Implementation of Various Search Algorithms and How They Differ in Random Settings

Bradley Kim, Trey Xiong, Chan Lu

December 17 2021

1 Abstract

For the purpose of this paper, we will be focusing on the 3 following search algorithms: A* Search, Breadth-First Search, and Depth-First Search. Our project will generate random nodes, which will serve as obstacles in our environment, and then generate a randomly placed agent whose purpose is to find the randomly placed goal node. We will program our agent to traverse the map in the three respective search algorithms, and record the time and distance it took for the agent to find the goal, along with the number of nodes the agent needed to open. By randomizing the obstacles, agent, and the goal node, we will be able to avoid any bias in our project, and effectively compare the averaged recorded time and distance of each search algorithm run numerous times in various environments.

2 Introduction

Our group decided to focus on the implementation of search algorithms and its benefits because we wanted to research and base our project on a topic that was relevant to the course. In the first half of the semester, we learned about various types of search algorithms and in which situations one specific algorithm would be advantageous compared to the rest. While we were given an understanding of this through the readings in the textbook and images of how each algorithm traverses the graph, we were never shown how a machine would interpret these algorithms and use them to actually traverse an unknown environment to find what it is looking for. To actually prove that each search is different and has its own benefits and hindrance, our group decided to physically map out these searches by showing every movement and the nodes the agent visits until it finds the goal node. Due to the lack of time and for the simplicity of this project, our group has decided to focus on just three search algorithms, being DFS, BFS and A* Search. We specifically chose these because DFS and BFS are the standard and most popular search methods whose runtime varies tremendously depending on the number of nodes and the complexity of the graph. By choosing these two algorithms, we will be able to easily identify their benefits and when each algorithm should be used, further proving the reliability of the information that was given to us from the textbook. We chose A* to be our last search because of its importance in this course, and as it is an algorithm that our group never studied prior to this semester, we believed it was appropriate for us to study and fully understand its purposes and how it compares to the other algorithms.

When people think of graphs and the way search algorithms work, many think that it is very straightforward and not interactive, in that the user simply waits for the search to run and for it to return either true or false depending on whether the goal state was found. For our project, our purpose is to prove that this is not the case by showcasing the complexity yet beauty of these algorithms and by making it very user-friendly through implementing many options for what the user wants the agent, the goal node, and the obstacles to be. For example, we have created an option for the traversal to be Super Mario themed, in that we have implemented our agent to be Mario, the goal node to be Princess Peach, and for the obstacles to be Super Mario blocks. By using various themes for our project, we hope to capture the user's interest in the way search algorithms work, and as they continue running the program with the various random maps generated, they will ultimately gain a better understanding of the differing search methods and when one might be better than the other. Furthermore, we have also added a function where the user may continue creating obstacles by tracking their cursor movements. When the program is running, in addition to the initially generated obstacles, the user can continue creating more obstacles by dragging their cursor around the map, which would mean more obstacles for the agent to keep track of and avoid while finding its path to the destination.

While we were taught various search methods and the way they uniquely traverse the graph, we were not shown how it is relevant to artificial intelligence and the ways an agent may use these algorithms to travel across unknown environments to find its destination. With everything we learned about DFS, BFS, and A* search from this course, our group decided to physically showcase this for our project and have opted for an interactive approach, where users can view in real-time the movements the agents are making depending on the search algorithm and how it may react as more obstacles are constantly getting added into its environment.

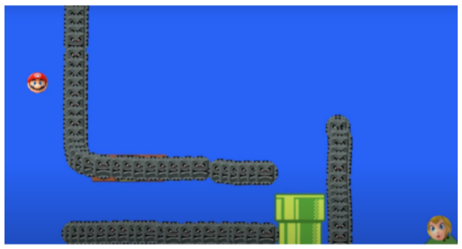


Figure 1: An example of one of the themes used

3 BFS

Breadth-First Search (BFS) is an uninformed search algorithm that has no awareness of how close it is to the goal state when traversing the graph. Traditionally, a BFS is first initialized from the root of the graph, and then visits all adjacent nodes from the selected node. Once it has traversed and visited an entire row of nodes, then it will move on to the next row and continue traversing the graph until a goal state is reached [RN21]. In order for BFS to be an optimal solution, all path costs must be non-negative, and therefore BFS is most commonly used when all actions have the same cost. Its time and space complexity, as stated by Peter Norvig, is completely dependent on the branching factor and the depth of the graph. This is because BFS is a complete and optimal algorithm for all graphs with non-negative costs, as it will explore all possible nodes, and therefore all possible paths that reach the goal state [RN21]. Therefore, there is a tradeoff between BFS' time and space complexity to how well the algorithm runs when compared to others.

