Subgoal Graphs for Optimal Pathfinding in Eight-Neighbor Grids *

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

Carlos Hernández
Depto. de Ingeniería Informática
Univ. Católica de la Ssma. Concepción
Concepción, Chile
chernan@ucsc.cl

Abstract

Grids are often used to represent maps in video games. In this paper, we propose a method for preprocessing eight-neighbor grids to generate subgoal graphs and show how subgoal graphs can be used to find shortest paths fast. We place subgoals at the corners of obstacles (similar to visibility graphs) and add those edges between subgoals that are necessary for finding shortest paths, while ensuring that each edge connects only subgoals that are easily reachable from one another. We describe a method for finding shortest paths by first finding high-level paths through subgoals and then shortest low-level paths between consecutive subgoals on the high-level path. Our method was one of ten entries in the Grid-Based Path Planning Competition 2012. Among all optimal path planners, ours was the fastest to find complete paths and required the least amount of memory.

Introduction

Grids with obstacles consisting of blocked cells are often used to represent maps in video games, and pathfinding on these maps usually needs to be fast. One can often preprocess maps before a game is released or while a map is loaded into memory, to speed up path planning on these maps. The data produced by preprocessing should use only a small amount of memory, and, in case the maps are generated during runtime, preprocessing should be fast.

In this paper, we propose a method for preprocessing eight-neighbor grids that lets us perform fast searches to find shortest paths. During preprocessing, we generate a subgoal graph that is similar to visibility graphs used for discretizing continuous environments (Lozano-Pérez and Wesley 1979). We place subgoals at the corners of obstacles and add those edges between subgoals that are necessary for finding shortest paths, while ensuring that each edge connects only subgoals that are easily reachable from one another. During search, we connect the given start and goal cells to the subgoal graph and find a shortest high-level path between them on the modified subgoal graph. We then determine a shortest low-level path by finding shortest paths between consecutive subgoals on the high-level path. Preprocessing can be done fast, and the memory requirements are fairly low. For example, on maps from the game StarCraft, which are the largest game maps in our benchmark with sizes up to 1024×1024, preprocessing takes less than 50 milliseconds, the amount of stored data is around 2.35 megabytes (including the maps), and shortest paths are found about 25 times faster than with A* on average. We also introduce a more sophisticated version of subgoal graphs where the subgoals are partitioned into *local* and *global* subgoals. During search, local subgoals that are not needed to connect the start and goal cells to the subgoal graph are ignored, resulting in an even smaller graph to search. The preprocessing of StarCraft maps now takes 6.7 seconds, the amount of stored data decreases to 2.15 megabytes, and shortest paths are found about 85 times faster than with A* on average. We can also use an additional 5.5 megabytes and 20 seconds during preprocessing to compute the pairwise distances between global subgoals to be around 165 times faster than with A* on average.

Methods for preprocessing grids have been studied before and can be grouped into several categories. Hierarchical abstractions (Botea, Müller, and Schaeffer 2004; Sturtevant and Buro 2005) reduce the size of the search space. They enable path planning to find high-level paths over abstract nodes and then refine them to low-level paths on the grid, which are not always optimal. More informed heuristics (Björnsson and Halldórsson 2006; Cazenave 2006; Sturtevant et al. 2009) guide the searches better to expand fewer states. Dead-end detection and other pruning methods (Björnsson and Halldórsson 2006; Goldenberg et al. 2007; Pochter et al. 2010) identify areas on the grid that do not need to be explored to find a shortest path. Another approach is to use subgoaling to reduce search to backtrack-free hill climbing (Bulitko, Björnsson, and Lawrence 2010; Hernández and Baier 2011). Some recent methods do not fit these categories. Compressed Path Databases (CPD) (Botea 2011) calculate and compress shortest paths between all

*This paper is based upon research supported by NSF (while Sven Koenig was serving at NSF). It is also based upon research supported by ARL/ARO under contract/grant number W911NF-08-1-0468 and ONR in form of a MURI under contract/grant number N00014-09-1-1031. The research was performed while Carlos Hernández visited the University of Southern California. 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.

pairs of cells during preprocessing and therefore avoid searching altogether during runtime. However, their memory requirements can be high even with high compression factors. Symmetry reducing methods (Harabor and Grastien 2010) identify and eliminate path symmetries on grids. For example, Jump Point Search (JPS) (Harabor and Grastien 2011) performs symmetry reduction during runtime (without any preprocessing) to find shortest paths faster than Hierarchical Pathfinding A* (Botea, Müller, and Schaeffer 2004).

Our method is a hybrid between hierarchical abstractions and symmetry reducing methods: It is hierarchical because we first find a high-level path and then refine it to a low-level path. It is symmetry reducing because an edge of the subgoal graph can correspond to many shortest paths between the subgoals that it connects (which cannot happen on visibility graphs because the straight line between vertices is the only shortest path between them). Our method does not abstract groups of cells, but rather all shortest paths between cells, which is similar to what JPS does. The jump points of JPS are similar to our subgoals, except that the agent always moves on a straight line from one jump point to the next one and thus moves in only one direction whereas the agent can move from one subgoal to the next one in a combination of two directions.

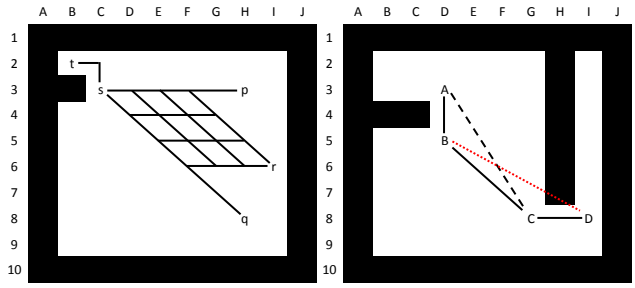
Our method was one of ten entries in the Grid-Based Path Planning Competition 2012 (GPPC 2012). Other entries included CPD and JPS as optimal path-planning methods, as well as several other optimal and suboptimal path-planning methods. On game maps and mazes, among all path planners, ours was the fastest to find complete paths and required the least amount of memory, except for one suboptimal entry which found paths that are around 50 percent longer than shortest paths. On the other maps, among all optimal path planners, ours was the fastest to find complete paths and required the least amount of memory.

