Identifying Hierarchies for Fast Optimal Search

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

Abstract

Search with Subgoal Graphs (Uras, Koenig, and Hernández 2013) was a non-dominated optimal path-planning algorithm in the Grid-Based Path Planning Competitions 2012 and 2013. During a preprocessing phase, it computes a Simple Subgoal Graph from a given grid, which is analogous to a visibility graph for continuous terrain, and then partitions the vertices into global and local subgoals to obtain a Two-Level Subgoal Graph. During the path-planning phase, it performs an A* search that ignores local subgoals that are not relevant to the search, which significantly reduces the size of the graph being searched.

In this paper, we generalize this partitioning process to any undirected graph and show that it can be recursively applied to generate more than two levels, which reduces the size of the graph being searched even further. We distinguish between basic partitioning, which only partitions the vertices into different levels, and advanced partitioning, which can also add new edges. We show that the construction of Simple-Subgoal Graphs from grids and the construction of Two-Level Subgoal Graphs from Simple Subgoal Graphs are instances of generalized partitioning. We then report on experiments on Subgoal Graphs that demonstrate the effects of different types and levels of partitioning. We also report on experiments that demonstrate that our new N-Level Subgoal Graphs achieve a speed up of 1.6 compared to Two-Level Subgoal graphs from (Uras, Koenig, and Hernández 2013) on maps from the video games StarCraft and Dragon Age: Origins.

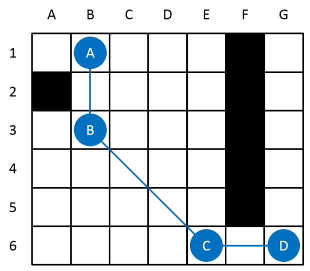
Introduction

For some search problems, the graph is known beforehand and there is time to preprocess the graph to make the search faster. One such example is video games, where one can often preprocess maps before a game is released or while a map is loaded into memory. The data produced by preprocessing should use only a small amount of memory, and, in case they are generated during runtime, preprocessing should be fast.

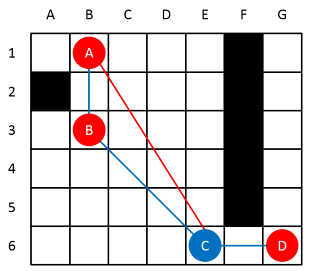
In this paper, we present a method of preprocessing any undirected graph to partition its vertices into several levels to create a hierarchy. These hierarchies satisfy one important property: Between any two vertices of the graph, there must be an *arching* path between them whose length is equal to the distance between them in the original graph. The levels of the vertices on an arching path strictly monotonically increase to some level, remain at this level, and then strictly monotonically decrease. During search, any vertex that cannot lie on an arching path between the start and goal vertices is ignored. Depending on the structure of the original graph and the hierarchy, the search can ignore a significant portion of the graph.

The idea of partitioning vertices into levels to speed up search has been used before in the context of Subgoal Graphs (Uras, Koenig, and Hernández 2013), but only to partition the vertices into two levels. The main contribution of this paper is the generalization of this partitioning idea to using undirected graphs and recursively partitioning the highest-level vertices to generate an arbitrary number of levels. We distinguish two types of partitioning: *Basic partitioning* simply assigns levels to the vertices of a graph, but otherwise leaves the graph unchanged. *Advanced partitioning*, on the other hand, may add new edges to the graph, which can increase the number of vertices ignored during search and, therefore, improve its performance. We provide algorithms for constructing and searching N-Level Graphs and prove their correctness. We show that Subgoal Graphs are instances of 2-Level Graphs on grids and use our recursive partitioning idea to construct N-Level Subgoal Graphs. Our experimental evaluations show that, by partitioning vertices into more levels, we can speed up search with Subgoal Graphs by a factor of 1.6 on maps from the video game StarCraft, making search with Subgoal Graphs 193 times faster than A* on these maps.

Methods for preprocessing graphs 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 by abstracting groups of vertices and finding high-level paths over these abstract vertices. These high-level paths are then refined to low-level paths, which are not guaranteed to be optimal. More informed heuristics (Björnsson and Halldórsson 2006; Cazenave 2006; Sturtevant et al. 2009) guide the searches better to expand fewer states. Hierarchies can also be used to derive heuristics during search (Leighton, Ruml, and Holte 2008; Holte et al. 1994). Dead-end detection and other pruning methods (Björnsson and Halldórsson 2006;



(a) Simple Subgoal Graph



(b) Two-Level Subgoal Graph

Figure 1: Simple and Two-Level Subgoal Graphs (local subgoals are shown in red).