There has been a lot of research and exploration on the fundamentals of a BFS ever since its creation in the mid 20th century, and one topic of discussion is the bias of a BFS towards the higher degree nodes. Studied by Kurant and her colleagues, there are two categories of graph exploration techniques, one being a Random Walk and the other being a Graph Transversal Technique. In Random Walk, all nodes can be revisited, with there being no bias towards which ones are selected. For Graph Transversal Technique, however, similar to a BFS, all nodes are only visited once [MKT10]. Through their study on various Graph Transversal Techniques, and specifically BFS, they have found that they all share a common trait in spending more time exploring nodes that are deeper into the tree than the shallower ones. They prove this through calculating the node degree distribution to the fraction of covered nodes, and after proving this bias, move towards introducing a refined version of BFS that does not account for such bias.

Another mention of the issues with BFS is in an article published by Lovasz, which covers the exact bias and solutions to lessen with bias to make BFS a better algorithm [Lov93]. Both articles state that while BFS is a Graph Transversal Technique and innately biased, a way to fix it is through the usage of BFS towards a Random Walk implementation. Through this implementation, it would be possible for BFS to revisit nodes and traverse both ways of the tree. As this would lead to a much larger time and space complexity, both articles state that this would not be ideal for very large datasets and should only be used for smaller sample sizes. For Kurant and her colleagues' experiment, they performed a study on this new BFS on the sampling of Facebook social networks and found that while their new BFS took much longer time and memory, they were able to quantify and minimize the bias the traditional BFS has for higher degree nodes. While they were able to reduce the bias through Random Walk, it is not a very efficient method, and while BFS is a very traditional, costly, and complete algorithm, it will be very difficult to find ways of improving it without costly runtime sacrifices.

4 A*

Another search algorithm that we will be analyzing is A* search. This search algorithm is the only informed search algorithm that we will be exploring in this literature review. An informed search algorithm is one that uses environment specific knowledge about the goal state to find more efficient solutions than other algorithms which do not know anything about the goal state [RN21].

The A* algorithm was originally created and presented by Hart, et al in 1968 to tackle the shortest path between origin and destination problem [Zen09]. This was done by maintaining a set of candidate nodes and applying a best-first method to select from this list of candidate nodes. This sounds very similar to Djisktsra's algorithm which was published in 1956 which also used a similar approach as described above [Zen09]. However, A* takes the environment specific knowledge to its full advantage and creates another weight (the heuristic) to add onto the cost of the length of the path to find a more efficient path than Djiskstra's. This heuristic calculation is actually the most crucial part of the A* search algorithm. This key factor plays a direct part in determining both the efficiency and accuracy of the path searching algorithm. This heuristic value can be thought of as a guess at the best path from the current state to the goal state. This value can come from many different ways or calculations, which is why selecting the most fitting heuristic calculation for the current problem is an important part of actual implementation. The one key property for selecting this heuristic is making sure that it is admissible, meaning that the estimation is never overestimating the cost to reach the goal [RN21].

In a study done by Zeng, many different search algorithms were created with efficiency first in mind to test against real world road networks. Specifically, Zeng wanted to measure the difference in performance of A* and Dijkstra's in an identical environment. These algorithms were run 500 times on the road networks and compared based on their optimality and speed. What the study found was that smaller data sets ran between 60-70 percent faster for A* and larger data sets were averaging half the amount of time needed for A* compared to Dijkstra's. This is important because this shows that A* is competitive in a spatial data environment where coordinates can be used as data for the domain specific environment. In this study, the heuristic used was the Euclidean distance formula. This efficiency comes from the fact that A* is selecting from a smaller set of nodes to explore since the weight is pulling in nodes that work towards getting to the goal.

This base A* algorithm with a constant heuristic value continues to be improved even to this day. In an article on one modified A* algorithm by Koenig and Likhachev, they recreated A* but with a way to iteratively update their heuristic after each search which made the heuristic more informed each time and ultimately proving more efficient. This has great implications for the future of path finding algorithms since finding an optimal way to create heuristics for complex problems can greatly improve the speed at which we can solve these problems. Koenig and Likhachev already talk about seeing improvements with their implementation of this adaptive A*, but also go into details in another way that would already be more efficient than the way they implemented it. These results are of course limited to the specific domain they tested, but show the potential that modifying A* search as a base has [KL05]. All of these factors together are the reason why A* search is a very competitive algorithm in path finding and why we're excited to see how it fares against BFS and DFS in our originally built domain/environment.

