Best practices for A* on grids

Chris Rayner

June 2, 2018

Description

This document describes ways to improve an A* implementation, focusing on pathfinding in four- and eight-connected grids. It's pitched at hobbyists and anyone looking for ways to make existing code a bit faster.

Some accompanying example code is available in C++.

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. 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` value:

`f_i = g_i + h_i`

which is an estimate of the cost of a path going through `i` and continuing to the goal. Here `g_i` is the cost of the cheapest path to state `i` that A* has generated so far, and `h_i` is an efficiently computed *heuristic estimate* for the cost to get from `i` to the goal.

(For further detail, visit the resources at the end of the document.)

Move costs

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 can also affect its efficiency.

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.

Avoid same-cost diagonal and cardinal moves

When the entity can 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.

Validate your move costs

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` 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 state expansions. Usually it is noticeably faster.

Heuristics

Here are some tips on selecting a good heuristic. I frequently see these details missed on many first implementations of A*.

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 states assuming no obstacles.

Use Manhattan distance on a 4-connected grid

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

```
h_i = C * (Δx + Δy)
```

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

Use octile distance on an 8-connected grid

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

```
h_i = 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.

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

```
h_i = (E * abs(Δx - Δy) + D * (Δx + Δy)) / 2
```

where `E = 2 * C - D`. You can see how this simplifies further, without floating point arithmetic, if `D` (and therefore `E`) is even.

- See an example implementation of the octile heuristic
- See an example implementation of the non-branching octile heuristic

Scale your heuristics up

Once you've selected a good heuristic, try multiplying all of the values it gives you by a constant `K > 1` (e.g. `10`). This simple change yields an algorithm called Weighted A*, which significantly improves run-time at the cost of small suboptimalities in your paths.

See an example implementation of a weighted octile heuristic.

Algorithmic details

Some small details that tend not to come up in textbook descriptions of A*.

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 tiebreaking code.

Avoid recomputing heuristics

To help keep the open list sorted, an implementation of A* might store the `f_i` and `g_i` values for every open state `i`. And since `f_i = g_i + h_i`, the value of `h_i` can always be recovered as `h_i = f_i - g_i` for any open state `i`. Using these stored values (a form of memoization) can be less expensive than recomputing `h_i`.

For instance, suppose `i` is on the open list with `f` and `g` values of `f_current` and `g_current`. Then A* iterates to a cheaper path to `i` with a cost of `g_new`. The corresponding value `f_new` can be determined *without* making another call to the heuristic function:

```
f_new = g_new + f_current - g_current
```

See an example of using memoized heuristics.

Know whether to use a heap

On larger grids with complex obstacles, implementing your open list as a binary heap (preferably on top of an array) can lead to dramatic performance gains. This is why it's generally considered a best practice to do so.

But heaps can hurt. On smaller grids with few obstacles, a linear scan of the entire open list can be much faster, especially if your implementation is written in a low-level language like C++.

- See an A* implementation that uses an array
- See an A* implementation that uses a heap
- See an example heap implementation

Consider Fringe Search

Fringe Search is a close cousin of A* that takes a different approach to growing and maintaining the open list. Just about all of the points in this document apply to Fringe Search, such as choosing a good heuristic, the choice of diagonal vs. cardinal move costs, and using memoized heuristic values.

With compiler optimizations on, I found Fringe Search to be slower than A*, albeit only if the methods in this document are applied. But with compiler optimizations off, Fringe Search can be faster than A*. It's reasonable to *predict* Fringe Search may be the faster choice in interpreted scripting languages.

See an example Fringe Search implementation.

Implementation

The following are some tips on the actual implementation of your pathfinder.

Maintain two pathfinders

During development you'll be constantly changing and refactoring your code. This can be dangerous – it is surprisingly easy to write a pathfinder that seems to work but has an invisible bug that isn't obvious until much later.

To prevent this you should write tested code: write a simple but *correct* pathfinder and use it to test your production pathfinder. For example, if you're finding optimal paths, both your simple pathfinder and your optimized pathfinder should return solutions of the same length, even if they visit different states.

Choose the right language

You'll get huge speed gains by writing your pathfinder in a compiled system-level language like C, or C++, or Rust.

If you're using a high-level scripting language, you're not necessarily out of luck. If you're using Python, for example, you could look into compiling your pathfinding module with Cython – it's surprisingly easy to do.

Pack your data structures

If you're coding in a low-level language like C, C++, or Rust, be aware of the effects of structure packing – *especially* if you're using an explicit graph to represent a large search space.

If you're using `gcc`, for example, try giving your compiler the `-Wpadded` argument and see how much it whines about having to pad your data structures with extra bytes. Eric Raymond has a great writeup on this topic.

Additional resources

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

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*.

Contributing and citing

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

If for any reason you want to cite this document, use the following:

```
@TECHREPORT{Rayner2017BestPracticesGrids,
    author = {D. Chris Rayner},
    title = {Best practices for A\* on grids},
    institution = {},
    year = {2017}
}
```