Explore of Dijkstra and A* Pathfinding Algorithms

Theo Grant
University of Utah EAE
332 1400 E, Salt Lake City, UT 84112
847-687-3096
theo.grant9@gmail.com

ABSTRACT

This study looked into the two path finding algorithms: Dijkstra and A*. In particular, analysis of metrics between the algorithms will be made regarding the number of nodes that each one explores and its time to complete the pathfinding search. A discussion of the effects of the different heuristics used in A* and their effects will also be had. This will also include the effects of the absence of a heuristic in Dijkstra.

Keywords

Dijkstra, A*, Pathfinding, Heuristics, Memory Efficient AI

INTRODUCTION

Pathfinding is simply determining the shortest path between two points on a weighted graph. This is of particular interest to us given the 2D nature of our play space. For an understanding of the exact mechanisms of Boids and the play space in general, refer to the previous study. Given a square undirected graph mapped to our playspace, we can create behaviors to navigate an environment and avoid obstacles. This is essential in the continued study of developing a variety of AI behavior.

GRAPHS

The immediately important information is that our play space is a square volume of 800 x 800 pixels. This means that a square undirected graph of considerable size can be used to map chunks of pixels to a path finding behavior. Before that, two graphs were created for a proof of concept on our two algorithms. Those are noted below.

Transactions of the Digital Games Research Association Vol 1 No 1, ISSN…., Feb 2022

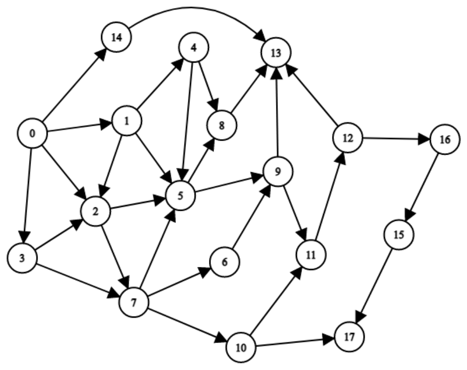

Figure 1: Larger 18 node directed graph with 2 terminating nodes and weights of 1

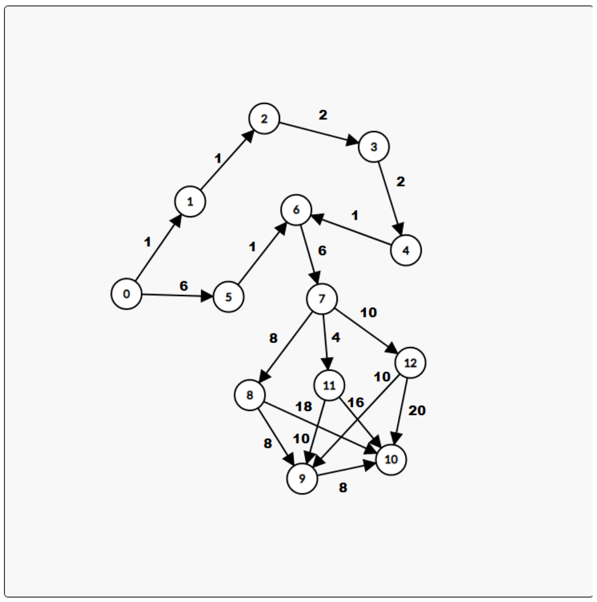

Figure 2: Smaller directed graph representing the optimal path of leveling the Mining Skill in ORSR

Both were utilized to easily discover if the implemented algorithms were accurate. This can be verified by hand quite easily by totaling the cost of any given path and comparing it to that of the algorithm output. This would be simply impossible on the large square undirected graph for the actual play space. In fact, it is impossible to even accurately depict in a picture. It can be described as a 10,000 node, undirected square graph(see Figure 3). This means it has a width of 100 nodes.

