time if canonical successors of nodes can be efficiently identified (Harabor and Grastien 2011; Sturtevant and Rabin 2016). In this section, we impose a canonical ordering on freespace paths to develop a new reachability relation on state lattices, called *canonical freespace reachability* (CR), with the aim to implement a faster CR-connect operation than FR-connect.

We can represent any path $\pi = (n_0, \ldots, n_k)$ on a state lattice as a sequence of primitives $\pi_p = (p_0, \ldots, p_{k-1})$ where each p_i induces the edge (n_i, n_{i+1}) in the state lattice. Let $<_L$ be a total ordering on all primitives. For any two paths $\pi_p = (p_0, \ldots, p_k)$ and $\pi'_p = (p'_0, \ldots, p'_{k'}) \neq \pi_p$, we say that π_p is *lexicographically smaller than* π'_p, denoted as $\pi_p <_L \pi'_p$ iff: $\exists j$ ($p_j <_L p'_j$ and $\forall i < j$ $p_i = p'_i$), or, if no such j exists, if $k < k'$. Among all symmetric freespace paths, we fix the lexicographically smallest one as the *canonical freespace path*. A node t is canonical freespace reachable from a node s iff the canonical freespace s-t-path is unblocked.

Lemma 1. *For any canonical freespace path π, any subpath π' of π is also a canonical freespace path.*

Proof. Suppose a symmetric path $\pi'' = (b_0, \ldots, b_m)$ of $\pi' = (a_0, \ldots, a_n)$ exists with $\pi'' <_L \pi'$. Let i be the smallest number for which $a_i \neq b_i$ (i must exist since π' and π'' are not prefixes of each other because, otherwise, one of them would be longer). Since $\pi'' <_L \pi'$, $b_i <_L a_i$. Then, substituting π'' for π' in π generates a path that is symmetric to π but lexicographically smaller, contradicting our assumption that π is a canonical freespace path. □

Lemma 2. *The collection of all canonical freespace paths that originate at a node s form an out-tree rooted at s (forward canonical tree). The collection of all canonical freespace paths that terminate at a node t form an in-tree rooted at t (backward canonical tree).*

Proof. Let π and π' be two canonical freespace paths that originate at s and intersect at a node n. By Lemma 1, the prefixes of π and π' up to n must also be canonical freespace paths. Since, by definition, there is a unique s-n-canonical path, the prefixes of π and π' up to n must be the same. We can similarly show that if two canonical freespace paths that terminate at t intersect at a node n, then their suffixes from n must be the same. □

Following Lemma 2, we implement CR$^{\rightarrow}$-connect as a breadth-first traversal of the forward canonical tree rooted

	Prep.	Memory (MB)		Size vs G			Query time (ms)			Speed	Suboptimality	
	time (s)	F	SG	Access	Nodes	Edges	Cn	Sr	Rf	up	Avg	Max
C-FR50	979.74	39.85	24.53	1.02%	1.78%	1.93%	6.357	17.787	0.295	128.52	1.116	1.637
C-FR75	1781.37	89.07	25.03	0.63%	1.18%	1.97%	22.025	11.413	0.436	92.72	1.092	1.358
C-FR100	2520.5	157.82	23.68	0.49%	0.94%	1.86%	47.353	9.117	0.747	54.89	1.087	1.374
C-FR125	3097.43	246.10	22.35	0.42%	0.82%	1.76%	79.590	8.381	1.067	35.28	1.095	1.728
C-FR150	3389.96	353.91	21.44	0.39%	0.76%	1.69%	99.687	7.535	1.394	28.92	1.093	1.427
C-CR50	207.45	89.66	62.80	1.14%	2.03%	2.47%	0.807	21.192	0.016	142.67	1.110	1.657
C-CR75	295.08	200.40	57.67	0.67%	1.27%	2.27%	2.777	12.874	0.015	200.49	1.092	1.481
C-CR100	355.78	355.09	51.44	0.51%	0.99%	2.02%	5.795	9.790	0.015	201.34	1.088	1.458
C-CR125	415.97	553.72	47.66	0.44%	0.86%	1.87%	9.534	8.943	0.015	169.85	1.089	1.760
C-CR150	475.05	796.30	44.78	0.40%	0.80%	1.76%	12.119	7.922	0.015	156.61	1.096	1.613

Table 4: Comparison of SC-SGs (C) using FR and CR with different reachabilty bounds on the Urban setup. The query times are split into connection (Cn), search (Sr), and refinement (Rf) times. F: Freespace information.

