Optimal Path-Finding on a Map

Jacynda Alatoma and Rajan Singh

4 May 2023

Abstract

In this report, we explore the question of which path-finding algorithm between A*, Greedy Best First Search (GBFS), and Dijkstra's algorithm has the best time and space complexity when finding the shortest path from one point to another in a map of a Minneapolis road system. We ran an experiment on the same map with different lengths of paths using each algorithm on each length, and then comparing the time and space complexities of each of the algorithms. Based off the results, we concluded that Dijkstra's algorithm performed the best in this context, but each algorithm has different space complexity. GBFS was the most efficient in terms of memory. Additionally, we concluded that more work needed to be done with a large map and a larger graph to see the differences in the algorithms better.

Research Question

When considering search algorithms, it can be overwhelming to see how many kinds of algorithms one can use for their application. For someone who does not understand the algorithms, it might seem unnecessary to have algorithms that do the same thing with slight differences. When applying these algorithms, however, it becomes clear why there are so many and that is because there are so many different applications that require specific benefits. If we look at our three algorithms of A*, Greedy Best First Search, and Dijkstra's algorithm, we see that there are small, but key differences that make them unique. Comparing these algorithms is not as simple as one may think because of the different requirements computer scientists want from an algorithm. There are many different metrics that can be considered and these myriad characteristics make this question of which algorithm is best such a complex problem. When we look at the theory of these functions, the utilization of a heuristic function in A* should give it the ability to get a solution faster than Dijkstra's algorithm, but space complexity should increase. While the theoretical analysis of the algorithms that gives us this result is important, it is also important to test the algorithms and confirm the theoretical results. In A*, the inclusion of the heuristic function means that more memory will be necessary to store the h(x) values, but it is possible that the heuristic will consistently give us solutions that require less use of the queue structure and this could

reduce the memory usage. These factors cause ambiguous changes to the memory usage of the algorithm, so we test the algorithm to observe how it will behave when applied to a real map. In some cases, time complexity is a more important metric, so exploring how the usage of heuristic functions in Greedy BFS and A* affect time complexity compared to Dijkstra's algorithm's lack of a heuristic function is another important result.

Analysis of these differences can help show the relative effectiveness of heuristic functions in path-finding. As stated earlier, there are many applications for path-finding algorithms and one interesting application is in graphics and video games. When a player selects a location to go to, it is up to the game to take the shortest path to get to the selected location. In this scenario, space and time complexity are incredibly important because there are so many computations occurring at the same time when a video game is being run. In video games, heuristics are often used to find the target and ensure that the routing is more focused on the target. However, if the target is unknown an uninformed algorithm can be better to find the target although it's time and space complexity is higher [1]. In addition to video games, this is a critical topic in most people's everyday lives. Many people use GPS applications such as Apple Maps and Google Maps. They rely on these applications to get them to their destinations as fast as possible. The functionality of these applications includes giving the user multiple options for routes they can take to their destination. The fastest paths, as well as the slowest paths, are provided as options to the user. Our research question is important to applications like this since developers can see which algorithms are the fastest, and then suggest those to the user first, and the slowest, or least optimal ones, last. Although our experiment is not on path-finding in video games, it is necessary to understand how the use of path-finding algorithms changes for different applications.

Background

Search algorithms have been an important topic in computer science for many years and with the prominence of artificial intelligence in the modern day, studying them has become more important than ever. The main question we are considering for our project is the comparison between the A* algorithm, Dijkstra's algorithm, and greedy best first search. A*, created by Nils Nilsson, Bertram Raphael, and Peter E. Hart from 1964 to 1968, utilizes heuristics to improve upon Dijkstra's algorithm which was created in 1959 [2]. Search algorithms can be applied to computer networks, spatial computing, and artificial intelligence, making them important for computer scientists to understand [3]. The most obvious application of these path-finding algorithms are in services such as Google and Apple maps. However, even with an application as established as Google maps, there is room for improvement. In the article "Dijkstra's Algorithm and Google Maps," the authors state how even adding some functionality for detecting negative costs would add more functionality to the algorithm in different situations [4]. Even beyond computer science, research for better path-finding algorithms is ongoing in many industries such as predicting the physical path of someone trying to breach a nuclear facility [5]. For this specific application, the researchers

found that the A* algorithm better suited their needs over Dijkstra's algorithm. This will not always be the case because there are many important metrics to consider for each specific application. It is our goal to be able to understand the general metrics of these algorithms and understand how they can be applied in different situations.