5 DFS

Depth-First Search(DFS) is a traditional uninformed search algorithm that was invented for its advantages when traversing finite graphs with large depths. Unlike other search algorithms, DFS has a much lower time and space complexity, with its time requirement being O(b*m) where b is the branching factor of the graph and m is the maximum depth of the graph. Furthermore, its memory requirement is linear in respect to the number of nodes and vertices in the graph.

As DFS has been continued to be used and implemented through programming languages throughout its development, a need of a backtracking implementation was necessary in that it would have to keep track of the nodes previously visited to ensure that the search would not get stuck in an infinite loop and to guarantee that every connected nodes would be explored. As explained by Karleigh Moore, "[DFS] goes as far as it can down a given branch(path), then backtracks until it finds an unexplored path, and then explores it." [Koz92]. Regardless of whether the graph is directed or undirected, DFS will need the implementation of a backtracking algorithm, as once it comes across a leaf node with no further branching factors, it will have to "backtrack" to the previously explored node and continue exploring until every node of the graph has been visited. For undirected graphs, a DFS would first pick a random node of the graph, visit that node, marking the node as explored, and pick another connected node that has yet to be explored. Because undirected graphs can traverse in any direction as long as the nodes are connected, it is especially important to mark the visited nodes as visited and implement a backtracking algorithm so that the search isn't continuously visiting already visited nodes and will eventually explore the entire graph. For directed graphs, a DFS would work by first choosing the root node as its starting point and traversing down the tree. However, because a tree does not contain any cycles and only moves in one direction, we would not need to keep track of the visited nodes, as each node can only be visited once throughout its search. From Kozen's analysis paper comparing DFS to other popular search algorithms, he states that because of how fast and efficient DFS is, it has been a very useful tool in analyzing graphs, but it does come with its disadvantages in that it will not always find the shortest possible path and will not guarantee a solution for infinite search trees [KG17]. While DFS is a popular algorithm, it may not always be optimal, as when it does find a solution, it will usually not be in the shortest amount of steps possible.

There have been further concerns regarding DFS and its implementation of a backtracking algorithm. Stated by Navneet, “ if a [DFS] algorithm is used for searching an element in the Directed Acyclic Graphs (DAGs), then a lot of time is wasted in the back-tracking” [Awe85]. Navneet suggests a different approach to using DFS, called Reverse Hierarchical Search (RHS) that is able to avoid the unnecessary backtracking search and allows for a faster and more efficient method rather than the traditional DFS. Similar to Navneet, there have been several publications adjusting the disadvantages of DFS and improving the function. DFS is one of the most traditional search methods created due to its advantages compared to other algorithms, but there are still ways that it can be improved, and more publications regarding its improvement will continue being published as the study in graph traversal continues growing.

6 Approach for our Project

To approach this project and effectively display the pathways different search algorithms take, we would first need to code the outline of our map and the obstacles within. We would need to develop a software that will randomly place various nodes of different radii around our focused grid to serve as our obstacles, and then randomly place our agent and our goal node. While this may sound straightforward on paper, because our project is an implementation of a probabilistic roadmap, we would also need to take into account collision and bounds checking for all obstacle, agent, and goal nodes to ensure that they are all inside the perimeter and are not overlapping one another when initially placed in our grid. Once all the randomly generated obstacle nodes are placed, we would then need to properly set our environment through connecting each node to its visible neighbors. Our group has decided to do this through a Ray-Disk intersection approach. Through a Ray-Disk approach, each node will send out a ray out to the nodes nearby, and if the ray does not detect any obstacles between the two nodes, it will connect them as neighbors. Because the ray is very sensitive to anything that lies in its pathway, we must further track the radius of each node and the width of the laser to properly assert a corresponding node’s neighbors and for our program to run without any issues. By having each generated node in our program go through this Ray-Disk approach, we will be able to efficiently track its neighboring nodes and convert the map into a proper probabilistic roadmap.

