Pathfinding Algorithms and Finding the Quickest Path

Jack Kraemer
Kraem098@umn.edu

May 11, 2020

Abstract

In this study, we will go over the perfomrance of two pathfinding algorithms: the A* (pronounced "A star") search algorithm and the Dijkstra algorithm. These algorithms are used to find the shortest path in a graph or table and were used to find the shortest, quckest path in maps, including the map of Minneapolis. By comparing the search time in both the base map as well as a larger map of Minneapolis, we can determine an algorithm that is superior in finding the shortest path. These results can then be applied to theories on pathfinding in modern video games and what can be done to improve NPC (Non-Player Character) movements throughout the video game world.

1 Introduction

A common goal in Artificial Intelligence is to improve the pathfinding capabilities of search algorithms. There are many different applications that could benefit increasing the time it takes to get from start to the goal. One example are GPS applications such as Google Maps and Apple Maps that require a fast, accurate pathfinding algorithm to bring the users to their destination in the fastest time possible while also keeping in mind the safety of the user by avoiding things such as car accidents and road blockages. Algorithms such as Dijkstra's and A* algorithm are the main algorithms used in these type of applications, but there are multiple studies looking into improving these algorithms to further optimize these already speedy algorithms as shown in [3] and [12]. Video games as well as real life maps can benefit from having algorithms that are efficient because video game maps are very important to the users and their speed and efficiency can be a key thing in whether the game is popular or not. Having NPCs being able to move from one path to another as quickly as possible willl both allow the NPCs to be where they need to but also reduce the amount of stress on the game to allow for other things such as the player memory or graphics of the game. With GPS applications and real world maps, increasing the speed and efficiency of the pathfinding algorithims can allow the algorithm to put more of a focus on other obstacles such as agents (traffic) or by following the rules of the road such as one way streets. It can be seen that either way, improving and optimizing the pathfinding algorithms can be a large benefit for both real world users of the application as well as video game players.

There are numerous experiments that can be ran to test the mutiple different algorithms to find the most optimal path to the goal, but this experiment focuses on the two most common algorithms: A* and Dijkstra. These two algorithms are first tested using a basic map implemented with arrays to calculate the runtime that both of the algorithms would have on a simple maze. The implementation of the A* algorithm and maze was given by [13] and the Dijkstra algorithm

given by [8], both implemented in Python. The runtimes of both the algorithms were recorded for different tests to obtain a better understanding of how these algorithms can function on real map data, which can also lead to a better understanding of the algorithms in virtual maps. Both algorithms have their advantages and disadvantages and this project will help narrow our search for the algorithm that can find the most efficient path to the goal in the quickest amount of time.

2 Literature Review

To begin, we must take an in depth look at both the A* algorithm and Dijkstra algorithm to fully understand the evidence that is given by the results of the experiment. The two most common 2D pathfinding algorithms, A* and Dijkstra, are similar in that they can both be used to find the shortest path [9]. Most of the pathfinding algorithms from AI are designed to be used with arbitrary graphs as opposed to grid-based games. The way that Dijkstra's algorithm works is that it visits the vertices of the graph by starting at the object's starting point. It then identifies the closest un-examined vertex, while adding its vertices to the list of vertices that have yet to be examined. In this way, it is able to extend outward from the starting point until it reaches its end goal using the shortest path possible. In an attempt to combine Dijkstra, a formal approach, with a heuristic approach such as Greedy Best-First-Search, A* was developed. Since A* is built on top of a heuristic, instead of only being able to give an approximation of the shortest path, it can guarantee the shortest path. A* is known for its wide range of possible contexts and its flexibility. In this example [9], A* was able to determine as short a path as Dijkstra when using a concave obstacle. A* accomplishes this by favoring vertices that are nearby the starting point, as well as favoring vertices that are nearby the goal. Both A* and Dijkstra can be seen in 1 and 2 respectively. The figures show the path and search space for each algorithm, showing what the algorithm takes into account while searching for the shortest path to the goal.

