Feasibility Study: Subgoal Graphs on State Lattices

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

Abstract

Search using subgoal graphs is a recent preprocessing-based path-planning algorithm that can find shortest paths on 8-neighbor grids several orders of magnitude faster than A*, while requiring little preprocessing time and memory overhead. In this paper, we first generalize the ideas behind subgoal graphs to a framework that can be specialized to different types of environments (represented as weighted directed graphs) through the choice of a reachability relation. Intuitively, a reachability relation identifies pairs of vertices for which a shortest path can be found quickly. A subgoal graph can then be constructed as an overlay graph that is guaranteed to have edges only between vertices that satisfy the reachability relation, which allows one to find shortest paths on the original graph quickly. In the context of this general framework, subgoal graphs on grids use *freespace-reachability* (originally called h-reachability) as the reachability relation, which holds for pairs of vertices if and only if their distance on the grid with blocked cells is equal to their distance on the grid without blocked cells (freespace assumption). We apply this framework to state lattices by using variants of freespace-reachability as the reachability relation. We provide preliminary results on (x, y, θ)-state lattices, which shows that subgoal graphs can be used to speed up path planning on state lattices as well, although the speed-up is not as significant as it is on grids.

Introduction

Path planning is the problem of finding a sequence of waypoints that an agent can follow in a continuous environment to its destination without colliding with obstacles. The path-planning problem has many real-world applications in video games, robotics, and GPS-based navigation. A typical approach to path planning is to discretize the environment into a graph and use search algorithms to find shortest paths on the graph. The choice of discretization depends mostly on the environment and the application. For GPS-based navigation, one can represent the environment as a road network. For video games characters in 2D environments, one can represent the environment as a 2D grid of blocked and unblocked cells. For more complex agents with kinematic constraints (for instance, automobiles that need to parallel park), one can capture their kinematically feasible motions with a state lattice, which represents the environment as a grid, but also accounts for other (discretized) features of their states, such as their orientations and velocities.

In recent years, there has been a surge of interest in preprocessing-based path planning. In some applications of path planning, the environment is static and known in advance, allowing one to analyze the environment (or its associated graph) in a preprocesing phase and cache information that can be used to speed up online path-planning queries. The 9th DIMACS Implementation Challenge (Demetrescu, Goldberg, and Johnson 2006) featured a competition on preprocessing the USA road network. As a result, a lot of new preprocessing-based path-planning algorithms were designed, such as Contraction Hierarchies (Geisberger et al. 2008; Dibbelt, Strasser, and Wagner 2014), Transit Routing (Bast, Funke, and Matijevic 2006; Arz, Luxen, and Sanders 2013), Highway Hierarchies (Sanders and Schultes 2005; 2006), Reach (Gutman 2004; Goldberg, Kaplan, and Werneck 2006; 2009), and Hub-labeling (Lauther 2004; Hilger et al. 2009; Bauer and Delling 2009; Abraham et al. 2011). Most of these algorithms are applicable to any graph and provide excellent experimental results on road networks, speeding up shortest-path queries by several orders of magnitude, while requiring only slightly more memory than the road network alone. Most of these algorithms are heuristic in nature, without good performance guarantees. Their success on road networks has been attributed to road networks having low highway dimensions (Abraham et al. 2010). When some of these methods were applied to grids, the achieved speed-up was less significant and their memory requirements were higher, even if the grids were represented as explicit graphs (Antsfeld et al. 2012; Storandt 2013; Sturtevant 2012b).

Search using subgoal graphs (Uras, Koenig, and Hernández 2013) is a recent preprocessing-based path-planning algorithm for grids. During the preprocessing phase, a subgoal graph is constructed by placing subgoals at the convex corners of all blocked cells and connecting all pairs of subgoals that are *direct-h-reachable*. The resulting graph is essentially a sparse visibility graph that can be used to find shortest paths by first connecting the given start and goal vertices to all of their respective direct-h-reachable subgoals to form a query subgoal graph, finding a shortest (high-level) path on the query subgoal graph and then replacing the edges of

the high-level path with corresponding shortest paths on the grid. Search using subgoal graphs can be several orders of magnitude faster than an A* search on the grid since any vertex that is not a subgoal (or the start or the goal) is ignored during the search. Furthermore, by exploiting the structure of grids, the procedures for connecting the start and goal vertices to the subgoal graph and for refining a high-level path to a low-level path can be implemented very efficiently. A further preprocessed version of subgoal graphs (Uras and Koenig 2014) have been an undominated entry in the Grid-Based Path-Planning Competition (GPPC) with respect to its runtime/memory trade-off.

Our first contribution in this paper is a generalization of the ideas behind subgoal graphs to a framework that can be specialized to different types of environments (represented as weighted directed graphs) through the choice of a reachability relation. Intuitively, a reachability relation identifies pairs of vertices for which a shortest path can be found quickly. A subgoal graph can then be constructed as an overlay graph that is guaranteed to have edges only between vertices that satisfy the reachability relation which allows one to find shortest paths on the original graph quickly.

Our second contribution in this paper is the application of this framework to state lattices. State lattices, similar to grids, systematically discretize the environment and can be considered as extensions of grids that can model kinematic constraints (Pivtoraiko and Kelly 2005; Likhachev and Ferguson 2009; Kushleyev and Likhachev 2009). Under the assumption that the environment does not contain any obstacles (freespace assumption), the shortest paths between any two vertices on state lattices can be computed quickly by using precomputed freespace distances. By exploiting the regular structure of state lattices, we can efficiently precompute and store freespace distances, which allows us to specialize the general framework of subgoal graphs to state lattices by using variants of freespace-reachability as the reachability relation. We provide preliminary results on (x, y, θ)-state lattices, which show that subgoal graphs can be used to speed up path planning on state lattices as well, although the speed-up is not as significant as on grids.