The following step will be to implement our agent and goal node in our roadmap, and just like the randomly generated nodes, we will also randomly generate an (x,y) coordinate for the nodes to lie on the map. Because we cannot have the agent and goal node overlapping with one another or with the obstacle nodes, we will need a dataset to keep track of all coordinates where a node is present while taking into account its radii, and place the agent and goal nodes in any space that is not occupied. Because we are approaching this project without any bias, we will also completely randomize the distance between the agent and goal node, meaning that there will be times when the agent and goal node are close together and when they are far apart. However, since we will be performing this program multiple times for each search algorithm and averaging out their run-time, path length, and path segments, it will balance out any major distance discrepancies amongst the search methods used.

The final segment of our project will consist of actually implementing Depth-First, Breadth-First, and A* Search for our agent to use to successfully traverse the grid and find the goal node. For the BFS algorithm, we will be using a queue that will take in the agent's initial position and then traverse the grid until it reaches the goal state (if the goal is not completely blocked). We will be polling nodes from the end of the queue to do this, as BFS is a LIFO algorithm. Another data structure that holds the visited nodes and parent nodes will also be created, allowing us to backtrack and review all the paths that the agent took. This, in itself, will get the goal node with the lowest number of nodes explored, as we will be going in a level-order traversal structure. In our specific implementation, this means that we will be starting at our starting position and looking at each of its neighbors. We will then start branching out of our map by continuing to look at the other neighboring nodes of all the nodes we are visiting.

The same algorithm will also be reused for DFS, except we will be changing the queue so that it expands the nodes most recently added, or in other words, a FIFO algorithm. By using the visited boolean array for each node in our DFS, it will ensure that the algorithm will never be stuck in an infinite loop by checking for cycles, and will successfully traverse the map. This algorithm will also be starting at the starting point, but will now be branching downwards as deep as possible. If it does not find the goal, then we will start to backtrack to the beginning node and continue branching another pathway until the goal node is reached.

For our A* approach, we will be using one more array to hold information on the current state of the simulation and change the queue to a priority queue. This extra array will be a heuristic value array that will store the heuristic value of that specific node. We will be using a priority queue that will prioritize the smallest (heuristic + distance from origin value) node to explore. Thus, if we choose an admissible heuristic, it should allow us to always get the shortest distance to the goal, regardless of how many nodes need to be explored. The heuristic we have decided to choose is the distance formula from the current node position to the goal. We chose this because the grid is in a 2D box and the agent is allowed to move anywhere needed. The distance formula takes into account all these degrees of freedom for movement. For our specific implementation, this should allow our algorithm to have a pretty clear pathway to the goal, which will, in the end, save us time and give us the optimal distance. This is our most prioritized algorithm, as we want to find the optimal path for our specific environment.

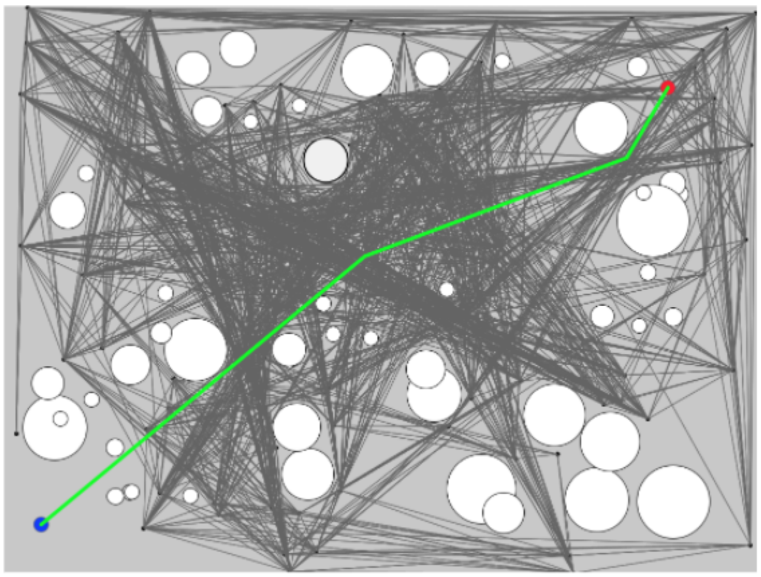


Figure 2: An outline of how our agent will find the goal

7 Description of the Design

When actually designing the experiment and coding the program to deliver the results we intend, many settings had to be taken into account. Some of the main factors that drove the design of our project included fairness, scalability, and repeatability, and we made sure to revolve our project on these three key points to deliver unbiased results.

