Grid pathfinding optimizations

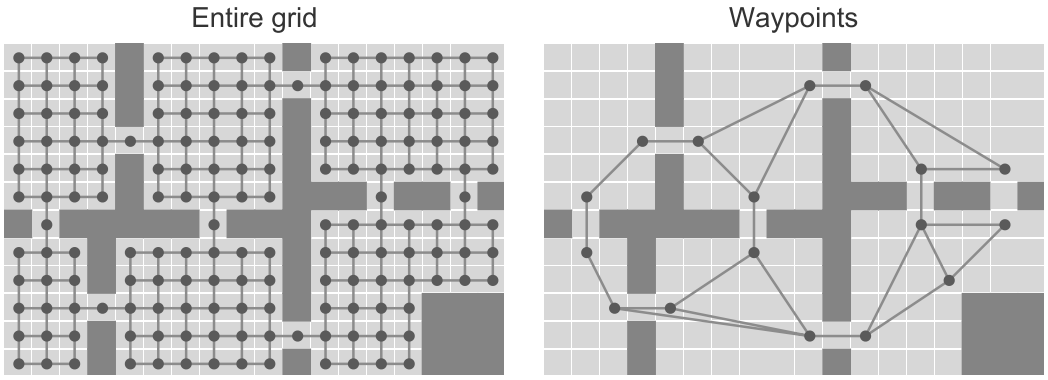
from Red Blob Games

Pathfinding algorithms like A* and Dijkstra's Algorithm work on graphs. To use them on a grid, we represent grids with graphs. For most grid-based maps, it works great. However, for those projects where you need more performance, there are a number of optimizations to consider.

I've not needed any of these optimizations in my own projects.

Change the map representation

The first thing to consider is whether you can use a **simpler graph for pathfinding**. Consider the same grid map with two different pathfinding graphs:

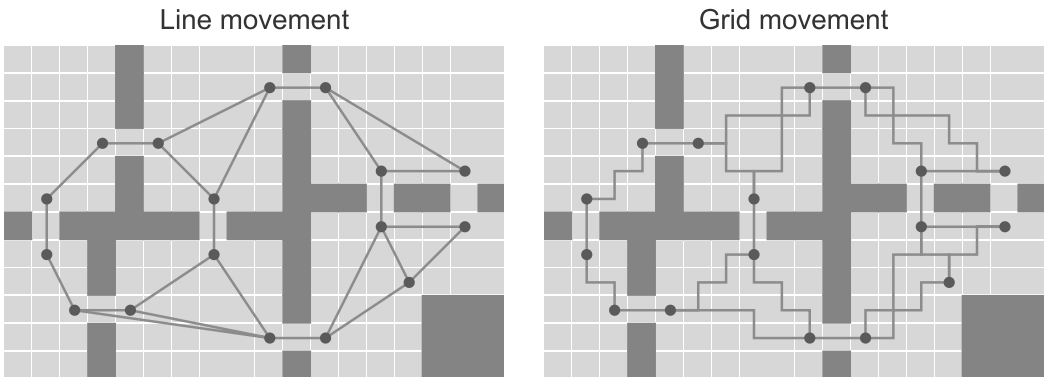


In general, the fewer nodes A* has to process, the faster it will run. Common ways to reduce the graph size:

- **Waypoints** are the key decision points where you might have to change direction. In the above diagram, I mark places where you have to go around a corner or wall. I call these "exterior corners".

- **Navigation meshes** are the walkable regions of the map. Regions can be any size and shape.
- **Hierarchical approaches** have coarse and fine level map representations. An approximate path is found with the coarse level, and then refined with the fine level.
- **Quad trees** use square regions of various sizes to represent the map. Large open areas can be represented by a few squares. Irregular edges can be represented by many small squares. See this[1]; please send me references.

Note that movement can be on grid even if your map representation is not. These two have the same set of waypoints:



There are multiple algorithms in computer science literature that can improve pathfinding for grid maps with grid ("Lɪ") movement. This paper[2] [PDF] is one example. See lɪ-path-finder[3] for a **fast implementation** and also more references to papers. There are also some approaches from the game developer community; see this paper[4] [PDF] for descriptions of those techniques (including JPS+).

Harness the grid structure

If you can't use an alternate graph structure for pathfinding, there are optimizations that can be applied to pathfinding running on a grid. Grids contain additional structure that graph search algorithms don't take advantage of.

For example, if you move east one space on the grid, it's likely that moving east again is going to be a good move. Graph algorithms however don't know about "east" or that two edges can be in the same direction. Another example: going north then east is usually the same as going east then north. Graph algorithms don't know this, and have to try both. These two examples suggest that there's additional efficiency to be gained by not using pathfinding directly on the grid.

