Best practices for A* on grids

Chris Rayner

November 15, 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.

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.)

Heuristics and move costs

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 (cheapest) path when its heuristic is admissible. A good, admissible grid heuristic is the "distance" between two states 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 = 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`.

- 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.

  (See an example implementation of the octile heuristic.)

- Non-branching octile heuristic 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 non-branching octile heuristic.)

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 can also affect its efficiency.

- 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.

- 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.

Scale your heuristics up

Multiply all heuristics by a constant `K > 1`. 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.)

Implementation details

Use a binary heap

. . . and implement the heap using an array.

This is enormously important on large grids, but admittedly less important for small grids – on the order of a couple thousand states in optimized C++. On grids with few obstacles, maintaining the heap might be more expensive than linear scans of the open list.

See an example heap implementation.

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 in favor of 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.

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.

Consider Fringe Search

Fringe Search is a close cousin of A* that takes a different approach to growing and maintaining the open list. The implementation is quite similar to A*, and 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.

See an example Fringe Search implementation.

With compiler optimizations on, Fringe Search is 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* that Fringe Search might be the faster choice in interpreted scripting languages...

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

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