Jake Ozer
IT279
Spring, 2024

The Use of the A* Shortest Path Algorithm for Grid Based Graphs

Pathfinding algorithms have been a staple of Computer Science research for years, and it's no mystery why. Navigating through mazes, plotting the fastest route on a map, or even guiding a virtual character through complex terrain are all tasks that demand efficiency and precision. Enter A* (pronounced "A star"), a pathfinding algorithm that not only revolutionized digital navigation but also mimics the way humans intuitively search for the best path. There have been many pathfinding algorithms created and tested before, but A* is one of the best in terms of efficiency and flexibility, primarily for point to point pathfinding on grid based graphs. To illustrate the reason as to why, we'll explore how this algorithm works and compare it to other pathfinding algorithms of the same vein.

Let's start by making a point about grid based graphs and how their metaphorical "edges" are weighted. Assuming our source is a square on the grid, and we can only move in 4 directions, we can naturally say that the adjacent squares are at a distance of 1, and diagonal squares are at a distance of 2:

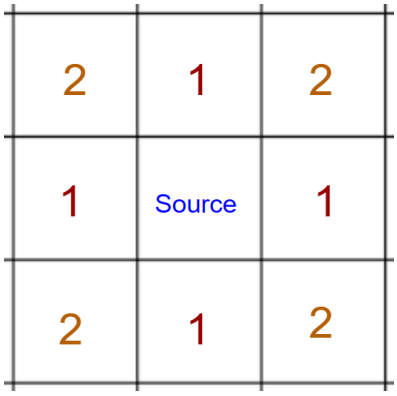


Figure 1: The metaphorical "edge" weights for a 4 directional grid graph

However, if you want to allow for diagonal movement (in 8 directions), we must use the standard 45-45-90 representation of a pythagorean right triangle, and use the hypotenuse of that ($\sqrt{2} \approx 1.4$) as our diagonal distance on our grid. In actual practice however, it is common strategy to replace 1 and 1.4 with integer values 10 and 14 respectively, to simplify calculations.

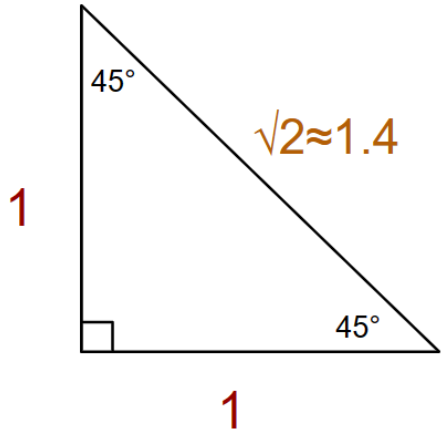


Figure 1: A standard 45-45-90 right triangle

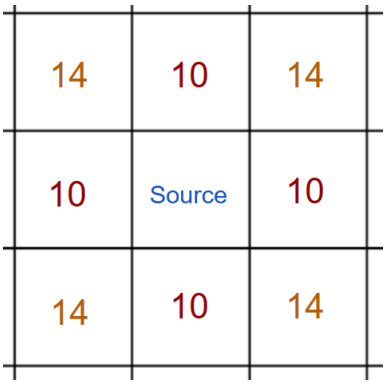
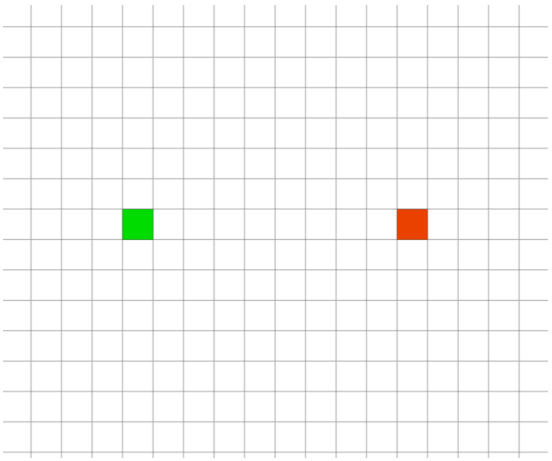


Figure 2: The metaphorical "edge" weights for an 8 directional grid

This is one of the main methods that allows us to use our traditional pathfinding algorithms, like Breadth First Search or Dijkstra's in a grid based format, as you'll see below.