Before we compare the algorithms, we want to understand each of them. A* has many advantages compared to other algorithms, as argued in the original paper the creators of A* wrote, "It might be argued that the algorithms of Moore, Busacker and Saaty, and other equivalent algorithms [...] are advantageous because they first encounter the goal by a path with a minimum number of steps. This argument merely reflects an imprecise formulation of the problem, since it implies that the number of steps, and not the cost of each step, is the quantity to be minimized" [6]. At the time, this algorithm was ahead of many others because of how it used heuristics to choose which nodes to expand. The equation for a path in a graph using A* can be shown as

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

with g(x) being the path cost and h(x) being the heuristic estimate of the remaining cost to reach the goal node [2]. Almost sixty years later, however, much more research has been done and new algorithms have been formulated. One popular algorithm that was created is D* Lite, which is an adapted version of A*. There are certain situations where D* is a better fit because it will expand less nodes than A*, but A* has the advantage of being more generalized [7].

On the other hand, Dijkstra's algorithm is a simpler path-finding algorithm that is even easier to implement. If we look back to the equation above we can alter it to represent Dijkstra's algorithm as

$$f(x) = g(x)$$

because Dijkstra's original algorithm does not utilize heuristic functions. As we can see, the main difference between the two algorithms is the usage of a heuristic function. Because of this, the efficiency of the A* algorithm is dependent on $h(x)$, also known as an evaluation function [8].

If the situation where we have no $h(x)$ results in Dijkstra's algorithm, we also want to know about the converse situation where we path-find based off of the equation

$$f(x) = h(x)$$

In this case, we have greedy best first search, where the algorithm expands nodes solely based on the estimated length of the path to the goal. Regardless of the immediate cost, greedy best first search will always choose the path that has the lowest estimation of the path to the goal node. Greedy best first search does not have one defined algorithm and can be treated as a family of algorithms that follow the aforementioned general rule. They

can differ in their tie breaking strategy. When we speak about tie breaking strategy, the issue is when we have two paths with equal $h(x)$ values. As Heusner, Keller, and Helmert state, "Given a state space topology and a tie-breaking strategy, GBFS performs a uniquely determined sequence of successive state expansions." When we have those two criteria, we get the individual greedy best first search algorithms [9].

With any algorithm, there will be drawbacks. The paper "How Good is Almost Perfect?" by Malte Helmert and Gabriele Röger explores the performance of A* in planning domains. Planning domains are a specific type of functionality that artificial intelligence can have [10]. For example, when we send a robot to Mars, we want it to be able to plan a path based off of information it senses. Planning remains costly due to its complexity, but it has uses that extend to areas such as scheduling construction [11]. As they studied how A* works in these situations, the authors found that in cases where we are dealing with a robot arm that needs to move balls, A* with near perfect heuristics does not have a significant difference in performance from breadth-first search. While this may make it seem that A* is not effective, there are many situations in which A* is better, but these algorithms are applied to many different situations and one algorithm cannot be the best for all situations. One point that the authors make is that this could change if there was a "perfect heuristic". In these situations, they had "almost perfect" heuristics such that if h* is a perfect heuristic, we can represent these almost perfect ones as

$$h^* - c$$

with c being some constant. For most situations, almost perfect heuristics are unattainable which shows that A* may not be the right algorithm for planning domains [12]. The goal of computing perfect heuristics is an ongoing pursuit. Another paper that Helmert contributed on, "Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning", shows a way to create a perfect heuristic. The Merge-and-Shrink (M and S) Abstraction operates in a abstract state space that is created by merging abstraction states and shrinking them by combining those states which gives us the name. M and S is very effective in creating heuristics for planning. The paper goes on to describe bisimulation and how label reduction can be used to simplify the algorithm so that we can produce perfect heuristics in polynomial time [13]. While this is a great advancement for heuristics, it is very difficult to find situations where we can apply these types of heuristics. In our project, we are applying heuristics to finding the shortest paths in a city and we are more concerned with heuristics as they relate to transportation. The paper, "Heuristic shortest path algorithms for transportation applications: State of the art" explores the applications of heuristics in transportation.