Goldenberg et al. 2010; Pochter et al. 2010) identify areas of the graph that do not need to be searched to find shortest paths. Search with contraction hierarchies (Geisberger et al. 2008) is an optimal and extremely hierarchical method, as every level of the hierarchy contains only a single vertex. It has been shown to be effective on road networks but seems to be less effective on graphs with higher branching factors, such as grid-based game maps (Storandt 2013).

Subgoal Graphs

In this section, we provide a brief introduction to Subgoal Graphs, which are used as a running example throughout the paper and as a testbed for our experiments. Subgoal Graphs apply to 8-neighbor grids 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: in order to move diagonally, both adjacent cardinal directions must be unblocked. For instance, in Figure 1, the agent cannot move from A1 to B2 because A2 is blocked. The lengths of cardinal and diagonal directions are 1 and $\sqrt{2}$, respectively. The heuristic $h(u, v)$ is the octile distance between cells u and v, which is the distance between u and v assuming that the grid is obstacle-free. We use the notation $u + c$ to denote the cell that is reached from cell u by moving in direction c.

Simple Subgoal Graphs (SSGs), which are analogous to visibility graphs in continuous terrain, are constructed by placing subgoals at the convex corners of blocked cells and adding edges between all direct-h-reachable subgoals. The following definitions make this statement more precise. Figure 1(a) shows an instance of a SSG.

Definition 1. *An unblocked cell u is a* ***subgoal*** *iff there are two perpendicular cardinal directions c_1 and c_2 such that $u+c_1+c_2$ is blocked and $u+c_1$, $u+c_2$ are unblocked. Two cells u and v are* ***h-reachable*** *iff there is a path of length $h(u, v)$ between them. Two h-reachable cells u and v are* ***direct-h-reachable*** *iff none of the shortest paths between them contain a subgoal $t \notin \{u, v\}$. The length of the edge between two h-reachable cells is equal to the octile distance between them.*

SSGs are used for finding shortest paths only if the given start and goal cells are not direct-h-reachable, which is checked before the search. If they are direct-h-reachable, a shortest path is returned by optimally following the direct-h-reachable edge between them. Otherwise, the shortest path is found by connecting the start and goal cells to their respective direct-h-reachable subgoals, searching the resulting graph for a shortest path that uses direct-h-reachable edges, and finally optimally following these edges.

Two-Level Subgoal Graphs (TSGs) are constructed from SSGs by partitioning the subgoals into *global* and *local* subgoals and adding extra edges between some pairs of h-reachable subgoals. The resulting graph has the following property, called the TSG property: Between any two subgoals u and v, there exists a shortest path $(s_0 = u, \dots, s_n = v)$ on the TSG such that the length of this shortest path is equal to the distance between u and v on the SSG and s_i is a global subgoal for all $i \in \{1, \dots, n-1\}$. Figure 1(b) shows an instance of a TSG. Search with TSGs is similar to search with SSGs, with the following exceptions: The search uses the global subgoals and the edges between them, called the *global subgoal graph*, instead of the SSG. The given start and goal cells are connected to the the global subgoal graph by first connecting them to their respective direct-h-reachable subgoals and then connecting the local subgoals among those subgoals to the global subgoal graph using existing edges in the TSG. This results in a smaller graph to search for a shortest path compared to SSGs because local subgoals that are not direct-h-reachable from the start or the goal cells are ignored. Search with SSGs and TSGs result in shortest paths.

N-Level Graphs

A TSG is constructed by partitioning the subgoals of an SSG into two levels. In this section, we generalize this idea to undirected graphs and an arbitrary number of levels. We give a formal definition of N-Level Graphs and describe the key ideas behind them, including how they are constructed and how they are searched.

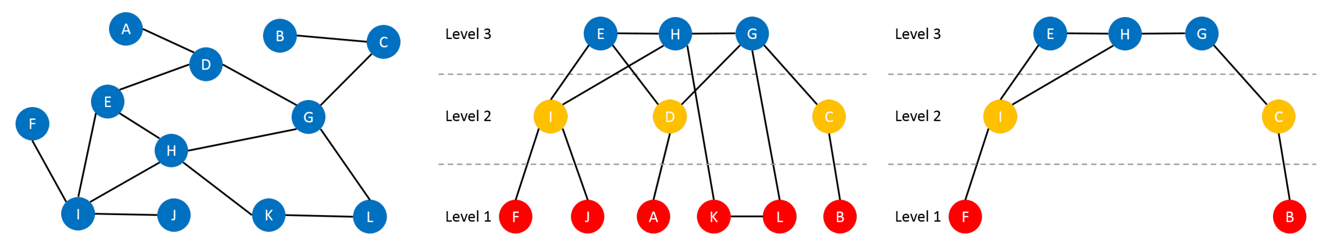
Definition 2. *An undirected graph G is a 3-tuple $G = (V, E, c)$, where V is a set of vertices, E is a set of edges, and c: $E \mapsto \mathbb{R}_{>0}$ is a function that assigns a length to each edge. A path $\Pi = (v_0, \dots, v_k)$ on G is a list of vertices such that $\forall i \in \{0, \dots, k-1\}$ $(v_i, v_{i+1}) \in E$. $d_G(u, v)$ is the distance between any two vertices u and v on G.*