So lets then discuss how A*'s predecessor and inspiration, *Dijkstra's Shortest Path Algorithm* works on these types of grids. In Dijkstra's Algorithm, assuming we need to get from point A to point B, we will always guarantee that we find the shortest path between those two points. However, in a scenario where a graph may be very dense and may have similarly weighted edges (like a grid based graph), we are bound to take many unnecessary steps to find our path. See the example below, where the green square is our source, the red square is our target, and the blue squares are the paths taken to get to our target:



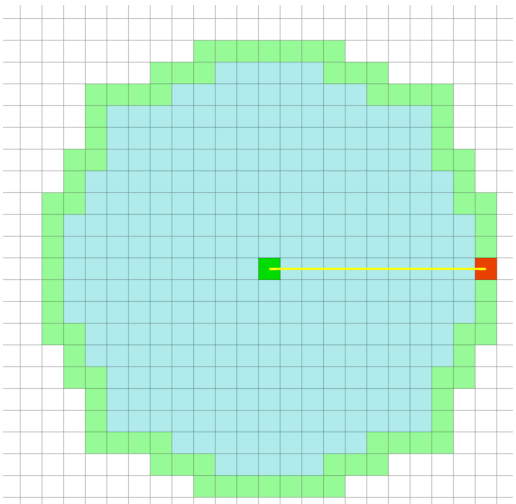


Figure 3: The grid before searching with Dijkstra's Algorithm (Xu 2015)

Figure 4: The result of searching with Dijkstra's Algorithm (Xu 2015)

In order to find our intended shortest path to the east, we searched far too much in every other direction, compromising our efficiency. This phenomenon is amplified to a greater extent when we have a supposed obstacle in our way on the grid:

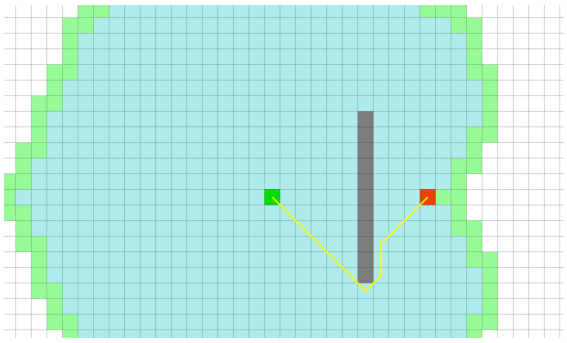
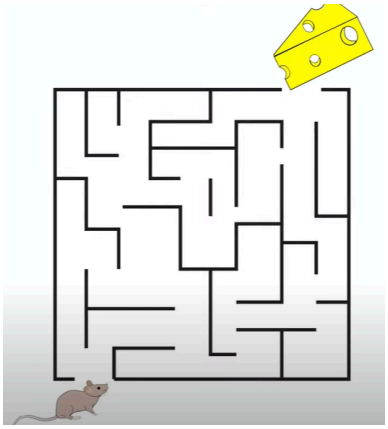
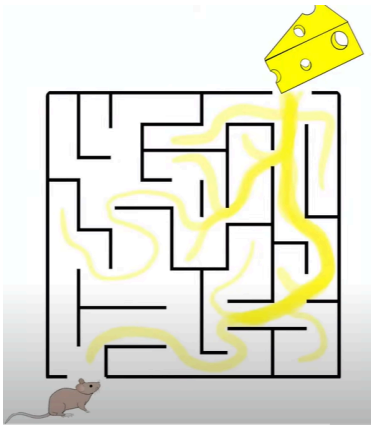


Figure 5: The result of searching with an obstacle in the way of Dijsktra's (Xu 2015)

So, a question presents itself: How can we close in on our shortest path more efficiently? The answer lies in *heuristic calculation*. To explain the concept of heuristics for pathfinding, let's use an analogy.

Imagine you are a mouse, tasked with completing a maze to find a piece of cheese. One option for finding the cheese might include exhausting all possible paths, until you reach your goal, but we've already seen that this is wildly inefficient in any practical scenario. But now imagine that the cheese had a rather potent smell that permeated through the maze. With this in mind, we have a much better chance of choosing a more correct route through the maze when given multiple options. We may deviate from the correct path slightly as we travel along, but as long as we follow the smell, we have a much better chance of finding our treat.





Figures 6 & 7: Diagrams of the mouse maze, without and with heuristics (Krishnan 2020)

This is the concept that lies at the heart of heuristics. In terms of our graph, assuming we are at any node, we can use the distance from that node to our target as our heuristic. Since there may be obstacles in that path, it will hardly ever be exact, but it does serve as a good estimate for where the correct shortest path may be (which is what heuristics is all about). In calculating these heuristics for A*, there are three values that are important.

- ***g***, the cost to get from the source node to the current node.
- ***h***, the cost to get from the current node to the target.
- ***f***, the combined cost of ***g*** and ***h***, that will serve as the basis of our estimate.

