15

Subgoal Graphs for Fast Optimal Pathfinding

Tansel Uras and Sven Koenig

15.1 Introduction
15.2 Preliminaries
15.3 Simple Subgoal Graphs
15.4 Two-Level Subgoal Graphs
15.5 N-Level Graphs
15.6 Conclusion
Acknowledgments
References

15.1 Introduction

Paths for game agents are often found by representing the map that the agents move on as a graph and using a search algorithm, such as A*, to search this graph. Pathfinding in games needs to be fast, especially if many agents are moving on the map. To speed up path planning, maps can often be preprocessed before games are released or when they are loaded into memory. The data produced by preprocessing should use a small amount of memory, and preprocessing should be fast if it is performed at runtime.

In this chapter, we present *subgoal graphs*, which are constructed by preprocessing maps that are represented as grids. Subgoal graphs use a small amount of memory and can be used to find shortest paths fast by ignoring most of the grid cells during search. We describe several variants of subgoal graphs, each being a more sophisticated version of the previous one and each requiring more preprocessing time in return for faster searches. Even though subgoal graphs are specific to grids, the ideas behind them can be generalized to any graph representation of a map.

Table 15.1 summarizes the results from the original paper on subgoal graphs [Uras 14] on maps from the video games *Dragon Age: Origins*, *StarCraft*, *Warcraft III*, and *Baldur's*

Table 15.1 Comparison of Subgoal Graph Variants on Game Maps

Subgoal Graph Variant	Preprocessing Time (s)	Memory Used (MBytes)	Runtime of A* on Subgoal Graphs Rather Than Grids	Optimal?
Simple subgoal graphs	0.022	1.172	24 times faster	Yes
Two-level subgoal graphs	2.031	1.223	71 times faster	Yes
N-level subgoal graphs	2.195	1.223	112 times faster	Yes

Note: The average runtime of A* is 12.69 ms on these maps.

Gate II [Sturtevant 12]. The two-level subgoal graph (TSG) entry was one of the nondominated entries in the Grid-Based Path Planning Competitions 2012 and 2013. That is, if another entry was faster, it either was suboptimal or required more memory.

15.2 Preliminaries

We assume that the map is represented as a uniform-cost eight-neighbor grid with obstacles consisting of contiguous segments 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: we assume that the agent is not a point and, therefore, can move diagonally only if both associated cardinal directions are also unblocked. For example, in Figure 15.1, the agent cannot move diagonally from B2 to A1 because A2 is blocked. The lengths of cardinal and diagonal moves are 1 and $\sqrt{2}$, respectively.

A* is a commonly used algorithm for pathfinding. It is an informed search algorithm that uses a heuristic to guide the search to find paths faster. The heuristic estimates the distance between any two locations on the map and, in order for A* to find shortest paths, needs to be admissible, that is, never overestimate the distance between two locations [Hart 68].

One common heuristic used when solving pathfinding problems is the Euclidean distance, which is the straight-line distance between two locations. For instance, the Euclidean distance between s and r in Figure 15.1 is $\sqrt{5^2+2^2} = 5.39$. The Euclidean

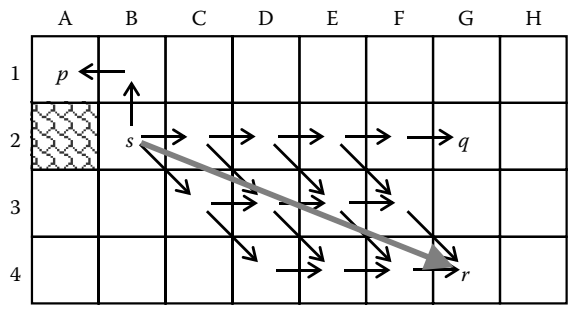


Figure 15.1

Shortest paths from s to some other cells on an eight-neighbor grid and the straight line between s and r.

distance is guaranteed to be admissible on maps with uniform traversal costs since the shortest path between two locations cannot be shorter than the straight line between them.

On eight-neighbor grids with uniform traversal costs, however, there is a more informed heuristic. The octile distance between two cells on the grid is the length of a shortest path between (that is, between their centers) assuming there are no obstacles on the grid. On a grid with no obstacles, the shortest path between two cells contains moves in only two directions. For instance, in Figure 15.1, all shortest paths between s and r contain exactly two diagonal moves toward the southeast and three cardinal moves toward the east. Therefore, the octile distance between two cells can be computed by simply comparing their x and y coordinates to figure out exactly how many diagonal and cardinal moves would be on a shortest path between them if the grid had no obstacles. For instance, the octile distance between s and r is $3 + 2 \times \sqrt{2} = 5.83$.

