Assignment 2 Report: Pathfinding Algorithms

Nate Strohmyer
University of Utah
Salt Lake City, Utah
United States of America
719-304-4780
nate.stroh@live.com

ABSTRACT

In this report I will be exploring pathfinding algorithms and the accompanying data structures and thought process for implementation. I will specifically be exploring Dijkstra's algorithm and A-star for pathfinding. For A-star heuristics, I will be comparing a random heuristic, a Manhattan distance heuristic, and a Euclidian distance heuristic. Lastly, this report with cover the implementation of boid path following and the abstraction scheme required to implement path following.

Keywords

AI, Artificial Intelligence, Pathfinding, AI for Games, A-star Pathfinding, Dijkstra's Algorithm

INTRODUCTION

My AI engine is largely based on the structure talked about in our course work and by Millington (2019). This structure focuses on four main algorithm categories being movement, pathfinding, decision-making, and strategy algorithms. One important concept behind this structure is the idea of delegation and a hierarchy of these systems. The hierarchy goes as follows (starting at the top): strategy algorithms, decision-making algorithms, pathfinding algorithms, then movement algorithms. Generally, algorithms will delegate to those below them. This assignment explores pathfinding algorithms, which are essentially ways to get your AI agent effectively navigating an environment. This means the agent will take the most direct path it can while avoiding potential obstacles or costly paths. Pathfinding also starts to explore the delegation relationship between the four main algorithms as pathfinding delegates to movement algorithms to follow the path that pathfinding algorithms return. Overall, pathfinding algorithms are an interesting challenge because not only are you trying to find the best path to a location, but we must do so in an effective and optimized manner. In addition, you must also find an effective way to translate this path to something your movement algorithms can handle. In order properly implement pathfinding algorithms, I will need to cover four main topics: building the graph, the pathfinding algorithm itself, heuristics and abstraction schemes/path following.

BUILDING THE GRAPH

The first challenge of this assignment was to get something that can store all the relevant information for representing nodes and connections between them. I decided to make this its own ConnectionGraph class and it would oversee keeping a two-dimensional array that represented each node and its connections. While this was fairly costly in terms of space (n^2 where n is the number of nodes in the graph) this allowed very easy management and very quick look up times. This graph also contains some extra information that can be used to translate nodes to and from world space. The main two variables it keeps track of in addition to the two-dimensional array are the number of rows and columns. Keeping track of these variables here allowed for ease of access and more importantly allowed the graph to be rectangular rather than being limited to a square.

The first graph I made was just based on my house and consisted of nine nodes (see figure 1). While this was a fairly small number of nodes, this allowed me to have a super simple graph to work with when debugging my pathfinding. Most of the costs are just associated with the amount of work it takes to go from room to room. For example, going from bedroom 1 to the hallway is just walking through a door so it gets a low cost of one. However, going from the hallway to the entry area involves stairs so it gets a higher cost.

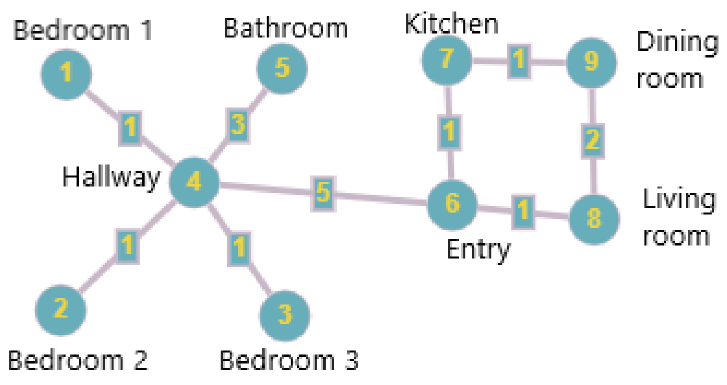

Figure 1: Above is the node graph representing the top floor of the house we're renting. It consists of 9 nodes with 9 connections.

However, this graph seemed too small, and I wanted to explore a graph that had some more nodes to offer more of a challenge for the pathfinding algorithm. For this second, smaller graph I decided to get a good start on a grid graph as I would be using a grid later for my abstraction scheme. I started with a simple five by five grid for a total of 25 nodes (see figure 2). When I initially made the graph, I just gave all the connections a value of one but had another version of it where I set values to different costs to test the pathfinding a bit more. I also found this grid connection graph to be very helpful to test some of A-stars heuristics as they depended on a world space of sorts.