Assumption 1. *We assume that the algorithms described in this paper operate on connected graphs $G = (V, E, c)$ and, if $\exists(u, v) \in E$, then $c(u, v) = d_G(u, v)$.*

Graphs that are not connected can be partitioned into their connected components. Edges that are not shortest paths between the vertices they connect can be removed from the graph without increasing the distance between those vertices.

Definition 3. *An* ***N-Level Graph*** *of the undirected graph $G = (V, E, c)$ is a 4-tuple $G_N = (V, E_N, c_N, l)$, where $G' = (V, E_N, c_N)$ is an undirected graph with $E_N \supseteq E$ and l: $V \mapsto \{1, \dots, N\}$ is a function that assigns levels to vertices, with the following properties:*

1. *$\forall u, v \in V$, there is an arching path between u and v on G_N with length $d_G(u, v)$; and*
2. *$\forall (u, v) \in E_N$, $c_N(u, v) = d_G(u, v)$.*



(a) An undirected graph with unit-length edges (b) 3-Level graph of the undirected graph (c) Graph searched for a shortest path between F and B

Figure 2: Idea behind N-Level graphs.

Algorithm 1 Constructing N-Level Graphs

```
 1: function GetNextLevel($G_i = (V, E, c, l)$, $P$)
 2: $V^* := \{v \in V : l(v) = i\}$;
 3: for all $v \in V^*$ do
 4:     $l(v) := i + 1$;
 5: for all $v \in V^*$ do
 6:     $S := \{u \in V^* : (v, u) \in E\}$;
 7:     $E^+ := \emptyset$;
 8:     $necessary := false$;
 9:     for all $u, t \in S$ do
10:         $d$ ←length of a shortest arching path between $u$ and $t$
            on $G_{i+1} = (V, E, c, l)$ that does not pass through $v$;
11:         if $d > c(u, v) + c(v, t)$ then
12:             if $(u, t)$ satisfies $P$ then
13:                 $E^+ := E^+ \cup \{(u, v)\}$;
14:             else
15:                 $necessary := true$;
16:                 break;
17:     if $\neg necessary$ then
18:         $E := E \cup E^+$;
19:         for all $(u, t) \in E^+$ do
20:             $c(u, t) := c(u, v) + c(v, t)$;
21:         $l(v) := i$;
22: return $G_{i+1} = (V, E, c, l)$;
23: function Partition($G = (V, E, c)$, $N$, $P$)
24: for all $v \in V$ do
25:     $l(v) := 1$;
26: $G_1 := (V, E, c, l)$;
27: for $i := 1, \ldots, N - 1$ do
28:     $G_{i+1} :=$ GetNextLevel($G_i$, $P$);
29: return $G_N$;
```

*Let $\Pi = (v_0, \ldots, v_k)$ be a path on G'. Π is an **ascending path** on G_N iff $\forall i \in \{0, \ldots, k-1\}$ $l(v_i) < l(v_{i+1})$. Π is a **descending path** on G_N iff $\forall i \in \{0, \ldots, k-1\}$ $l(v_i) > l(v_{i+1})$. Π is an **arching path** on G_N iff $\exists i, j \in \{0 \ldots k\}$ with $i \leq j$ such that $(v_0, \ldots, v_i)$ is an ascending path, $(v_j, \ldots, v_k)$ is a descending path, $l(v_i) = \cdots = l(v_j)$, and, if $l(v_i) < N$, then $i = j$ or $i + 1 = j$.*

N-Level Graphs are constructed by a process called partitioning. The key idea behind partitioning is to create a hierarchy among the vertices of a graph by assigning levels to them, such that there is always an arching path between any two vertices u and v on the N-Level Graph whose length is equal to the distance between u and v on the original graph.