One of the main problems with using heuristics in transportation is that it is too complex to give updates in real time. However, there are many ideas on how to limit the complexity to make heuristics more useful for transportation. Currently, one heuristic algorithm that is used in many situations is a label-correcting (LC) algorithm which utilizes a list structure to

scan the set of nodes when finding the shortest path. In the paper, the authors explain how this heuristic is different, "The major feature of an LC algorithm is that it cannot provide the shortest path between two nodes before the shortest path to every node in the network is identified." It is clear how this algorithm has a large amount of complexity and it would be computationally difficult to check every path when we're trying to going from point A to point B in a city. The authors go on to explain some possible solutions. One of the proposed solutions is to limit the search area. For an LC algorithm, it is clear how this would reduce the complexity of the algorithm. It is unlikely that the best route from one location to another goes outside of the main roads that connect the two, so limiting the search area will help reduce the number of nodes expanded while still finding the best path most of the time. Another proposed solution is the branch pruning method. Similar to limiting the area, branch pruning reduces the need to expand nodes that won't lead to the optimal solution. One equation that is suggested by Fu is

$$L_{(i)} + e_{(i,d)} \leq E_{(o,d)}$$

where $L_{(i)}$ is the current minimum travel time from origin node o to node i; $e_{(i,d)}$ the estimated travel time from node i to destination node d; $E_{(o,d)}$ an estimated upper bound of the minimum travel time from the origin node to the destination node." Combining this with an LC algorithm creates a branch pruning LC algorithm that prunes based off of travel time while exploring many nodes [14]. There are many different heuristics that can be applied to this situation, but we will just consider A*, a greedy best first search algorithm, and Dijkstra's algorithm to simplify our experiment.

We must compare and consider other experiments that have also attempted to solve the problem we are trying to solve. There is some research currently that tests the problem we are trying to solve. In the article "Finding shortest paths on real road networks: the case for A*," the researchers are interested in finding optimal paths on a map, similar to what we are interested in doing. They discovered that traversing a map of Santa Barbara using a priority queue for Dijkstra and A^* algorithms performed poorly [15]. However, this was the case when they used a double-bucket implementation. When testing on a smaller map of Santa Barbara, the run times for A^* were 60-70% lower than those of Dijkstra. Path-finding algorithms are also used frequently when planning road network systems. In the article "Fast Shortest Path Algorithms for Large Road Networks" the authors propose a how to use symmetrical Dijkstra for planning road networks. Symmetrical Dijkstra differs from the Naive implementation of Dijkstra by using a bi-directional approach. This was first introduced by Pohl to reduce the time complexity to $O(b^{d/2})$ from $O(b^d)$ in the traditional Dijsktra's algorithm [16]. This new bidirectional method implements two search trees. First, from the origin, giving a tree spanning a set of nodes L_f for which the minimum distance and time from the origin is known, and second, from the destination that gives a tree spanning a set of nodes L_b for which the minimum distance and time to the destination is known. After creating these trees they iteratively add one node or the other until an arc is created for either one of the trees. They also take into account a weighted A^* approach, which takes into account a

multiplicative factor to weight the evaluation function. It is important to note that weighted A^* doesn't always find the optimal path. Their results showed that using a symmetric Dijkstra their results did not improve, and the regular Dijkstra algorithm outperformed all of the ones they tested. Both of the experiments above had contradicting results, which is as expected since the performance of an algorithm depends on the environment in which it's being tested. These experiments, and ones we find along the way, will be very helpful in our understanding and learning of our own experiment.