Preliminaries and Notation

We use C to denote the set of cardinal directions (North, East, South, West) and D to denote the set of diagonal directions (NorthEast, SouthEast, SouthWest, NorthWest). We assume that an agent operates on an eight-neighbor grid with obstacles consisting of blocked cells. The agent moves from grid center to grid center and can move to an unblocked cell in any cardinal or diagonal direction, with one exception: It can move diagonally only if both associated cardinal directions are also unblocked since we assume that it has the same diameter as a grid cell. For example, in Figure 1(a), the agent cannot move diagonally from C3 to B2 because B3 is blocked. The respective lengths of cardinal and diagonal moves are 1 and $\sqrt{2}$.

We distinguish between paths and trajectories as follows: A trajectory between two cells s and s' is a series of moves on the grid that would take an agent from s to s' if all obstacles were removed. A path is a trajectory that takes an agent from s to s' on the actual grid.

We use the octile distance as heuristic $h(s, s')$, that estimates the distance between cells s and s'. The octile distance between s and s' is the length of a shortest trajectory between the cells. Let dx and dy denote the respective distances between s and s' along the x and y axes. Then, the octile distance between s and s' is $\sqrt{2} \times min(dx, dy) + |dx - dy|$ because a shortest trajectory between s and s' has exactly $min(dx, dy)$ diagonal and $|dx - dy|$ cardinal moves, for a total of $max(dx, dy)$ moves.



(a) Shortest paths between s and other cells (b) Subgoals and connections between them

Figure 1: Eight-neighbor grids and subgoals

We introduce the following notation to formalize such reasoning. Given a cell s, a direction d and an integer k, we use $s + kd$ to denote the cell s' that is reached from s on a trajectory with k moves in direction d. Similarly, we use $s+kd+mc = s'+mc$ to denote the cell that is reached from s' on a trajectory with m moves in direction c. Therefore, we also use $d = c_1 + c_2$ to denote that adding two perpendicular cardinal directions c_1 and c_2 results in the corresponding diagonal direction d (for example, North + East = NorthEast).

Key Idea

Instead of finding a shortest path from a given start cell to a given goal cell, we identify a sequence of cells, called subgoals, such that an agent moves on a shortest path from the given start cell to the given goal cell if it always moves on a shortest path to the next subgoal. The idea is to split the overall path-planning problem into several easier ones that can be solved simply by moving in the direction of the next subgoal. We generate a subgoal graph from a given grid during preprocessing to be able to identify suitable sequences of subgoals during search. Assume, for example, that an agent needs to move on a shortest path from B5 to I5 on the grid from Figure 1(b). A suitable sequence of subgoals then is G8, I8 and I5.

Simple Subgoal Graphs

In this section, we describe *simple subgoal graphs*, where we put subgoals at the corners of obstacles because they can then be used to circumnavigate obstacles. We add edges between subgoals that are *direct-h-reachable* from one another, resulting in a simple subgoal graph. These edges are sufficient for finding shortest paths and easy to follow. To find a shortest path between given start and goal cells, we connect them to their respective direct-h-reachable subgoals and then find a shortest (high-level) path between the start and goal cells on the modified subgoal graph. We then find the shortest (low-level) path between the start and goal cells on the grid by first finding shortest paths on the grid between

Algorithm 1 Constructing simple subgoal graphs

```
1: procedure ConstructSubgoalGraph()
2: V_S := E_S := ∅;
3: for all unblocked cells s do
4:     for all perpendicular cardinal directions c_1 and c_2 do
5:         if s + c_1 + c_2 is blocked then
6:             if s + c_1 and s + c_2 are unblocked then
7:                 V_S := V_S ∪ {s};
8: for all s ∈ V_S do
9:     S ←GetDirectHReachable(s);
10:    for all s' ∈ S do
11:        E_S := E_S ∪ {(s, s')};
12: G_S := (V_S, E_S);
```

consecutive subgoals on the path and then appending them. We now formally define simple subgoal graphs:

Definition 1. *An unblocked cell s is a **subgoal** iff there are two perpendicular cardinal directions c_1 and c_2 such that $s + c_1 + c_2$ is blocked and $s + c_1$ and $s + c_2$ are not blocked.*

Definition 2. *Two cells s and s' are **h-reachable** iff there is a path of length $h(s, s')$ between them. Two h-reachable cells are **safe-h-reachable** iff all shortest trajectories between them are also paths. Two h-reachable cells s and s' are **direct-h-reachable** iff none of the shortest paths between them contains a subgoal $s'' \notin \{s, s'\}$.*

Definition 3. *A **simple subgoal graph** $G_S = (V_S, E_S)$ is an undirected graph where V_S is the set of subgoals and E_S is the set of edges connecting direct-h-reachable subgoals. The lengths of the edges are the octile distances between subgoals they connect.*

Figure 1(b) shows a grid with subgoals and the connections between them. A, B, C and D are the subgoals. A-B, B-C and C-D are direct-h-reachable. A-C is safe-h-reachable but not direct-h-reachable since there is a shortest path between A and C through B. B-D is h-reachable but not safe-h-reachable since the path D5-E5-F5-G6-H7-I8 is blocked. A-D is not even h-reachable. Any two direct-h-reachable cells s and s' are also safe-h-reachable, for the following reason: If s and s' were not safe-h-reachable, then there would exist an obstacle that blocks at least one shortest trajectory between them. This obstacle would either block all shortest trajectories between them (and they would not be direct-h-reachable) or would introduce a subgoal that lies on one of the shortest paths between them, contradicting the definition of direct-h-reachability.