The first thing we seeked in our project was to ensure that each algorithm we chose (A*, BFS, DFS) had an equal playing field, and to do this, we needed to somehow create a probabilistic roadmap that would not give inherent advantages to any of the algorithms. The way this was accomplished was by randomness. We incorporated randomness into the environment by randomly selecting where obstacles, vertices, and start/end points were. By doing this, we were able to create an environment that we had no say in, further limiting the bias in our project. We only controlled the amount of nodes/obstacles being randomly produced and the size of the environment. With us only controlling these two factors, we would not be responsible for any differences in performance measure, and we would be able to accurately find the differences between the three search methods.

The next thing that was taken into consideration was the scalability of the experiment. For now, we are only comparing three search algorithms, but in the future, what if we wanted more? The code was written and designed in a way where we could implement any search algorithm and quickly apply it to our fair environment and see the results that it produces. This wasn't a requirement in the scope of this project, but was more for our personal interests in creating a successful program. Gathering and analyzing data is interesting and important, and this allows us to come back at any time and see in a real environment how different search algorithms compare to each other.

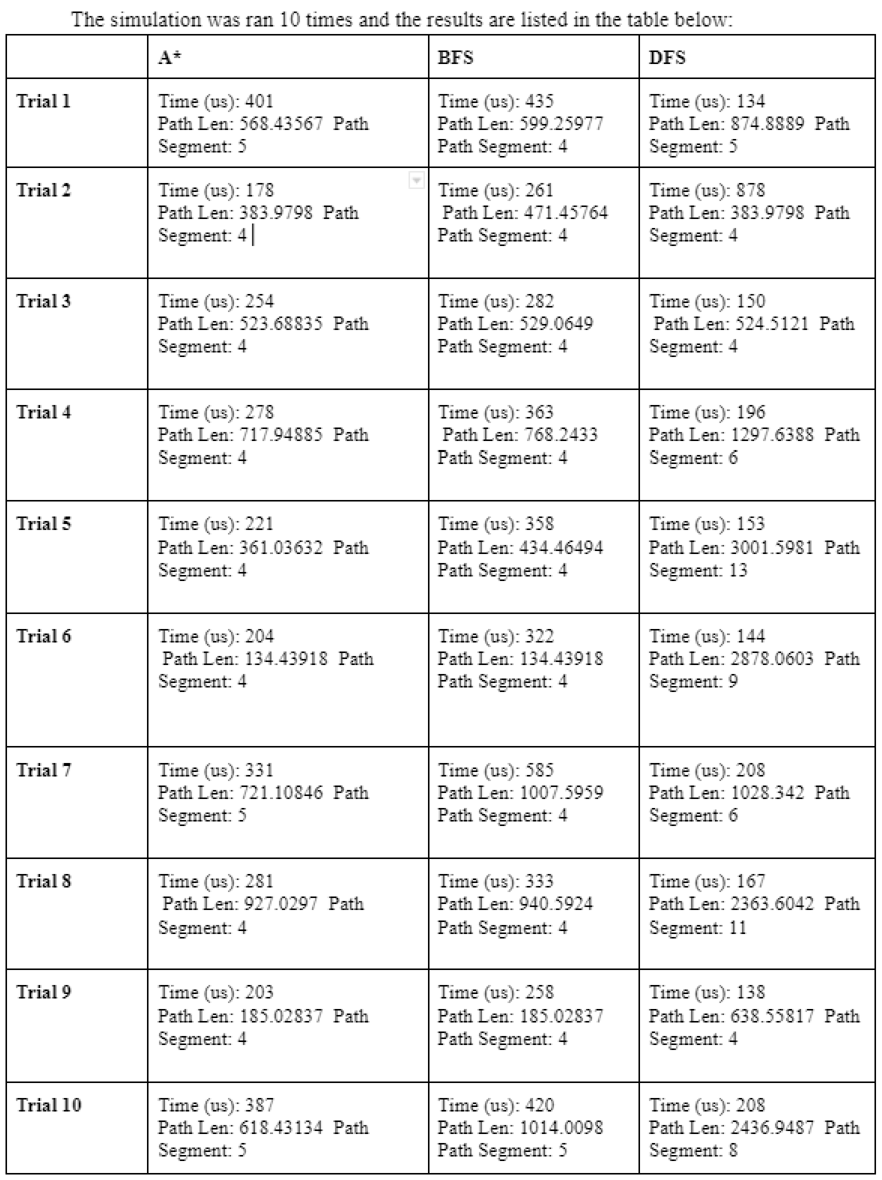
One of the last major things we accounted for is repeatability. We did not want to have to manually set up each trial run and compare each data in an inefficient manner. This would have been very tedious and time consuming, so instead, we opted for a way that our code could automatically run and compare each algorithm to each other in the same environment. This was important because we wanted to have a large amount of data before comparing, as our entire project is based on gathering enough trial data to draw significant conclusions.