Algorithm 2 Searching N-Level Graphs

```
 1: function FindArchingPath($G_N = (V, E, c, l)$, $u$, $v$)
 2: $V^A = \{t \in V$: there is an ascending path from $u$ to $t$ on $G_N\}$;
 3: $V^D = \{t \in V$: there is an ascending path from $v$ to $t$ on $G_N\}$;
 4: $V^M = \{t \in V : l(t) = N\}$;
 5: $V' = V^A \cup V^D \cup V_M$;
 6: $E' = \{(t, s) \in E : t, s \in V'\}$;
 7: $\Pi$ ← find a shortest path between $u$ and $v$ on $G' = (V', E', c)$;
 8: return $\Pi$;
 9: function FindPath($G$, $G_N$, $u$, $v$)
10: $\Pi$ ← FindArchingPath($G_N$, $u$, $v$);
11: $\pi$ ← empty path;
12: for all edges $(v_i, v_{i+1})$ in $\Pi$ (in order) do
13:     $\pi'$ ← find a shortest path between $v_i$ and $v_{i+1}$ on $G$;
14:     $\pi$ ← append $\pi'$ to $\pi$;
15: return $\pi$;
```

This allows a search to find a shortest path between any two vertices by finding an arching path between them while ignoring vertices that cannot be on an arching path between them. Figure 2 illustrates this idea with an example of a 3-Level Graph. The graph searched for a shortest path contains all highest-level vertices. Lower level vertices are usually used to connect the start and goal vertices to the highest-level vertices. An analogy can be made to traveling between two locations in different cities. One can use the streets of the first city to get to a highway, travel to the second city using the network of highways, and finally use the streets of the second city to get to one's destination. A shortest path does not always need to use highest-level vertices. For instance, one does not always need to use highways to travel between two locations in the same city. Such is the case with the shortest paths between K and L (or F and J) in Figure 2.

We distinguish two types of partitioning: *Basic partitioning* simply assigns levels to the vertices of a graph, but otherwise leaves the graph unchanged, which is the case in the example in Figure 2. *Advanced partitioning*, on the other hand, may add new edges to the graph, whose lengths are equal to the distances between the vertices they connect. This might allow partitioning to move more vertices into lower levels, which might increase the number of vertices ignored during search. For instance, by adding an extra edge of length 2 between B and G, we can decrease C's level to 1, in which case C does not need to be part of the graph shown in Figure 2(c). Such edges can be refined after a search if they are on the shortest path found by the search, by replacing them with a corresponding shortest path on the original graph. One needs to be careful when adding extra edges, as they can increase the branching factor of the graph and the memory required to store the graph. In the extreme case, partitioning can add edges between all pairs of vertices, in which case partitioning can move all vertices to the lowest level. We leave it to the user to provide a property P that all extra edges need to satisfy in order to get a good performance/memory trade-off. Figure 3 shows examples of N-Level Graphs constructed from SSGs using basic partitioning and a version of advanced partitioning that is allowed to add h-reachable edges.

Algorithm 1 shows how to construct N-Level Graphs from undirected graphs. Given an undirected graph G, a positive integer N, and a user-defined property P that extra edges need to satisfy, Function Partition(G,N,P) returns an N-Level Graph G_N of G, where all extra edges satisfy P. It obtains G_1 by setting the level of all vertices of G to 1 (Lines 24-26). It then constructs G_N by incrementally adding levels to G_1 (Lines 27-28). Function GetNextLevel(G_i, P) builds an $(i+1)$-Level Graph G_{i+1} from an i-Level Graph G_i by partitioning all ith level vertices of G_i into levels i and $i+1$, and is allowed to add extra edges to the graph that satisfy the user-specified edge property P. It first promotes all ith level vertices to level $i+1$, at which point the graph satisfies the properties of $(i+1)$-Level Graphs (Lines 2-4). It then iterates over all $(i+1)$st level vertices v and tries to demote them back to level i without violating the properties of $(i+1)$-Level Graphs (Lines 5-21). Demoting v to level i would cause any arching path with the sub-path (u, v, t), where u and t are neighbors of v with level i or higher, to no longer be an arching path, which might violate Property 1 of $(i+1)$-Level Graphs. In order to make sure that the only shortest arching path between any two vertices is not lost by demoting v, Function GetNextLevel(G_i,P) checks if v is necessary to optimally connect at least one pair of its neighbors u and t of level i or higher (Lines 6 and 9). v is not necessary to optimally connect its neighbors u and t iff (a) there is an arching path between u and t with length at most $c(u, v) + c(v, t)$ that does not pass through v (Lines 10-11), or (b) an edge can be added between u and t that satisfies P (Line 12). If (b) is satisfied but not (a), then the edge (u, t) is added to E^+, the set of extra edges to be added to the graph if v gets demoted to level i (Line 13). If v is not necessary to optimally connect any pair of its neighbors, then v is demoted to level i and all edges $(u, t) \in E^+$ are added to the graph with length $c(u, v) + c(v, t)$ (Lines 17-21).