Algorithm 1 shows how to construct simple subgoal graphs. GetDirectHReachable(s) (explained in the next section) returns the set of subgoals that are direct-h-reachable from cell s. Algorithm 2 shows how to find a shortest path between a given start cell s and a given goal cell s'. We first call TryDirectPath(s, s') to try to find a shortest path between s and s' fast by arbitrarily choosing a shortest trajectory between s and s' and checking whether it is also a path (Lines 14-16). If it is, then we have found a shortest path between them. Otherwise, we conclude that s and s' are not safe-h-reachable and thus also not direct-h-reachable. (Verifying that s and s' are not direct-h-reachable is important, as we will explain later.) We then call FindAbstractPath(s, s')

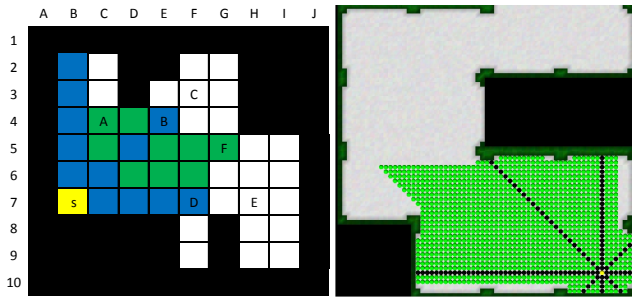
Algorithm 2 Searching simple subgoal graphs

```
1: procedure ConnectToGraph(cell s)
2: if s ∉ V_S then
3:     V_S := V_S ∪ {s};
4:     S ← GetDirectHReachable(s);
5:     for all s' ∈ S do
6:         E_S := E_S ∪ {(s, s')};
7: function FindAbstractPath(cells s, s')
8: ConnectToGraph(s);
9: ConnectToGraph(s');
10: Π ← find a shortest path from s to s' over the modified graph;
11: restore original graph;
12: return Π;
13: function FindPath(cells s, s')
14: π ← TryDirectPath(s, s');
15: if π ≠ nopath then
16:     return π;
17: Π ← FindAbstractPath(s,s');
18: if Π = nopath then
19:     return nopath;
20: π := emptypath;
21: for all segments (s_i, s_{i+1}) in Π, in increasing order of i do
22:     π := append(π,FindHReachablePath(s_i, s_{i+1}));
23: return π;
```

to connect s and s' to the simple subgoal graph by adding edges between them and all of their respective direct-h-reachable subgoals (Lines 8-9), use an A* search (Hart, Nilsson, and Raphael 1968) to find a shortest path between s and s' on the modified subgoal graph and return this high-level path. We then call FindHReachablePath(s_i, s_{i+1}) for each segment (s_i, s_{i+1}) of the high-level path to compute the corresponding low-level path. This function (whose pseudocode is omitted) performs a depth-first search from s_i to s_{i+1} that considers only moves in the two directions needed to find a path of length $h(s_i, s_{i+1})$. Let dx and dy denote the respective distances between s_i and s_{i+1} along the x and y axes. Then, the runtime complexity of FindHReachablePath(s_i, s_{i+1}) is $O(dx \times dy)$ since at most $dx \times dy$ cells lie on shortest trajectories between s_i and s_{i+1} (see, for example, cells C3 and I6 in Figure 1(a)) and depth-first search expands every such cell at most once. If s_i and s_{i+1} are safe-h-reachable (which is always the case in simple subgoal graphs), the runtime complexity is $O(max(dx, dy))$ since depth-first search finds a path without backtracking and thus expands at most $max(dx, dy)$ nodes.

Theorem 1. *FindPath(s,s') from Algorithm 2 finds a shortest path between cells s and s' on the grid.*

Proof. We first show that there exists a shortest path between any reachable cells s and s' that can be divided into segments between subgoals (plus s and s'), where each segment connects h-reachable cells. Consider any shortest path between s and s'. If the length of the path is $h(s, s')$, then s and s' are h-reachable and the path has the required property. Otherwise, every shortest trajectory between s and s' is blocked by at least one obstacle. Thus, every shortest path between s and s' needs to circumnavigate at least one obstacle and therefore contains some subgoal s''. The shortest path is then divided into two segments, a shortest path between cells s and s''



(a) Small example (b) Large example

Figure 2: Direct-h-reachable area

and a shortest path between cells s'' and s', and the procedure is recursively applied to both segments until all segments connect h-reachable cells.

We now show that there exists a shortest path between any h-reachable cells s and s' that can be divided into segments between subgoals (plus s and s'), where each segment connects direct-h-reachable cells. Consider any shortest path between cells s and s'. If they are also direct-h-reachable, then the path has the required property. Otherwise, there exists a shortest path between them that contains some subgoal s'' according to the definition of direct-h-reachable. This path is then divided into two segments, a shortest path between h-reachable cells s and s'' and a shortest path between h-reachable cells s'' and s', and the procedure is recursively applied to both segments until all segments connect direct-h-reachable cells.

Combining the two steps shows that there exists a shortest path between any two reachable cells that can be divided into segments between subgoals (plus s and s'), where each segment connects direct-h-reachable subgoals. The modified subgoal graph on Line 10 of Algorithm 2 contains all subgoals (plus s and s') and all direct-h-reachable edges between them, except possibly for an edge between s and s' if neither s nor s' are subgoals (because ConnectToGraph only adds edges to direct-h-reachable subgoals). Therefore, Line 10 finds a shortest high-level path between s and s' unless s and s' are direct-h-reachable but neither of them is a subgoal. However, TryDirectPath(s,s') on Line 14 would already have found a shortest path between them if they were direct-h-reachable, which proves the theorem. □

Finding Direct-H-Reachable Subgoals

An important part of constructing and searching simple subgoal graphs is identifying all direct-h-reachable subgoals from a given unblocked cell s. This is done by exploring the direct-h-reachable area around s, that contains all cells that are direct-h-reachable from s. The exploration can be sped up by precomputing clearance values for cells in each direction, which are their distances to obstacles or subgoals. GetDirectHReachable(s) from Algorithm 3 uses these clearance values to find direct-h-reachable subgoals fast from any given unblocked cell s. We use Figure 2(a) (with subgoals A to F) as running example to explain the method. Figure 2(b) shows part of a larger example to get a better picture of how the direct-h-reachable area looks like. In both figures, the direct-h-reachable area is given by the colored cells.