All of these factors in our design come together to create an efficient experiment that presents us with results that hold value. If the environment was not random, we could have introduced some error in a way that affected the validity of the experiment. If we did not account for scalability, we would have a very constrained project with little room to improve in the future, and if we did not account for repeatability, it would have increased the man hours needed for implementing the search methods and keeping track of the results it produces. The design of this experiment was to make everything automated through the compiler while gathering enough valid data so that the experiment could provide interesting insight on different search algorithms.

8 Analysis of the Results

The experiment was run a total of 10 times, and certain metrics were taken for each run. We took into account the time to complete the search, the length of the path from the start to the goal, and the amount of nodes explored (or the path segments) in each path found. While the only data we needed was the optimal path, we also took into account various metrics to further compare the results.

When comparing the time completed for each algorithm, there is no clear winner, but there is a clear loser. From the ten trials, we found that BFS is always the slowest path finding algorithm upon completion compared to the rest. The second slowest to completion was the A* search. We found this a little odd, as we would have expected A* to be the fastest due to the nature of the heuristics. Our theory was that A* would have the lowest time since the heuristic values would allow the algorithm to have a "clear" shot to the goal consistently. However, this was not the case, as DFS consistently outperformed A* in speed. This might be for a variety of reasons, but the main being suboptimal optimization with Processing, the language we used for our project, as its interaction with priority queues throughout each trial was unpredictable and eccentric. Even though DFS being ranked first in speed was unpredictable, its low average completion time was not. Because our map consisted of a lot of vertices within the structure, it meant that the goal node was usually found to be deeper into the branches of each path, and as DFS is able to explore these



The simulation was ran 10 times and the results are listed in the table below:

	A*	BFS	DFS
Trial 1	Time (us): 401 Path Len: 568.43567 Path Segment: 5	Time (us): 435 Path Len: 599.25977 Path Segment: 4	Time (us): 134 Path Len: 874.8889 Path Segment: 5
Trial 2	Time (us): 178 Path Len: 383.9798 Path Segment: 4	Time (us): 261 Path Len: 471.45764 Path Segment: 4	Time (us): 878 Path Len: 383.9798 Path Segment: 4
Trial 3	Time (us): 254 Path Len: 523.68835 Path Segment: 4	Time (us): 282 Path Len: 529.0649 Path Segment: 4	Time (us): 150 Path Len: 524.5121 Path Segment: 4
Trial 4	Time (us): 278 Path Len: 717.94885 Path Segment: 4	Time (us): 363 Path Len: 768.2433 Path Segment: 4	Time (us): 196 Path Len: 1297.6388 Path Segment: 6
Trial 5	Time (us): 221 Path Len: 361.03632 Path Segment: 4	Time (us): 358 Path Len: 434.46494 Path Segment: 4	Time (us): 153 Path Len: 3001.5981 Path Segment: 13
Trial 6	Time (us): 204 Path Len: 134.43918 Path Segment: 4	Time (us): 322 Path Len: 134.43918 Path Segment: 4	Time (us): 144 Path Len: 2878.0603 Path Segment: 9
Trial 7	Time (us): 331 Path Len: 721.10846 Path Segment: 5	Time (us): 585 Path Len: 1007.5959 Path Segment: 4	Time (us): 208 Path Len: 1028.342 Path Segment: 6
Trial 8	Time (us): 281 Path Len: 927.0297 Path Segment: 4	Time (us): 333 Path Len: 940.5924 Path Segment: 4	Time (us): 167 Path Len: 2363.6042 Path Segment: 11
Trial 9	Time (us): 203 Path Len: 185.02837 Path Segment: 4	Time (us): 258 Path Len: 185.02837 Path Segment: 4	Time (us): 138 Path Len: 638.55817 Path Segment: 4
Trial 10	Time (us): 387 Path Len: 618.43134 Path Segment: 5	Time (us): 420 Path Len: 1014.0098 Path Segment: 5	Time (us): 208 Path Len: 2436.9487 Path Segment: 8

Figure 3: Simulation on 10 Trials

deep goals without needing to worry about the number of nodes explored or the optimization of the shortest path, it would almost always outperform both BFS and A* Search.