Algorithm 2 shows how to search undirected graphs, using N-Level Graphs. Given an undirected graph G, an N-Level Graph G_N of G, a start vertex u and a goal vertex v, Function FindPath(G, G_N, u, v) finds a shortest arching path between u and v on G_N (Line 10) and then refines it to a shortest path on G (Lines 11-14). Function FindArchingPath(G_N, u, v) first constructs a set of vertices V' that contains only vertices of level N and any vertices that can be reached from u or v by ascending paths (Lines 2-5). Any vertex of a level lower than N that cannot be reached from u or v by ascending paths cannot be on an arching path between u and v and, therefore, is excluded from V'. Then, a subgraph G' of G_N, that includes all edges of G_N that connect vertices in V', is constructed and searched for a shortest path between u and v (Lines 6-7). The implementation details of Algorithm 2 are crucial for the performance of the search. We discuss them in the following sections, in the context of N-Level Subgoal Graphs.

Theoretical Results

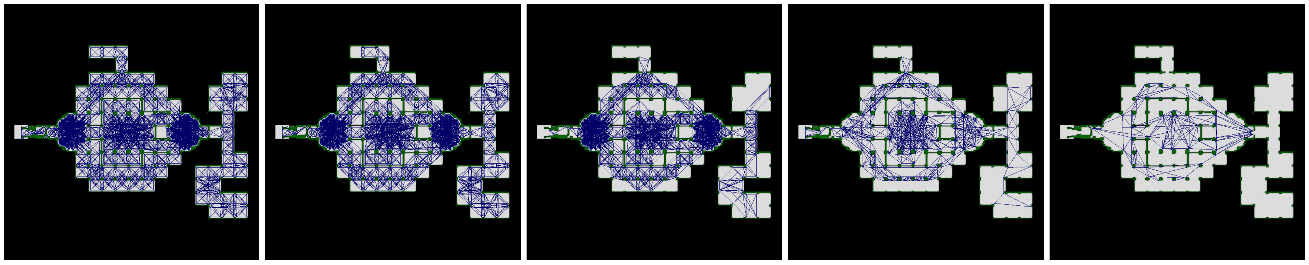
Theorem 1. *Function Partition(G, N, P) returns an N-Level Graph G_N of the undirected graph G, where N is a positive integer and P is a user-defined property that the extra edges of G_N need to satisfy.*

Proof. Let $(G_1, \ldots, G_N)$ be the series of graphs constructed by function Partition(G, N, P). Let $\mathcal{S}_i$ be the statement that G_i is an i-Level Graph of G. We show that $\mathcal{S}_i$ holds for all $i \in \{1, \ldots, N\}$ when function Partition(G, N, P) returns, by using induction on i to show that both properties of i-Level Graphs hold (Definition 3). G_1 is constructed from G on Lines 24-26. Since $E_1 = E$ and $c_1 = c$, Property 2 holds. Since $l_1(v) = 1$ for all vertices v, all paths on the graph are arching paths and, therefore, Property 1 holds. Consequently, $\mathcal{S}_1$ holds. To prove the induction step, we assume that $\mathcal{S}_{i-1}$ holds when function GetNextLevel(G_i, P) is called and show that $\mathcal{S}_i$ holds when it returns.

We show that Properties 1 and 2 hold when the function returns by using a second induction, this time on the number of times Line 5 is executed. Lines 2-4 increment the levels of all vertices with level i. Since E and c are unchanged, Property 2 holds when Line 5 is executed for the first time. Since the levels of all vertices with level i are incremented, any arching path remains an arching path and, therefore, Property 1 holds the first time Line 5 is executed. To prove the induction step, we assume that Properties 1 and 2 hold when Line 5 is executed for the jth time and show that they hold when Line 5 is executed for the $j+1$st time.

We start with Property 2 by showing that, when Line 20 is executed, $c(u, t) = d_G(u, t)$. Since $(u, t) \in E^+$ (Line 19), (u, v), $(v, t) \in E$ (Lines 6, 9, and 13) and, therefore, (u, v, t) is a path. Since $l(v) = i + 1$, (u, v, t) is an arching path (Definition 3). Since we assume that Property 2 holds when Line 5 is executed for the jth time, (u, v, t) is a shortest arching path between u and t that passes through v. Since $(u, t) \in E^+$ (Line 19), the condition on Line 11 is satisfied for u and t and, therefore, (u, v, t) is a shortest arching path between u and t. Since we assume that Property 1 holds when Line 5 is executed for the jth time, $d_G(u, t) = c(u, v) + c(v, t) = c(u, t)$. Therefore, Property 2 holds when Line 5 is executed for the $j+1$st time.