We first describe the idea behind clearance values. Clearance(s, d) computes how many moves an agent can make from cell s in direction d until it either reaches a subgoal or can no longer move due to a blocked cell.

Algorithm 3 Finding direct-h-reachable subgoals

```
1:  function Clearance(cell s, direction d)
2:  i := 0
3:  while true do
4:      if movement not possible from s + id to s + id + d then
5:          return i;
6:      i := i + 1;
7:      if IsSubgoal(s + id) then
8:          return i;
9:  function GetDirectHReachable(cell s)
10: S := ∅;
11: for all directions d do
12:     if IsSubgoal(s+Clearance(s, d) × d) then
13:         S := S ∪ {s+Clearance(s, d) × d};
14: for all diagonal directions d do
15:     for all cardinal directions c associated with d do
16:         max ←Clearance(s, c);
17:         diag ←Clearance(s, d);
18:         if IsSubgoal(s + max × c) then
19:             max := max − 1;
20:         if IsSubgoal(s + diag × d) then
21:             diag := diag − 1;
22:         for i = 1 . . . diag do
23:             j :=Clearance(s + id, c);
24:             if j ≤ max and IsSubgoal(s + id + jc) then
25:                 S := S ∪ {s + id + jc};
26:                 j := j − 1;
27:             if j < max then
28:                 max := j;
29: return S;
```

s+Clearance(s, d) × d is always an unblocked cell according to the definition of the clearance values. For example, in Figure 2(a), Clearance$(s, North) = 5$ and Clearance$(s, East) = 4$. The clearance values can be precomputed and cached since they do not depend on the locations of the start and goal cells.

GetDirectHReachable(s) works in two phases. The first phase (Lines 11-13) uses the clearance values of cell s in each direction to find direct-h-reachable subgoals that can be reached via moves in only a single direction. For example, when the function checks the North direction of cell s in Figure 2(a), it checks whether cell s+Clearance$(s, North) \times North = s + 5North = B2$ contains a subgoal. Since it does not, no subgoal is added to the set S of direct-h-reachable subgoals. When the function checks the North-East direction, it finds that E4 contains B and adds it to S. Any subgoal found this way is direct-h-reachable from s because there is only a single shortest trajectory to any cell on the same horizontal, vertical or diagonal line as s, and this shortest trajectory is also a path that does not contain subgoals according to the definition of the clearance values. In Figure 2(a), D is also direct-h-reachable from s but C and E are not since B and D are subgoals on shortest paths between s and C and E, respectively.

The second phase (Lines 14-28) finds direct-h-reachable subgoals that can be reached via moves in two directions. In Figure 2(b), the yellow dot represents s. The black lines are explored in the first phase while the green areas are explored in the second phase. There are eight areas to explore,

one for each of the combination of a cardinal and a diagonal direction (for example, the North-NorthEast area).

The exploration uses the following key observations: First, for cells s and $s' = s + id + jc$ with $i, j > 0$ to be direct-h-reachable, both $s' - c = s + id + (j-1)c$ and $s' - d = s + (i-1)d + jc$ must be direct-h-reachable from s (because, if a shortest trajectory between s and $s' - c$ (or $s' - d$) were not a path or contained a subgoal, then one could extend this trajectory to a shortest trajectory between s and s' that would not be a path or contained a subgoal, which contradicts the definition of direct-h-reachability). Second, for s and s' to be direct-h-reachable, neither $s' - c$ nor $s' - d$ can contain a subgoal (because otherwise there would exist a shortest path between s and s' through a subgoal, which contradicts the definition of direct-h-reachability).

We now explain Lines 16-28 in the context of exploring the East-NorthEast area around s = B7 in Figure 2(a), which is done by sweeping it with horizontal lines from West to East starting with cells on the NorthEast diagonal line. Variable max contains an upper bound on the length of the horizontal line swept currently, which is usually equal to the length of the horizontal line swept last (due to the first observation, namely that s and a cell s' are direct-h-reachable only if s and $s' - d$ are also direct-h-reachable). In case the horizontal line swept last ended with a subgoal, variable max is decremented by one (due to the second observation, namely that s and a cell s' are direct-h-reachable only if $s' - d$ does not contain a subgoal). Variable max is set to Clearance$(s, East) = 4$ (Line 16) and then decremented to 3 since F7 contains a subgoal (Lines 18-19). Variable $diag$ contains an upper bound on the length of the diagonal line and thus on the number of horizontal lines to sweep. It is set to Clearance$(s, NorthEast) = 3$ (Line 17) and then decremented to 2 since E4 contains a subgoal (Lines 20-21). We sweep the first horizontal line, starting with C6. The clearance value of C6 in the East direction is 6, which is larger than variable $max = 3$ (Line 24). Thus, three cells to the East of C6 are direct-h-reachable from s, namely D6, E6 and F6. (G6 is not direct-h-reachable from s since there exists a shortest path between s and G6 through D.) We then sweep the second horizontal line, starting with D5. The clearance value of D5 in the East direction is 3, which is equal to variable $max = 3$. Thus, three cells to the East of D5 are direct-h-reachable from s, namely E5, F5 and G6.. G5 contains F. Therefore, F is added to the set S of direct-h-reachable subgoals and variable max is decremented to 2 (Lines 23-28). We do not sweep additional horizontal lines because the limit given by variable $diag = 2$ has been reached.