Preliminaries and Notation

Our generalized framework of subgoal graphs is applicable to any weighted directed graph $G = (V, E, c)$, where V is the set of vertices, E is the set of edges, and $c : E \to (0, \infty)$ is a function that assigns a non-negative length to each edge. We use $d_G(s, t)$ to denote the s-t-distance on G, or simply $d(s, t)$ if G can be inferred from the context. We say that a vertex u *covers* an s-t-path $\pi = (v_0 = s, \dots, v_n = t)$ (where, for all $i = 1, \dots, n$, $(v_{i-1}, v_i) \in E$) if and only if $u = v_i$ for some $i \in \{1, \dots, n-1\}$. (This implies that $u \neq s$ and $u \neq t$ if π has no loops.) We assume that any edge $(u, v) \in E$ is the unique shortest u-v-path. This is not a limiting assumption when finding shortest paths since any edge (u, v) that does not satisfy the assumption can be removed from the graph without changing the length of any shortest path.

Subgoal Graphs: General Framework

In this section, we formally introduce the generalized version of subgoal graphs. The general framework of subgoal graphs can be specialized to different types of graphs through the choice of a reachability relation, which is defined as follows.

Definition 1. (R-Reachability) *Given a graph $G = (V, E, c)$, a reachability relation on G is a relation $R \subseteq V \times V$ that satisfies:*

1. $\forall s \in V$, $(s, s) \in R$,
2. $\forall (s, t) \in E$, $(s, t) \in R$ (edge property),
3. $\forall (s, t) \in R$, $d(s, t) < \infty$.

We use "t is R-reachable from s" and "s and t are R-reachable" (if R is symmetric) as synonyms for $(s, t) \in R$.

The vertices of a subgoal graph, called subgoals, form a subset of V that satisfies the following property: For any two vertices s and t in V such that $d(s, t) < \infty$, at least one shortest s-t-path can be split into R-reachable segments by subgoals. This property can be captured in the following definition.

Definition 2. (Shortest Path Cover) *Given a graph $G = (V, E, c)$ and a reachability relation R on G, a set of vertices $C \subseteq V$ is an R-Shortest-Path-Cover (R-SPC) of G if and only if, for all $s, t \in V$ such that $d(s, t) < \infty$ and $(s, t) \notin R$, there exists a $u \in C$ that covers at least one shortest s-t-path.*

Lemma 1. *Given an R-SPC C of $G = (V, E, c)$, for all $s, t \in V$ such that $d(s, t) < \infty$, the vertices in C split at least one shortest s-t-path into R-reachable segments.*

Proof. Let Π be a sequence of pairs of vertices, initialized to $((s, t))$. Let P be the procedure that operates on Π and performs the following operation until Π no longer contains a pair $(u, v) \notin R$: Pick a pair $(u, v) \notin R$ and replace it with the pairs $(u, p), (p, v)$ for some $p \in C$ that covers a shortest u-v-path. Such p must exist since C is an R-SPC of G.

P terminates because (1) $|V|$ is finite, (2) edge lengths are positive, and (3) $p \neq u$ and $p \neq v$ since p covers a shortest u-v-path.

After P terminates, all pairs (u, v) in Π satisfy $(u, v) \in R$ (otherwise, P would not have terminated). As P operates on Π, it maintains the invariant that the concatenation of all shortest u-v-paths of all pairs (u, v) in Π is a shortest s-t-path, since p covers a shortest u-v-path when P replaces a pair (u, v) with the pairs (u, p) and (p, v). □

Lemma 1 implies that, in order to find a shortest s-t-path using subgoal graphs, it is sufficient to have edges in the (query) subgoal graph corresponding to all R-reachable segments in a shortest s-t-path. This can be guaranteed by adding edges between all pairs of R-reachable subgoals in the preprocessing phase and, in the query phase, connecting s to every subgoal (and t) that is R-reachable from s and connecting t to every subgoal from which t is R-reachable. However, with this construction, the resulting (query) subgoal graph can have redundant edges. We now define the concept of direct-R-reachability, which avoids redundant edges.

Definition 3. (Direct-R-Reachability) *Given a graph $G = (V, E, c)$, a reachability relation R on G, and a subset $C \subseteq V$, a direct reachability relation on G is a relation $D_C(R) \subseteq V \times V$ that satisfies: $(s,t) \in D_C(R)$ if and only if (1) $(s,t) \in R$ and (2) there does not exist a $u \in C$ that covers a shortest s-t-path.*

For simplicity, we use "direct-R-reachable" instead of $D_C(R)$-reachable if C can be inferred from the context.

Lemma 2. *An R-SPC C of G is also a $D_C(R)$-SPC of G.*

Proof. Let $s, t \in V$ with $d(s,t) < \infty$ and $(s,t) \notin D_C(R)$. We show that there exists a vertex $u \in C$ that covers a shortest s-t path in G. If $(s,t) \notin R$, then u must exist since C is an R-SPC of G (Definition 2). Otherwise, u must exist because otherwise $(s,t) \in D_C(R)$ (Definition 3). □

We now formally define a subgoal graph and an s-t-query subgoal graph. In the definitions below, $R_s^{\rightarrow}$ denotes the set of vertices that are R-reachable from s, and $R_s^{\leftarrow}$ denotes the set of vertices from which s is R-reachable.