We prove by contradiction that Property 1 holds when Line 5 is executed for the $j+1$st time. Let s and r be arbitrary vertices. Since we assume that Property 1 holds when Line 5 is executed for the jth time, there must be an arching path Π between s and r with length $d_G(s, r)$. Assume that, when Line 5 is executed for the $j+1$st time, there is no longer an arching path between s and r with length $d_G(s, r)$. Since Lines 5-21 never remove edges, Π is still a path. Since



(a) A Simple Subgoal Graph (b) 2-Level (Basic partitioning) (c) 5-Level (Basic partitioning) (d) 2-Level (Advanced partitioning) (e) 5-Level (Advanced partitioning)

Figure 3: N-Level Subgoal Graphs constructed using basic partitioning and advanced partitioning that is allowed to add h-reachable edges. (Only the highest level vertices and the edges between them are shown.)

we have shown that Property 2 holds when Line 5 is executed for the $j + 1$st time, the length of Π is unchanged. Therefore, Π must no longer be an arching path. This can only happen if v, the vertex chosen on Line 5, lies on Π, and Line 21 reduces v's level from $i + 1$ to i. For the reduction in v's level to cause Π to no longer be an arching path, there must exist vertices v^- and v^+, preceding and succeeding v on Π, respectively, such that $l(v^-), l(v^+) \in \{i, i+1\}$ (Definition 3). Therefore, $v^-, v^+ \in S$ (Line 6), Lines 10-16 are executed for $u = v^-$ and $t = v^+$, and, either the condition on Line 11 is not satisfied or the condition on Line 12 is satisfied (because, otherwise, $necessary = true$ and Line 21 is not executed). Since Π is a shortest path between s and r, its sub-path $\Pi' = (v^-, v, v^+)$ must be a shortest path between v^- and v^+. If the condition on Line 11 is not satisfied, then there must be a path $\Pi'' = (v^-, \ldots, v^+)$, which does not pass through v, with length $c(v^-, v) + c(v, v^+)$, and that contains only vertices of level $i + 1$, with the possible exceptions of v^- and v^+. Therefore, we can replace Π' with Π'' in Π to get another arching path between s and r, with the same length as Π. Otherwise, if the condition on Line 12 is satisfied, we add the edge (v^-, v^+) with length $c(v^-, v) + c(v, v^+)$ to the graph. Therefore, we can replace Π' with edge (v^-, v^+) in Π to get another arching path between s and r, with the same length as Π. This contradicts our initial assumption, and, therefore, Property 1 holds when Line 5 is executed for the $j + 1$st time. Therefore, S_i holds when the function GetNextLevel(G, P) returns and, consequently, when function Partition(G,N,P) returns. $\square$

Theorem 2. *Function FindPath(G, G_N, u, v) returns a shortest path from u to v on G, where G is an undirected graph, G_N is an N-Level Graph of G, and u, v are vertices on G.*

Proof. Let $\Pi = (v_0 = u, \ldots, v_k = v)$ be an arching path on G_N with length $d_G(u, v)$ (Definition 3). We show that Π is contained in G', constructed on Lines 2-6. Let $\Pi^A = (v_0, \ldots, v_i)$, $\Pi^D = (v_j, \ldots, v_k)$ and $\Pi^M = (v_i, \ldots, v_j)$, where Π^A is an ascending path, Π^D is a descending path, and Π^M is the middle part where $l(v_i) = \cdots = l(v_j)$. Any vertex v_m on Π^A is in V^A, since $(v_0, \ldots, v_m)$ is an ascending path from v_0 to v_m (Line 2). By a similar argument, any vertex on Π^D is in V^D. Let n denote the level of the vertices on Π^M. If $n < N$, then $l(v_i) = l(v_j) < N$ and, therefore, $i = j$ or $i + 1 = j$ (Definition 3). This means that the only vertices on Π^M are v_i and v_j, which are already in V^A and V^D, respectively. If $n = N$, then all vertices on Π^M are in V^M (Line 4). Therefore, all vertices on Π are in V' (Line 5). Since E' contains all edges between vertices in V', the graph $G' = (V', E', c)$ contains all edges on Π. Therefore, the search must find Π or another arching path between u and v with length $d_G(u, v)$. Any edge on Π corresponds to a shortest path on G (Definition 3) and, therefore, Lines 9-15 find and return a shortest path on G. $\square$

N-Level Subgoal Graphs

In this section, we show that SSGs and TSGs are instances of N-Level Graphs and provide implementation details on search with N-Level Subgoal Graphs.

