Variants of A*

From Amit's Thoughts on Pathfinding

Beam search

In the main A* loop, the OPEN set stores all the nodes that may need to be searched to find a path. The **Beam Search** is a variation of A* that places a limit on the size of the OPEN set. If the set becomes too large, the node with the worst chances of giving a good path is dropped. One drawback is that you have to keep your set sorted to do this, which limits the kinds of data structures you'd choose.

Iterative deepening

Iterative Deepening is an approach used in many AI algorithms to start with an approximate answer, then make it more accurate. The name comes from game tree searches, where you look some number of moves ahead (for example, in Chess). You can try to deepen the tree by looking ahead more moves. Once your answer doesn't change or improve much, you assume that you have a pretty good answer, and it won't improve when you try to make it more accurate again. In IDA*, the "depth" is a cutoff for `f` values. When the `f` value is too large, the node won't even be considered (*i.e.,* it won't be added to the OPEN set). The first time through you process very few nodes. Each subsequent pass, you increase the number of nodes you visit. If you find that the path improves, then you continue to increase the cutoff; otherwise, you can stop. For more details, read these lecture nodes on IDA*.

I personally don't see much need for IDA* for finding paths on game maps. ID algorithms tend to increase computation time while reducing memory requirements. In map pathfinding, however, the "nodes" are very small—they are simply coordinates. I don't see a big win from not storing those nodes.

Dynamic weighting

With dynamic weighting, you assume that at the beginning of your search, it's more important to get (anywhere) quickly; at the end of the search, it's more important to get to the goal.

```
f(p) = g(p) + w(p) * h(p)
```

There is a weight (`w >= 1`) associated with the heuristic. As you get closer to the goal, you decrease the weight; this decreases the importance of the heuristic, and increases the relative importance of the actual cost of the path.

Bandwidth search

There are two properties about *Bandwidth Search* that some people may find useful. This variation assumes that `h` is an *overestimate*, but that it doesn't overestimate by more than some number `e`. If this is the case in your search, then the path you get will have a cost that doesn't exceed the best path's cost by more than `e`. Once again, the better you make your heuristic, the better your solution will be.

Another property you get is that if you can drop some nodes in the OPEN set. Whenever `h+d` is greater then the true cost of the path (for some `d`), you can drop any node that has an `f` value that's at least `e+d` higher than the `f` value of the best node in OPEN. This is a strange property. You have a "band" of good values for `f`; everything outside this band can be dropped, because there is a guarantee that it will not be on the best path.

Curiously, you can use different heuristics for the two properties, and things still work out. You can use one heuristic to guarantee that your path isn't too bad, and another one to determine what to drop in the OPEN set.

Note: When I wrote this in 1997, Bandwidth search looked potentially useful, but I've never used it and I don't see much written about it in the game industry, so I will probably remove this section. You can search Google for more information, especially from textbooks.

Bidirectional search

Instead of searching from the start to the finish, you can start two searches in parallel—one from start to finish, and one from finish to start. When they meet, you should have a good path.

It's a good idea that will help in some situations. The idea behind bidirectional searches is that searching results in a "tree" that fans out over the map. A big tree is much worse than two small trees, so it's better to have two small search trees.

The *front-to-front* variation links the two searches together. Instead of choosing the best forward-search node—`g(start,x) + h(x,goal)`—or the best backward-search node—`g(y,goal) + h(start,y)`—this algorithm chooses a pair of nodes with the best `g(start,x) + h(x,y) + g(y,goal)`.

The *retargeting* approach abandons simultaneous searches in the forward and backward directions. Instead, it performs a forward search for a short time, chooses the best forward candidate, and then performs a backward search—not to the starting point, but to that candidate. After a while, it chooses a best backward candidate and performs a forward search from the best forward candidate to the best backward candidate. This process continues until the two candidates are the same point.

Holte, Felner, Sharon, Sturtevant's 2016 paper *Front-to-End Bidirectional Heuristic Search with Near-Optimal Node Expansions* is a recent result with a near-optimal bidirectional variant of A*. Pijls and Post's 2009 paper *Yet another bidirectional algorithm for shortest paths* proposes an unbalanced bidirectional A* that runs faster than balanced bidirectional search.

Dynamic A* and Lifelong Planning A*

There are variants of A* that allow for changes to the world after the initial path is computed. D* is intended for use when you don't have complete information. If you don't have all the information, A* can make mistakes; D*'s contribution is that it can correct those mistakes without taking much time. LPA* is intended for use when the costs are changing. With A*, the path may be invalidated by changes to the map; LPA* can re-use previous A* computations to produce a new path.

However, both D* and LPA* require a lot of space—essentially you run A* and keep around its internal information (`OPEN/CLOSED` sets, path tree, `g` values), and then when the map changes, D* or LPA* will tell you if you need to adjust your path to take into account the map changes.

For a game with lots of moving units, you usually don't want to keep all that information around, so D* and LPA* aren't applicable. They were designed for robotics, where there is only one robot—you don't need to reuse the memory for some other robot's path. If your game has only one or a small number of units, you may want to investigate D* or LPA*.

- Overview of D*
- D* Paper 1

- D* Paper 2
- Lifelong planning overview
- Lifelong planning paper (PDF)
- Lifelong planning A* applet

Jump Point Search

Many of the techniques for speeding up A* are really about reducing the number of nodes. In a square grid with uniform costs it's quite a waste to look at all the individual grid spaces one at a time. One approach is to build a graph of key points (such as corners) and use that for pathfinding. However, you don't want to precompute a waypoint graph, look at Jump Point Search, a variant of A* that can skip ahead on square grids. When considering children of the current node for possible inclusion in the OPEN set, Jump Point Search skips ahead to faraway nodes that are visible from the current node. Each step is more expensive but there are fewer of them, reducing the number of nodes in the OPEN set. See this blog post for details, this blog post for a nice visual explanation, and this discussion on reddit of pros and cons.

Also see Rectangular Symmetry Reduction, which analyzes the map and embeds jumps into the graph itself. Both techniques were developed for square grids. Here's the algorithm extended for hexagonal grids.

Theta*

Sometimes grids are used for pathfinding because the map is made on a grid, not because you actually want movement on a grid. A* would run faster and produce better paths if given a graph of key points (such as corners) instead of the grid. However if you don't want to precompute the graph of corners, you can use Theta*, a variant of A* that runs on square grids, to find paths that don't strictly follow the grid. When building *parent* pointers, Theta* will point directly to an ancestor if there's a line of sight to that node, and skips the nodes in between. Unlike path smoothing, which straightens out paths after they're found by A*, Theta* can analyze those paths as part of the A* process. This can lead to shorter paths than postprocessing a grid path into an any-angle path. This article is a reasonable introduction to the algorithm; also see Lazy Theta*.

The ideas from Theta* likely can be applied to navigation meshes as well.

Also see Block A*, which claims to be much faster than Theta* by using a hierarchical approach.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 25 Sep 2018