Next, we will be analyzing the remaining factors: path length and nodes explored. Our worst algorithm for finding an optimal path in terms of both path length and nodes explored was DFS, as this algorithm made the agent search all across the map, only to return the goal node regardless of how inefficient the path is. This was in line with our predictions for the results as well, as DFS has no guarantee of finding an optimal path and will simply return the first path that succeeds. After DFS, our second worst search algorithm in terms of path length was BFS, but while it was just short of being on par with A*, it still had a much lower average path length when compared to DFS. The one thing that BFS does best is finding the optimal solution through the lowest number of nodes explored, and our project evidenced this information, resulting in BFS being the best performer for this metric.

While one way of finding the optimal path is through the number of nodes explored, another way would be through finding the shortest distance covered by the agent. A* is an explicit example of this, as it will always find the lowest path in terms of cost, even if it has to explore more nodes than other search algorithms such as BFS. We see this evidenced with our results, as while BFS always had the lowest amount of nodes explored, A* always had the shortest optimal path. This is especially prevalent in Trial 7, where we see that while BFS explored 4 path segments and traveled a total distance of 1007.5959, A* explored 5 path segments but only had a total distance of 721.10846. While A* had more path segments than BFS, its total path length was much lower, resulting in A* being the better choice of the two.

From these results, it is clear that A* is the most suited search algorithm for what we are trying to achieve within the scope of this project. It is guaranteed to provide us with the optimal path and succeeds at a satisfactory time of completion in all environments.

9 Conclusion

Based on the results of our project, Breadth-First Search will always find the shortest path in terms of nodes explored, but this does not guarantee the optimal solution in terms of distance covered. For Depth-First Search, we know that the algorithm is not guaranteed to find either the shortest path in terms of nodes explored or distance traveled, but if the solution is further away from the starting point, it will usually perform better just from the nature of how DFS traverses the grid. This is because BFS' time complexity is based on the diameter of the grid, whereas DFS is based on the depth of the grid. For A* Search, while it is not guaranteed to explore the least amount of nodes to reach the path, it will always find the optimal solution through the shortest distance actually covered, meaning that, at worst, A* should always tie with either BFS or DFS and will never cover a greater distance than the two methods. Although our implementation of A* was not the fastest in terms of time, we believe that this had to do with the usage of Processing and its underlying application of priority queues being inherently slower than regular queues. If we were to replicate this project with a different language that is more optimized with graph traversals, the time of completion for both A* and DFS should be very similar for each trial.

When comparing the three algorithms used in our project, A* Search is the best contender for our virtual environment, as it always guarantees the shortest path to the goal node, which is the most important aspect when traversing a directed, weighted graph.

This was a very enjoyable project because we had the opportunity to test the knowledge we learned in this course and grasp a better understanding on how differing search algorithms are implemented for agents in artificial intelligence, and we believe that it was a strong way to tie the materials we learned together into a final paper.

References

[Awe85] Baruch Awerbuch. A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147–150, 1985.

[KG17] Navneet Kaur and Deepak Garg. Analysis of the depth first search algorithms. *Data mining and knowledge engineering*, 4(1):37–41, 2017.

[KL05] Sven Koenig and Maxim Likhachev. Adaptive a*. In *International Joint Conference on Autonomous Agents and Multiagent Systems*, 2005.

[Koz92] Dexter C Kozen. Depth-first and breadth-first search. In *The design and analysis of algorithms*, pages 19–24. Springer, 1992.

[Lov93] L. Lovasz. Random walks on graphs: A survey. *Bolyai Society Mathematical Studies 2*, 2(1):14–23, 1993.

[MKT10] A. Markopoulou M. Kurant and P. Thiran. On the bias of bfs (breadth first search). *22nd International Teletraffic Congress*, 1(1):1–8, 2010.

[RN21] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach.* Pearson Education, London, United Kingdom, 2021.

[Zen09] Zeng. Finding shortest paths on real road networks: the case for a*. *International Journal of Geographical Information Science*, 23(4):531–5413, 2009.

10 Contributions

Description, Writing 4, Conclusion - Chan Lu
Approach, Writing 4, Design and Analysis - Trey Xiong
Abstract, Introduction, Writing 4, Analysis, Conclusion - Bradley Kim