A* is preferred for a couple main reasons. Firstly, A* is considered a complete algorithm, meaning that if a solution exists, it will always find it [1]. Its focus is also to reach the goal node as quickly as possible from its initial state, while avoiding every other node. Therefore, A* only expands on a node if it will likely lead to a solution. This reduces the time it takes to reach the goal while performing efficiently. The downside of A* occurs when you have a multitude of target vertices while not knowing which of those is closest to the main one. This is because the algorithm would need to be run several times in order to reach every one of them.

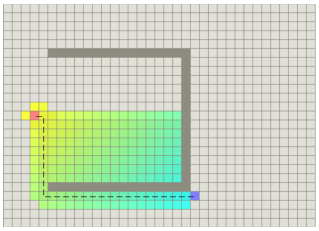

Figure 1: A* Search

On the other hand, Dijkstra is useful when the goal state is unknown. This makes the algorithm beneficial for when there is no prior graph knowledge and therefore cannot approximate the distance between each vertex and the target [1]. Dijkstra also covers a large area of the graph by picking edges with the lowest cost at each step. This is optimal for when there are multiple target vertices present, but it is unknown which is the closest. If there are negative edge weights present, Dijkstra will be unable to evaluate them, making this a con of running the algorithm.

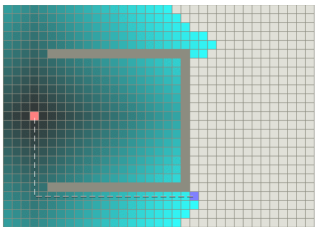

Figure 2: Dijkstra Algorithm

Efficiency of Dijkstra and A*

The complexity for both of the algorithms are expressed in Big-O notation and are shown in more detail in [11]. The efficiency for Dijkstra's algorithm is dependant on the nodes and edges of the graph. The worst case for Dijkstra's algorithm is when the graph is a complete graph, where the total edges = v(v-1)/2 where v is the number of nodes. In the case of adjacency lists and priority queues, the best case scenario is $O((|V|+|E|)\log|V|)$ and the worst case scenario would be $O(|E|\log|V|)$. To improve the efficiency, one can implement a Fibonacci Heap along with an Adjacency list to achieve the best bound of $O(|E|+|V|\log|V|)$.

To make things more complicated, A* algorithm has a very dependant complexity as it can be changed depending on the heuristic used in the search. Different heuristic functions can lead to different complexities, causing a situation where A* can only sometimes outperform Dijkstra's algorithm. For example, if the target is a straight line distance away from the initial start, then one can use a straight line distance heuristic which will lead to A* having a $O(1)$ complexity. If one would use A* to search a graph that they do not have any information on, A* would not be as efficient as Dijkstra's algorithm would be since they would not know which heuristic to use. This is one of a few pitfalls that can cause a worse complexity. For Dijkstra's algorithm, one problem that can cause issues with complexity is that the algorithm can only see the neighbors of the immediate node [11]. Since the algorithm does not backtrack, it could run into a infinite loop where there are no suitable neighbors to visit. This can consume and waste time and resources that could be used elsewhere.

More on A*

A* search along with others were the leading algorithms used in agents in video games. As the technology used in video games have advanced, the algorithms struggle to keep up and new ones need to be developed and tested to improve multi-agent environments. Compared to other algorithms, it is seen that A* search is optimal and other optimizations would improve AI in video games [4]. With the growth of new video games also includes the growth of different type of Real-Time Strategy games that require different types of AI agents. Although A* is useful for simple problems, by combining different variations of the Local Repair A* will make the WHCA* algorithm. This algorithm is efficient at calculating routes and ongoing processing and it will apply equally to general pathfinding domains [12].