The octile distance is guaranteed to be admissible on eight-neighbor grids with uniform traversal costs since the shortest path between two cells cannot be shorter than on a grid with no obstacles. It is more informed than the Euclidean distance because the octile distance between two cells cannot be smaller than the Euclidean distance but is sometimes larger. Searching with a more informed heuristic means that the search typically performs fewer expansions before it finds a path and is then faster.

15.3 Simple Subgoal Graphs

Simple subgoal graphs (SSGs) are an adaptation of visibility graphs to grids. Visibility graphs abstract continuous environments with polygonal obstacles. The vertices of a visibility graph are the convex corners of obstacles, and the edges connect vertices that are visible from each other. The length of an edge is equal to the Euclidean distance between the vertices it connects. To find a shortest path between given start and goal locations in a continuous environment, one simply adds the vertices for them to the visibility graph, connects them to all vertices visible from them, and searches the resulting graph for a shortest path from the start vertex (which corresponds to the start location) to the goal vertex (which corresponds to the goal location). Figure 15.2 shows an example of a visibility graph and a path found by searching this graph. If an optimal search algorithm is used to search this graph (such as a suitable version of A* with the

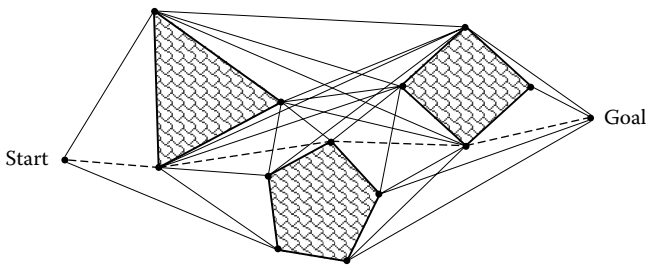


Figure 15.2

A visibility graph and the shortest path between the start and goal vertices.

Euclidean distance as heuristic), the resulting path is also a shortest path from the start location to the goal location in the continuous environment.

SSGs, on the other hand, abstract grids. The vertices of an SSG are called subgoals and are placed at the convex corners of obstacles on the grid. The edges connect subgoals that satisfy a certain connection strategy that we describe later. The length of an edge is equal to the octile distance between the vertices it connects. To find a shortest path between given start and goal cells on a grid, one can use the following steps: First, vertices are added to the SSG for the start and goal cells. Then, edges are added to connect them to the other vertices using the given connection strategy. Finally, the resulting graph is searched with A* with the octile distance as heuristic. The resulting high-level path is a series of subgoals connecting the start and goal vertices. One can then connect the subgoals on this high-level path on the grid to obtain a shortest low-level path on the grid.

Visibility graphs have strengths that SSGs aim to preserve. For instance, they can be used to find shortest paths and can be precomputed and stored. Visibility graphs also have some weaknesses that SSGs aim to fix. For instance, they can result in search trees with high branching factors, which is bad for both memory consumption and search time. The construction of visibility graphs can also be time consuming since one needs to perform visibility checks between all pairs of vertices. Even if preprocessing time is not an issue, visibility checks need to be performed when connecting the start and goal vertices to the visibility graph at runtime, namely, from the start and goal vertices to all other vertices of the visibility graph.

15.3.1 Constructing Simple Subgoal Graphs

Similar to visibility graphs, SSGs place subgoals at the corners of obstacles. Formally, we say that a cell s is a subgoal if and only if s is unblocked, s has a blocked diagonal neighbor t, and the two cells that are neighbors of both s and t are unblocked. For instance, in Figure 15.1, B1 is a subgoal because A2 is blocked and both A1 and B2 are unblocked. The idea is the same as for visibility graphs, namely, that one can use the convex corners of obstacles to navigate around them optimally.

We now introduce the concept of *h-reachability*. We say that two vertices of a graph are h-reachable if and only if there is a shortest path between them whose length is equal to the heuristic between them. H-reachability is a generalization of the concept of visibility in visibility graphs. Since visibility graphs abstract continuous environments, the heuristic used is the Euclidean distance. Thus, two vertices in a continuous environment are h-reachable if and only if they are visible from each other. Therefore, edges in visibility graphs connect exactly the h-reachable vertices.

Now, we discuss how h-reachable subgoals are connected. Since SSGs abstract grids, we use the octile distance as heuristic, and the length of an edge is equal to the octile distance between the subgoals it connects. Figure 15.3 shows an SSG constructed by connecting all h-reachable subgoals.

We now explain three properties that a connection strategy should possess and check whether the strategy of connecting h-reachable subgoals satisfies them:

1. *Edges are easy to follow on the grid*: If two cells on the grid are h-reachable, we can navigate from one cell to the other by moving in only two directions, as discussed in the preliminaries. This certainly makes it easier to follow h-reachable edges (edges that

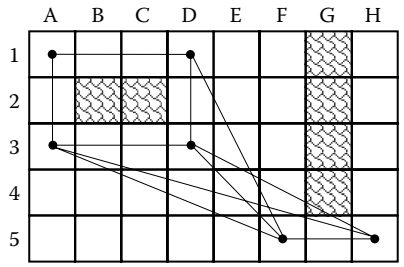


Figure 15.3

An SSG constructed by connecting all h-reachable subgoals.

connect two h-reachable cells) compared to doing an A* search on the grid between the two subgoals they connect since we know how many cardinal and diagonal moves to make in which directions. All we have to figure out is the order of the moves.

