Best practices for A* on grids

Chris Rayner

January 28, 2017

Description

This document describes ways to improve A*, focusing on pathfinding on four- and eight-connected grids. It's pitched at hobbyists and anyone looking for ways 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

Forgoing a complete description, recall that A* is essentially a loop that expands a list of *open* states that reach toward a goal state. (For more detail, visit the resources at the end of the document.)

Each iteration of the A* loop expands the *open list* with the neighbors of a state already on the open list. The open state `i` that gets chosen is one with the lowest `f` cost,

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

which is an estimate of the cost of a path starting at `s`, going through `i`, and arriving at `j`. Here `g(s,i)` is the *ground truth* cost to get from `s` to the open

state `i` (which A* will have determined by this point). `h(i,j)` is a cheaply computed *heuristic* estimate for the cost to get from `i` to the goal `j`.

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.

Break ties in favor of path depth

It is common for more than one state on the open list to have the lowest `f` cost. When this is the case it's better to make A* focus on deep solutions rather than a breadth of shallow solutions by tie-breaking on larger `g` values. My Ph.D. co-supervisor Nathan Sturtevant created a video that demonstrates this. (See example code.)

Implement the open list using a binary heap

. . . and implement the heap using an array.

This is enormously important on large grids, but less important for small grids (on the order of a couple thousand states in a C++ implementation). On grids with very few obstacles, maintaining the heap might be more expensive than linear scans of the open list. (See example code.)

Use a non-overestimating heuristic

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

On a 4-connected grid

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

```
h(i,j) = C * (Δx + Δy)
```

where `Δx` and `Δy` 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 * Δx + B * Δy   if Δx > Δy
         C * Δy + B * Δx   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(Δx - Δy) + D * (Δx + Δy)) / 2
```

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

Choose cardinal/diagonal move costs carefully

On an 8-connected grid, the cost of a single diagonal move (`D`) relative to the cost of a cardinal move (`C`) not only affects the appearance of the paths A* generates, but also its efficiency. (A* isn't alone in this; the Fringe Search algorithm is also very sensitive to changes in cost structure.)

Avoid same-cost diagonal and cardinal moves

When entities move cardinally *or* diagonally once per time-step, the instinct is to tell A* that cardinal and diagonal moves cost the same (e.g., `C = D = 1`). While technically true, this increases the number of unique optimal paths across the grid; A* is more efficient when it has fewer options.

Ensure C < D < 2C

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 like you'd see on a 4-connected grid. Paths tend to look best when the costs lie between these two extremes.

Use high-performing move costs

The following cost structures work well in practice. Results can vary depending on the obstacles in the grid, so test before using.

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

`D = 3, C = 2` This is still close to a `D/C` ratio of `sqrt(2)` and remains integral. Moreover, if `h(i,j)` is admissible but non-integral for whatever reason, then its ceiling is admissible and can be used instead. Nathan Sturtevant showed me this when we wrote Euclidean Heuristic Optimization (Rayner, Bowling, Sturtevant), and it made a noticeable difference.

`D = 99, C = 50` This gives something close to rectilinear costs but retains a preference for diagonal moves over pairs of cardinal moves. On average this keeps the size of the open list smaller, but it can also increase node expansions. Usually it is noticeably faster.

Additional Resources

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 have also been called "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.

Amit Patel's variants of A* A listing of some alternatives to A*.

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