Some studies focused on the idea of using an API that will act as a neural network to create agents based on "parents" that carry the specific instructions for the agents in the game. This can help agents use the information given to them already to help make new decisions once obstacles come in their path. The goal of developers is to train a neural network to be able to overcome the limitations of standard pathfinding approaches [12] As most experiments try to show how different variations of A* will improve pathfinding and others show that it is the evironment around A* that needs to be improved, one study shows that A* can be outperformed on the grids typical of computer-game worlds. [3] uses the Fringe Search algorithm to improve efficiency. Compared to other studies done on A* search, Fringe search is shown to outperform A*, running from 25% to 40% faster. [3] also states that there could be an even better search that can be implemented in the future that will be even more efficient. To test the A* algorithm with other algorithms such as

the IDA*, [3] used a game-world map from the video game "Baldur's Gate II", shown in 3. This experiment also used obstacles such as immovable objects that would be in the way of the agent. This can help relate it to a more real world study as maps in the real world will have objects that would not be in a video game map.

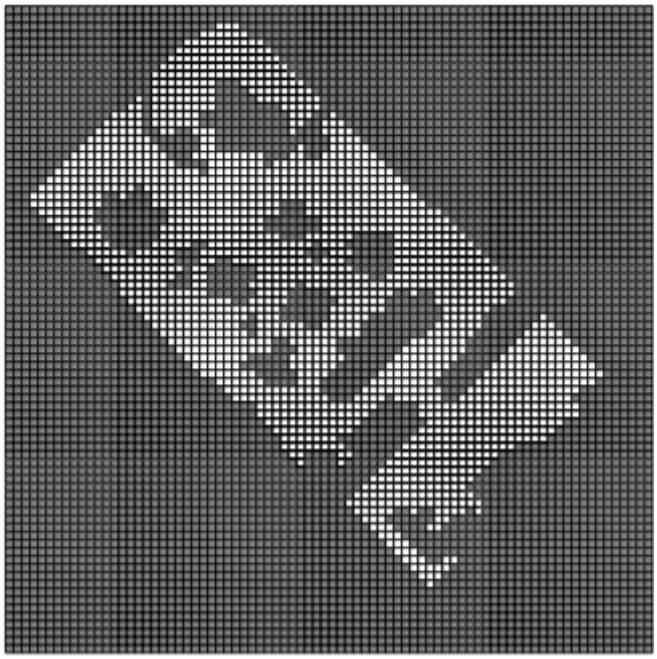

Figure 3: Example Map

With the many different versions of A* search being used to find the most efficient pathfinding algorithm, it seems that the issue is not with the algorithm but with the environment around it. The different studies on pathfinding in video games shows that A* and it's many variations are efficient, but there is always something better. If one would turn to focus on the environment like [6], you can see that A* is plenty efficient and the bottlenecks revolve around the memory usage and the use of a neural network. Implementing and improving a base for the agent to work and grow from will increase pathfinding by allowing agents to learn from previous agents while also learning from new things thrown into their environment

In a very odd but interesting way to look at pathfinding is a study done by [7] which focuses on improving "state-of-the-art dominated by hierarchical pathfinding algorithms." by optimizing these otherwise suboptimal algorithms. These algorithms are very fast and do not require much memory, but they need an increase in optimization. A strategy found by [7] has shown to optimize the algorithms, but it is an interesting concept. Sticking to certain nodes in a grid map, the algorithm will go on the path from node to node instead of following a path. These nodes are called "jump points" and the algorithm jumps from node to node and the pathway in never expanded. [7] has found that using this algorithm will increase the time it takes for A* to find the fastest path in the grid map. Altough this is a study that is mainly used for video game maps or in robotics, a "Jump point" is an interesting concept and could possibly be developed into something more futuristic for human travel. For now, Jump Points will not be relative in finding the fastest path in a city.

After using the algorithms to test runtime on a simple maze, we can move to a more complex

"maze" to see how the algorithms work as well. This complex maze will be modeled after the Minneapolis streets using the coordinates given in map.lisp. Both code implementations will have to be modified to take into account the one-way streets, but this will still not be as complex as it could be because there are no other agents included to simulate other cars that may be on the road. Excluding the other agents will still be able to show how efficient both the algoritms can be in finding the shortest path and will give a base to estimate how it could be ran with other obstacles in the path. Using the information found in [3] studies of A* search on game maps, it can be seen that A* does well with no obstacles in the environment. Using that information, we can assume that if Dijkstra's algorithm can outperform A* in solving grid based maps, we can also assume that Dijkstra's algorithm can outperform A* when there are other agents in the environment.