2. *Searches find shortest paths*: Adding edges between all h-reachable subgoals (plus the start and goal cells) also allows us to find shortest paths on the grid. The proof follows from the observation that, if two cells are not h-reachable, then there must be an obstacle that makes the path between the two cells longer than the octile distance between them. This obstacle introduces a subgoal that can be used to optimally circumnavigate it [Uras 13].
3. *Search trees have low branching factors*: Unfortunately, with this connection strategy, SSGs can have many more edges than the corresponding visibility graphs. For instance, in Figure 15.3, D3 and H5 are not visible from each other, but they are h-reachable.

The branching factors are a deal breaker for us, so we need to modify our connection strategy. H-reachability is still a valuable concept that will be used later when we generate two-level subgoal graphs from SSGs.

Fortunately, it is easy to address this issue. Consider the edge between D3 and H5 in Figure 15.3. This edge corresponds to the grid path D3-E4-F5-G5-H5. But there is already a subgoal at F5, and there are edges in the SSG between D3 and F5 and between F5 and H7. The sum of the lengths of these two edges is equal to the length of the edge between D3 and H5. Therefore, the edge between D3 and H5 is redundant and can be removed from the SSG without affecting the optimality of the resulting paths. When we remove all such redundant edges from the SSG in Figure 15.3, we end up with the SSG in Figure 15.4. We call the remaining edges *direct-h-reachable* edges and the subgoals they connect direct-h-reachable subgoals.

Formally, we say that two cells are direct-h-reachable if and only if they are h-reachable and none of the shortest paths between them pass through a subgoal. Direct-h-reachable edges are easier to follow than h-reachable edges. As mentioned before, h-reachable edges are easy to follow because we know exactly how many cardinal and diagonal moves we have to make in each direction. The problem is that we have to figure out the order of the moves. This is not the case for direct-h-reachable edges. Observe that, in Figure 15.1, all shortest paths between *s* and *r* cover a parallelogram-shaped area. As an equivalent definition of direct-h-reachability, we say that two cells are direct-h-reachable if and only if the parallelogram-shaped area between them does not contain any subgoals and that the

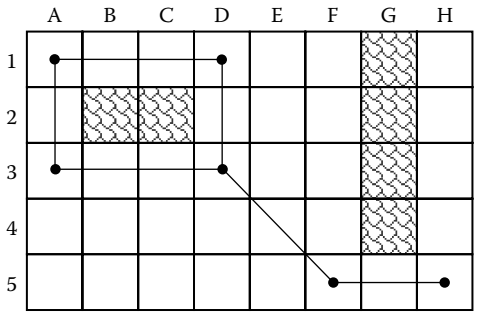


Figure 15.4

An SSG constructed by connecting all direct-h-reachable subgoals.

movement inside the parallelogram-shaped area is not blocked. Therefore, when following direct-h-reachable edges, we can make the cardinal and diagonal moves in any order. The equivalence of the two definitions follows from the observation that obstacles that block movement in the parallelogram-shaped area between two cells either introduce subgoals in the parallelogram-shaped area (meaning that the two cells are not direct-h-reachable) or block all shortest paths between the two cells (meaning that the two cells are not even h-reachable) [Uras 13].

The definition of parallelogram-shaped areas is also useful for showing that the branching factors of the search trees generated when using SSGs are lower than when using visibility graphs. If the movement inside the parallelogram-shaped area between two cells is not blocked, then the straight line between the two cells cannot be blocked either, which means that they must be visible from each other. Therefore, every direct-h-reachable edge in an SSG corresponds to a straight-line edge in the visibility graph. The converse is not true. For instance, in Figure 15.4, A3 and F5 are visible from each other but not direct-h-reachable.

To summarize, SSGs are constructed by placing subgoals at the corners of obstacles and adding edges between subgoals that are direct-h-reachable. Section 15.3.3 describes an algorithm to identify all direct-h-reachable subgoals from a given cell.

15.3.2 Searching Using Simple Subgoal Graphs

Once an SSG has been constructed, it can be used to find shortest paths between any two cells on the grid. Given a start cell and a goal cell, we connect them to their direct-h-reachable subgoals using the algorithm described in the next section. We then search the resulting graph with an optimal search algorithm, such as a suitable version of A* with the octile distance heuristic, to find a shortest high-level path. We then follow the edges of this path by simply moving in the direction of the next cell on the high-level path, to find a shortest low-level path on the grid. This process is illustrated in Figure 15.5.