By precomputing the clearance values, the runtime of identifying all direct-h-reachable subgoals from a cell s is linear in the sum of the four clearance values of s in the diagonal directions because, when exploring an area, GetDirectHReachable(s) only looks up clearance values of cells that lie on the diagonal line bordering the area. Our implementations store only clearance values in the cardinal directions. Clearance values in the diagonal directions are used only in the first phase of GetDirectHReachable(s) but the runtime of their calculation during search is also linear in the sum of the four clearance values of s in the diagonal directions.

Algorithm 4 Constructing two-level subgoal graphs

```
1: function CostOtherPath(subgoals s, s', s'')
2: G'_T := ((V_T ∪ {s', s''}) \ {s}, E_T);
3: return the cost of a shortest path from s' to s'' over G'_T;
4: function IsNecessaryToConnect(subgoals s, s', s'')
5: if IsHReachable(s', s'') then
6:     return false;
7: if CostOtherPath(s, s', s'') ≤ h(s, s') + h(s, s'') then
8:     return false;
9: return true;
10: procedure PruneSubgoal(subgoal s)
11: V_T := V_T \ {s};
12: for all (s, s'), (s, s'') ∈ E_T do
13:     if h(s', s'') = h(s, s') + h(s, s'') then
14:         if CostOtherPath(s, s', s'') > h(s, s') + h(s, s'') then
15:             E_T := E_T ∪ {(s', s'')};
16: procedure PruneSubgoals()
17: V_T := V_S; E_T := E_S;
18: for all s ∈ V_T, in any order do
19:     necessary := false;
20:     for all (s, s'), (s, s'') ∈ E_T do
21:         if IsNecessaryToConnect(s, s', s'') then
22:             necessary := true;
23:             break;
24:     if ¬necessary then
25:         PruneSubgoal(s)
26: G_T := (V_T, E_T);
```

Two-Level Subgoal Graphs

In this section, we describe *two-level subgoal graphs*, that ignore parts of simple subgoal graphs depending on the locations of the start and goal cells. After constructing a simple subgoal graph, the subgoals are partitioned into *global* subgoals (in V_T) and *local* subgoals (in $V_S \setminus V_T$). Edges connect (local or global) subgoals. Only global subgoals are part of a two-level subgoal graph. Before an A* search, the local subgoals that are needed to connect the start and goal cells to the two-level subgoal graph (because they are direct-h-reachable from the start or goal cells) are temporarily made global subgoals. The A* search then ignores all remaining local subgoals as well as edges not between global subgoals.

Definition 4. *A* **two-level subgoal graph** $G_T = (V_T, E_T)$ *for a given simple subgoal graph* $G_S = (V_S, E_S)$ *is an undirected graph with the following properties:*

(a) $V_T \subseteq V_S$, *and* $E_S \subseteq E_T \subseteq V_S \times V_S$.

(b) For all $(s, s') \in E_T$, s *and* s' *are h-reachable.*

(c) For all $s, s' \in V_S$, *the distance between* s *and* s' *in the two-level subgoal graph* $G'_T = (V_T \cup \{s, s'\}, E_T)$ *is equal to the distance of* s *and* s' *on the simple subgoal graph.*

Algorithm 4 shows how to construct two-level subgoal graphs from simple subgoal graphs. The global subgoals and edges of the initial two-level subgoal graph are equal to the subgoals and edges, respectively, of the given simple subgoal graph. For each global subgoal s of the two-level subgoal graph, we check whether it is needed to connect one or

more pairs of its neighboring (local or global) subgoals. s is not needed to connect its neighboring subgoals s' and s'' (Lines 4-9) iff (a) there is a path between s' and s'' through only global subgoals, excluding s, that is no longer than $h(s', s)+h(s, s'')$ or (b) s' and s'' are h-reachable. If s is not needed to connect any pair of its neighboring subgoals, we prune it by removing it from V_T (so that it becomes a local subgoal) and add edges between all pairs of its neighboring subgoals that satisfy Condition (b) but not Condition (a) (Lines 10-15). The initial two-level subgoal graph satisfies the properties of Definition 4, and the above transformations preserve them.

The two-level subgoal graph resulting from a given simple subgoal graph depends on the order in which the global subgoals are checked. Consider the simple subgoal graph in Figure 1(b) for illustration, with subgoals A,B,C,D and edges A-B, B-C and C-D. We check the subgoals in the order A, B, C and D. A has only one neighboring subgoal and is thus trivially not needed to connect any pair of its neighboring subgoals. Consequently, A is pruned. B has only one pair of neighboring subgoals, namely (A,C), which satisfies Condition (b) but not (a). Consequently, B is pruned and edge A-C is added. C has three pairs of neighboring subgoals, of which the pair (A,D) does not satisfy Conditions (a) and (b). Thus, C is not pruned. Finally, D has only one neighboring subgoal and is thus trivially not needed to connect any pair of its neighboring subgoals. Consequently, D is pruned. The resulting two-level subgoal graph has one global subgoal (C) and one additional edge (A-C). If we had checked the subgoals in the order D, C, B and A, then the resulting two-level subgoal graph has one global subgoal (B) and one additional edge (B-D). Figure 3(a) shows a simple subgoal graph for part of a larger example (together with all edges between subgoals), and Figure 3(b) shows the resulting two-level subgoal graph (together with all edges between global subgoals).