Definition 4. (Subgoal Graph) *Given a graph $G = (V, E, c)$, and a reachability relation R on G, a graph $S = (V_S, E_S, c_S)$ is a subgoal graph on G with respect to R if and only if: (1) V_S is an R-SPC on G and (2) for all $u, v \in V_S$ with $u \neq v$, there exists an edge $(u,v) \in E_S$ with $c_S(u,v) = d_G(u,v)$ if and only if v is direct-R-reachable from u.*

Definition 5. (s-t-Query Subgoal Graph) *Given a subgoal graph $S = (V_S, E_S, c_S)$ on G with respect to reachability relation R and start and goal vertices $s, t \in V$, the s-t-query subgoal graph $S_{s,t} = (V'_S, E'_S, c'_S)$ is defined as follows:*

- $V'_S = V_S \cup \{s, t\}$
- $E'_S = E_S \cup E_s^{\rightarrow} \cup E_t^{\leftarrow}$, *where:*
  - $E_s^{\rightarrow} = \{(s,u) : u \in (D_{V_S}(R)_s^{\rightarrow} \cap V'_S) \setminus \{s\}\}$
  - $E_t^{\leftarrow} = \{(u,t) : u \in (D_{V_S}(R)_t^{\leftarrow} \cap V'_S) \setminus \{t\}\}$
- $\forall (u,v) \in E'_S$, $c'_S(u,v) = d_G(u,v)$

Theorem 1. *Given a graph $G = (V, E, c)$, a reachability relation R on G, a subgoal graph $S = (V_S, E_S, c_S)$ on G with respect to R, and start and goal vertices $s, t \in V$, the s-t-distance on the s-t-query subgoal graph $S_{s,t} = (V'_S, E'_S, c'_S)$ is equal to the s-t distance on G.*

Proof. According to Definition 5, $c'_S(u,v) = d_G(u,v)$ for any edge $(u,v) \in E'_S$. Therefore, $d_{S_{s,t}}(s,t) \geq d_G(s,t)$. Consequently, $d_{S_{s,t}}(s,t) = \infty$ if $d_G(s,t) = \infty$. We now prove that $d_{S_{s,t}}(s,t) = d_G(s,t)$ if $d_G(s,t) < \infty$.

According to Definition 4, V_S is an R-SPC on G. According to Lemma 2, V_S is also a $D_{V_S}(R)$-SPC on G. Then, by Lemma 1, the vertices of V_S split at least one shortest s-t-path into $D_{V_S}(R)$-reachable segments. According to Definitions 4 and 5, $S_{s,t}$ contains edges between all $D_{V_S}(R)$-reachable pairs $u, v \neq u \in V'_S$, with length $d_G(u,v)$. Therefore, $d_{S_{s,t}}(s,t) = d_G(s,t)$. □

Algorithm 1 outlines how subgoal graphs can be used for finding shortest paths. There are three main phases, and the performance of each phase is affected by the choice of R:

Algorithm 1 Finding shortest paths using subgoal graphs.

1: **function FindPath**(G, S, s, t)
2: Construct the s-t-query subgoal graph (by identifying all subgoals that are R-reachable from s and all subgoals from which t is R-reachable)
3: $\Pi \leftarrow$ find a shortest s-t-path on the s-t-query subgoal graph
4: $\pi := ()$
5: **for all** pairs $(u_i, u_{i+1}) \in \Pi$, in increasing order of i **do**
6: $\quad \pi := append(\pi,$ R-Reachable-Path$(u_i, u_{i+1}))$
7: return π

(1) *Connect* (Line 2): Being able to quickly identify the subgoals that are R-reachable from a given start vertex (and the subgoals from which a given goal vertex is R-reachable) allows for the efficient construction of query subgoal graphs. (2) *Search* (Line 3): Since the set of subgoals is required to be an R-SPC, the choice of R affects the size of the resulting subgoal graph, which in turn affects the runtime for finding a high-level path on a query subgoal graph. (3) *Refine* (Lines 4-6): Since the high-level path contains only R-reachable segments (since, by definition, query subgoal graphs contain edges only between R-reachable vertices), being able to quickly find paths between vertices that are known to be R-reachable allows for the efficient refinement of the high-level path into a low-level path.

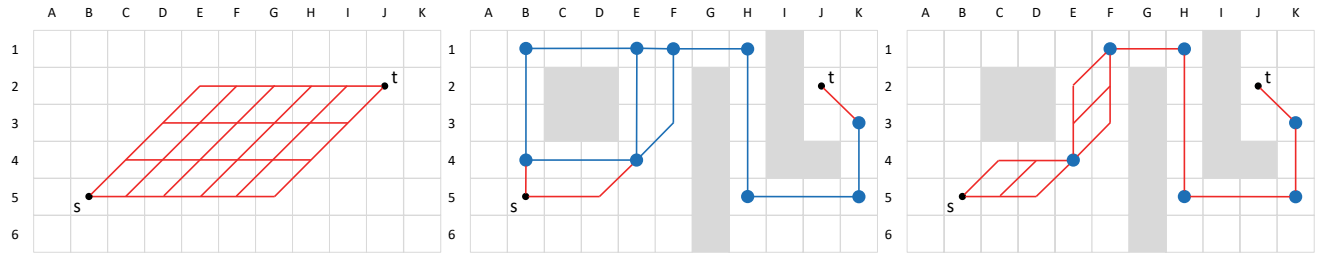
Subgoal Graphs on Grids and Freespace-Reachability

In this section, we review previous work on subgoal graphs, and introduce the concept of freespace-reachability.

Subgoal graphs on grids have been developed as an entry to the GPPC, which uses the following specification for grids: An agent can move from the center of an unblocked cell to the center of an unblocked neighboring cell in any cardinal (with cost 1) or diagonal direction (with cost $\sqrt{2}$) with the following exception: Diagonal moves are only possible if the neighboring cells in both associated cardinal directions are unblocked. For instance, in Figure 1 (b), the agent cannot move from D4 to E3 since D3 is blocked.