Consider any grid graph. Let $G_2 = (V, E, c, l)$, such that: V is the set of unblocked cells; E is the set of edges between all direct-h-reachable cells; $\forall (u, v) \in E\ c(u, v) = h(u, v)$; and, $\forall v \in V$, if v is a subgoal, $l(v) = 2$, otherwise, $l(v) = 1$. The distance between two direct-h-reachable cells u and v is $h(u, v)$. Therefore, G_2 satisfies Property 2 of 2-Level Graphs. (Uras, Koenig, and Hernández 2013) shows that, between any two cells u and v, there is a shortest path that can be divided into segments between direct-h-reachable subgoals (plus u and v). Such paths are arching paths in G_2 since E contains edges between all direct-h-reachable subgoals and all subgoals have level 2. Therefore, G_2 satisfies Property 1 of 2-Level Graphs. Consequently, G_2 is a 2-Level Graph of the grid graph.

The memory required to store all edges between direct-h-reachable cells can be very large. As an implementation trick, one can discard all edges between level 1 vertices and check before a search if the start and goal vertices are direct-h-reachable. This works because the start and goal vertices are the only vertices with level 1 in the graph that is searched. Furthermore, one can discard all edges between level 1 and level 2 vertices and reconstruct the necessary edges before a search by identifying the direct-h-reachable subgoals from the start and goal vertices. This allows one to store only the edges between subgoals, which is exactly what SSGs do. Therefore, *SSGs are 2-Level Graphs of grid*

	Runtime per Instance (ms)					Average Level		Partition Time (ms)		
	A*	S	TL	S_N	S_N^+	S_N	S_N^+	TL	S_N	S_N^+
bg512	2.69	0.07	0.05	0.06	0.05	26.17	7.59	267	398	284
DAO	5.45	0.36	0.13	0.26	0.08	27.70	10.06	229	837	257
starcraft	24.87	0.94	0.30	0.63	0.18	71.12	14.43	8657	59334	8912
wc3maps512	5.52	0.08	0.06	0.07	0.06	28.44	9.22	313	376	379
maze1	15.73	3.84	3.39	0.47	0.45	1263.60	1138.40	45	22697	21230
maze2	29.95	2.53	1.78	0.33	0.30	832.20	675.60	48	9536	7864
maze4	42.99	1.04	0.49	0.21	0.18	407.80	179.90	36	1654	697
maze8	52.65	0.39	0.23	0.16	0.15	228.60	101.30	12	299	140
maze16	59.24	0.18	0.14	0.12	0.12	105.70	47.20	4	47	29
maze32	59.23	0.09	0.10	0.08	0.09	39.00	17.60	2	5	6
random10	3.93	1.52	1.44	1.52	1.30	7.60	13.70	875	1900	6430
random15	6.32	2.70	2.40	2.67	2.17	7.80	11.10	685	2003	4268
random20	8.37	3.69	3.12	3.56	2.75	9.70	12.20	533	2332	3706
random25	10.00	4.40	3.57	4.03	2.87	12.20	13.50	421	2554	3172
random30	11.15	4.77	3.58	3.98	2.57	17.60	14.30	324	2984	2509
random35	12.65	5.27	3.61	3.66	2.09	27.30	22.70	244	3340	2647
random40	12.16	4.84	3.05	2.51	1.22	39.50	32.30	126	2281	1645
room8	15.37	0.65	0.57	0.64	0.54	8.40	7.40	35	164	206
room16	16.75	0.17	0.16	0.17	0.15	6.90	7.10	9	33	54
room32	19.60	0.07	0.07	0.07	0.06	7.30	5.70	3	8	13
room64	23.61	0.04	0.05	0.04	0.04	6.40	6.70	1	2	5

Table 1: Results of different subgoal graphs

graphs.

The properties of TSGs satisfy the properties of 2-Level Graphs of SSGs. Algorithms 1 and 2 allow us to construct N-Level Graphs of SSGs, called N-Level Subgoal Graphs, and search with them to find shortest paths on the SSG, which can then be refined to shortest paths on grids. Edges are stored as directed edges in both directions, with the following exception: Between different-level vertices, one adds an *ascending edge* from the lower-level vertex to the higher-level vertex. This allows one to construct the graph to search much faster, by only following the ascending edges from the goal vertex and then reversing their directions to form descending paths to the goal vertex. A shortest path on the resulting graph is found by a forward A* search using a binary heap as priority queue. The search ignores edges that connect vertices of the same level, unless they connect level N vertices or lead to vertices that have descending paths to the goal vertex.