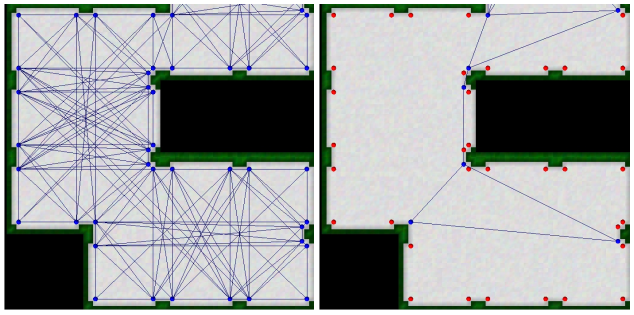
Finding a shortest path between a given start cell s and a given goal cell s' with a two-level subgoal graph is similar to finding a shortest path with a simple subgoal graph. Algorithm 5 shows how to connect s and s' to the two-level subgoal graph. FindPath and FindAbstractPath from Algorithm 2 are used without any modifications, except that FindAbstractPath now uses the modified two-level subgoal graph instead of the modified simple subgoal graph. The computation of the low-level path might take longer for two-level subgoal graphs than for simple subgoal graphs because the edges now connect h-reachable but not necessarily direct-h-reachable cells and the depth-first search thus might have to backtrack. However, our experimental results show that this increase in runtime is usually offset by the decrease in runtime due to having to search smaller graphs.

Theorem 2. *FindPath(s,s′) from Algorithm 2 that uses ConnectToGraph from Algorithm 5 finds a shortest path between cells s and s′ on the grid.*

Proof. Consider any high-level shortest path between s and s' on the modified simple subgoal graph. Let s'' and s''' denote the first and last subgoals on the path, respectively. The path can be divided into three segments, namely (a) the direct-h-reachable edge between s and s'', (b) the shortest path between s'' and s''' and (c) the direct-h-reachable edge between s''' and s'. (a) and (c) are part of the modified two-level-subgoal graph since ConnectToGraph connects both s and s' to all of their (local or global) direct-h-reachable subgoals. The distance between s'' and s''' on the two-level subgoal graph with s'' and s''' added is equal to their distance on the simple subgoal graph according to Property (c) of Definition 4. Thus, the distance between s and s' on the modified two-level subgoal graph is no larger than their distance on the modified simple subgoal graph and thus also no larger than their distance on the grid according to Theorem 1 (and it cannot be longer per construction). □

Algorithm 5 Searching two-level subgoal graphs

```
1: procedure ConnectToGraph(cell s)
2: if s ∉ V_T then
3:     V_T := V_T ∪ {s};
4: if s ∉ V_S then
5:     S ← GetDirectHReachable(s);
6:     for all s′ ∈ S do
7:         E_T := E_T ∪ {(s, s′)};
8:         if s′ ∉ V_T then
9:             V_T := V_T ∪ {s′};
```



(a) Simple subgoal graph (b) Two-level subgoal graph

Figure 3: Simple and two-level subgoal graphs

Other Improvements and Trade-offs

Discarding edges between local subgoals in two-level subgoal graphs: Edges between local subgoals can be up to 70 percent of all edges. Any shortest high-level path that contains an edge between local subgoals does not contain any other subgoals. Thus, an edge between local subgoals is relevant only if one of them is direct-h-reachable from the start cell and the other one is direct-h-reachable from the goal cell. We can discard all edges between local subgoals at the cost of giving up completeness, a loss that can be mitigated somewhat as follows: After a search for a high-level path between two cells s and s', we pick local subgoals s'' and s''' such that s and s'' are direct-h-reachable, s' and s''' are direct-h-reachable, and the sums $h(s, s'') + h(s'', s')$ and $h(s, s''') + h(s''', s')$ are minimal. If the sum $h(s, s'') + h(s'', s''') + h(s''', s')$ is smaller than the length of the path found by the search, then we check whether s'' and s''' are h-reachable. If they are, then we use $s-s''-s'''-s'$ as the shortest high-level path. (Special cases apply if s or s' are local or global subgoals.) This method does well in practice, rarely misses shortest high-level paths

and mostly misses them in easy instances. In cases where we fail to find a path, we try to find local subgoals s'' and s''' such that s and s'' are direct-h-reachable, s' and s''' are direct-h-reachable, and s'' and s''' are h-reachable. If we find such a pair, then we use $s - s'' - s''' - s'$ as the high-level path. This ensures that we never fail to find a path between s and s', if there exists one.

Precomputing pairwise distances: For simple subgoal graphs, we can precompute all pairwise distances between subgoals, which might result in feasible memory requirements since the number of subgoals is often much smaller than the number of unblocked cells. To find a shortest high-level path between cells s and s', we pick subgoals s'' and s''' such that $s'' = s$ or s'' is direct-h-reachable from s, $s''' = s'$ or s''' is direct-h-reachable from s', and the sum of $h(s, s'')$, the precomputed distance between s'' and s''' and $h(s''', s')$ is minimal. Finding the minimizing subgoals s'' and s''' requires a number of distance lookups, each of which takes constant time. For two-level subgoal graphs, we can proceed similarly by precomputing all pairwise distances between global subgoals, which requires even less memory than storing all pairwise distances between subgoals. We do not need to store the shortest paths themselves since they can be found fast with an A* search that uses the pairwise distances as heuristics.

Implementation Details

Edges: We store all subgoals (plus the start and goal cell) in a (subgoal) array together with lists of their neighboring subgoals on the subgoal graph, with the following exception: For an edge between a global and a local subgoal in two-level subgoal graphs, we store the global subgoal as a neighboring subgoal of the local subgoal but not vice versa. This way, a search does not reach a local subgoal from a global subgoal by default. Only when a local subgoal is temporarily added to the two-level subgoal graph, we store the local subgoal also as a neighboring subgoal of the global subgoal. Our data structure makes it easy to add subgoals and edges to subgoal graphs, remove them again and look up neighboring subgoals.

Clearance values: For all unblocked cells that are not subgoals, we store their clearance values in all cardinal directions. For all subgoals, we store pointers to their locations in the subgoal array instead because we do not need their clearance values during runtime. We use 8 bits to store each clearance value. For cells whose clearance values are too large to store in 8 bits, we simply store the highest possible value k. If a cell s has the highest possible clearance value k in direction d when we look up its clearance value during runtime, we also look up the clearance value of the cell k moves away from s in direction d. We keep repeating this procedure until we reach a cell with a clearance value below k, at which point we add up the clearance values of all visited cells and return the results as the clearance value of s in direction d.