On grids, one can make the freespace assumption by assuming that the grid contains no blocked cells. Figure 1(a) shows all shortest s-t-paths under the freespace assumption (freespace-s-t-paths). Each path has the same number of diagonal moves and the same number cardinal moves, which can be computed in constant time from the relative positions of s and t. We use $fd(s,t)$ to denote the distance between two vertices under the freespace assumption (freespace distance). We say that two vertices s and t are **freespace-reachable** (previously called h-reachable) if and only if $fd(s,t) = d(s,t)$. Equivalently, two vertices s and t are freespace-reachable if and only if at least one of the shortest freespace-s-t-paths is unblocked. We say that s and t are **safe-freespace-reachable** if and only if all shortest freespace-s-t-paths are unblocked.

Subgoal graphs on grids use safe-freespace-reachability



(a) Freespace paths. (b) Query subgoal graph. (c) Subgoals splitting a shortest path into safe-freespace-reachable segments.

Figure 1: Subgoal graphs on grids.

as the reachability relation, which has the following advantages: (1) The subgoals can be identified quickly by placing them at the convex corners of all obstacles since the set of all convex corners of obstacles are a (safe-)freespace-reachability-SPC (Uras, Koenig, and Hernández 2013). Figure 1(b) shows a subgoal graph and Figure 1(c) shows subgoals splitting a shortest path into safe-freespace-reachable segments. (2) A high-level path found on the subgoal graph can quickly be refined into a low-level path on the grid since, given two vertices that are known to be safe-freespace-reachable, a shortest path between them can be computed by simply determining the number of cardinal and diagonal moves to get from one vertex to the other and executing them in any order. (3) Subgoals that are direct-freespace-reachable from any given vertex can be identified quickly by using clearance values (Uras, Koenig, and Hernández 2013). This operation can be used to identify the edges of a subgoal graph as well as connect any given start and goal vertices to the subgoal graph. (4) The freespace distance between two vertices can be computed in constant time as the Octile distance, allowing one to avoid storing edge costs, which can significantly reduce the memory requirement of storing the subgoal graph.

State Lattices

State lattices can be used to model agents with kinematic constraints. Vertices of a state lattice are defined by discretizing the environment into a grid, as well as discretizing other features of the agent, such as its orientation and velocity. Each vertex $u = (x_u, y_u, p_u)$ in a state lattice represents the agent's *location*, where a reference point on the agent coincides with the center of the grid cell (x_u, y_u), as well as the agent's *pose* p_u, which combines the remaining discretized features of the agent.

The edges of a state lattice are defined by a precomputed set of motion primitives that discretize the kinematically feasible motions of the agent. Each motion primitive m is defined by a tuple $(p^s_m, p^e_m, x_m, y_m, l_m, C_m)$, where p^s_m is the start pose of m, p^e_m is the end pose of m, x_m and y_m are the relative coordinates of the start and end cells of m, l_m is the length of m, and C_m is a set of cells, relative to the start cell, that need to be unblocked in order to apply m. For any two vertices $s = (x_s, y_s, p_s)$ and $e = (x_e, y_e, p_e)$ in the state lattice, m induces an edge (s, e) with length l_m if and only if: (1) $p_s = p^s_m$, $p_e = p^e_m$, (2) $x_s + x_m = x_e$, $y_s + y_m = y_e$, and (3) for all $(x, y) \in C_m$, the cell (x_s+x, y_s+y) is unblocked.

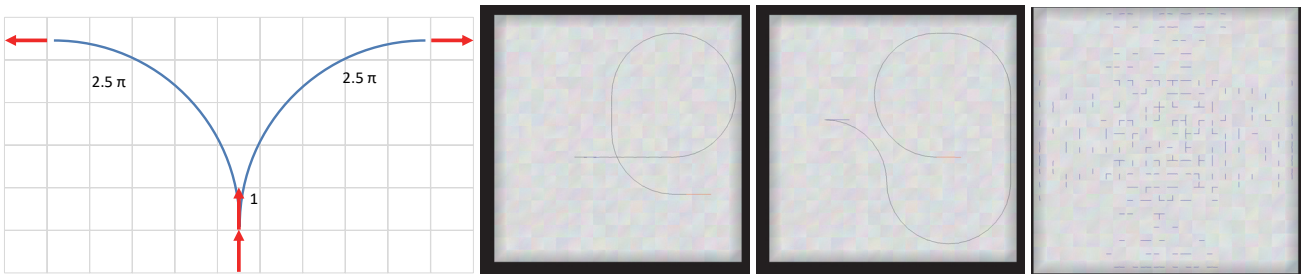
Figure 2(a) shows an (x, y, θ)-state lattice (where θ denotes the orientation of the agent) with four possible poses for the agent (facing North, East, South, and West). For each pose, the agent has three available motions: A straight motion that moves the agent one cell in the direction that it is facing without changing its pose, and two motions that follow quarter circles with radii of five cells, that change the agent's pose either clockwise or counterclockwise. We use this set of motion primitives as a running example throughout the paper as well as in our experiments.

Freespace-Reachability on State Lattices

Similar to grids, one can make the freespace assumption on state lattices by assuming that the underlying grid contains no blocked cells (and ignoring the boundaries of the environment), which allows us to define freespace-reachability on state lattices as follows: A vertex t is freespace-reachable from a vertex s if and only if the s-t-distance on the state lattice is equal to the freespace-s-t-distance. However, different from grids, it is not clear how to efficiently calculate freespace distances on state lattices since they can be defined with respect to an arbitrary set of motion primitives.

We therefore use precomputed freespace distances to implement freespace-reachability on state lattices. Let $s = (x_s, y_s, p_s)$ and $t = (x_t, y_t, p_t)$ be two vertices of the state lattice. Ignoring the boundaries of the environment, the freespace distance from s to t is equal to the freespace distance from the vertex $s' = (0, 0, p_s)$ to the vertex $t' = (x_t - x_s, y_t - y_s, p_t)$. Therefore, if we know the freespace distances from a vertex $(0, 0, p)$ to all other vertices, we can easily determine the freespace distance from any vertex (x, y, p) to any other vertex, for any value of x, y, and p. We exploit this translation invariance of freespace distances on state lattices to store the freespace distances between all pairs of vertices in a (finite) state lattice using memory that is only linear in the size of the underlying grid representation (and quadratic in the number of possible poses, as we discuss below).