Experimental Results

We compare A*, S, TL, S_N and S_N^+, where S and TL are the implementations of SSGs and TSGs used in (Uras, Koenig, and Hernández 2013) and S_N, S_N^+ are our implementations of N-Level Subgoal Graphs. TL is a state-of-the art algorithm and was one of the undominated entries in the Grid-Based Path Planning Competition (GPPC) in 2012[1]. It trades optimality for reduced memory requirements and improved search performance, by discarding all edges between local subgoals. In GPPC 2012, the average suboptimality of TL was no more than 1%. S_N is constructed from an SSG using simple partitioning, whereas S_N^+ is constructed from an SSG using a version of advanced partitioning that is allowed to add h-reachable edges. For both variants, function GetNextLevel(G_i, P) is called until the highest level vertices can no longer be partitioned.

The experiments are run on a PC with a dual-core 3.2GHz Intel Xeon CPU and 2GB of RAM. We compare the methods on different map types[2], 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. For each map type, Table 1 shows the average runtime per instance for all methods, the average levels for S_N and S_N^+, and the average preprocessing time for TL, S_N, and S_N^+ needed for partitioning. The memory requirements are very similar and thus not reported.

The results show that, in general, S_N^+ is the fastest method, followed by TL, S_N, S, and finally A*. S_N^+ is faster than TL by a factor of 1.6 on Dragon Age: Origins and StarCraft maps, a factor of 7.5 on maze maps with corridor width 1, and a factor of 2.5 on maps with 40% randomly blocked cells. Its performance is comparable to TL on other game maps and room maps. It is faster than A* by a factor of 193 on StarCraft maps. S_N^+ is generally faster than S_N, especially on Dragon Age: Origins and StarCraft maps (by a factor of 3.5 and 3.2, respectively), which demonstrates the benefits of adding extra edges during partitioning. S_N is generally slower than TL, except on maze maps. TL is faster than S_N by a factor of 2.1 on StarCraft maps. On the other hand, S_N is faster than TL by a factor of 7.2 on maze maps with corridor width 1. These results show that the structure of the graph can have a significant impact on the performance increase obtained by adding extra edges during partitioning and by partitioning the vertices into more levels. For instance, StarCraft maps have lots of diagonal obstacles that result in a large number of subgoals. Many of these can be partitioned into the lowest level, by adding h-reachable edges between them. On the other hand, mazes have lots of bending corridors, which limits the number of h-reachable edges that can be added.

Conclusions

N-Level Graphs are constructed from undirected graphs by partitioning the vertices into levels to create a hierarchy, which allows searching for shortest paths while ignoring parts of the graph. We gave a formal definition of N-Level Graphs and provided algorithms for constructing and searching them. We proved the correctness of these algorithms and demonstrated their effectiveness on Subgoal Graphs, by improving the state-of-the-art of path planning on grids. Future research includes application of these methods to different domains, investigation of whether stopping partitioning early increases performance, exploration of new techniques for partitioning and search, and generalization of these methods to directed graphs.

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

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

Acknowledgments

Our research was supported by NSF under grant number IIS-1319966 and ONR under grant number N00014-09-1-1031. 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

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 9–14.

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

Cazenave, T. 2006. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games*, 27–33.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*.

Goldenberg, M.; Felner, A.; Sturtevant, N.; and Schaeffer, J. 2010. Portal-based true-distance heuristics for path finding. In *Proceedings of the Third Annual Symposium on Combinatorial Search*.

Holte, R. C.; Drummond, C.; Perez, M. B.; Zimmer, R. M.; and Macdonald, A. J. 1994. Searching with abstractions: A unifying framework and new high-performance algorithm. In *Proceedings of the 10th Canadian Conference on Artificial Intelligence*.

Leighton, M.; Ruml, W.; and Holte, R. C. 2008. Faster optimal and suboptimal hierarchical search. In *Proceedings of the First International Symposium on Combinatorial Search*.

Pochter, N.; Zohar, A.; Rosenschein, J.; and Felner, A. 2010. Search space reduction using swamp hierarchies. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 155–160.

Storandt, S. 2013. Contraction hierarchies on grid graphs. In *Proceedings of the 36th Annual Conference on Artificial Intelligence*.

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *Proceedings of the Twentieth AAAI Conference on Artificial Intelligence*, 1392–1397.

Sturtevant, N.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, 609–614.

Uras, T.; Koenig, S.; and Hernández, C. 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling*.