3 Approach

With the many different variations of A* search, it is best for this experiment to stick with the base version of A* algorithm. I also chose to focus on Dijkstra's algorithm as that is a popular and commonly used algorithm in pathfinding, especially as it relates to map traversal. There are plenty of tests ran to compare the algorithms such as in [5] where the algorithms are used on a simple grid map to test the runtime of reaching the goal. This is expanded in the project by using a simple array map created by [8] along with their python implementation of the A* algorithm. I will also be using a python implementation of Dijkstra's algorithm created by [2]. Using the same array map on both python implementations of the algorithms, I will be running tests on different start and end coordinate combinations These tests along with previous tests done in other studies can give a better look at what we would be expecting from using these algorithims on real world data, such as the road information of Minneapolis. For the project, I will be attempting to find the shortest path from different variations of start point to goal point in the map of Minneapolis 4.

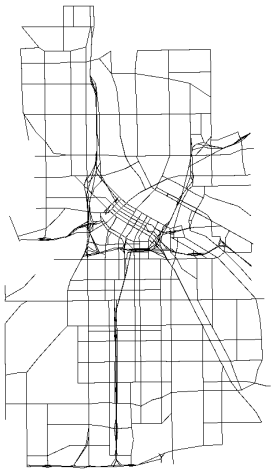

Figure 4: Map of Minneapolis

The Minneapolis road data is presented as a lisp file and given in [10] by Professor James Parker. Taken from the project instructions, the given map data contains 1357 lines, where each line corresponds to a road segment. Each line contains information on the road segment shown in 5 numbers in parentheses. The first number is either a 1 or 2 to determine if the road is a one way or two way street. The following two numbers are the X and Y coordinates of the start of the segment of the road, and the last two numbers are the X and Y coordinates of the end of the segment of the road. For example, (1 2884 8977 2708 8865) represents a one way road that starts at the point [2884, 8977] and ends at [2708, 8865]. Since the algorithms used are implemented in Python, it would only make sense to convert the lisp data into usable Python data. The easiest way I found to do this is to create a Node class in Python that can store the street direction value, the start coordinates, and the end coordinates. For example, one set of usable data would be under a new node name Street1 where Street1.direction will equal 1 to show that it is a one way street, Street1.start will be the beginning coordinates (i.e. [2884, 8977]), and Street1.end will be the ending coordinates (i.e [2708, 8865]). We would then have to do that for all 1357 lines of data, which was made easier by parsing through the tuples to assign the appropriate information. We can also use the information gathered through parsing the data to find which nodes will be connected to link the streets using the .nextStreet value. The use of the nodes required a slight edit in the code to be able to traverse the nodes.

Previously, we looked at multiple different variations of the classic A* algorithm that has been implemented, mainly in [6], [12], and [4]. These studies showed that even though A* can be optimized fully by combining it with other algorithms or slightly adjusting the original, the base A* search algorithm is a highly optimal pathfinding algorithm and is one of the best at finding the quickest path. The reason for this is the heuristic function that is used in A* search to assist the algorithm in finding an optimal path. The cost that is associated with a node is f(n) = g(n) + h(n) where h(n) is the estimated heuristic of the distance from n to the goal and g(n) is the actual cost of the path from the initial starting position to n. For this experiment, the heurisitc function is the Euclidean distance, the straight-line distance between two points, from start position to goal position. This is already included in the A* Python implementation created by [13].