Given a grid with $\mathcal{X}$ rows and $\mathcal{Y}$ columns and a set of motion primitives defined for an agent with $\mathcal{P}$ possible poses, we compute and store freespace distances between all pairs of vertices as follows: For each possible pose p, we run Di-



(a) A set of three motion primitives for an agent with four possible orientations. (b) A shortest freespace path. (c) A shortest path. (d) A freespace-reachability-SPC.

Figure 2: State lattices.

jkstra's algorithm from $(\mathcal{X}, \mathcal{Y}, p)$ on a freespace state lattice, where the underlying grid has dimensions $2\mathcal{X} \times 2\mathcal{Y}$ and no blocked cells, and store the distances to all other vertices. This way, we store $4\mathcal{X}\mathcal{Y}\mathcal{P}^2$ distances since this freespace state lattice has $2\mathcal{X} \times 2\mathcal{Y} \times \mathcal{P}$ vertices and we store distances from $(\mathcal{X}, \mathcal{Y}, p)$ to all other vertices. The freespace distance from any vertex $s = (x_s, y_s, p_s)$ to any other vertex $t = (x_t, y_t, p_t)$ on the original state lattice can then be determined by retrieving the stored distance from $(\mathcal{X}, \mathcal{Y}, p_s)$ to $(\mathcal{X} + x_t - x_s, \mathcal{Y} + y_t - y_s, p_t)$.

However, the memory requirements can still be prohibitive. For instance, a state lattice with 16 possible poses and a 512 $\times$ 512 grid requires around 1GB of memory to store all freespace distances (we use 4 bytes for each distance in our implementation). We therefore introduce **bounded-freespace-reachability**: A vertex t is bounded-freespace-reachable from a vertex s if and only if $d(s,t) = fd(s,t) \leq b$ for a given bound b. This allows us to store the freespace distances only up to the given bound, which requires only a constant amount of memory for a given bound b and a given set of motion primitives.

(Bounded-)safe-freespace-reachability can be implemented on state lattices in a similar way: For a pair of vertices (s,t), we store (in addition to the freespace-s-t-distance) the number of predecessors of t that cover a shortest freespace-s-t-path (we use 1 byte to store this information in our implementation). We discuss in the next section how these values, together with the freespace distances, can be used to implement efficient methods for constructing query subgoal graphs and refining high-level paths on query subgoal graphs into low-level paths on state lattices.

Subgoal Graphs on State Lattices

As discussed in the previous section, there are four variants of freespace-reachability that we can use to construct subgoal graphs on state lattices (regular, safe, and bounded variants for both). Rather than committing to a reachability relation, we experiment with all four variants. In the following subsections, we discuss three key methods for constructing and searching subgoal graphs and how these methods can be implemented for different variants of freespace-reachability. We focus on regular freespace-reachability, but also comment on how they can be extended to safe-freespace-reachability. The extensions to the bounded versions are straightforward.

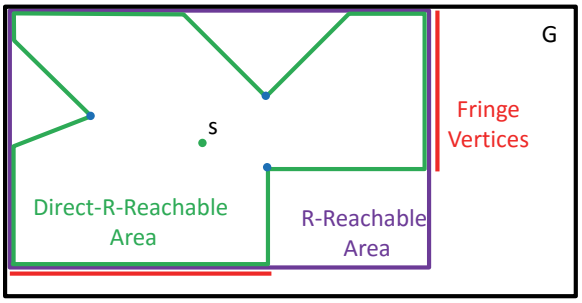


Figure 3: Illustration of the (direct-)R-reachable area around a vertex.

Identifying Subgoals

On grids, the set of convex corners of obstacles is a freespace-reachability-SPC, allowing one to quickly construct a subgoal graph. On state lattices, however, this subgoal placement strategy does not work. Consider the two paths shown in Figures 2(b) and 2(c) for an agent that moves according to the motion primitives shown in Figure 2(a). The path shown in Figure 2(b) is a freespace path, whereas the path shown in Figure 2(c) is not, even though the relative locations and poses of the start and goal vertices in both figures are the same. This example demonstrates that, even if the underlying grid has no blocked cells (and therefore has no convex corners), the set of subgoals cannot be empty since a subgoal is necessary to cover a shortest path between the start and goal vertices in Figure 2(c) (according to Definition 2). Figure 2(d) shows subgoals that satisfy the requirements of a freespace-reachability-SPC (lines extend from the center of a cell in the direction that the agent is facing). This example also demonstrates that walls (different from the boundaries of the environment) can also introduce subgoals, which might explain why subgoal graphs on state lattices are not as effective as they are on grids, as we later observe in the experimental results.

Remember that the set of subgoals has to be a freespace-reachability-SPC and thus contain vertices that cover at least

one shortest path between any pair of vertices that are not freespace-reachable. We construct such a set of subgoals C incrementally, by iterating over all vertices s and adding vertices to C that cover at least one shortest s-t-path for any t with $d(s,t) < \infty$ and $(s,t) \notin R$. As illustrated in Figure 3, the set of all freespace-reachable vertices from s form an area around s. Any shortest s-t-path that leaves this area has to contain a vertex that is freespace-reachable from s, immediately followed by a vertex that is not freespace-reachable from s. It is sufficient to cover at least one shortest path to any of these vertices (the ones that are not freespace-reachable from s) in order to cover at least one shortest path to all vertices that are not freespace-reachable from s. However, there might already be subgoals in C (shown as blue dots) that cover shortest paths to some of these vertices. We call the remaining vertices the *fringe* vertices, formally defined as follows: A vertex t is a fringe vertex of s (with respect to the current elements of C) if and only if (a) $(s,t) \notin R$ (where R is freespace-reachability), (b) no shortest s-t-path is covered by a vertex $u \in C$, and (c) there exists a shortest s-t-path $\pi = (v_0, \ldots, v_n)$ with $(s, v_{n-1}) \in R$.

