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

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. 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 over the subgoal graph that ignores local subgoals that are not relevant to the search, which significantly reduces the size of the graph being searched. This paper is an abstract of (Uras and Koenig 2014), which generalizes this partitioning process to any undirected graph and shows that it can be recursively applied to generate more than two levels, which reduces the size of the graph being searched even further. We call these graphs N-Level Graphs.

N-Level Graphs

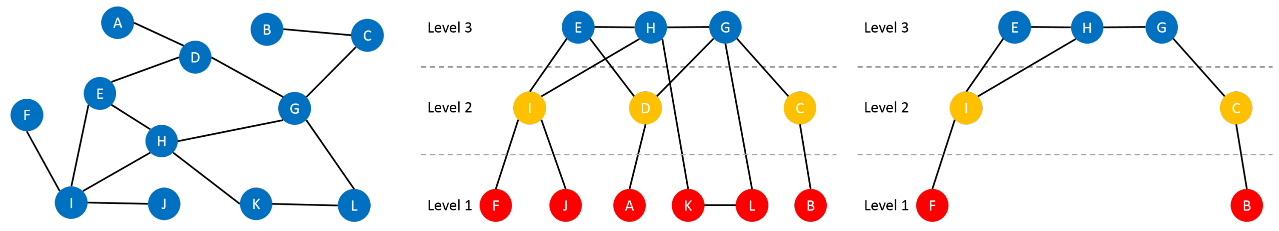
Graphs can contain vertices that are not necessary to optimally connect other vertices of the graph. For instance, vertex A in Figure 1(a) cannot appear on any shortest path unless it is the start or the goal vertex. Vertex K can appear on a shortest path between H and L, but if it is removed from the graph, there is an alternative shortest path between H and L through G. Therefore, unless K is the start or the goal vertex, it can safely be excluded from the search without increasing the length of the resulting path. More generally, we can partition the vertices of a graph into two levels, levels 1 and 2, that satisfy the following property: Between any two vertices of the graph, there is a shortest path that uses only level 2 vertices, the start, and the goal. This property allows us to search a smaller graph by ignoring level 1 vertices, except for the start and the goal, while still guaranteeing optimality.

We can take this idea a little further by partitioning the level 2 vertices into levels 2 and 3 with the property that, between any two vertices of level 2 or 3, there is a shortest path that uses only level 3 vertices, the start, and the goal. To find a shortest path between any two vertices of the original graph, we can search an even smaller graph that contains only level 3 vertices, the start, the goal, and any level 2 vertices that are neighbors of the start or the goal. We can keep adding levels to the graph by moving some of the highest level vertices into a new level, thereby reducing the size of the graph to be searched, until no more partitioning is possible. This is the main idea behind N-Level Graphs, which is illustrated in Figure 1.

We call the partitioning method described above *simple partitioning*, which simply assigns levels to the vertices of a graph. A more sophisticated partitioning method, which we call *advanced partitioning*, is allowed to add new edges to the graph whose lengths are equal to the distances between the vertices they connect. Advanced partitioning can move more vertices into lower levels, which might increase the number of vertices ignored during search. For instance, if we add an extra edge of length 2 between B and G, C can be moved to level 1, in which case C does not need to be part of the graph shown in Figure 1(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 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 level 1. We leave it to the user to provide a property P that all extra edges need to satisfy in order to obtain a good performance/memory trade-off. A formal description of N-Level Graphs, along with algorithms to construct and search them, are provided in (Uras and Koenig 2014).

Experimental Results

We compare A*, S, TL, S_N and S_N^+, where S and TL are the implementations of Simple Subgoal Graphs (SSGs) and Two-Level Subgoal Graphs used in (Uras, Koenig, and Hernández 2013) and S_N and S_N^+ are N-Level Graphs of SSGs. TL is a state-of-the art algorithm and was one of the non-dominated entries in the Grid-Based Path Planning



(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 1: Idea behind N-Level Graphs.

	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

Competition (GPPC) in 2012[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 (defined in (Uras, Koenig, and Hernández 2013)). We compare the methods on different map types, all of which are available from Nathan Sturtevant's repository[2]. For each map type, Table 1 shows the average runtime per instance for all methods, the average number of levels for S_N and S_N^+, and the average preprocessing time for TL, S_N, and S_N^+ needed for partitioning.

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 factors 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 them 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. The main idea of using hierarchies to ignore parts of the graph during search is similar to Multi-Level Graphs, but the means of achieving this is very different. For instance, for N-Level Graphs, one needs to specify an edge connection strategy whereas, for Multi-Level Graphs, one needs to specify the levels of the vertices. We demonstrate the effectiveness of N-Level Graphs on grids 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, and exploration of new techniques for partitioning and search.

Acknowledgments

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

References

Uras, T., and Koenig, S. 2014. Identifying hierarchies for fast optimal search. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*.

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*.

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

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