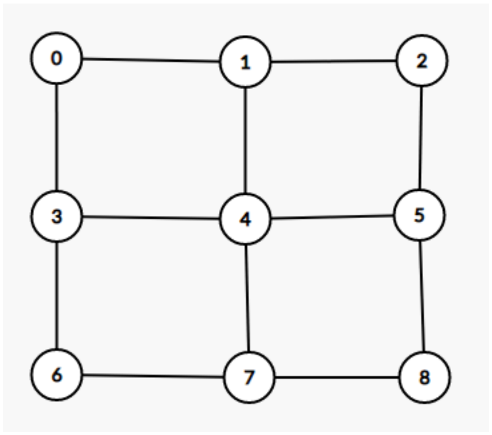

Figure 3: A square undirected graph. This example has 9 total nodes and a width of 3.

This graph is created every time at runtime by the GraphMaker class, which can create any square graph of width **W** up to the hardware limitations of the system it is used on. Additionally, the class can also create obstacles at run time. It is important to note that the obstacles are simply disconnected regions inside the larger graph. All nodes inside the bounds of the obstacle are still connected, essentially creating a new, localized graph. (see Figure 4)

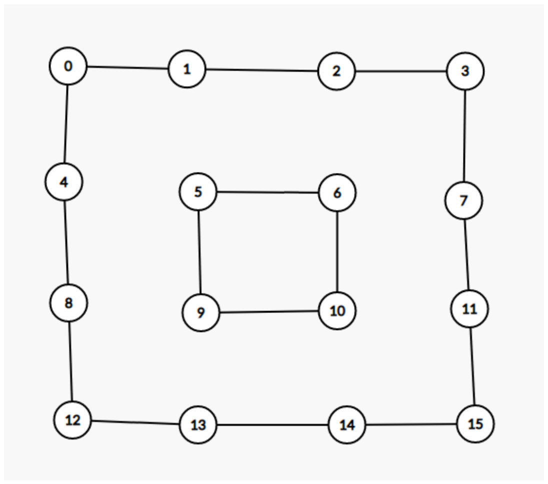

Figure 4: Example of square obstacle of width 4

There are also a handful of important things to note when assessing this graph in our play space. Our initial 0 node lives in the top left corner of the screen. As you move horizontally in the +X direction (the right side of the screen), the next node you move to increases by 1 in index until you hit the edge of the play space. Likewise, moving horizontally in the -X direction (the left side of the screen), the next node you move to will decrease by 1. Given the nature of OpenFrameworks, moving vertically in the +Y position (toward the bottom of the screen), the next node index will be increased by the width of the square graph. For example in Figure 3, the node directly below node 0 is node 3. This means in our complete graph of 10,000 nodes, node 5533 is directly below node 5433 given its width of 100.

Finally, it is important to understand the scale that 2 nodes represent. As mentioned before, the play space is a size of 800 x 800 pixels. Therefore, we can calculate that every node is exactly 8 pixels apart from each other. 4 nodes in a square presents an area of 64 pixels or .01% of the total area available. This allows for extreme precision in navigation of our Boids in exchange for a much more costly path finding algorithm call. This will be discussed further later.

DIJKSTRA AND A*

Dijkstra is an early pathfinding algorithm created by and named after Edsger Dijkstra in 1956. Its core design focuses on a weighted graph and seeks to move towards nodes of the lowest total cost from a source node. It is interesting to note that using Dijkstra on a graph where all edges have the same cost will essentially become a breadth-first search and defeats any advantage that it provides. An example of the Dijkstra algorithm can be seen below