There are some subtle points to the algorithm. If either the start or goal cell is a subgoal, we do not need to identify the subgoals that are direct-h-reachable from them since they are already in the SSG. Also, since the algorithm described in the next section finds only the subgoals that are direct-h-reachable from a given cell, it might not connect the start cell to the goal cell if they are direct-h-reachable but neither of them is a subgoal. In this case, we might not be able to find a shortest path between them. Before connecting the

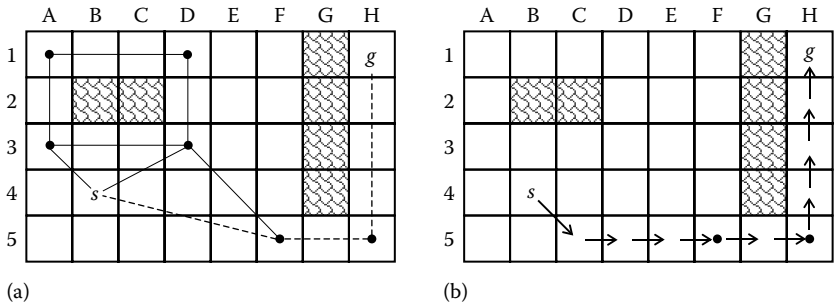


Figure 15.5

Connecting the start and goal cells to the SSG and finding a shortest high-level path on the resulting graph (a). Then, following the edges on this high-level path to find a shortest low-level path on the grid (b).

start and goal cells to the SSG, we therefore first generate a possible shortest path between them (for instance, by first moving diagonally and then moving cardinally). If the path is not blocked, we return it as the shortest path. Otherwise, the start and goal cells cannot be direct-h-reachable, and we therefore search using the SSG as described earlier.

15.3.3 Identifying All Direct-H-Reachable Subgoals from a Given Cell

Being able to identify all direct-h-reachable subgoals from a given cell quickly is important both during the construction of SSGs and when connecting the start and goal cells to SSGs. The algorithm we propose is a dynamic programming algorithm that identifies all direct-h-reachable cells from a given cell s and returns all subgoals among them. Figure 15.6 shows an example.

Our algorithm uses clearance values. The clearance value of a cell s in a direction d, called *Clearance*(s, d), is the maximum number of moves the agent can make from s

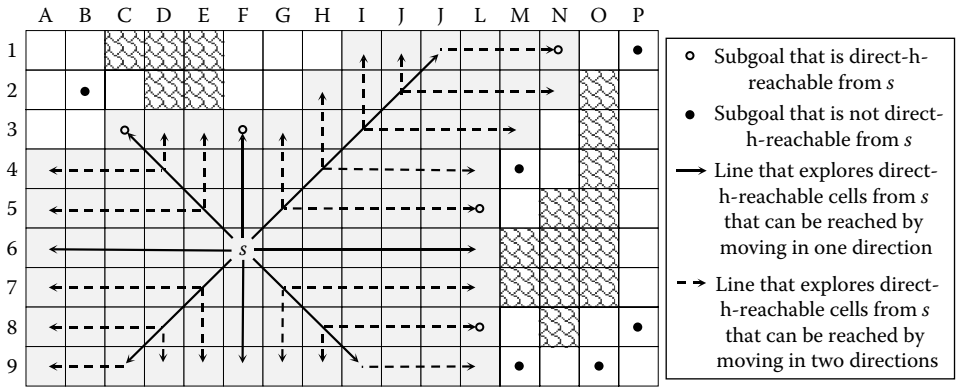


Figure 15.6

Identifying the direct-h-reachable area around s (shown in gray), which contains all direct-h-reachable subgoals from s.

toward *d* without reaching a subgoal or being blocked. For instance, the east clearance of *s* in Figure 15.6 is 6 because M6 is blocked. The north clearance of *s* is 2 because F3 is a subgoal. Clearance values can be either computed at runtime or be precomputed, although the algorithm does not benefit much from storing clearance values in diagonal directions. The algorithm works in two phases:

- The first phase identifies all direct-h-reachable cells from *s* that can be reached by moving in only one direction. This is achieved by looking at the clearance values of *s* in all eight directions to figure out if there are direct-h-reachable subgoals in these directions. For instance, since the north clearance of *s* in Figure 15.6 is 2, the algorithm checks the cell 2 + 1 moves north of *s*, F3, to see if it is a subgoal. In Figure 15.6, the first phase determines that C3 and F3 are direct-h-reachable subgoals from *s*.
- The second phase identifies all direct-h-reachable cells from *s* that can be reached by a combination of moves in a cardinal and a diagonal direction. There are eight combinations of cardinal and diagonal directions that can appear on a shortest path between two direct-h-reachable cells, and each of them identifies an area. Figure 15.6 shows these combinations, divided by the solid lines emanating from *s* in eight directions. The algorithm explores each area line by line (using the dashed lines in Figure 15.6). Assume that it is exploring the area that is associated with cardinal direction *c* and diagonal direction *d*. For each cell that is direct-h-reachable from *s* by moving toward *d*, it casts a line that starts at that cell and travels toward *c*. It starts with the line closest to *s* and continues until all lines are cast.