Dijkstra's algorithm is the other algorithm that is tested in this experiment. This is a very popular algorithm that is used in a lot of popular mapping applications, so comparing Dijkstra's algorithm with A* sounds like it would lead to some more interesting results. Previously implemented in Python by [8], and using pseudocode from [11] to implement it using nodes, Dijkstra's algorithm works exactly how one would expect. We made sure to keep an array with a list of the nodes that are visited and unvisited to keep with the fact that Dijkstra's algorithm is a non-backtracking algorithm. The Python implementation first marks all nodes as unvisited and stores them in a list to keep track of which nodes we are allowed to visit. Then we will set the distance for our initial node to zero and the others to infinity. We will then select an unvisited node that has the lowest distance and if it is infinity, we will break the loop on set that node as the current node to search for unvisited neighbors to calculate their distances from the current node. By comparing the values with other neighbors, we can find the smallest distance and to select that node as the next node we will want to visit. First, we will have to remove the current node from the unvisited list to make sure we do not revisit this node and then we will continue until we remove the goal node from the unvisited list to the visited list, thus terminating the program.

4 Experiment Design and Results

To begin, the experiment was first tested on a simple maze that was created by [13] and shown in 5. The maze is an array that is a list of arrays that are in the shape of a grid. I used five different sets of coordinates to test the runtime of both the A* and Dijkstra's algorithm. In the code, we have to set the start coordinates and the end coordinates manually which is then put into the A* function. This function searches the graph from start coordinates to end coordinates, storing each coordinate in a list, where the function will print out the path that it took. I also implemented the time module in the python implementation to obtain the runtime of the searches, which could hinder the results but that is what I get for running it on a Windows computer. I used a varity of close coordinates and longer distance coordinates to ensure that the data we get back is accurate to what we would expect from the algorithms. Below in 1 is the results from running the two algorithms on the simple maze.

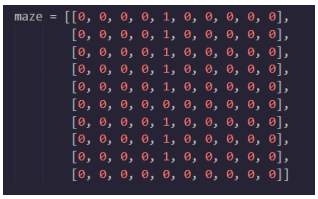

Figure 5: Simple Maze

Table 1: Simple Maze Results

Start	End	A* (Seconds)	Dijkstra (Seconds)
(9, 3)	(8, 1)	0.0039	0.0040
(2, 3)	(8, 1)	0.0060	0.0068
(9, 3)	(3, 1)	0.0059	0.0066
(5, 9)	(4, 2)	0.0069	0.0081
(1, 1)	(9, 9)	0.0070	0.0088

As you can see, Dijkstra's algorithm is just slightly slower in all of the tests done on the simple maze. Previously, in the Literature Review, I stated that A* algorithm would be dominant in an evironment without any agents, and this has proven that is correct as A* algorithm is a faster algorithm on this environment. We will go into more detail in the analysis section further in the paper. After the results, we would then go replicate an experiment done by [5] where it is a very similar test but with a more advanced maze shown in an example run here 6. This maze has obstacles in the middle of the maze that would require the algorithms to work avoid the obstacles to reach the goal. These tests are very similar to the ones done on the simple maze, but I had to adjust the algorithms to take into account obstacles that are included in the advanced graph. This again will be ran five times, but this maze is a little different as the start coordinates and end coordinates will always be the same, (0, 0) and (39, 39) respectively. The results are shown in 2 where cases 1 and case 2 are ran with a verticle hurdle, cases 3 and 4 are ran with a horizontal hurdle, and case 5 is ran with both a horizontal and verticle hurdle to create a plus sign type of hurdle.

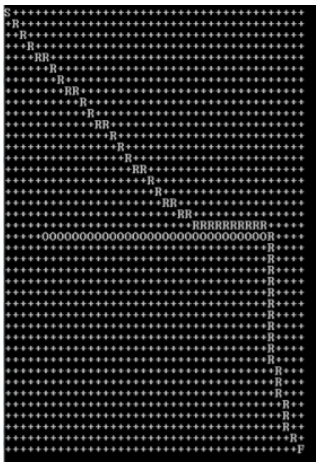

Figure 6: Advanced Maze

Again in this more advanced maze, it can be seen that Dijkstra's algorithm does not go as fast as A* algorithm does on a more advanced maze with obstacles. Now we move on to the more important tests that we would like to focus on: the map of Minneapolis. Again, we will be choosing