As discussed above, these three algorithms are very important to research as many people rely on them for their everyday activities. To highlight some of the points above, we see that A^* and Dijkstra are related through the absence or presence of a heuristic function, and greedy best-first search expands solely on the length of the path to the goal. As with any algorithm, there will be drawbacks. Most them depend on the characteristics of the environment in which the algorithm is searching. A^* and GBFS perform almost identical when A^* is operating under perfect heuristics. However, realistically no heuristics are ever perfect. Changing the heuristics plays a great role in the performance of algorithms. Many past experiments have implemented A^*, Dijkstra, and greedy best-first search in various applications. Referencing similar experiments and past work in optimal path-finding algorithms will be crucial in designing any experiment to see how the results compare with others. Ultimately, every situation varies, so different algorithms will perform differently. In designing a new experiment, it will vary whether A^* and its variations in heuristics, Dijkstra, or GBFS will find the most optimal solution on the map that one chooses.

Experimental Approach

We used our own code for all three algorithms and used a map of the Dinkytown, Cedar-Riverside, and the UMN Campus areas. We completed each algorithm twice on a short, medium, and long path. The results that we are seeking are time complexity and space complexity. This project was coded in coding language C. The first thing to do when coding this was to translate the map to a graph/grid-like structure. This was done by dividing the map into a grid of nodes and connecting adjacent nodes with edges. The size of the grid is important in terms of accuracy and efficiency. Larger graphs are more detailed since they have more edges and nodes, but there is a trade-off with missing specific obstacles in a path and completely disregarding them. Our goal was to create a graph that wouldn't miss obstacles, such as buildings, to provide a better estimation of each algorithm's path. There are other more advance methods such as Voronoi diagrams that could also describe maps and graphs in more detail [17]. However, this was not implemented due to time constraints and the simplicity of the map we used, it wouldn't have added much value for this project and what we were researching.

Next, the algorithms were coded after the map was translated into a graph. A* is an informed search algorithm that expands nodes in accordance with their anticipated cost to

reach the goal, taking into account both the actual cost of getting to the node and the estimated cost to reach the goal. The estimate is derived from a heuristic function, which is both admissible and consistent. To be admissible and consistent is to never overestimate the cost to the a node. A* uses this heuristic to prioritize expanding nodes that are more likely to be closer to the optimal solution, and avoiding nodes that might not. In practice, A* tends to be more efficient in finding a path over BFS, since it can quickly prune the map's search space of nodes that aren't likely to lead to the optimal solution. Taking this into account, we thought to use a Euclidean distance heuristic. The main reason for choosing this was because it was the easiest for us to find resources online for how to implement it. After doing some research, it also seemed like Euclidean distance was a good heuristic to use, as long as it was implemented correctly, even if it made A* run a little slower. Some webpages suggested to make it faster by getting rid of the "expensive" square root in the Euclidean function, but doing this can cause the function to either degrade into GBFS or Dijsktra's algorithms, so we made sure to avoid doing that [18]. Pseudo-code for what we wrote is below in Figure 1, for the A* algorithm. Before the section that you see there, there are error checks for the start and end nodes, as well as initialization of a priority queue for paths.

```
while possible_paths is not empty
    candidate = possible_paths.top()
    possible_paths.pop()

    path_end = candidate.path.top()
    if path_end is not in visited
        visited.add(path_end)

        if path_end is equal to to
            // we found our result
            return candidate.path.toList()

        path_end_node = graph.getNode(path_end)
        if path_end_node is null
            print "Encountered node not in graph, named: " + path_end
            continue

        next_steps = path_end_node.getNeighbors()
        for next in next_steps
            next_name = next.getName()

            // add new candidate paths to the queue
            possible_paths.push(new CandidatePath(
                candidate.path.push(next_name),
                candidate.distance + cost.calculate(path_end_node.getPosition(), next.getPosition()),
                heuristic.calculate(next.getPosition(), terminal_node.getPosition())
            ))

// path not found
return null
```

Figure 1: A* Pseudo-Code Algorithm

Dijkstra's algorithm is the other search algorithm that we coded for this project. It works by maintaining a set of visited nodes and a priority queue of unvisited nodes. Dijkstra's is guaranteed to find the optimal path in a weighted graph. That being said, it might not find the shortest path and in practice, Dijkstra's can be slower than A* and GBFS. A disadvantage of this algorithm is that its prone to getting stuck in infinite loops or explore long paths before finding the optimal path. Taking this into account when planning the coding we thought to initialize the starting node with a small cost and all other nodes in the graph with a large cost. The cost of its neighbors could then be updated by adding the cost of moving from the current node to the neighbor. If the updated cost is lower than the previous cost of the neighbor, we could add the neighbor to the priority queue with its updated cost. This would continue until the goal node is reached (the destination).