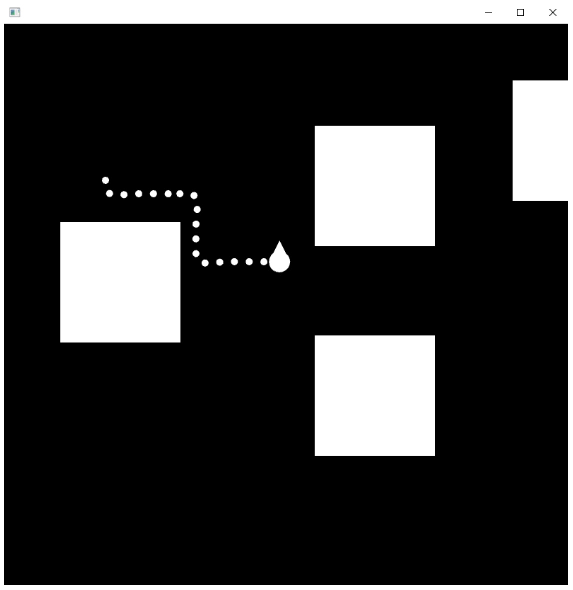

Figure 5: Dijkstra pathing around given obstacles

In comparison, A* is essentially Dijkstras with an added heuristic. A heuristic is some function that evaluates the potential of a given node. We evaluate this in the form of an additional cost added to the weighted edge cost. This can be seen in the following expression:

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

g(n) being the cost associated with the edge and **h(n)** being the added cost of the heuristic. This added heuristic can have a variety of effects on the A* compared to Dijkstra. Unlike Dijkstra, a graph consisting of all the same weighted edges may not be a simple breadth-first search. This depends entirely on the heuristic used, but it adds some prediction to hopefully ensure a more accurate path finding. This certainly comes at a cost of performance at run time though. Comparing time in milliseconds:

$$startTime_{A*} = 2793\,,\ finishTime_{A*} = 4526,\ completionTime_{A*} = 1733$$

$$startTime_{Dijkstra} = 2863\,,\ finishTime_{Dijkstra} = 3221, completionTime_{Dijkstra} = 358$$

Simply put, the Dijkstra algorithm is marginally faster in its call time across very large data sets. This can be attributed to the heuristic in A* having to be called so frequently in the algorithm. Not only are you calling that heuristic function on every child of every node you explore to find the next smallest cost, you have to also call it on every connection to that next child in order to determine if you have found a shorter path to it. This is why your heuristic functions runtime will greatly affect its speed of execution. On the other hand, Dijkstra will use explore about 29% ($1 - 1079/1429 = 29\%$) more nodes before it gets to its target when compared to A*. Additionally, it will generally be less accurate due to its missing heuristic and the fact that the algorithm stops searching for a better path once it has found its destination. Each has their trade offs.

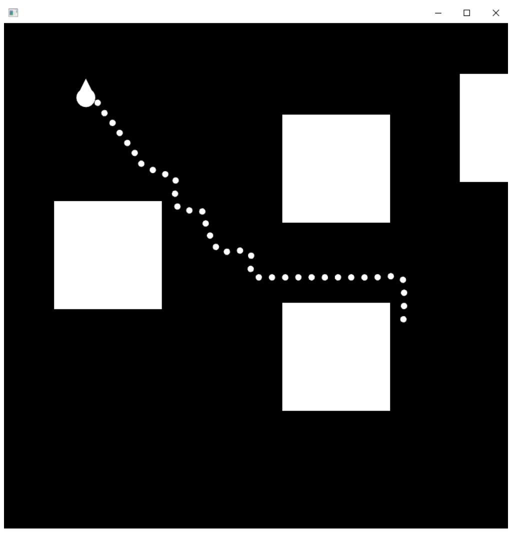

Figure 6: A* pathing around given obstacles with Euclidean distance heuristic

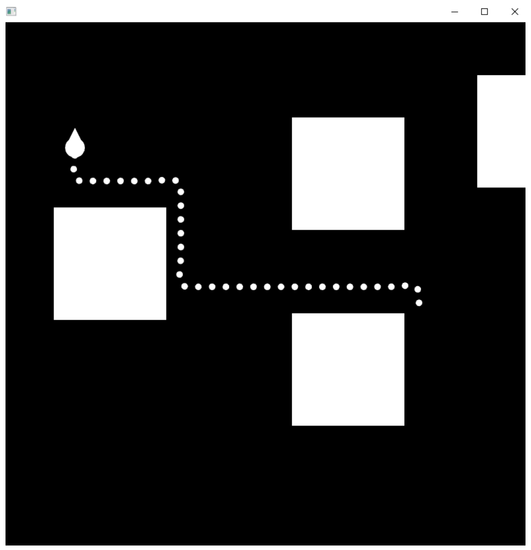

Figure 7: A* pathing around given obstacles with Manhattan distance heuristic

Figure 6 and 7 show the drastic effect of a given heuristic. It is important to note that this **large graph of 10,000 nodes all have weighted edges of 1.** This essentially proves the point that heuristic will prevent a simple breadth-first search in A*.