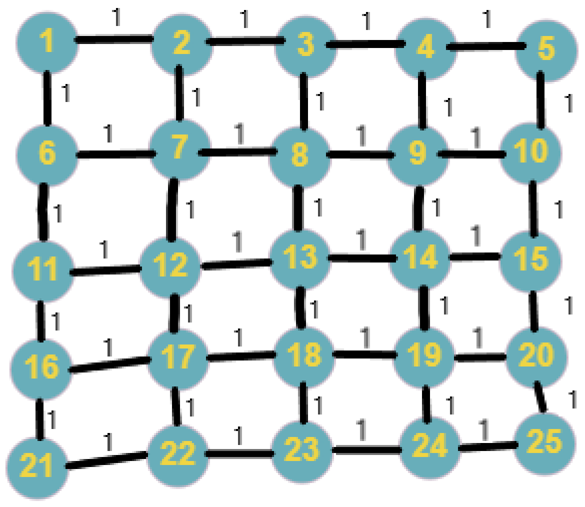

Figure 2: Above is the node graph representing a five-by-five grid for a total of 25 nodes.

At this point I was still doing this by hand which was quite a pain but thankfully I found a pretty helpful graph generator through Rutgers DIMACS shortest path challenge. While it gave me a good starting point the generator needed a fair amount of work to get working, especially in visual studios. Once I fixed up the file, I was able to get a grid generator that allowed me to specify rows, columns and costs. This is what I used to generate the grid for my final play space as well as my limit-testing graph. To really test my algorithms and show the difference between Dijkstra's and A-star I made a graph of 10,000 (100 by 100) and 40,000 (200 by 200) nodes where each connection had a random cost between one and ten. This also required me to write a quick function to read in these graphs from file, but thanks to OpenFrameworks file IO this ended up being pretty easy. Loading graphs this big also pointed out a problem my ConnectionGraph class initially had. Early on, I was storing costs as unsigned integers and when allocating 40,000 unsigned integers you hit the max amount of data you can allocate quite quickly. I decided that I shouldn't really need the initial costs of nodes to be more than 255, so I brought the unsigned integer down to a uint8_t which means it was only taking one byte to store rather than four. It's also worth mentioning that my pathfinding algorithms still keep track of costs in terms of unsinged integers, so I should have plenty of room to store the accumulated costs of nodes for big graphs.

PATHFINDING

The second part of this assignment was developing the two major pathfinding algorithms I will be using. Developing Dijkstra's algorithm and A-star provided some interesting and unique challenges. To start off, I had to figure out the best way to keep track of all the data I needed. I needed to keep track of a frontier to manage unvisited nodes, a way to keep track of costs and a way to store the best path. Finding the best structure for each of these items was by far the hardest part of making these pathfinding algorithms.

As mentioned before the graph structure I made was just a two-dimensional array that held all nodes and all possible connections between them. This allows for super quick look up time and if your graph is small enough, it doesn't take up too much memory. In addition, this simple format is pretty easy to wrap your head around and populate with data. Finding the best way to store the frontier was pretty straight forward as a priority queue suits it very well. Making the frontier a priority queue allows for the pathfinding algorithm to easily get the node with the lowest cost. The insertion may be a bit more costly, but it's better than sorting the frontier, or looping through the whole frontier every time you need a node. Getting the lowest cost node also allows you to be much more efficient as you are guaranteed to explore the most promising nodes first. Keeping track of costs and the path so far had very similar requirements, so I used the same structure to represent them. At first I considered making a map for each set of data. The map for costs would be used to just keep track of the best cost to get to a node, where the key would be a node id and the value would be the best cost so far to get to that node. I also thought this would be perfect for keeping track of the path, as I could easily store a node id for the key with the value being what node id led to it. After looking at maps for a little while I noticed that all the node ids are going to be unique and rather than making a whole map for these two sets of data, I can just store them in arrays. In these arrays the index of each item matches up with what index they are in the graph. The values at these indices are what I would be storing as the values in the map (the cost so far and where it came from). It's important to note that these indices match up with the first array of the graph, as the second level of the graph just keeps track of connections to other nodes. With the structures chosen the rest implementation went smoothly. I was surprised to see that the only difference I had to make between A-star and Dijkstra's was another addition for the heuristic when calculating cost. I found it super interesting that such a seemingly small change can make such a big difference.