We now present three rules to determine how far each line extends. The first rule is simple: *a line stops when it reaches a subgoal or directly before it reaches an obstacle*. This is so because the additional cells the line would reach cannot be direct-h-reachable from s according to the definition of direct-h-reachability. Otherwise, the parallelogram-shaped area between s and the next cell the line would reach contained a subgoal or obstacle. The second rule follows from the following observation: if cell *t* is direct-h-reachable from cell *s*, then any cell *u* that lies in the parallelogram-shaped area between *s* and *t* is also direct-h-reachable from *s*. This is so because the parallelogram-shaped area between *s* and *u* is a subarea of the parallelogram-shaped area between *s* and *t* and, therefore, does not contain any subgoals and the movement inside the area is not blocked (since *s* and *t* are direct-h-reachable). Therefore, the area of cells that are direct-h-reachable from *s* is a union of parallelogram-shaped areas, each area between *s* and some other cell. This results in the second rule: *a line cannot be longer than the previous line*. Otherwise, the area cannot be a union of parallelogram-shaped areas. The third rule is a refinement of the second rule: *a line cannot be longer than the previous line minus one cell if the previous line ends in a subgoal*.

The algorithm uses these rules to determine quickly how far each line extends. For instance, in Figure 15.6, when the algorithm explores the east–northeast area around *s*, the first line it casts travels along row 5 toward east and reaches subgoal L5 after 5 moves. Since the first line ends in a subgoal, the second line can only travel 5 − 1 = 4 moves and stops before reaching subgoal M4, which is not direct-h-reachable from *s*. Instead of explicitly casting lines, the algorithm can use the clearance values of the cells in which the lines originate. Listing 15.1 shows pseudocode that uses the clearance values.

Listing 15.1. Identify all direct-h-reachable subgoals in an area.

```
GetDirectHReachable(cell s, cardinal dir. c, diagonal dir. d)
    SubgoalVector list = {};
    int maxLineLength = Clearance(s,c);
    int nDiagMoves = Clearance(s,d);
    for int i = 1 … nDiagMoves
        s = neighbor of s toward d;
        l = Clearance(s,c);
        if (l < maxLineLength)
            maxLineLength = l;
            s' = the cell l+1 moves away from s toward c;
            if (s' is a subgoal)
                list.add(s');
    return list;
```

15.4 Two-Level Subgoal Graphs

Searches using SSGs are faster than searches of grids because SSGs are smaller than grids and searching them expands fewer cells on average. In a way, SSGs partition the cells into subgoals and nonsubgoals, and the search ignores all nonsubgoals other than the start and goal cells.

Two-level subgoal graphs (TSGs) apply this idea to SSGs instead of grids. TSGs are constructed from SSGs by partitioning the subgoals into local and global subgoals. The search ignores all local subgoals that are not direct-h-reachable from the start or goal cells, allowing us to search even smaller graphs. TSGs satisfy the following property, called the *two-level property* (*TLP*):

> The length of a shortest path between any two (local or global) subgoals s and t on the SSG is equal to the length of a shortest path between *s* and *t* on the graph consisting of all global subgoals of the TSG plus s and t (and all edges between these subgoals in the TSG).

In other words, if we remove all local subgoals except for *s* and *t* (and their associated edges) from the TSG, then the length of a shortest path between *s* and *t* does not change.

This property guarantees that TSGs can be used to find shortest paths on grids [Uras 13]. Figure 15.7 shows an SSG and a TSG constructed from the SSG. Observe that the subgraph of the TSG consisting of A1, D1, D3, and H5 contains the shortest path between A1 and H5. Also, observe that the edge between D3 and H5 is not direct-h-reachable. During the construction of TSGs, h-reachable edges can be added to the graph if this allows classifying more subgoals as local subgoals.

15.4.1 Constructing Two-Level Subgoal Graphs

Constructing TSGs from SSGs is different from constructing SSGs from grids. When constructing SSGs from grids, we identify some cells as subgoals and connect them with a connection strategy that allows them to be used to find shortest paths on the grids. This is possible because grids have structure, and visibility graphs provide some intuition for exploiting it.