To decide which vertices to add to C to cover shortest paths originating at vertex s, we first run a modified version of Dijkstra 's algorithm from s to identify the set of fringe vertices. We then greedily add vertices to C that cover shortest paths to fringe vertices. We omit the specifics of our vertex selection strategy (both due to space restrictions and because it is still work in progress). We basically assign a score to every vertex u that covers a shortest path from s to a fringe vertex, pick the vertex that maximizes this score, add it to C, update the set of fringe vertices, and repeat this procedure until the set of fringe vertices is empty. The score of a vertex u is based on the number of fringe vertices that can be eliminated when adding it to C as well as its distances to these fringe vertices. We have observed that the successors of s get the highest scores if we only use the number of fringe vertices eliminated.

Identifying Direct-Freespace-Reachable Subgoals from a Given Vertex

In this section, we describe how to connect given start and goal vertices to the subgoal graph to create a query subgoal graph. Algorithm 2 identifies a superset of direct-freespace-reachable subgoals from a given start vertex s. A modified version of Algorithm 2 can be used to connect the goal to the subgoal graph in a similar way.

A straightforward way of identifying exactly the set of freespace-reachable subgoals from a given start vertex s would be to run a variant of Dijkstra's algorithm from s that ignores any vertex u that is not freespace-reachable from s (which can be determined by checking if $g(u) \neq fd(s,u)$). This algorithm can be further modified to identify exactly the set of direct-freespace-reachable subgoals from s by maintaining $covered(u)$ values for every generated vertex u, which specify whether the search has found a shortest s-u-path that is covered by a subgoal. At the end of the search, the set of all expanded subgoals u with $covered(u) = false$ is the set of direct-freespace-reachable subgoals from s. The search can be terminated when all vertices in OPEN have

Algorithm 2 Identifying Direct-Freespace-Reachable Subgoals

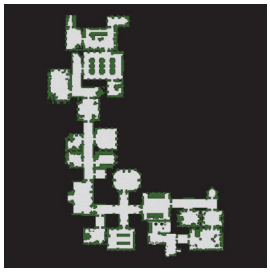
```
1:  function IdentifyDFRSubgoals(G, C, s)
2:  for all u ∈ V do
3:      g(u) := ∞, covered(u) := false
4:  g(s) := 0
5:  OPEN := {s} // Implemented as a FIFO-queue for BFS
6:  CLOSED := {}
7:  D := {} // direct-freespace-reachable subgoals
8:
9:  Invariant: ∀u ∈ OPEN, u is freespace-reachable from s
10: while OPEN contains a vertex u with ¬ covered(u) do
11:     u := any vertex in OPEN
12:     Move u from OPEN to CLOSED
13:     if u ∈ C \ {s} ∧ ¬ covered(u) then
14:         D := D ∪ {u}
15:     for all successors v of u such that v ∉ CLOSED do
16:         if g(u) + c(u,v) = fd(s,v) then
17:             g(v) := fd(s,v)
18:             if covered(u) ∨ u ∈ C \ {s} then
19:                 covered(v) := true
20:             if v ∉ OPEN then
21:                 Add v to OPEN
22: return D
```

$covered(u) = true$ since all new vertices v generated afterwards also have $covered(v) = true$ and thus are not direct-freespace-reachable from s.

We only need to determine a superset of all subgoals that are direct-freespace-reachable from s. Algorithm 2 uses all techniques discussed above, but also changes the search method from Dijkstra's algorithm to breadth-first search to avoid having to maintain a priority queue. The expansion order of Dijkstra's algorithm guarantees that $g(u) = d(s,u)$ when a vertex u is expanded. However, since we are only interested in expanding vertices u that are freespace-reachable from s (that is, for which $d(s,u) = fd(s,u)$), we can ensure that all vertices u placed in OPEN have $g(u) = d(s,u) = fd(s,u)$ (Line 16), which allows us to expand the vertices in any order while ensuring that $g(u) = d(s,u)$ when u is expanded.

The expansion order of Dijkstra's algorithm also guarantees that any vertex v that covers a shortest s-u-path is expanded before u, which in turn guarantees that the $covered(u)$ values are propagated correctly. By using breadth-first search instead of Dijkstra's algorithm, Algorithm 2 no longer has this guarantee and can fail to label some vertices as covered, therefore returning a superset of the direct-freespace-reachable subgoals from s.

A possible modification of Algorithm 2 is to avoid using $covered(u)$ values, terminate the search only when OPEN is empty, and not expand any subgoals (since their children are not direct-freespace-reachable). However, in our experiments, we have observed that maintaining $covered(u)$ values to terminate the search early is significantly more efficient.



(a) State lattice has 68,208 vertices and 135,310 edges.

	Subgoal Graph		Prep.	Memory	Avg. Query Time (ms)			Speed
	Vertices	Edges	Time (s)	(MB)	Connect	Search	Refine	up
FR-25	16,304	63,642	2.14	0.58	0.027	2.256	0.093	2.45
FR-50	8,461	75,477	8.02	0.98	0.117	1.204	0.081	4.15
FR-75	8,377	75,188	9.39	1.74	0.137	1.189	0.083	4.14
FR-100	8,378	75,609	9.26	2.80	0.150	1.242	0.087	3.94
FR-inf	8,427	74,796	9.89	19.61	0.138	1.164	0.084	4.20
SFR-25	16,645	64,478	2.08	0.62	0.013	2.235	0.092	2.49
SFR-50	11,054	72,196	5.25	1.15	0.032	1.583	0.093	3.41
SFR-75	10,986	71,996	5.96	2.09	0.030	1.436	0.086	3.75
SFR-100	10,916	73,257	5.58	3.43	0.031	1.370	0.079	3.94
SFR-inf	10,944	72,689	6.27	24.44	0.030	1.387	0.088	3.87

(b) Preprocessing time, memory consumption, and query time.