The ***g*** cost will be calculated the same way that you would in Dijkstra's algorithm, adding the total cost it took to reach the previous node, to the cost it takes to reach the current node.

$$g(n) = cost_n + cost_{n\_previous}$$

The ***h*** cost can be calculated in a number of different ways, depending on what directions you are allowed to move on your grid, as I stated earlier. The simplest of these calculations, called the *Manhattan Distance*, is used when you are only allowed to move in 4 directions, like the first example I showed at the beginning of the paper. It is as follows (Zhang, Li, & Bi, 2016):

In a plane with the **node** at *(x1, y1)* and the **goal** at *(x2, y2)*,

$h(n) = |x1 - x2| + |y1 - y2|$

In terms of pseudo-code, it can look like this, where *D* is the scale of your graph units (Patel 2024):

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

For movement in 8 directions, we will use the *Diagonal Distance*. **This is the one I will be using for the example in the next section**. The process follows the same principle as the Manhattan Distance but accounts for the diagonal movement (Zhang, Li, & Bi, 2016):

In a plane with the **node** at *(x1, y1)* and the **goal** at *(x2, y2)*,

$dx = |x1 - x2|$

$dy = |y1 - y2|$

$h(n) = max(dx, dy) + (\sqrt{2}-1) * min(dx, dy)$

In terms of pseudocode, it can look like this, where *D* is the scale of your graph units, and *D2* is the scale of your diagonal movement cost. In our case, *D* = 10 and *D2* = 14 (Patel 2024):

