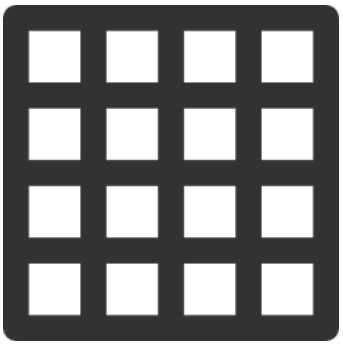
Best practices for A* on grids

Chris Rayner

January 15, 2017



Description

Here are some simple notes on ways to improve A*, focusing on pathfinding on four- and eight-connected grids. This document is pitched at hobbyists and anyone looking for ideas on how to make an existing implementation a bit faster.

Check out the source on github for some example code in C++.

Contributing

If you have any (much appreciated) corrections or contributions, feel free to get in touch or simply make a pull request.

Preliminaries

Skip this if you're very familiar with A*. If you're not, you might want to check out the resources section at the end of the document.

Forgoing a complete description, recall that A* is essentially a loop that expands a list of *open* states that reach toward the goal. Each iteration

expands the *open list* with the neighbors of an already open state. The open state `i` that gets chosen is one with the lowest `f` cost,

```
f_i(s,g) = g(s,i) + h(i,g)
```

This `f` cost is an estimate of the cost of an optimal path between `s` and `g` that goes through `i`. Here `g(s,i)` is the *ground truth* distance to get from the start state `s` to open state `i` (the value of which A* knows exactly). `h(i,g)` is a cheaply computed heuristic estimate for the distance from `i` to the goal state `g`.

Implement the open list using a binary heap

. . . and implement the heap using an array See example code.

Depending on the implementation language, this isn't as important for small grids (on the order of a couple thousand cells in a C++ implementation). On larger grids, this can cut execution time in half – or better.

Note it is common for multiple states on the open list have the lowest `f` cost, so there is an opportunity to add smart tie-breaking on larger `g` values. This makes A* focus on deep solutions rather than a breadth of shallow solutions. My Ph.D. co-supervisor Nathan Sturtevant has a video demonstrating this.

Avoid floating point arithmetic

Prefer integral data types wherever possible. This is not only faster but helps to avoid the numerical imprecision that can confuse debugging attempts.

Use a non-overestimating heuristic

Heuristics that don't overestimate are called *admissible*. A* recovers an optimal path when it's using an admissible heuristic. A good, admissible grid heuristic is the "distance" between two cells assuming no obstacles.

On a 4-connected grid

The distance between two cells on a 4-connected grid, assuming no obstacles, is the **rectilinear** (or **L1-norm** or **Manhattan**) distance:

```
h(i,j) = C * (x + y)
```

where `x` and `y` are absolute distances between grid cells along the `x` and `y` axes and `C` is the cost to take a cardinal move, which may as well be `1`.

On an 8-connected grid

When pathfinding on an 8-connected grid, use the **octile** heuristic:

```
h(i, j) = C * x + B * y if x > y
          C * y + B * x else
```

where `B = D - C` with `C` being the cost to take a cardinal move and `D` being the cost to take a diagonal move. See example code.

Note the octile heuristic can be written without a conditional (albeit with an absolute value), which may help improve instruction level parallelism:

```
h(i, j) = (E * abs(x - y) + D * (x + y)) / 2
```

where `E = 2 * C - D`. You can see how to simplify this further if both `D` and `E` are even numbers. See example code.

Avoid same-cost diagonal and cardinal moves

. . . for instance, by having diagonal and cardinal moves both cost `1`. This greatly increases the number of optimal paths and subsequently the average number of nodes A* expands.

Modified move costs

On an 8-connected grid, the cost of a diagonal move (`D`) relative to the cost of a cardinal move (`C`) affects the appearance of the resulting paths. If a diagonal move costs *less* than a cardinal move, A* prefers zigzagging paths. If a diagonal move costs more than *two* cardinal moves, A* prefers rectilinear paths.

Paths tend to look best when the costs lie between these two extremes, but some algorithms (like Fringe Search) are hugely sensitive to changes in cost structure. You might try one of the following.

Diagonal: 99/Cardinal: 70

If you prefer a diagonal move to cost `sqrt(2)` relative to a cardinal move, try using `D = 99` for the cost of a diagonal move and `C = 70` for the cost of a cardinal move. This close approximation helps to avoid floating point arithmetic.

Diagonal: 3/Cardinal: 2

If your heuristic can return fractional values, `D = 3/C = 2` gives you the ability to take the *ceiling* of those fractional heuristics since all distances must be integral. This is still reasonably close to a `D/C` ratio of `sqrt(2)`, and again you avoid floating point arithmetic.

Nathan Sturtevant showed me this when we wrote Euclidean Heuristic Optimization (Rayner, Bowling, Sturtevant), and it made a noticeable difference.

Diagonal: 99/Cardinal: 50

`D = 99/C = 50` gives something very close to a rectilinear cost structure. On average this can keep the size of the open list smaller, but it can also increase node expansions, depending on the layout of the grid. This can be beneficial in many cases, but you should test it out before you use it.

Resources

Additional resources that might also be helpful:

Patrick Lester's A* for beginners A good starting point.

Nathan Sturtevant's movingai.com Benchmark problems, tutorials, and videos covering fundamental and advanced topics.

Dijkstra Maps Dijkstra Maps are also known as "differential heuristics", "ALT heuristics", or "Lipschitz embeddings". We looked at smart ways to set these heuristics up in Subset Selection of Search Heuristics (Rayner, Sturtevant, Bowling) but this article describes some extremely novel ways to use these mappings to control game entities.

Variants of A* Amit Patel lists some alternatives to A* search.

A* on Wikipedia Wikipedia gives a thorough description of A*.