Table 2: Advanced Maze Results

Case	A* (Seconds)	Dijkstra (Seconds)
1	0.054	0.134
2	0.059	0.130
3	0.103	0.122
4	0.102	0.129
5	0.066	0.147

points that are both short distances and long distances to better gauge the performance of the algorithms. To stay constant, there are still five test cases ran on the map of Minneapolis that I have implemented into Python. The longest path that is in the tests is 2497.17 and the shortest path is 208.61. I also included a path that would not complete because the path is not an actual path in the graph. For the Table 3, we can see the paths that the algorithms will go through as well as the distance from the start node to the target node. In Table 4 we see the runtimes of each case for both of the algorithms.

Table 3: Minneapolis Path

Path	Start $\rightarrow$ Target Node	Distance
1	$(2884, 8977) \rightarrow (2708, 8865)$	208.61
2	$(1001, 8171) \rightarrow (0997, 7961)$	210.03
3	$(1073, 8231) \rightarrow (0164, 7700)$	1052.73
4	$(221, 7166) \rightarrow (930, 8377)$	1403.28
5	$(810, 10500) \rightarrow (2775, 8959)$	2497.17

Table 4: Minneapolis Results

Path	A* (Seconds)	Dijkstra (Seconds)
1	0.0023	0.0039
2	0.0025	0.0038
3	null	null
4	0.021	0.032
5	0.0113	0.0251

5 Analysis

A* algorithms and Dijkstra's algorithm were both tested an three different maps a consistent 5 times each variety of map. To begin, I tested the two algorithms on a simple maze with no obstacles from one start position and one end position. Each case had a different set of beginning coordinates and target coordinates to have a variety of distances that we can test. After observing the results, it can be noted that A* algorithm was just slightly faster in all of the tests. These

results support our initial hypothesis that the A* algorithm is speedier than Dijkstra's algorithm in a simple maze without other agents. Since the results show that A* is only slightly faster than Dijkstra's algorithm, it isn't the most relliable source to assume that A* is the superior algorithm. As we take a look at 2, we can see the runtimes of the two algorithms start to diverge as A* shows its true potential. Once we start adding obstacles in to the mazes, we can see that Dijkstra's algorithm does not do well as it's performance starts to dwindle compared to A*. Switching the ways that the obstacles are oriented can cause some lag time in the speed of A* while also increasing the speed of Dijkstra's algorithm, but these tests on the advanced maze do show that A* search is still more efficient. Lastly, as we test on the real world road data of the Minneapolis maps, we can finally confirm that A* algorithm is the top pathfinding algorithm compared to Dijkstra's algorithm. The results show that, although not by a whole lot, A* algorithm on average is about 14 ms faster than Dijkstra's algorithm. Even if the path distance is large or small, there is still a decent gap between A* and Dijkstra runtimes on the maps.

The runtimes of these algorithms on the three different maps stays pretty consistent as the maps become increasingly more complicated. For both algorithms, they returned null on the third test while searching a path that does not even exist in the data set. This can either show that the algorithms are working and that they showed that there is not a path towards the target node, or it can show that the target node not being in the data set caused an error with the search and returned null to show that it cannot find a point with those coordinates. All in all, we can see that as the distance of the initial node and the target node increase, the runtime of both of the algorithms increase as well, except for a few cases. This can be seen from case 4 to case 5 as the distance increase from 1403.28 to 2497.17 and the runtimes for both algorithms decrease from A*'s 0.021 seconds and Dijkstra's 0.032 to A*'s 0.0113 and Dijkstra's 0.0251 seconds. This can be attributed to some roads being more connected towards the target road compared to other roads. Dijkstra's algorithm being much slower than A* algorithm can be linked to the fact that Dijkstra's algorithm typically favors nodes that are close to where it began, causing the algorithm to search more around the initial starting position and if the target node is closer to the begining node, it will have a quicker runtime. It can also cause issues where there are negative edges but luckily there are no negative edges in the Minneapolis road data.