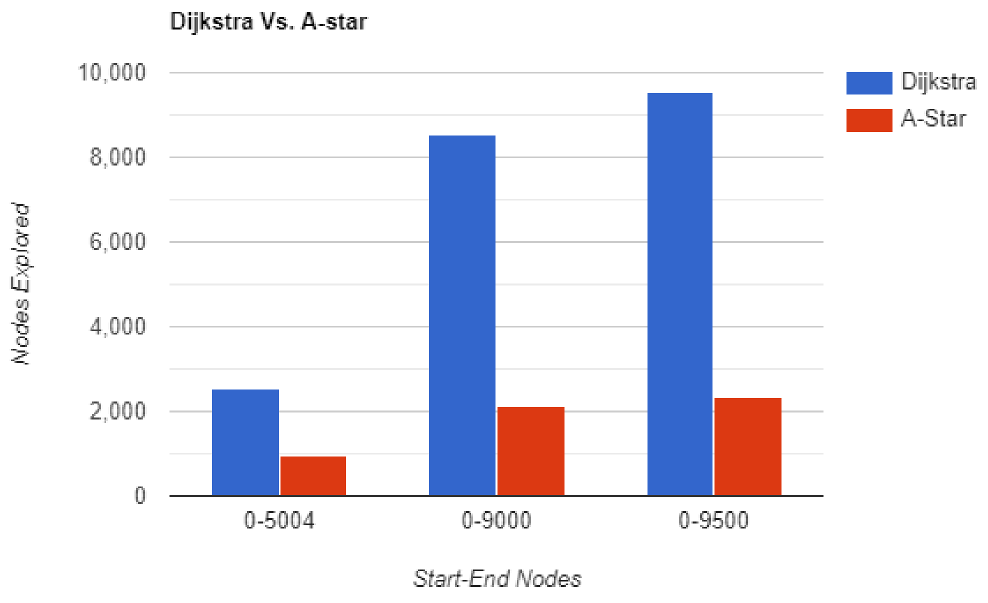

Figure 3: Above is a graph showing the number of nodes explored by Dijkstra's algorithm vs A-star. Each pair of bars represents a pathfinding call with start and end node.

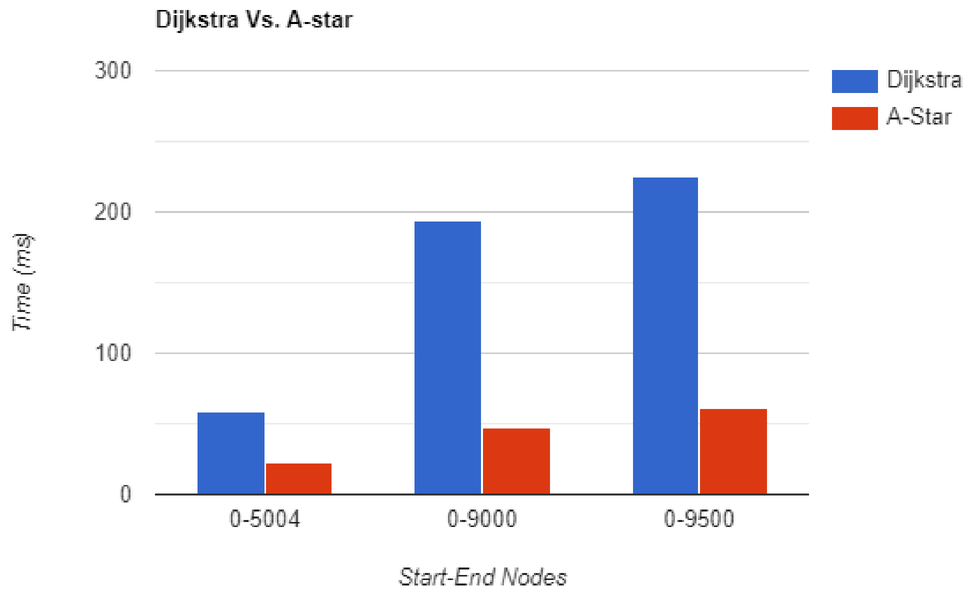

Figure 4: Above is a graph showing the time (in milliseconds) it took to find the best path. Each pair of bars represents a pathfinding call with start and end node. One thing to note is how similar figures 3 and 4 are.

