Pathfinding with A*

Alexander Bake

05/29/2014

1 A* search algorithm

The A* search algorithm is a very well known pathfinding algorithm, widely used for a variety of applications. It is a modified version of *Dijkstra's Algorithm*, but with an added **heuristic function**. The cost of visiting a node using the A* search algorithm is made of two parts: the past path-cost function, which we will refer to as g, and the future path-cost function (the heuristic) which we will refer to as h. Together, we get the cost of visiting any node in our graph as $f = g + h$.

1.1 Parameters to consider

The A* search algorithm can be applied to different contexts, and it is important to consider the context when considering the various parameters of our search.

The first piece of our context is the *world* in which we run the algorithm. Abstractly, the algorithm needs nothing more than the constructions provided by basic graph theory — a set of nodes and a set of edges between nodes. Practically (and in two-dimensions), the world is often expressed through some sort of division mechanism, usually in the form of a square grid, although other polygons that interlock nicely are acceptable (e.g. hexagons, triangles). For the sake of simplicity, we'll restrict our conversation here to square grids. Note as well that while A* isn't suited exactly for a continuous world, we can approximate that suitability by making our square grids smaller and smaller (at the expense of some performance). Of course, there do exist pathfinding algorithms for continuous worlds, but we will not discuss them here.

Another parameter to consider is whether the *traversal of diagonals* is allowed. Conventionally, they are not allowed, and this can make searching our world a bit faster as we don't have to consider an additional four nodes. This can have some interesting effects in terms of path efficiency, which we will explore more when we talk about heuristics. However, when we do allow diagonal traversal, our paths will certainly be more efficient (by the Pythagorean theorem). This can also produce some interesting results in terms of path construction. For example, in Figure 1 we can see that when diagonals are allowed, we get a somewhat strange traversal through a set of walls, which would normally be a bit unintuitive.

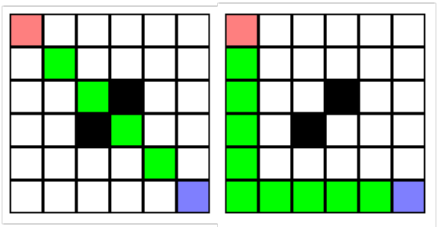


Figure 1: Diagonals allowed versus not allowed

Finally, if we allow diagonal traversal, we can consider the cost of moving to a node's proximity as a parameter. That is, the cost of going north, west, south, east, (nwse), compared to the cost of going

north-west, north-east, south-west, south-east, (diagonal). The "correct" way to do this is to base the cost of going diagonally off of the cost of going nwse. That is, if the cost of going nwse is x, then the cost of going diagonally should be $\sqrt{x^2 + x^2}$. However, if we decide to ignore this convention, we can produce some interesting results which might be useful for certain applications, which we will talk about in section 1.3.

1.2 Heuristics

One very interesting aspect of the A* algorithm is its use of heuristics. There are quite a few different heuristics available, and they all have trade-offs for performance in finding the shortest path and being efficient when exploring nodes.

An important aspect for our heuristics is the parameter for scale, s, which scales our heuristic function in relation to the cost of moving to a node's proximity. For simplicity, we will denote the cost of going nwse as s and the cost of going diagonally as s_d. Also, for the sake of simplicity, we'll denote the difference in the x and y coordinates for two points as $d_x = x_2 - x_1$ and $d_y = y_2 - y_1$.

1.2.1 Zero Distance

This is the most basic heuristic. Here, we simply have $h((x_1, y_1), (x_2, y_2)) = 0$, $\forall x_1, x_2, y_1, y_2$. Using this heuristic, the only measure of cost for choosing one node or another is the cost of moving to a node's proximity. Using this heuristic, A* is reduced to Dijkstra's search algorithm.

1.2.2 Manhattan Distance

Manhattan distance (or taxicab distance) may make the most sense when the world is a square grid. This heuristic is simply the sum of the distances to the goal in the x and y directions. It is expressed as:

$$h((x_1, y_1), (x_2, y_2)) = s\left(|d_x| + |d_y|\right)$$

Manhattan distance is quite efficient when it comes to what nodes it explores, but it can sometimes provide slight overestimations, which can lead to an inadmissible heuristic, producing suboptimal paths. However, this is rarely the case (and especially when diagonals are not allowed) and it proves to be admissible.

1.2.3 Euclidean Distance

This is the most intuitive distance, especially if our world is continuous (or approximates being continuous). It is expressed as:

$$h((x_1, y_1), (x_2, y_2)) = s\sqrt{d_x^2 + d_y^2}$$

While euclidean distance is more intuitive, and will certainly provide efficient paths, it sacrifices a bit of efficiency in what nodes it explores and some more expensive computation.

1.2.4 Squared Euclidean Distance

Some believe that it would be more efficient, computationally, to eliminate the expensive square root operation from the euclidean distance and reduce it to:

$$h((x_1, y_1), (x_2, y_2)) = s\left(d_x^2 + d_y^2\right)$$

If we don't include our scale s in this heuristic, it becomes inadmissible as it produces overestimations. This can cause the algorithm to approach greedy-best-first-search. However, if we try to rectify this, using instead use a smaller scale, we run into the problem of the heuristic approaching the zero heuristic, i.e. the algorithm approaching Dijkstra's.