wA*-Euc; w =	1	1.5	2	2.5	3
Speed up	1.00	3.78	7.24	10.60	13.09
Avg. subopt.	1.000	1.052	1.140	1.214	1.286
Max. subopt.	1.000	1.370	1.731	1.920	2.156
wA*-2D; w =	1	1.5	2	2.5	3
Speed up	4.88	36.08	53.12	61.60	61.45
Avg. subopt.	1.007	1.073	1.139	1.196	1.433
Max. subopt.	1.030	1.227	1.574	1.731	2.965

Table 5: Comparison of weighted A* searches using the Euclidean distance heuristic (wA*-Euc) and the 2D heuristic (wA*-2D) with different suboptimality bounds w on the Urban setup.

at the start node. CR$^{\rightarrow}$-connect does not perform duplicate detection and therefore requires less time to expand each node compared to FR$^{\rightarrow}$-connect. Our implementation uses precomputed canonical successor lookup tables (similar to freespace distance lookup tables used in FR): Each entry (x_e, y_e, θ_e) in the canonical successor lookup table T_θ is a bitfield where the ith bit indicates whether extending the canonical freespace $(0, 0, \theta)$-(x_e, y_e, θ_e)-path with the ith primitive (for pose θ_e) results in a canonical freespace path. We populate the canonical successor lookup tables by depth-first searches that generate freespace paths in increasing lexicographic order. CR$^{\leftarrow}$-connect is implemented similarly by using precomputed canonical predecessor lookup tables.[5] We implement CR-refine as a series of *canonical parent* lookups to generate the canonical freespace path between a given pair of CR-reachable nodes. Whereas canonical successor lookup tables store the children of nodes in forward canonical trees, canonical parent lookup tables store their parents instead. Each entry (x_e, y_e, θ_e) in the canonical parent lookup table T_θ identifies the last primitive (using its ID) on the canonical freespace $(0, 0, \theta)$-(x_e, y_e, θ_e) path.[6]

We compare SC-SGs using FR and CR on the Unicycle (Table 3) and Urban (Table 4) setups. We observe the following trends: (1) CR-connect is 2.6-2.9 times faster than FR-connect (for the same reachability bound) on the Unicycle setup and 7.8-8.2 times faster on the Urban setup. Compared to the Unicycle setup, FR-connect is 60.9-81.8 times slower and CR-connect is 20.6-28.9 times slower on the Urban setup, which has up to 8 times more primitives per pose. These results suggest that connection times scale better with the number of primitives per pose when using CR rather than FR. (2) CR-refine is 4.8-9 times faster than FR-refine on the Unicycle setup and 18.9-96.1 times faster on the Urban setup. Although this is a significant improvement, it is not reflected in query times since refinement times make up at most 4.6% of query times for FR. (3) SC-SGs using CR have up to 35% more subgoals on the Unicycle setup and up to 14% more subgoals on the Urban setup. As a result, searches using CR are up to 34% slower on the Unicycle setup and up to 17% slower on the Urban setup. CR induces more subgoals in SC-SGs because if a canonical freespace s-t-path is unblocked, then it must be the case that a freespace s-t-path is unblocked (that is, if $(s, t) \in$ CR, then $(s, t) \in$ FR) and, therefore, the SC-SG for CR can be used as an SC-SG for FR. (4) Paths found by using FR and CR are of similar length. (5) Preprocessing using CR is 1.6-3.5 times faster than FR on the Unicycle setup and 4.7-7.4 times faster on the Urban setup. These results mirror the speed-up CR-connect achieves over FR-connect, since R-connect is used when identifying the access subgoals and edges of SC-SGs. (6) Using CR requires slightly less memory than using FR on the Unicycle setup (where we can use 1 Byte to store each entry in the canonical successor/predecessor tables), but requires more than double the memory on the Urban setup.

Table 5 shows a comparison of weighted A* searches on the Urban setup's state lattice, using the Euclidean Distance heuristic (wA*-Euc) and the 2D heuristic[7] (wA*-2D), using different suboptimality bounds w. wA*-2D with $w = 2$ achieves a speed-up of 53.1 and finds paths that are 13.9% longer than optimal. C-CR100 achieves a speed-up of 201.3 and finds paths that are only 8.8% longer than optimal.

[5] Since there are at most 4 (32) primitives per pose in the Unicycle (Urban) setup, we use 1 Byte (4 Bytes) to store each entry in the canonical successor/predecessor lookup tables.