A-star simply blows Dijkstra's algorithm out of the water when it comes to the number of nodes explored. This is especially noticeable as the size of the graph and distance between the start and goal increase. For example, starting at node zero and pathfinding to node 5004 saw Dijkstra's explore 2.6 times more nodes than A-star in the same situation (see figure 3). This only gets worse as the distance between start and goal gets bigger. For example, when pathfinding from node zero to node 9,500 Dijkstra's explores 9,523 nodes. A-star on the other hand only explores 2,344 nodes, which means Dijkstra's explores around four times as many nodes. When looking at the time each algorithm takes, we get very similar results (see figure 4). A-star beats Dijkstra's many times over as the time each algorithm takes to complete is largely related to how many nodes they explored. I figured the extra calculations A-star does for the heuristics might slow it down. However, given that my heuristics are simple calculations, it doesn't seem to affect its performance too much.

HEURISTICS

Heuristics are what set A-star apart from Dijkstra's algorithm and essentially are educated guesses/estimates of how much work is left to get to the goal. There are two main things to keep in mind when deciding on what heuristic to use: is a heuristic admissible and is a heuristic consistent. A heuristic is admissible if it doesn't overestimate the cost of getting to the goal. A heuristic is consistent as long as it always moves the algorithm closer to the goal. I ended up exploring Manhattan distance, Euclidian distance and a random heuristic for heuristics in my A-star algorithm.

Manhattan distance is something that really only works on a grid graph since it measures the difference between row and columns between two points. This can be calculated with the following formula: Manhattan distance = absolute value(x1 – x2) + absolute value(y1 – y2). This heuristic is admissible as it will never overestimate the actual cost to the goal. This heuristic is essentially calculating a straight path (as straight as it can be on a grid) to the goal so at worst this heuristic will give you the actual cost to the goal. For example, say there is a obstacle in the way or a cost higher than one, the actual cost will be higher than the heuristic making the heuristic an underestimate. This heuristic is also always consistent as it factors in a nodes distance from the goal. This makes nodes that are closer to the goal more likely to be explored. This makes sure that A-star is always picking a node that moves it closer to its goal.

Euclidian distance is another heuristic, and it is very similar to Manhattan distance. However, rather than measuring the difference between rows and columns, Euclidian distance directly measures the actual distance between the current node and the goal. Euclidian distance can be calculated with the following formula: sqrt((x1 - x2)^2 + (y1 - y2)^2). Like Manhattan distance, this heuristic is both admissible and consistent. This formula gets the exact distance between the current node and goal so even if you had a straight path to the goal this heuristic wouldn't overestimate it. In the case when you don't have a straight path (say you must go around a wall or have a costly path) this heuristic underestimates the cost, remaining admissible. Again, Euclidian distance is consistent as it will result in prioritizing nodes that are closer to the goal which makes sure A-star is moving forward.

The random cost heuristic is exactly what it sounds like, just adding a random number in the range of one to ten to the cost of that node. I mostly just explored this heuristic as a way to see what a bad heuristic does to A-star. This heuristic is inadmissible as it can easily overestimate the actual cost given that it just randomly guesses on a cost. Given that the heuristic is so random, it is also inconsistent as it can easily assign a node that is further away a low cost which may make A-star explore a node that doesn't bring it forward.

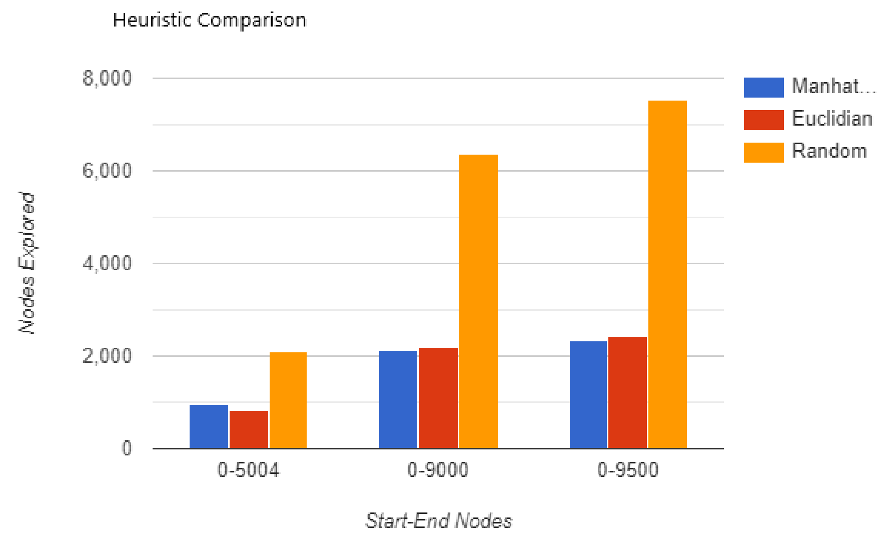

Figure 5: Above is a graph showing number of nodes explored in each call to A-star. Each cluster represents starting and ending node.