1.2.5 Diagonal Distance

This distance is best suited for contexts that allow diagonal movement. The idea behind this heuristic is to compute the cost of getting to the goal with diagonals not allowed, and then subtract the cost of getting there with the diagonals allowed. This distance is expressed as:

$$h((x_1, y_1), (x_2, y_2)) = s\,(d_x + d_y) + (2s - s_d) \cdot \min\,(d_x, d_y)$$

1.3 In Gaming

The A* search algorithm is used heavily in gaming, generally for providing the NPCs with the appearance of "intelligence." This illusion is achieved through an NPC's ability to move towards a target without dumbly hitting walls or exiting the world.

A* can be used for agents that follow other agents, agents that race to some goal, enemy agents that track a player, and many other applications. Of course, the fastest and most efficient A* isn't always desirable! Playing with the parameters of the algorithm can produce funky results that could potentially be used as gameplay elements.

An easy example of this is altering the cost of traveling to proximity nodes. Typically, this isn't really a parameter you want to mess with, simply because the cost of going diagonally is mathematically defined by the cost of going nwse. However, by taking these parameters to an extreme, we could potentially produce an interesting level of gameplay. Take the example in Figure 2, where the cost of going diagonally is drastically less than the cost of going nwse. This situation could potentially be used for an enemy's pathfinding, where the enemy moves in a zig-zag, making it harder to predict its next move.

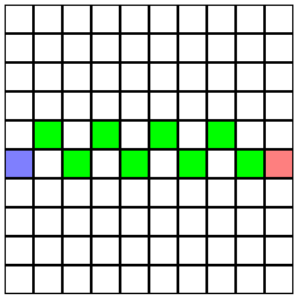


Figure 2: Cost of going diagonally is much lower than that of nwse

Maybe this is still too predictable. We could go further still, and use a random heuristic function, where we compute a random number and scale it to the cost of traversing the proximity. This will provide an even more erratic path that will certainly be harder to predict. Figure 3 shows how this might look like.

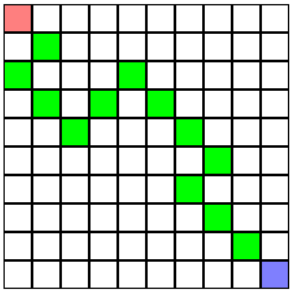
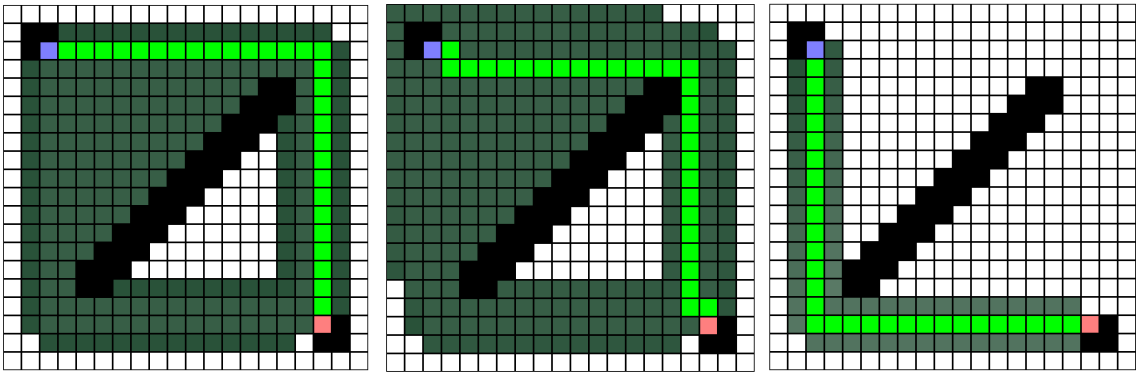


Figure 3: Using a random heuristic function



(a) Manhattan distance (b) Euclidean distance (c) Diagonal distance

Figure 4: Finding the same path with different heuristics (dark green indicates explored nodes)

1.4 Trade-offs in choice of heuristic function

Choosing the right heuristic function is important. The trade-off here is path cost efficiency versus efficiency in finding that path. Of course, this all depends on the context of your search.

Let's look at an example where diagonals are *not* allowed. When diagonals are not allowed, we get surprisingly uniform path cost efficiency. The real difference in choosing a heuristic in this context comes from how efficient the algorithm is in finding a path. As a metric, we can look at how many nodes the algorithm searches before it determines its path.

Figure 4 demonstrates this. Using the same context and start and end nodes, changing only the heuristic function, we can clearly see how some heuristics are more efficient when it comes to choosing which nodes to look at than others. Note that all three examples produce paths of the same length.

2 Genetic Algorithms

Genetic algorithms aren't suited for every application (say, pathfinding), however, they can produce interesting results that can spawn new ideas. Genetic algorithms emulate evolution, where selection occurs over many generations. The question then becomes, how can they be applied to pathfinding?

2.1 A* as an evaluation mechanism

An example of pathfinding and genetic algorithms working together is one where agents in a generation work towards reaching some goal, and upon failure or completion, that agent's progress is evaluated using A* .

In this situation, we can imagine a generation of agents performing a random walk through the world. Whenever an agent's next move causes him to collide with a wall, or exit the world, we evaluate the agent's progress by running A* from it's last good step to the end node and taking the path length as our metric. Then, we simply take the top 10% of agents, and perpetuate their knowledge to successive generations.

3 Concluding remarks

The A* algorithm is widely used, and for good reason. What makes it interesting is it's parameters, and namely, the heuristic functions it uses. These parameters can be tweaked to achieve optimal performance, or return the most efficient paths, but they can also be tweaked in unconventional ways that can provide potentially interesting aspects of gameplay.