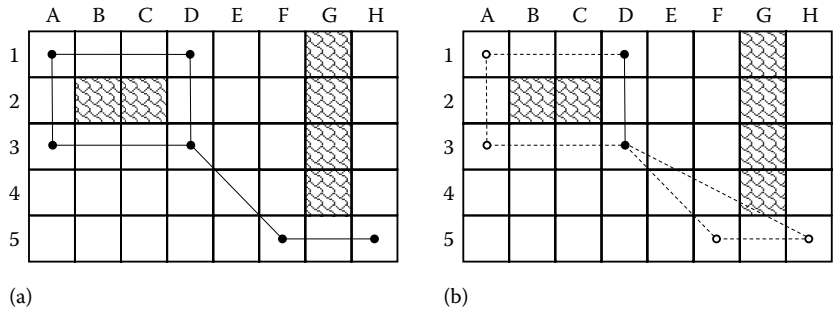


Figure 15.7

An SSG (a) and a TSG constructed from the SSG (b). Hollow circles indicate local subgoals, and solid circles indicate global subgoals.

On the other hand, there is little structure to exploit when constructing TSGs from SSGs. Therefore, we start by assuming that all subgoals are global subgoals. At this point, the TLP is satisfied since the TSG is identical to the SSG. We then iterate over all global subgoals and classify them as local subgoals if doing so does not violate the TLP. We are allowed to add edges between h-reachable subgoals if doing so helps to preserve the TLP and allows us to classify a global subgoal as a local subgoal.

The question remains how to determine quickly whether a subgoal *s* can be classified as a local subgoal. The straightforward method is to check if removing *s* from the TSG increases the length of a shortest path between two other subgoals that are not h-reachable (if they are h-reachable, we can simply add an edge between them). It is faster to check if removing *s* from the TSG increases the length of a shortest path between two of its neighbors that are not h-reachable, because any path that passes through *s* must also pass through its neighbors. The process of removing *s* in this way is called a contraction [Geisberger 08]. Listing 15.2 shows pseudocode for constructing a TSG from an SSG.

For each global subgoal *s*, we accumulate a list of edges that would need to be added to the TSG if *s* were classified as a local subgoal. We iterate over all pairs of neighbors of *s*. If there exists a pair of neighbors such that all shortest paths between the two neighbors pass through *s* and the neighbors are not h-reachable, then *s* cannot be classified as a local subgoal because doing so would violate the TLP. Otherwise, we classify *s* as a local subgoal and add all necessary edges to the TSG.

SSGs do not necessarily have unique TSGs since the resulting TSG can depend on the order in which the subgoals are processed. For instance, in Figure 15.7, if D1 were a local subgoal and A3 were a global subgoal, the resulting TSG would still satisfy the TLP. No research has been done so far on how the order in which the subgoals are processed affects the resulting TSG.

15.4.2 Searching Using Two-Level Subgoal Graphs

Search using TSGs is similar to search using SSGs. We start with a core graph that consists of all global subgoals and the edges between them. We then connect the start and goal cells to their respective direct-h-reachable (local or global) subgoals. Next, local subgoals

Listing 15.2. Constructing a TSG from an SSG.

```
ConstructTSG(SSG S)
    SubgoalList G = subgoals of S; //Global subgoals
    SubgoalList L = {}; //Local subgoals
    EdgeList E = edges of S;
    for all s in G
        EdgeList E+ = {}; //Extra edges
        bool local = true; //Assume s can be a local subgoal
        for all p, q in Neighbors(s) //Neighbors wrt E
            d = length of a shortest path between p and q
                (wrt E) that does not pass through s or any
                subgoal in L;
            if (d > c(p,s) + c(s,q))
               if (p and q are h-reachable)
                   E+.add((p,q));
               else //s is necessary to connect p and q
                   local = false; //Can't make s local
                   break;
        if (local)
            G.remove(s); //Classify s as a local subgoal
            L.add(s);
            E.append(E+); //Add the extra edges to the TSG
    return (G, L, E);
```

that are direct-h-reachable from the start or goal cells using edges not in the core graph are added to the graph. Once a high-level shortest path from the start cell to the goal cell is found on this graph, we follow its edges to find a low-level path on the grid. We might have to follow edges between cells that are h-reachable but not direct-h-reachable. This means that we have to identify the order of cardinal and diagonal moves, which can be achieved with a depth-first search. Figure 15.8 shows an example of this search graph. The number of subgoals excluded from the search can be much larger for larger TSGs.

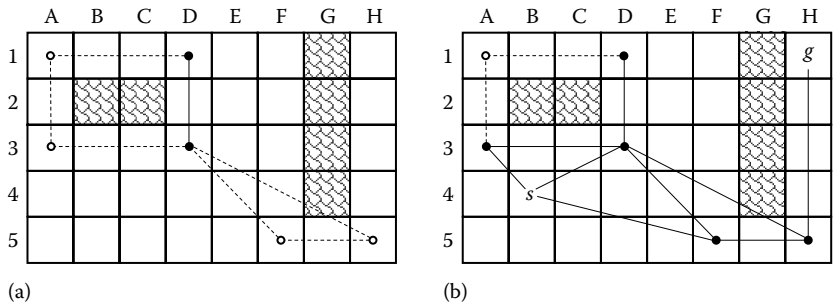


Figure 15.8

A TSG (a) and a search using this TSG (b). The graph that is searched consists of the solid circles.