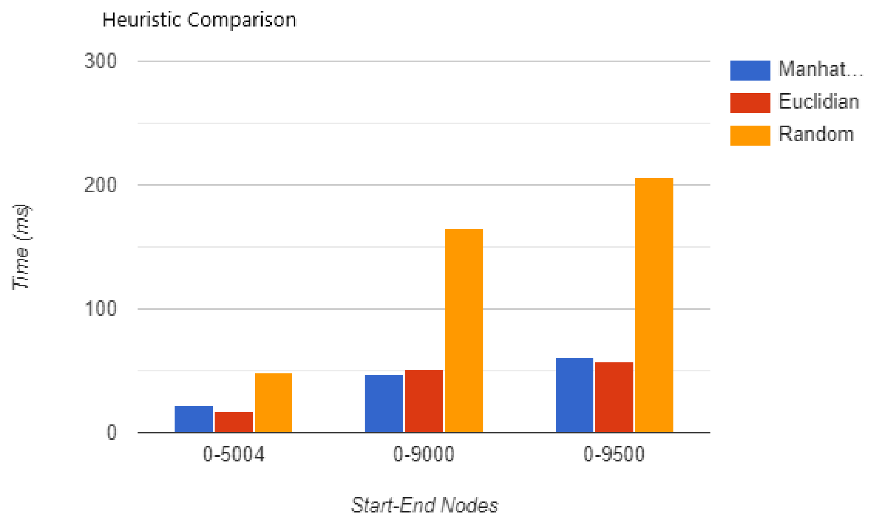

Figure 6: Above is a graph showing time taken (in milliseconds) for each call to the pathfinding algorithm with different heuristics. Each cluster represents starting and ending node.

When it comes to the performance of these heuristics, random was the worst by far and usurpingly, Manhattan distance and Euclidian distance preformed very similarly. Manhattan distance and Euclidian distance often resulted in a very similar number of nodes explored and alternated on which one explored less nodes (see figure 5). For a heuristic working on a grid, I feel that Manhattan distance suits A-star better given that it can give a better guess as to what lines/connections to explore. I also thought Euclidian distance would take more time to process given that it was a slightly more complicated heuristic, but it seems it actually took less time (see figure 6) when compared to Manhattan distance. However, this difference was minimal as the main indicator of time to complete the path finding was the number of nodes explored.

ABSTRACTION SCHEMES AND PATH FOLLOWING

An abstraction scheme is essentially how the connection graph is represented in the world. More importantly, an abstraction scheme dictates how you convert indices to and from world space. This is also known as quantization and localization. Since I was using a simple grid system (15 rows and 20 columns) quantization and localization was actually fairly easy. Localization or converting indices to the world position can be found with the following formulas: x = colums * width + width/2, y = rows * height + height/2. Quantification or converting a world position to rows and columns can be found with the following formulas: column = floor(x/width), row = floor(y/height). My pathfinding algorithms returned a vector of integers that represented what node it visited. Once converting that node to the row and column it belongs to, I can convert that into a world position.

Now that I have a way to get these node indices to world positions, I can just visit each position in the list that pathfinding gave me. This is essentially what path following is responsible for. I do this by keeping an index of what node I'm on and update that index once I reach the position that the current index translates to. My path following also takes in a smoothing radius, which is just a variable dictating how close I have to get to the point the boid is going to before it counts as visited. This helps the path following look much smoother as it can handle 90 degree turns much smoother as it cut corners in a way. I found that delegating to arrive looked best, but it required some interesting inputs to look right. For example, one of the changes I made was setting the slow radius of arrive to the smoothing radius on all nodes to arrive at except the last one. This resulted in a consistent speed and smooth travel between nodes, while still allowing for a smooth stop on the final node to arrive at.