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * max(dx, dy) + (D2-D) * min(dx, dy)
```

For a scenario where you can move in any direction, you would use the *Euclidean Distance*, more commonly known as just the standard “Distance Formula”:

In a plane with the **node** at *(x1, y1)* and the **goal** at *(x2, y2),*

$h(n) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$

Lastly, the ***f*** cost will simply be the result of the previous two added together.

$f(n) = g(n) + h(n)$

See the example below for the start of our search, where we calculate our ***h*** cost with *Diagonal Distance*:

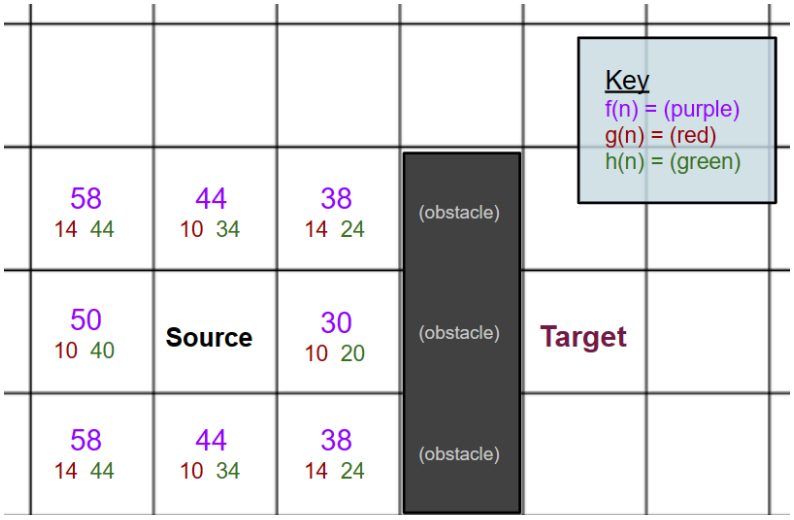


Figure 8: The start of our A* search, with costs displayed.

We then proceed with a greedy approach similar to Dijkstra's, picking the node with the lowest ***f*** cost.

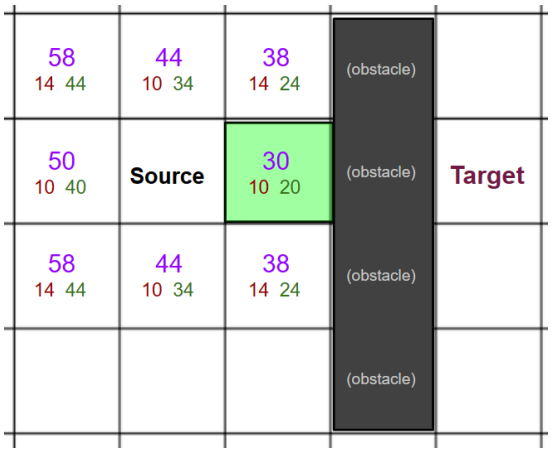


Figure 9: The first step of our algorithm, choosing the lowest ***f*** cost node.

From here, the algorithm is rather straightforward, as we continue to choose the lowest ***f*** costs as we traverse through the graph (or lowest ***h*** cost, if ***f*** costs are the same at any given step in the traversal). Note also that like in Dijkstra's, we will update our ***g*** cost if through our traversal we find a shorter path to a node from our source. Here is the final result of our search, where the green highlight indicates nodes traveled to, and blue indicates our shortest path.

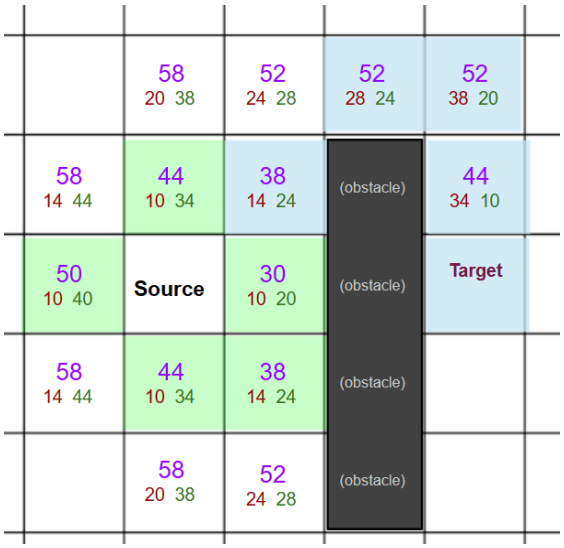


Figure 10: The result of performing an A* search on the previous example.

With the basics of how the algorithm and heuristics works, let's address why this type of approach is ideal for this type of scenario. You may argue, that given the power of heuristics, that we can just use a greedy, best-first search approach and ignore any kind of the backtracking you would perform in the A* search. This certainly works if there are no obstacles, and is of similar efficiency to A* without the extra overhead. However, it must be stated that a greedy best-first search will *not* guarantee that you find the shortest path. Consider a scenario like this, where we use best-first search to make a path to our target (red square is source, blue square is our target):

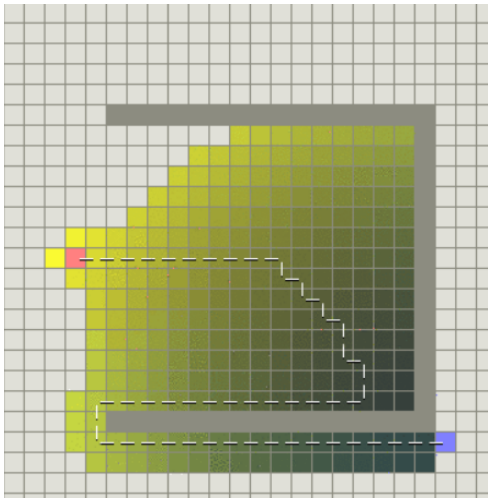


Figure 11: greedy, best-first search used to create a path to target (Patel 2024)

Our search does generate a path to our target, but it is by no means the shortest path. Now examine the same scenario with A*:

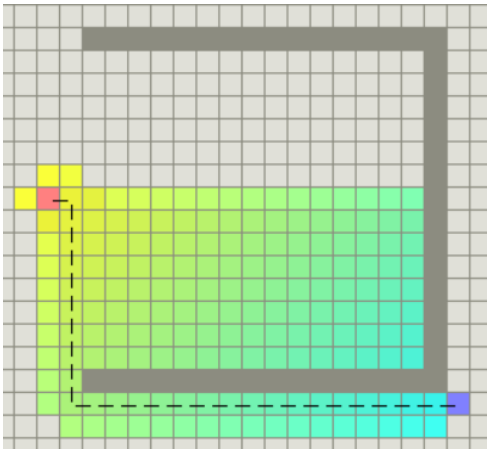


Figure 12: A* used to create the shortest path to the target (Patel 2024)

As you see, A* takes the best of both worlds considering Dijsktra's algorithm and greedy best-first search, always guaranteeing the shortest path, while still using heuristics to speed up the process. This is one of the main reasons why it is still widely preferred for its main applications in contemporary pathfinding.

In conclusion, the A* algorithm shines brightly among pathfinding methods, especially for grid-based graphs. By cleverly blending efficiency and heuristics, A* not only reliably finds the shortest path but also does so with impressive speed.

References

Candra, A., Budiman, M. A., & Pohan, R. I. (2021). Application of A-star algorithm on Pathfinding Game. Journal of Physics: Conference Series, 1898(1), 012047. https://doi.org/10.1088/1742-6596/1898/1/012047

Krishnan, A. (2020, May 26). A* (a star) search and heuristics intuition in 2 minutes. YouTube. https://www.youtube.com/watch?v=71CEj4gKDnE

Lester, P. (2005). A* pathfinding for beginners. https://csis.pace.edu/~benjamin/teaching/cs627/webfiles/Astar.pdf

Patel, A. (2024). Heuristics for A*. https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Zhang, A., Li, C., & Bi, W. (2016). Rectangle expansion a* pathfinding for grid maps. Chinese Journal of Aeronautics, 29(5), 1385–1396. https://doi.org/10.1016/j.cja.2016.04.023

Xu, X. (2015). A comprehensive path-finding library for grid based games. Pathfinding.js. https://qiao.github.io/PathFinding.js/visual/