15.5 N-Level Graphs

We now generalize the ideas behind the construction of TSGs to be able to generate graphs with more than two levels from any given undirected graph [Uras 14].

Observe that only the terms "subgoal" and "h-reachable" are specific to subgoal graphs in the partitioning algorithm shown in Listing 15.2 and can be replaced by the terms "vertex" and "a property *P* that all extra edges need to satisfy," making the partitioning algorithm applicable to any undirected graph. The lengths of the edges added to the graph should always be equal to the lengths of shortest paths on the original graph between the two vertices they connect. We need to specify a property *P* that all extra edges need to satisfy. Otherwise, all vertices of a graph can be classified as local by adding edges between all vertices, which would create a pairwise distance matrix for the graph. *P* should be chosen such that the extra edges can easily be followed on the original graph and the branching factors of the search trees do not increase too much. H-reachability satisfies these criteria for subgoal graphs, although other properties could exist that would result in even faster searches. If it is hard to come up with such a property, one can always choose *P* such that no extra edges are added to the graph, resulting in fewer vertices being excluded from the search. *Two-level graphs* can be constructed by applying the general version of the partitioning algorithm described in Listing 15.2 to any undirected graph. Figure 15.9 shows an example with an undirected graph with unit edge costs and a two-level graph constructed from the undirected graph without adding extra edges.

The general version of the algorithm described in Listing 15.2 partitions the vertices of an undirected graph into local and global vertices. Call local vertices *level 1 vertices* and global vertices *level 2 vertices*. Level 2 vertices and the edges between them form a graph, and one can apply the general version of the algorithm described in Listing 15.2 to this graph to partition the level 2 vertices into level 2 and level 3 vertices. Figure 15.10 shows an example of a *three-level graph*, constructed from the two-level graph shown in Figure 15.9 by partitioning its level 2 vertices into level 2 and level 3 vertices.

One can keep adding levels to the graphs by recursively partitioning the highest-level vertices until they can no longer be partitioned. Adding more levels to the graph means

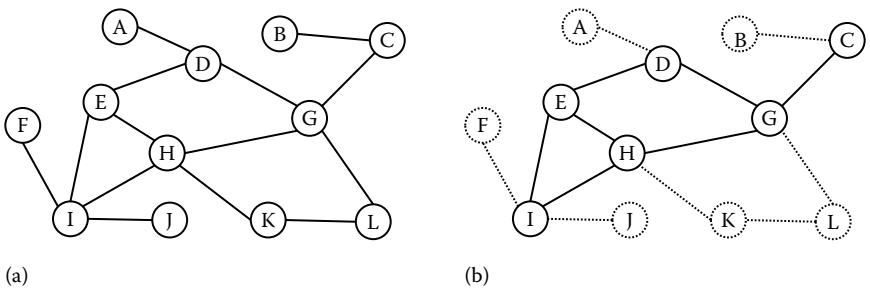


Figure 15.9

An undirected graph with unit edge costs (a) and a two-level graph constructed from the undirected graph without adding extra edges (b). Solid circles indicate global vertices, and dashed circles indicate local vertices.

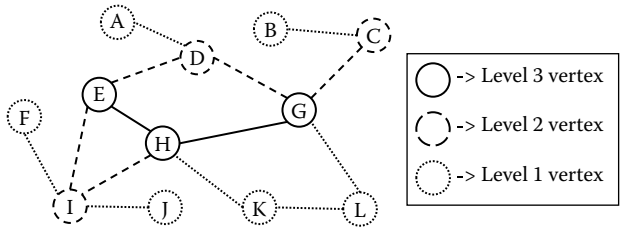


Figure 15.10

A three-level graph constructed from the undirected graph in Figure 15.9.

that the graphs that are searched are getting smaller but also that one needs to spend more time constructing the graph for each search.

Once an n-level graph has been constructed, it can be used to find shortest paths between any two vertices of the original graph. Call the graph consisting of all highest-level vertices (and the edges between them) the *core graph* since it appears in every search. When using a two-level graph to find a shortest path between given start and goal vertices, one adds them to the core graph and searches the resulting graph with an optimal search algorithm to find a high-level path. When using a three-level graph to find a shortest path between given start and goal vertices, one also adds any level 2 vertices that are neighbors of the start or goal vertices. This process is illustrated in Figure 15.11. Listing 15.3 shows pseudocode to determine which vertices need to be added to the core graph when using n-level graphs, for any value of N. This algorithm needs to be called for both the start and goal vertices. When this algorithm is called for an SSG, it creates an n-level subgoal graph.