Lastly, we planned the approach to GBFS. This algorithm prioritizes the nodes by heuristic value without looking at the actual cost to reach the node. The lower heuristic value, the better. Because of how GBFS operates it makes it a fast algorithm in many cases, but there is a possibility of it getting stuck in in a local optimum or a sub-optimal path. In comparison

to A*, it isn't guaranteed to always find the optimal path in a weighted graph. GBFS can also sometimes expand too many nodes, causing the search to become very slow. Since GBFS uses a heuristic function, and we ultimately want to compare it to A*, we used the same Euclidean distance heuristic to compute the paths. Our implementation of GBFS is similar to A*, with a few key differences. One of those being the priority queue being sorted based off the heuristic value alone, and not considering the cost to the node. In Figure 2 below, the pseudo-code for this different priority queue is shown.

```
unordered_set<string> visited;
priority_queue<CandidatePath*, vector<CandidatePath*>,
    function<bool(const CandidatePath*, const CandidatePath*)>>
    possible_paths([this](const CandidatePath* a, const CandidatePath* b) {
        return heuristic->Calculate(a->path->Top()->GetPosition(), terminal_node->GetPosition()) <
               heuristic->Calculate(b->path->Top()->GetPosition(), terminal_node->GetPosition());
    });

possible_paths.push(
    new CandidatePath(
        new FStack<IGraphNode*>(start_node),
        heuristic->Calculate(start_node->GetPosition(), terminal_node->GetPosition())
    )
);
```

Figure 2: GBFS Priority Queue

Experimental Design and Results

As mentioned earlier, the setup of our experiment was to run three different length paths on each of the three algorithms, totaling to nine different results. We wanted to make sure to choose paths that would have lots of different turns in them, so we weren't just testing and looking for comparisons in straight line paths, and the algorithms would have higher possibilities of choosing different paths. Another thing we had to consider when coding this project was how we were going to collect the data from the algorithms and the paths. To do so we had to code a data-collection class. This class was able to track the path as it was created while logging all the data we wanted into a CSV (comma-separated-values) file. We could then use those files to look at the results and draw conclusions.

We used real-time (seconds) values to measure how long it took to travel the path, and we used the code's way to measure distance (we'll just call them units) which is calculated at every point in time. Each part of the map is assigned values to measure the distance of the path and is updated with vector functions throughout the code. The approximate conversion from the code's use of distance vs. real-life distance is a factor of 0.00717. For example, if the code's distance is 100, then the real-life distance is approximately 0.7 miles. This conversion was found by comparing a straight-line path on the code's map against what Google Maps said for the distance of the same path. Another thing to make our results more reliable is

we ran each of the three algorithm's multiple times on the same path, and averaged out the times and distances. The numbers you see in the tables below are the averages of each algorithm on that specific path.

The first path that we tested was our shortest whose starting point was Amundson Hall and destination was Walter Library ([44°58'26.1"N 93°13'56.8"W] to [44°58'30.9"N 93°14'08.6"W]). For A*, the total time to find the path from the two points took 15.23 seconds. The length of the path was 478.801 units. For Dijkstra's the time to find the path was 15.73 seconds and the length of the path was 473.996 units. And finally, GBFS found a path in 15.1 seconds, with the path length being 481.23 units. Below is a table with all of this information.

Algorithm	**Time (seconds)**	**Length of Path (units)**	**Space Complexity**
A*	15.23	478.801	$O(\|V\| + \|E\|)$
Dijkstra	15.73	473.996	$O(\|V\| + \|E\|)$
GBFS	15.10	481.23	$O(\|V\|)$

Table 1: Results of Shortest Path

The second path that we tested was our medium length path that's starting point was Kolthoff Hall and destination was the University Recreation and Wellness Center ([44°58'26.2"N 93°14'08.6"W] to [44°58'30.4"N 93°13'49.4"W]). Starting off with A*, the time it took for the algorithm to find its optimal path was 18.57 seconds. The length of the path found was 556.8 units. For Dijkstra's algorithm, the time it took to find the path was 17.48 seconds and the length of the path was 524.102 units. For GBFS, it took the algorithm 19.32 seconds and the path was 531.224 units. A table is below summarizing these results. Again, these are averages from running each algorithm three times on the path.