A final important note is the effect the walls have on the A* algorithm. As we mentioned before all walls are created by isolating those nodes from the rest of the play space. Those isolated nodes do still exist in the two dimensional array that stores all nodes representing the play space. The consequence of this is when a Boid is given a direction to move inside any wall, its A* algorithm will search the entire graph before realizing it is impossible to reach, causing a massive run-time for the call. A similar thing happens for the Dijkstra but to a lesser extent due to Dijkstra's quick run time which takes only about .5 seconds to search the whole graph. This situation points out a great limitation of game AI in many situations. While it would be ideal to have a particular Boid to move to any given pixel, it is simply too costly to have perfect control over an agent.

HEURISTICS

The two heuristics compared in this study were Euclidean Distance and Manhattan Distance. The differences in behavior can be seen in Figure 6 and 7 respectively. Euclidean distance can be described as the straight line distances from one node to another. This equation for that distance is as follows:

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

The Manhattan distance can be described as the absolute difference between any 2 nodes X-values and Y-values, summed together. On a grid, this essentially tells you how many blocks away from the target node a given node is. This is where the heuristic gets its name. The equation for any two nodes is as follows:

$$|p_1 - q_1| + |p_2 - q_2|.$$

Both of these two heuristics were specifically chosen due to their efficiency on a grid. Manhattan distance in particular basically only works on a grid style graph and Euclidean distance becomes very easy to calculate since each node has a simple coordinate associated with it. Additionally, they are fairly simple mathematical operations which helps ease up the run time of A*. Finally, both of these heuristics are admissible and consistent. This is important on any given A* implementation because it ensures that the algorithm is optimal. Admissibility is dependent on if the cost it calculates does not overestimate the cost to the target node. A heuristic is consistent if it continues to make progress towards the target node on every given step.

Looking at the Euclidean distance, we know that it is admissible with the triangle inequality.

$$z \leq x + y,$$

Since the distance of Euclidean distance is essentially the hypotenuse of a triangle and our Boid is not allowed to move in a diagonal direction, its horizontal **(x)** and vertical **(y)** distance summed together will always be a larger distance than the Euclidean distance. This means it’s admissible.

We can prove that Manhattan distance is admissible because of its relationship to our grid movement. Since we are inside a grid, every neighboring node is a distance of **X** away from one another. Since it can only ever move that distance of **X**, it will never overestimate the distance from a given node. It simply points the Boid either the closer **X** or **Y** direction, never both.

Both can be proved to be consistent simply because they rely on distances. A node in between the Boid and the target node will always be closer than a node behind the Boid and its target, thus producing a smaller cost for nodes in the direction of the target.

As for actual performance difference between the two, it is fairly negligible and not consistent. They can outperform each other depending on the initial position and target in terms of nodes searched with around 10% variance, but Euclidean distance A* does tend to run around .5 seconds longer than Manhattan distance A*. This is probably due to the square root function call inside that heuristic, whereas Manhattan only uses addition/subtraction.

Conclusion

As I mentioned before in the analysis of A* versus Dijkstra algorithm, we cannot create a perfect AI in games. Not surprisingly, representing intelligence is an overwhelmingly costly process, but it is not something we can really control with our given technology. Understanding the importances of the cost and benefits of each system is essential to creating something that is truly believable for the players. This becomes increasingly apparent as we begin to layer these systems into a deeper and deeper hierarchy. Cost will continue to add up and prevent further precision. Luckily, all we really need to do is create the perception of intelligence in a given environment and I am sure it will become more and more of a focus and we continue to move into decision making.