[6] We use 1 Byte to store each entry in the canonical parent lookup tables for both the Unicycle and Urban setups, which can distinguish between 256 primitive IDs per pose.

[7] The 2D-heuristic is the distance between the cells corresponding to two nodes in the underlying (8-neighbor) grid and is not necessarily admissible.

Conclusions and Future Work

We have tried to improve the query times of SGs on state lattices by using a construction strategy which guarantees that the set of subgoals is minimal and by pruning the edges of SGs while maintaining a suboptimality bound. Both methods failed to significantly improve the query times of SGs. Motivated by these results, we proposed a variant of SGs that try to minimize the number of subgoals while maintaining completeness, which we believe to be a good first step in the direction of developing a bounded-suboptimal version of SGs that not only modifies the edges of SGs but also the placement of the subgoals. Our new variant, SC-SGs, achieved a speed-up of 200 over A* on a state lattice with 15 million nodes and 333 million edges, using our new reachability relation, canonical freespace reachability. Although SC-SGs do not guarantee bounded suboptimality, they seem to find paths that are close to optimal in practice.

We believe that there are several ways to improve SC-SGs or augment them to achieve different trade-offs of query times, memory, and suboptimality. For instance, our preliminary experiments showed that iterating over nodes in a different order in Algorithm 1 can result in fewer subgoals, or adding more subgoals to SC-SGs can result in slower queries that find shorter paths. We believe that a similar result can be achieved with a smaller increase in runtime by *smoothing* subgoal paths using CR-refine with a higher reachability bound, by iteratively identifying two CR-reachable nodes on the subgoal path that are not consecutive and bypassing the nodes between them. We also envision a version of SC-SGs that uses a smaller reachability bound for connecting start and goal nodes to the SC-SG and a larger reachability bound for strongly connecting the subgoals.

Acknowledgements

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, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the International Symposium on Experimental Algorithms*, 230–241.

Althöfer, I.; Das, G.; Dobkin, D.; Joseph, D.; and Soares, J. 1993. On sparse spanners of weighted graphs. *Discrete & Computational Geometry* 9(1):81–100.

Arz, J.; Luxen, D.; and Sanders, P. 2013. Transit node routing reconsidered. In *Proceedings of the International Symposium on Experimental Algorithms*, 55–66.

Bast, H.; Funke, S.; and Matijević, D. 2006. TRANSIT— ultrafast shortest-path queries with linear-time preprocessing. In *Proceedings of the DIMACS Implementation Challenge – Shortest Paths*.

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

Funke, S.; Nusser, A.; and Storandt, S. 2014. On k-path covers and their applications. *Proceedings of the VLDB Endowment* 7(10):893–902.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the International Workshop on Experimental Algorithms*, 319–333.

Harabor, D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the of AAAI Conference on Artificial Intelligence*, 1114–1119.

Holte, R. C., and Burch, N. 2014. Automatic move pruning for single-agent search. *AI Communications* 27(4):363–383.

Karaman, S., and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research* 30(7):846–894.

Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; and Overmars, M. H. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation* 12(4):566–580.

Kuffner, J. J., and LaValle, S. M. 2000. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, 995–1001. IEEE.

Kushleyev, A., and Likhachev, M. 2009. Time-bounded lattice for efficient planning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1662–1668.

Likhachev, M., and Ferguson, D. 2009. Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research* 28(8):933–945.

Pivtoraiko, M., and Kelly, A. 2005. Generating near minimal spanning control sets for constrained motion planning in discrete state spaces. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3231–3237.

Prim, R. C. 1957. Shortest connection networks and some generalizations. *Bell Labs Technical Journal* 36(6):1389–1401.

Sturtevant, N., and Rabin, S. 2016. Canonical orderings on grids. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 683–689.

Sturtevant, N.; Traish, J.; Tulip, J.; Uras, T.; Koenig, S.; Strasser, B.; Botea, A.; Harabor, D.; and Rabin, S. 2015. The grid-based path planning competition: 2014 entries and results. In *Proceedings of the Symposium on Combinatorial Search*.

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1(2):146–160.

Taylor, L. A., and Korf, R. E. 1993. Pruning duplicate nodes in depth-first search. In *Proceedings of the AAAI National Conference*, 756–761.

Uras, T., and Koenig, S. 2014. Identifying hierarchies for fast optimal search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 878–884.

Uras, T., and Koenig, S. 2017. Feasibility study: Subgoal graphs on state lattices. In *Proceedings of the Symposium on Combinatorial Search*.

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