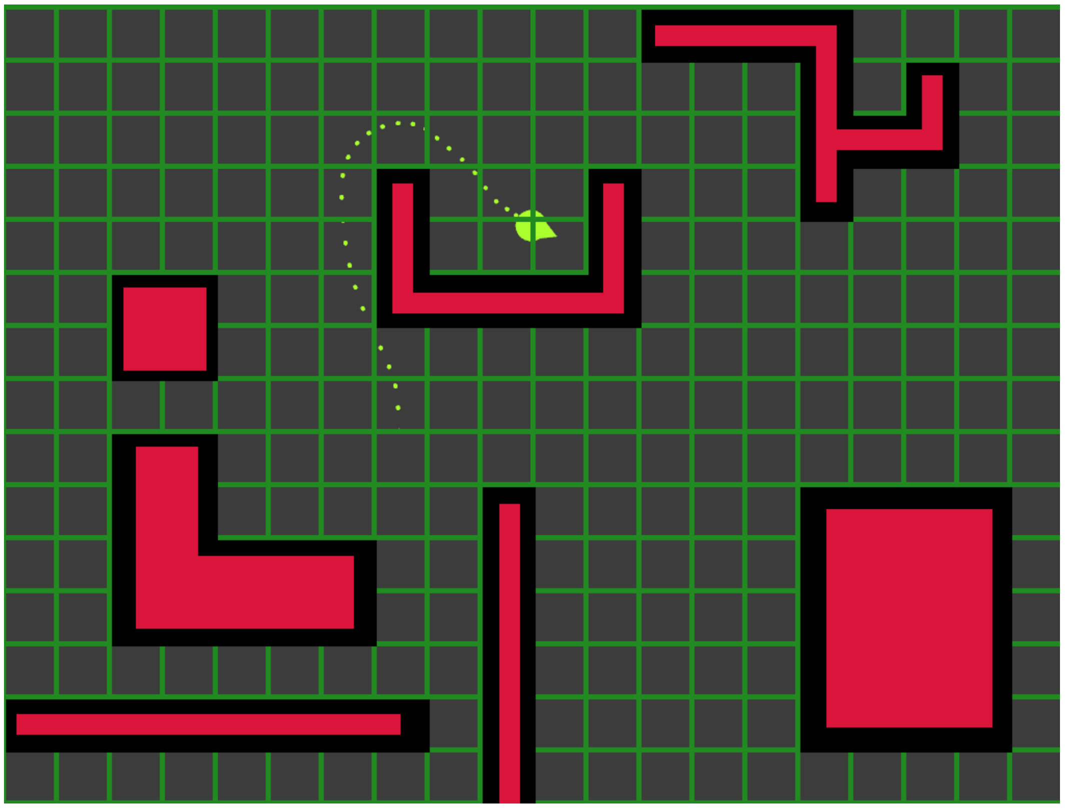

Figure 7: Picture of the boid following the path given to it by a A-star call running a Manhattan distance heuristic. Note the difference between the red and black. The red represents the actual wall, whereas the black represents an area pathfinding considers off-limits.

I found it very challenging to avoid running into walls when they were the size of a full tile. Due to the nature of our dynamic boids, they can't turn super quickly resulting in some overlap in the walls. In the end, I solved this problem by making the representation of walls in the world be smaller than a full tile. This allowed the boid to have some wiggle room when taking sharp turns. Overall, this ended up looking pretty good since humans (and most animals) tend to walk a distance away from walls so keeping this extra space between walls and walled-off tiles looks pretty natural. You can see the extra space given to boids in figure 7. The black is the entire space the pathfinding is blocked off from, while the red is the actual wall.

CONCLUSION

Overall, this assignment gave me a much better understanding of pathfinding algorithms and how much heuristics can affect an algorithm. In addition, this assignment taught me how delegation in my AI system can work and be super effective. For example, getting my boid to follow the path returned by A-star ended up being very easy given I can just use polished functionality already made. Inadvertently, this assignment also got me thinking a lot about storing data and the balancing of speed and storage.

ACKNOWLEDGMENTS

Bar graphs were made using the following link: www.rapidtables.com/tools/bar-graph.html

BIBLIOGRAPHY

Millington, I. *AI for Games Third Edition* CRC Press, 2019

Demetrescu, Camil. “9th DIMACS Implementation Challenge - Shortest Paths.” DIMACS. Rutgers DIMACS, June 14, 2010. http://www.diag.uniroma1.it//~challenge9/download.shtml#benchmark.