Algorithm	**Time (seconds)**	**Length of Path (units)**	**Space Complexity**
A*	18.57	556.8	$O(\|V\| + \|E\|)$
Dijkstra	17.48	524.102	$O(\|V\| + \|E\|)$
GBFS	19.32	531.224	$O(\|V\|)$

Table 2: Results of Medium Length Path

Our third and last path that we tested was our longest length path that's starting point was Blegen Hall and destination was 3M Arena at Mariucci ([44°58'19.1"N 93°14'35.2"W] to [44°58'39.3"N 93°13'42.0"W]). Starting with A*, this found a path in 53.61 seconds and the path distance was 1607.98 units. Dijsktra found a path in 53.51 seconds and the path was 1605.14 units. Lastly, GBFS found a path in 53.45 seconds and the path length was 1607.23 units. Table 3 below shows all the information from this path.

Algorithm	Time (seconds)	Length of Path (units)	Space Complexity
A*	53.61	1607.98	$O(\|V\| + \|E\|)$
Dijkstra	53.51	1605.14	$O(\|V\| + \|E\|)$
GBFS	53.45	1607.23	$O(\|V\|)$

Table 3: Results of Long Length Path

Most of what we collected agrees with the general knowledge of what these algorithms should produce, but some go against what one would normally hypothesize. Time complexity will be discussed in more detail in the following section. In addition, these results will be discussed and analyzed in the context of our research question.

Analysis

Based off our experiment and results, we have gathered data about these three types of path-finding algorithms. An important point to note is that the results from a shorter path are not as statistically reliable as a longer path, but some elements of the algorithms are shown in the data from the shorter paths. Looking at Table 1, Dijkstra's algorithm has the longest time taken at 15.73 seconds even though it has a shorter length of path than Greedy BFS. Overall, the data is very close together and we can see that the time complexity does not come into play as much in a smaller sample.

In the medium length path, we see more key elements of the algorithms in the data. A noticeable difference is that of the time. A* takes 1.09 seconds longer than Dijkstra's algorithm and the length of path is still relatively optimal. We know that the worst case runtime for A* is dependent on the heuristic that is used, so we can see from the Greedy BFS runtime of 19.32 that the heuristic function ran much slower on this specific path.

In the long length path, we get the best results since we are less likely to have outliers skew the results. In this data, we see that Greedy BFS is the fastest at 53.45 seconds which is 0.16 seconds faster than A* search which is the slowest of the three. The length of all three paths is within one percent of each other, so we can confidently say that they all found the shortest path as theory dictates. As far as time difference, there is only a 0.3 percent difference, but we must also consider space complexity and how that factors into our decision on which algorithm to use. Overall, our data shows that longer paths actually have fewer differences in terms of path length and time to find the path. This is important to note since if this was implemented in a real-life system, it might not be so important which algorithm an application uses when finding a path for the user.

For space complexity, we can see the space complexity for the algorithms based off of our analysis of the functions in the algorithms. It is notable is that there are different situations

where the algorithms are in their worst case time and space complexity. For example, if we look at these space complexities, we know that they behave differently and each run of the paths will have different outcomes for memory used. What we see in column 4 of each table is that the difference in space complexity will be much larger when there are more edges. Larger paths allow for this difference to show, so we know that the large length of path will cause more memory usage for A* and Dijkstra's algorithm than Greedy Best First Search. In our map, the paths are not long enough to show large differences in time complexity, but we know that Greedy Best First Search has a better space complexity from the length of the path.

Conclusions

Our research question was to use A*, Dijkstra, and Greedy Best-First-Search algorithms to find the optimal paths on a map of Minneapolis. We used Euclidean distance for the heuristic for the informed algorithms, and trialed each on three different length paths. Our results showed that with the two shortest length paths, Dijkstra outperformed A* and GBFS in terms of time to find a path, and the length of the path found. GBFS had the worst path lengths, so it is probable that it didn't find the best path to the destination. When looking at our longest path, we see very small differences between the three algorithms, they all almost performed the same when looking at time to find the path, and the length of the path found. Space complexity was not able to be adequately measured since the size of our graph and paths were not long enough. Overall, the algorithm that performed the best on our map and experiment was Dijkstra's algorithm. However, in a real life application in something such as Google or Apple maps, all paths generated by this algorithm would need to be given as options to the user since different people might want to take longer or shorter routes to their destinations.