Search: We implement the search as a standard A* search with a binary heap as the priority queue. We store the values needed by the A* search, such as the g-values and parent pointers, in the subgoal array and initialize them lazily during runtime.

	Memory (MB)			Prep Time (ms)			Solution Time per Map (s)			
Map type	S	TL	TLP	S	TL	TLP	A*	S	TL	TLP
bg512	1.12	1.10	1.20	10	225	292	2.149	0.060	0.043	0.038
DAO	0.61	0.59	0.89	5	179	535	3.541	0.252	0.087	0.032
starcraft	2.35	2.15	7.67	34	6,746	26,008	44.132	1.678	0.518	0.267
wc3maps512	1.15	1.13	1.39	13	260	386	4.380	0.071	0.054	0.047
maze1	2.19	2.17	-	40	67	-	120.360	36.189	30.062	-
maze2	1.80	1.76	-	33	61	-	199.727	18.843	12.449	-
maze4	1.35	1.32	20.68	22	44	75,885	264.360	7.028	3.030	0.967
maze8	1.15	1.15	2.96	17	24	1,903	329.499	2.536	1.380	0.762
maze16	1.10	1.09	1.25	14	17	67	334.355	1.018	0.754	0.587
maze32	1.08	1.08	1.09	13	14	15	241.170	0.357	0.402	0.373
random10	4.39	4.55	-	97	636	-	4.423	1.679	1.527	-
random15	4.66	4.76	-	98	509	-	7.142	3.004	2.551	-
random20	4.62	4.65	-	96	411	-	9.468	4.154	3.354	-
random25	4.39	4.38	-	92	337	-	11.336	5.078	3.876	-
random30	4.06	4.02	-	88	274	-	12.700	5.666	3.987	-
random35	3.64	3.58	-	81	220	-	15.813	6.941	4.486	-
random40	2.55	2.50	-	46	117	-	24.345	10.813	6.397	-
room8	1.52	1.53	-	25	45	-	19.072	0.846	0.709	-
room16	1.19	1.19	11.27	16	22	28,024	20.840	0.218	0.195	0.055
room32	1.10	1.10	1.67	14	15	351	24.394	0.085	0.086	0.048
room64	1.08	1.08	1.11	13	13	18	32.613	0.058	0.079	0.069

Table 1: Results of different subgoal graphs

Experimental Evaluation

We first compare A*, simple subgoal graphs (S), two-level subgoal graphs (TL) and two-level subgoal graphs with pairwise distances (TLP). For TL and TLP, we discard all edges between local subgoals. A* uses the same data structures as our subgoal graph variants. The experiments were run on a dual quad-core Xeon E5430 PC with a 2.66GHz CPU and 16GB of RAM. We compare the methods on different map types[1], namely maps from the games Baldur's Gate II and Warcraft III (resized to 512×512), maps from the game Dragon Age: Origins (ranging from 22×28 to 1260×1104), maps from the game StarCraft (ranging from 384×384 to 1024×1024), room maps (of varying room sizes), maze maps (of varying corridor widths), and maps with randomly blocked cells (of varying blockage percentages), all of size 512×512. We did not run TLP on all map types due to its long preprocessing times (of more than 30 minutes per map). For each map type, Table 1 shows the average amount of memory for the cached subgoal graph (except for A*), the average preprocessing time (except for A*) and the average time to solve all instances on a map. The results show that, in general, TLP is faster than TL, and TL is faster than S. S speeds up A* by factors from 2.2 (on maps with 25 percent randomly blocked cells) to 675.5 (on maze maps with corridor width 32). Preprocessing to generate subgoal graphs for S and TL is usually fast, except for StarCraft maps due to them containing many diagonal walls, which results in many local subgoals. Overall, subgoal graphs speed up A* the most on structured maps (such as room and maze maps, especially as their room sizes and corridor widths increase) because they result in few subgoals.

Not shown in Table 1 are the following results: First, TL and TLP find shortest paths in almost all cases even though

[1] All maps are available from Nathan Sturtevant's repository at http://movingai.com/benchmarks/.

	Memory (MB)		Runtime per Instance (ms)			Path Length	
Entry	Pre	Post	Total	Max	20	Total	%
BFS	11.5	17.0	1.141	1.140	1.140	298.3	1.15
BlockA*	58.0	64.1	0.482	0.482	0.482	298.3	1.15
CPD-full	3,040.1	3,043.2	0.050	0.045	0.006	258.7	1.00
CPD-mbm	67.8	81.7	0.190	0.006	0.016	258.7	1.00
JPS-offline	79.7	82.7	0.343	0.343	0.343	258.7	1.00
JPS-online	63.5	67.4	2.437	2.437	2.437	258.7	1.00
PDH	17.7	23.6	0.128	0.009	0.017	302.8	1.16
PPQ	47.3	50.9	0.830	0.830	0.830	259.1	1.00
SUB	12.8	16.7	0.024	0.024	0.024	258.7	1.00
Tree	11.9	16.2	0.004	0.004	0.004	303.6	1.51

Table 2: GPPC results on game maps

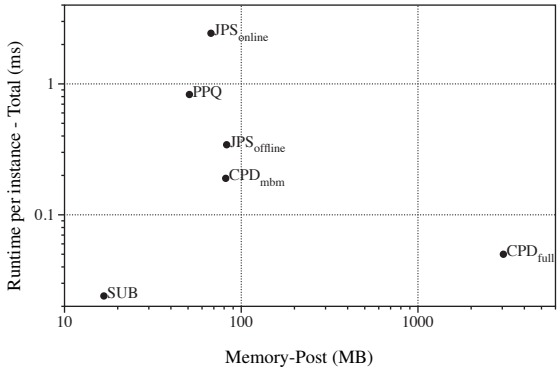


Figure 4: GPPC optimal planners on game maps

they cannot guarantee optimality since all edges between local subgoals are discarded, The average length of the paths found by TL and TLP is no more than 1 percent longer than the average shortest path length. Second, the percentage of the total runtime spent by ConnectToGraph varies a lot with the map type. On maze maps, for example, where the direct-h-reachable areas around cells are small and total runtimes are high, ConnectToGraph uses only around 0.07 percent of the total runtime of TL. On Baldur's Gate maps with open spaces and small total runtimes, ConnectToGraph uses around 31 percent of the total runtime of TL.