6 Conclusion and Future Work

The results of the experiments support what we have read in other articles, showing that A* search is a much better algorithm than Dijkstra's algorithm through all of the tests I have ran. From simple, small mazes that contain no obstacles to larger mazes with a variation of obstacles to real world data that shows the road systems of Minneapolis, both pathfinding algorithms showed that they can find the quickest, best path in a very short amount of time. That being said, we have proven and supported our intial thoughts that A* algorithm would be a better pathfinding algorithm when it comes to a more simple, single-agent environment. Even as we add obstacles and other things to look for such as one way streets we can see that the A* search algorithm can outperform Dijkstra's algorithm consistantly. This can mainly be attributed to the fact that A* algorithm is a complete algorithm and can even be morphed into other algorithms. A* is a complete algorithm because it is guarenteed to return a correct answer no matter the input and it will guarantee a false (or in this case null) if there is no correct answer. Another advantage that A* has is the fact that is thrives in an known environment because of the use of a heuristic function

in the algorithm. By knowing the environment and what kind of things to look out for, the user can adjust the heuristic function to better suit the environment and allow for an optimal solution. With that being said, Dijkstra's algorithm should not be ignored as it is very efficient in many ways and can keep up easily with A* runtimes. The advantages of Dijkstra's algorithm can be seen when it is used in an unknown environment, allowing Dijkstra's algorithm to outperform A* in the case where you do not know where or what the target node is. It has also been tested and shown in other experiements that Dijkstra's algorithm can outperform A* when there are other agents in the environment. This is because dijkstra's algorithm usually covers a larger area of the graph compared to A*, allowing it to either avoid the agents as it searches through the graph or in the case of multiple targets that you do not know.

As for future work, there is a lot of testing that can be done with these two algorithms. As noted in the Literature Review, we can see that there are many different variations of A* algorithm that can be compared to the base A* algorithm to see if there are ways to improve it. With a modified heuristic function along with a well known environment and modifications to the algorithm, we could have an even more optimal algorithm that greatly outperform A* algorithm on more complex graphs. Future work can also include testing multiple agent environments to determine if agents do in fact have an impact on if A* or Dijkstra's is a more efficient algorithm. Using that data, we could then move on to a more exciting type of experimentation where we can use these algorithms on NPC characters in live video games to test which one could complete a task the quickest in a very complex environment. The future of pathfinding is bright and there can be many things to learn to increase what we know today.

References

[1] Dijkstra's algorithm vs a* algorithm detailed comparison as of 2020.

[2] Python program for dijkstra's shortest path algorithm — greedy algo-7 - geeksforgeeks. (Accessed on 04/14/2020).

[3] Y. Björnsson, M. Enzenberger, R. C. Holte, and J. Schaeffer. Fringe search: Beating a* at pathfinding on game maps. *CIG*, 5:125–132, 2005.

[4] X. Cui and H. Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[5] A. Goyal, P. Mogha, R. Luthra, and N. Sangwan. Path finding: A* or dijkstra's? *International Journal in IT & Engineering*, 2(1):1–15, 2014.

[6] R. Graham, H. McCabe, and S. Sheridan. Pathfinding in computer games. *The ITB Journal*, 4(2):6, 2003.

[7] D. D. Harabor and A. Grastien. Online graph pruning for pathfinding on grid maps. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[8] K. Hong. Python tutorial: Dijkstra's shortest path algorithm - 2020. (Accessed on 05/10/2020).

[9] A. P. Introduction to a*, May 2014.

[10] J. Parker. projectdescription.pdf. (Accessed on 05/10/2020).

[11] H. Reddy. Path finding-dijkstra's and a* algorithm's. *International Journal in IT and Engineering*, pages 1–15, 2013.

[12] D. Silver. Cooperative pathfinding. *AIIDE*, 1:117–122, 2005.

[13] N. Swift. Easy a* (star) pathfinding - nicholas swift - medium. (Accessed on 05/10/2020).