For future work, it is important to test these algorithms on a larger map and a larger graph. This is so more accurate conclusions can be made about the algorithms and space complexity can be examined more in-depth. It would also be interesting to test other heuristics for our informed algorithms to see which ones would produce more efficient and optimal paths. Another final suggestion would be to test multiple maps with different terrains, and to consider using a graph such as Voronoi Diagrams, which were mentioned in the "Experimental Approach" section. Overall, there are many possible ways to extend this project into future work, but these ideas stated are a few that we think would be the most beneficial.

References

[1] Azad Noori and Farzad Moradi. Simulation and comparison of efficency in pathfinding algorithms in games. *Ciência e Natura*, 37:230–238, 2015.

[2] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. Investigation of the*(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming*, 2(4):251–256, 2012.

[3] Adeel Javaid. Understanding dijkstra's algorithm. *Available at SSRN 2340905*, 2013.

[4] Jin Wang Daniel R. Lanning, Gregory K. Harrell. Dijkstra's algorithm and google maps. *ACM SE '14: Proceedings of the 2014 ACM Southeast Regional Conference*, pages 1–4, 2014.

[5] D Andiwijayakusuma, A Mardhi, I Savitri, and T Asmoro. A comparative study of the algorithms for path finding to determine the adversary path in physical protection system of nuclear facilities. In *Journal of Physics: Conference Series*, volume 1198, page 092002. IOP Publishing, 2019.

[6] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[7] Sven Koenig and Maxim Likhachev. D* lite. *Aaai/iaai*, 15:476–483, 2002.

[8] Yan Li, Hongyan Zhang, Huaizhong Zhu, Jianwei Li, Wenjie Yan, and Youxi Wu. Ibas: Index based a-star. *IEEE Access*, 6:11707–11715, 2018.

[9] Manuel Heusner, Thomas Keller, and Malte Helmert. Understanding the search behaviour of greedy best-first search. In *Proceedings of the International Symposium on Combinatorial Search*, volume 8, pages 47–55, 2017.

[10] Roni Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2):125–148, 1999.

[11] Sofiat O Abioye, Lukumon O Oyedele, Lukman Akanbi, Anuoluwapo Ajayi, Juan Manuel Davila Delgado, Muhammad Bilal, Olugbenga O Akinade, and Ashraf Ahmed. Artificial intelligence in the construction industry: A review of present status, opportunities and future challenges. *Journal of Building Engineering*, 44:103299, 2021.

[12] Malte Helmert, Gabriele Röger, et al. How good is almost perfect?. In *AAAI*, volume 8, pages 944–949, 2008.

[13] Raz Nissim, Jörg Hoffmann, and Malte Helmert. Computing perfect heuristics in polynomial time: on bisimulation and merge-and-shrink abstractions in optimal planning. 2011.

[14] Liping Fu, Dihua Sun, and Laurence R Rilett. Heuristic shortest path algorithms for transportation applications: State of the art. *Computers & Operations Research*, 33(11):3324–3343, 2006.

[15] R. L. Church W. Zeng. Finding shortest paths on real road networks: the case for a*. *International Journal of Geographical Information Science*, 23(4):531–543, 2009.

[16] Chitaranjan Sharma Aijaz Magray. Fastest shortest path finding algorithms for large road network. *International Journal of Advanced Scientific Research and Management*, 4(1), 2016.

[17] Vadim Stadnik. Simple approach to voronoi diagrams. *Code Project*, 2020.

[18] Amit Patel. Heuristics. *Theory, Stanford*, 2016.

Team Member Contributions

1. Jacynda Alatoma: Background, Experimental Approach, Experimental Design and Results, Conclusion
2. Rajan Singh: Abstract, Research Question, Background, Experimental Design and Results, Analysis