We also show preliminary results from the Grid Based Path Planning Competition (GPPC) 2012[2]. We only show results for game maps since some of the other entries did not solve all instances on other types of maps. These game maps include some maps from our experiment as well as maps from the game Dragon Age II. SUB is our entry. We entered TL instead of TLP since the preprocessing time and memory requirements of TLP do not scale well with the number of subgoals. The other entries that solved all instances on game maps are as follows: BFS performs breadth-first search without considering diagonal moves. Block A* (BlockA*) precomputes all paths in 3×3 blocks of cells and then performs an A* search over blocks rather than cells. Compressed Path Databases (CPD-full and CPD-mbm) precompute the directions of the first moves for all pairs of start and goal cells and compress the resulting tables, with different compression factors. Jump Point Search (JPS-online and JPS-offline) performs symmetry detection, online and offline. Precomputed Direction Heuristics (PDH) partition the grid into areas and store distances between areas. Pseudo-Priority Queues (PPQ) use search with a pseudo-priority queue to access the best node in the open list fast. Finally, Tree Cache (TREE) caches a spanning tree to make the search needed to return a path between cells on the spanning tree trivial. More information on most entries can be found in the online proceedings of the International Symposium on Combinatorial Search 2012[3]. The competition was run on a dual quad-core Xeon E5620 PC with a 2.4GHz CPU and 12GB of RAM. The entries could report individual moves one at a time or complete paths. Table 2 (Sturtevant 2012) reports memory, time and suboptimality statistics. Two memory statistics are reported, namely the average amount of memory for cached information and the average amount of memory needed during the search. Three time statistics are reported, namely the average time to find a complete path, the average maximum time to report a move and the average time to report the first twenty moves. Excluding TREE, whose paths are around 50 percent longer than shortest paths, our entry was the fastest to find complete paths and requires the least amount of memory. Figure 4 (Sturtevant 2012) shows a graph plotting the average time to find complete paths versus the memory requirements of all (near) optimal entries.

Conclusions and Future Work

We developed a new method for preprocessing eight-neighbor grids to generate subgoal graphs, that can be used to find shortest paths fast. Simple subgoal graphs and two-level subgoal graphs are specialized data structures for eight-neighbor grids with uniform movement costs and specific diagonal movement rules. However, the pruning idea behind generating two-level subgoal graphs from simple subgoal graphs generalizes beyond this scenario. For example, generating simple subgoal graphs from grids can be viewed as an application of the same pruning idea. Initially, the graph corresponds to the grid. We then repeatedly prune cells and add edges to ensure that the properties of Definition 4 continue to hold with the change that edges now need to connect direct-h-reachable subgoals instead of h-reachable ones. The subgoals of the simple subgoal graph correspond to the global subgoals, and the rest of the cells correspond to the local subgoals. The edges of the simple subgoal graph correspond to the edges between global subgoals. The edges that connect cells to their direct-h-reachable subgoals correspond to edges between local and global subgoals although simple subgoal graphs compute them during runtime using GetDirectHReachable rather than store them.

Future research includes extending two-level subgoal graphs to n-level subgoal graphs with 1st to nth level subgoals instead of only local and global subgoals, developing more sophisticated pruning rules for obtaining two-level subgoal graphs from simple subgoal graphs that prune the maximum number of cells, and applying our pruning ideas to additional domains, such as grids with non-uniform movement costs and lattices for motion planning.

[2]http://movingai.com/GPPC/

[3]http://www.aaai.org/Library/SOCS/socs12contents.php

References

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. In *Proc. of AIIDE*, 9–14.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1:7–28.

Botea, A. 2011. Ultra-fast optimal pathfinding without runtime search. In *Proc. of AIIDE*, 122–127.

Bulitko, V.; Björnsson, Y.; and Lawrence, R. 2010. Case-based subgoaling in real-time heuristic search for video game pathfinding. *JAIR* 39(1):269–300.

Cazenave, T. 2006. Improved heuristics for optimal path-finding on game maps. In *Proc. of CIG*, 27–33.

Goldenberg, M.; Felner, A.; Sturtevant, N.; and Schaeffer, J. 2007. Portal-based true-distance heuristics for path finding. In *Proc. of SoCS*.

Harabor, D., and Grastien, A. 2010. Breaking path symmetries in 4-connected grid maps. In *Proc. of AIIDE*, 33–38.

Harabor, D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proc. of AAAI*, 1114–1119.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 2:100–107.

Hernández, C., and Baier, J. A. 2011. Fast subgoaling for pathfinding via real-time search. In *Proc. of ICAPS*.

Lozano-Pérez, T., and Wesley, M. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Commununication of the ACM* 22(10):560–570.

Pochter, N.; Zohar, A.; Rosenschein, J.; and Felner, A. 2010. Search space reduction using swamp hierarchies. In *Proc. of AAAI*, 155–160.

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *Proc. of AAAI*, 1392–1397.

Sturtevant, N.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *Proc. of IJCAI*, 609–614.

Sturtevant, N. 2012. Grid-based path planning competition results [slides]. Technical report, University of Denver, Computer Science Department. http://www.movingai.com/GPPC/GPPC.pdf.