Figure 4: Results on den005d.

We can also modify Algorithm 2 to identify direct-safe-freespace-reachable subgoals from s by keeping track of the number of direct-safe-freespace-reachable predecessors for every generated vertex and only adding a vertex to OPEN if (1) it is not a subgoal and (2) both its distance from s and its number of direct-safe-freespace-reachable predecessors match their respective values in the freespace. This variant does not need to maintain $covered(u)$ values, terminates when OPEN is empty, and guarantees to never expand a vertex u that is not direct-safe-freespace-reachable from s (proof omitted).

Finding Freespace-Reachable Paths

In this section, we describe how to find a shortest path from a vertex s to a vertex t that is known to be freespace-reachable from s. The efficiency of this operation determines the time needed for refining a high-level path on a query subgoal graph into a low-level path on the state lattice.

Our proposed algorithm (pseudocode omitted) is a depth-first search that only generates a successor v of an expanded vertex u if $fd(s,u) + c(u,v) + fd(v,t) = fd(s,t)$. This rule effectively prunes any vertex u with $fd(s,u) + fd(u,t) \neq fd(s,t)$ (which is justified since then u cannot cover a shortest freespace-s-t-path) and guarantees that any expanded vertex u satisfies $g(u) = fd(s,u)$ (similar to Algorithm 2). The expansion order of depth-first search also allows the search to find a path quickly. This algorithm can also be used unchanged to find shortest paths between safe-freespace-reachable vertices. Since safe-freespace-reachability guarantees that all shortest freespace-s-t-paths are unblocked on the state lattice, the depth-first search is guaranteed to find a shortest path without backtracking.

Experimental Results

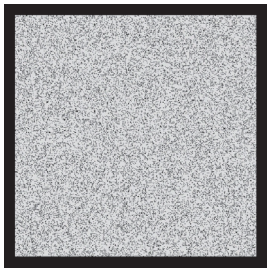
We provide preliminary results on our application of subgoal graphs to state lattices, on three different maps (Figures 4, 5, and 6) from Nathan Sturtevant's benchmarks (Sturtevant 2012a). For each map, we first construct a state lattice using the motion primitives shown in Figure 2(a) and then construct subgoal graphs with respect to either bounded-freespace-reachability (FR) or bounded-safe-freespace-reachability (SFR) with bounds of 25, 50, 75, 100, and infinity (that is, (safe-)freespace-reachability with no bound). The subgoal graphs are named by combining the reachability relation and the bound. For instance, FR-25 denotes a subgoal graph constructed with respect to bounded-freespace reachability with bound 25. For each map, we generate 1,000 random instances by selecting random start and goal states. We use Dijkstra's algorithm (implemented with a binary heap as priority queue) for searching both query subgoal graphs and state lattices. The experiments were run on a PC with a 2.6GHz Intel i7-4720HQ CPU and 16GB of RAM.

For each map and each subgoal graph constructed for the map, we report the number of vertices and edges of the subgoal graph, the memory required to store the subgoal graph (including memory required to store freespace distances and, for SFR, the number of direct-safe-freespace-reachable predecessors), preprocessing time, query time (divided into the time for connecting the start and goal vertices to the subgoal graph, searching the query subgoal graph, and refining the high-level path on the query subgoal graph into a low-level path on the state lattice), and speed-up achieved over running Dijkstra's algorithm on the state lattice.

The results show that search with subgoal graphs can speed up path planning on state lattices by roughly $\sim$2-4 times. This speed up is not as impressive as the speed up gained by using subgoal graphs on grids (where it can be, for example, $\sim$25 times on game maps), which can be attributed to the sizes of the subgoal graphs relative to the size of the state lattice. For instance, $\sim$12.2% of all vertices of den005d are subgoals (in the best case, using FR-75). $\sim$19.3% of all vertices are subgoals for 64room-000 (in the best case, using FR-100), and the subgoal graph has $\sim$4.5 times as many edges as the state lattice. This can be explained by our previous observation that even the walls of a room can introduce subgoals.

The query time is dominated by the time to search the query subgoal graph, whereas the connect and refine times are relatively small in comparison. This suggests that speeding up searches on subgoal graphs by a given factor (such as by using A* searches with a good heuristic or other techniques which we outline as part of our future work) could improve the overall query time by almost the same factor.

In general, SFR subgoal graphs have more subgoals than FR subgoal graphs (for the same bound), and using

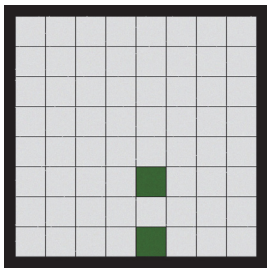


(a) State lattice has 914,512 vertices and 1,633,280 edges.

	Subgoal Graph		Prep.	Memory	Avg. Query Time (ms)			Speed
	Vertices	Edges	Time (s)	(MB)	Connect	Search	Refine	up
FR-25	231,965	857,801	23.44	6.07	0.017	58.085	0.105	3.27
FR-50	178,605	978,677	119.07	6.38	0.043	53.932	0.110	3.52
FR-75	177,665	982,573	193.73	7.14	0.043	55.766	0.109	3.41
FR-100	177,868	983,003	246.44	8.20	0.043	55.948	0.108	3.40
FR-inf	178,319	983,459	433.41	69.67	0.045	59.470	0.106	3.20
SFR-25	279,275	864,472	21.78	6.68	0.009	84.345	0.111	2.26
SFR-50	269,600	874,475	56.06	7.17	0.010	78.105	0.106	2.44
SFR-75	269,984	874,592	65.10	8.12	0.009	76.770	0.101	2.48
SFR-100	270,012	875,360	68.39	9.45	0.010	77.995	0.100	2.44
SFR-inf	270,346	874,413	69.91	86.27	0.010	88.000	0.109	2.16

(b) Preprocessing time, memory consumption, and query time.