If movement is along grid nodes and you have large areas of uniformly weighted nodes, **Jump Point Search** jumps ahead in the grid instead of processing nodes one by one. Read about the algorithm here[5] and here[6]. Read a discussion of JPS pros and cons here[7].

If movement is *not* along grid nodes and you are pathfinding on a grid, you'll want to straighten the paths[8]. However, the straightened paths aren't guaranteed to be the shortest. To have the pathfinder directly analyze the straightened paths, look at the **Theta* Algorithm**, described here[9].

Traverse nodes faster

Another generality in weighted graphs is the weights (varying movement costs). Dijkstra's Algorithm and A* analyze the weights when finding the paths. Many situations don't call for weights, and set them to 1 everywhere. This suggests that there's additional efficiency gains that we could get if you have uniform movement costs in your map. For example, Breadth First Search is a better choice than Dijkstra's Algorithm if you don't need weights.

If your movement costs are uniform, or in a narrow range, consider these:

- Use a **priority queue with bins** to take advantage of the limited range of movement costs. For example, if using Dijkstra's Algorithm, if your current node has priority p, the OPEN set will contain priorities from p to $p+E$, where E is the maximum movement cost. You don't need a priority queue that supports arbitrary priorities. You can use binning to create a more efficient data structure, the same way that radix sort can be faster than quicksort.
- Use a **partitioned priority queue**. The nodes close to the top need to be sorted, but the nodes that aren't likely to be expanded soon don't need to be. A HOT Queue (Heap On Top) divides the OPEN set into those nodes that will be expanded soon (these get sorted accurately) and those that will not (these get binned but not sorted). I believe a HOT Queue is probably overkill for most games, but partitioning the queue may give most of the benefit.
- Consider **batch processing** the nodes in the priority queue. In a game, we rarely need the absolute best path. We just need a reasonably good path. We don't need to process every node in exact order, only in approximate order. Fringe Search[10] uses batch operations to make processing the OPEN set much faster. Many nodes are processed at once without inserting new nodes one by one into the queue. Although Fringe Search may be overkill for most games, batch processing the queue can be a big win. A bucket-based priority queue likely gives similar gains.

Improve the heuristic

The purpose of the heuristic is to give an estimate of the length of the shortest path. The closer the heuristic is to the actual shortest path length, the faster A* runs. When the heuristic is 0, A* becomes Dijkstra's Algorithm. When the heuristic is equal to the shortest path length, A* immediately finds the shortest path and doesn't have to explore other areas. When the heuristic is larger than the shortest path length, A* no longer guarantees shortest paths. For grids, we typically use **distance** as the heuristic. It's greater than 0, but less than the shortest path length, so it gives us shortest paths, but not the best speed.

For maps in general, not only grid maps, we can analyze the map to generate better heuristics. Pick a set of *pivot points* and then find the shortest paths between them. The path length between pivot points can then be used in the heuristic to calculate a better estimate of the shortest path length, with significant speedups possible. For more, read this paper[11] [PDF]. I implemented this myself and got a significant speedup with under 10 lines of code.

Summary

---

1. Grids can be viewed as a special case of a graph.
2. Grids contain additional structure not in all graphs. Standard graph search algorithms don't take advantage of that structure.
3. Often our maps have uniform movement costs, leading to graphs with unweighted edges. Many standard graph search algorithms spend time analyzing the weights.
4. We use distance for the A* heuristic. It's easy to understand and calculate, but it's not the best heuristic. Most maps can be preprocessed to generate a better heuristic.
5. Often the map is a grid but object movement isn't limited to a grid. Standard graph search algorithms don't take this into account.

Most of the time, the standard graph search algorithms are all you need. But if you need more, I hope this list gives you some ideas. Which approaches work best will depend on your game, size and style of map, number of units pathfinding, and so on. I'd love to hear what you've used and how well it worked.

Email me redblobgames@gmail.com, or tweet @redblobgames, or comment:

Endnotes

[1]: https://www.youtube.com/watch?v=95aHGzzNCY8

[2]: http://www.cs.uu.nl/research/techreps/repo/CS-1986/1986-20.pdf

[3]: https://mikolalysenko.github.io/l1-path-finder/www
[4]: http://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-icaps14.pdf
[5]: http://harablog.wordpress.com/2011/09/07/jump-point-search/
[6]: http://aigamedev.com/open/tutorial/symmetry-in-pathfinding/
[7]: http://www.reddit.com/r/programming/comments/1es39b/speed_up_a_using_jump_point_search_explained/
[8]: http://www.gamasutra.com/view/feature/3096/toward_more_realistic_pathfinding.php
[9]: http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/
[10]: https://en.wikipedia.org/wiki/Fringe_search
[11]: http://research.microsoft.com/pubs/64511/tr-2004-24.pdf

---

RSS Feed

Created 2 Mar 2014 with D3.js and TypeScript; Last modified: 24 Jul 2019