SSGs are essentially *two-level grid graphs*. If one were to allow the addition of extra edges between direct-h-reachable vertices of the grid graph, the general version of the algorithm described in Listing 15.2 could classify all subgoals as level 2 vertices and all nonsubgoals as level 1 vertices [Uras 14], although it could take a long time to run and the extra edges between direct-h-reachable cells could require a lot of memory to store. The construction of SSGs, as described previously, avoids these issues by exploiting the structure of grids and by only storing direct-h-reachable edges between subgoals. Direct-h-reachable edges that are not stored are reconstructed

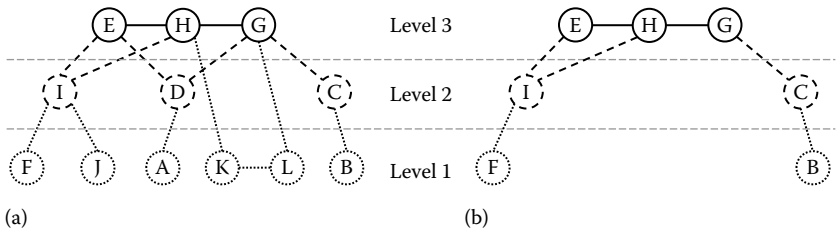
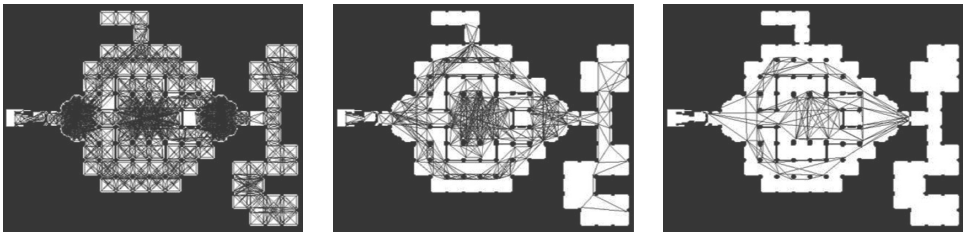


Figure 15.11

The three-level graph from Figure 15.10 with vertices rearranged in layers (a) and the graph searched for a shortest path between vertices F and B of the three-level graph (b).

Listing 15.3. Identifying which vertices need to be added to the core graph during search.

```
IdentifyConnectingVertices(vertex s, Graph G, int graphLevel)
    VertexList open = {s};
    VertexList closed = {};
    while (open != {})
        vertex p = open.getVertexWithSmallestLevel();
        open.remove(p);
        if (p.level == graphLevel)
             break;
        if (!closed.contains(p))
             closed.add(p);
             for all neighbors q of p in G
                 if (q.level > p.level && !closed.contains(q))
                     open.add(q);
    return closed;
```



(a) (b) (c)

Figure 15.12

An SSG (a), TSG (b), and a five-level subgoal graph (c). Only the highest-level subgoals and the edges between them are shown.

before a search when connecting the start and goal cells to the SSG or when checking if the start and goal cells are direct-h-reachable. Figure 15.12 shows an SSG, a TSG, and a five-level subgoal graph (six-level grid graph).

The ideas behind n-level graphs are also closely related to contraction hierarchies [Geisberger 08], where the vertices of a graph are first ordered by "importance" and then iteratively contracted, starting from the least important vertex. Contracting a vertex v means replacing unique shortest paths between the neighbors of v that go through v by shortcut edges. The resulting graphs are searched with a bidirectional search algorithm, where the forward search uses only edges leading to more important vertices and the backward search uses only edges coming from more important vertices. Vertex contraction is used during the construction of n-level graphs whenever the level of a vertex is decreased. In essence, n-level graphs are contraction hierarchies where there are constraints on adding new edges to the graph but each level is not limited to contain only one vertex.

15.6 Conclusion

Subgoal graphs are generated by preprocessing grids and can be used to significantly speed up searches on grids, with little memory overhead. The ideas behind them apply to any undirected graph, although it might need some ingenuity to figure out a suitable property that all extra edges need to satisfy. We have, so far, only tested n-level graphs on grids.

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

[Geisberger 08] Geisberger, R., Sanders, P., Schultes, D., and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. *Proceedings of the International Workshop on Experimental Algorithms,* Provincetown, MA, 319–333.

[Hart 68] 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* 4(2):100–107.

[Sturtevant 12] Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144–148.

[Uras 13] Uras, T., Koenig, S., and Hernandez, C. 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. *Proceedings of the International Conference on Automated Planning and Scheduling*, Rome, Italy, 224–232.

[Uras 14] Uras, T. and Koenig, S. 2014. Identifying hierarchies for fast optimal search. *Proceedings of the AAAI Conference on Artificial Intelligence*, Quebec City, Quebec, Canada, pp. 878–884.