Figure 5: Results on random512-10-0.



(a) State lattice has 984,512 vertices and 2,640,088 edges.

	Subgoal Graph		Prep.	Memory	Avg. Query Time (ms)			Speed
	Vertices	Edges	Time (s)	(MB)	Connect	Search	Refine	up
FR-25	496,856	3,361,203	153.75	18.65	0.022	134.086	0.144	1.62
FR-50	286,161	8,603,245	1692.02	36.69	0.599	98.866	0.140	2.18
FR-75	213,856	11,937,807	4998.03	49.34	2.590	94.418	0.192	2.24
FR-100	190,218	11,337,573	5009.79	47.84	2.921	77.960	0.209	2.68
FR-inf	190,371	11,355,569	5235.48	109.37	3.624	90.567	0.234	2.30
SFR-25	497,154	3,385,113	140.03	18.79	0.013	134.548	0.136	1.61
SFR-50	300,484	8,785,303	1593.41	37.70	0.071	112.922	0.126	1.92
SFR-75	225,687	13,785,215	4538.78	56.86	0.702	109.575	0.141	1.97
SFR-100	215,867	14,039,801	4540.00	59.05	0.770	97.855	0.142	2.20
SFR-inf	215,926	14,039,151	4459.24	135.87	0.779	102.561	0.138	2.10

(b) Preprocessing time, memory consumption, and query time.

Figure 6: Results on 64room-000.

higher bounds results in fewer subgoals. This can be explained as follows: Safe-freespace-reachability is a *stronger* reachability relation than freespace-reachability, in the sense that any pair of vertices that are safe-freespace-reachable are also freespace-reachable (but not vice versa). In the same sense, bounded-safe-freespace-reachability becomes stronger as the bound gets smaller. A stronger reachability relation implies that there are more pairs of vertices that do not satisfy the reachability relation, which requires the introduction of more subgoals in order to cover the shortest paths between them.

Conclusions and Future Work

We have introduced a general framework for subgoal graphs that can be specialized through the choice of different reachability relations, and applied this framework to state lattices by using variants of freespace-reachability as the reachability relation.

Although our preliminary experimental results demonstrate a small speed-up, we think that it is possible to achieve larger speed-ups through better subgoal graph construction strategies, which we consider as future work. For example, we plan to investigate a less greedy construction strategy that performs multiple freespace-reachable area explorations at the same time and identifies subgoals in a more informed way. We also plan to investigate a different construction strategy through vertex contractions, that initializes the subgoal graph to the original graph and then removes subgoals while maintaining the invariant that the set of subgoals is a freespace-reachability-SPC.

Subgoal graphs could also be used as a base graph for more involved preprocessing strategies. We plan to investigate constructing contraction hierarchies on subgoal graphs (as opposed to N-level subgoal graphs, which is the subgoal graph entry in GPPC), where we allow the addition of extra edges only between freespace-reachable subgoals to avoid having to store edge lengths and refinement information. Finally, given that only a small percentage of vertices of the original graph become subgoals (especially on grids), it might be possible to develop a memory-friendly version of hub-labeling that only stores labels for the subgoals.[1]

Acknowledgments

The research at USC was supported by NSF under grant numbers 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.; Fiat, A.; Goldberg, A. V.; and Werneck, R. F. 2010. Highway dimension, shortest paths, and provably ef-

[1]We thank Ben Strasser for his suggestion, as well as helpful discussions over the years.

ficient algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 782–793.

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.

Antsfeld, L.; Harabor, D. D.; Kilby, P.; and Walsh, T. 2012. Transit routing on video game maps. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2–7.

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 Matijevic, D. 2006. Transit ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge – Shortest Paths*.

Bauer, R., and Delling, D. 2009. Sharc: Fast and robust unidirectional routing. *Journal of Experimental Algorithmics* 14:4.

Demetrescu, C.; Goldberg, A.; and Johnson, D. 2006. 9th dimacs implementation challenge–shortest paths.

Dibbelt, J.; Strasser, B.; and Wagner, D. 2014. Customizable contraction hierarchies. In *Proceedings of the International Symposium on Experimental Algorithms*, 271–282.

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.

Goldberg, A. V.; Kaplan, H.; and Werneck, R. F. 2006. Reach for a*: Efficient point-to-point shortest path algorithms. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 129–143.

Goldberg, A. V.; Kaplan, H.; and Werneck, R. F. 2009. Reach for a*: Shortest path algorithms with preprocessing. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge* 74:93–139.

Gutman, R. J. 2004. 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*, 100–111.

Hilger, M.; Köhler, E.; Möhring, R. H.; and Schilling, H. 2009. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge* 74:41–72.

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.

Lauther, U. 2004. 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.

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.

Sanders, P., and Schultes, D. 2005. Highway hierarchies hasten exact shortest path queries. In *Proceedings of the European Symposium on Algorithms*, 568–579.

Sanders, P., and Schultes, D. 2006. Engineering highway hierarchies. In *Proceedings of the European Symposium on Algorithms*, 804–816.

Storandt, S. 2013. Contraction hierarchies on grid graphs. In *Proceedings of the German Conference on Artificial Intelligence*, 236–247.

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

Sturtevant, N. 2012b. Grid-based path-planning competition.

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.; 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*.