Path Finding Algorithms

Abstract

In this paper, we compare the performance of three shortest path algorithms, namely the A*, Dijkstra, and Bellman-Ford algorithms. These algorithms were used to find the shortest paths in a map of Minneapolis and the search times of these algorithms were compared. The experimental results showed that the A* search algorithm performed the best, the Dijkstra algorithm performed slightly worse, and the Bellman-Ford algorithm performed the worst.

1 Introduction

When it comes to emergency situations, one of the most critical issues is finding the best route to reach a destination. In the life of a college student, one of the most critical issues is making it to class on time. Be it an emergency situation or a day-to-day problem, finding the shortest path in a map is a relevant and important problem to explore and solve. Much research and effort has gone into finding the best way to solve this classic problem. These research efforts have resulted in the development of several different algorithms and experimental results about their performances. First, a class of modified A* search algorithms were explored in [7], and their performances were compared to current state-of-the-art shortest path finding algorithms. Several other variations of A* search were also presented in [4], [12] and [9]. Then, the possibility of using genetic algorithms to find solutions to shortest path problems was considered in [6] and [2] . Finally, the performances of several different cutting-edge algorithms were compared using real road network data in [13].

For this experiment, we compare three of the more common shortest path algorithms, which are A*, Dijkstra's algorithm and Bellman-Ford algorithm. The performances of these algorithms were compared using road data of Minneapolis to find the shortest node-to-node paths in the map of Minneapolis. These three algorithms were implemented using Java, and their runtimes were recorded for different test cases. The motivation in doing this is to obtain a better understanding of how different algorithms perform on the real road data of Minneapolis. While these algorithms are often used in shortest problems, they differ in terms of their trade-off between precision and speed. For this project, we expect to find the algorithm that will find the optimal shortest path from a start point to a goal point in the shortest amount of time.

2 Related Work

One of the most common shortest path problem is one that finds the shortest path from one node to another node in a directed graph. The main goal of [7] was to find the fastest algorithm to compute the solution for this node-to-node problem. The solution to this problem can be found by searching only a part of the graph, and that means that the run time of the algorithm used only depends on how many nodes were visited [7]. Therefore, the performances of the algorithms in [7] were measured as a function of the number of vertices in the solution path. When the classic A* search is used to solve the node-to-node problem, distance bounds are implicit in the domain description, and no preprocessing was required [7]. In contrast, the authors of [7] developed a new pre-processing technique for computing the distance bounds instead of just letting them be implicit in the domain description. For this technique, they chose a number of landmarks, and then calculated the shortest path distances between all vertices of each of these landmarks. Then, they used these lower bounds, A* search, and the triangle inequality to develop new algorithms which were named ALT algorithms. They tested the performances of these algorithms against the A* algorithm with Manhattan distances as lower bounds and the Dijkstras algorithm. The tests were run on several different synthetic and real-road data sets. For the real-road data, experimental results showed that the ALT algorithm outperformed the other two by a factor of two in efficiency [7]. However, for the randomly generated data, the ALT algorithms did not outperform the other algorithms, and the Dijkstra's algorithm performed the best instead.

As A* is one of the most popular path-finding algorithms, it is of no sur-

prise that several variations of this algorithm have been proposed throughout the years. Botea, Muller and Schaeffer [4] present another near-optimum path-finding algorithm called HPA* (Hierarchical PathFinding A*) which is a variation of the traditional A* algorithm. HPA* has shown to be up to 10 times faster than A*, while finding paths that are within one percent of the optimal solution. This technique abstracts a map into linked local clusters. The optimal distances for crossing the cluster are pre-computed at the local level. At the global level, clusters are traversed in a single big step. In contrast to A*, which returns a complete path, HPA* returns a complete path of sub-problems. This is advantageous because if we decide to change the destination, not all effort will be wasted. Hence, this method adapts to dynamically changing environments.Su, Li and Shiu [12] propose another variation of A* named the Genetic Convex A* (G-CA*) algorithm. This variant automatically cuts the original map into several convex maps. The distance of the shortest path between any two tiles within a convex map is proven to equal their Manhattan distance, where Manhattan distance is the distance between two points in a grid measured along axes at right angles. Genetic Convex A* employs the genetic algorithm to merge adjacent convex maps and reduce the number of selected key nodes. Experiments conducted in [12] showed that G-CA* searched fewer nodes than HPA* while preserving the optimality of A*. A drawback, however, is that G-CA* takes more time than HPA*.

Since the experimental results of [7] suggested that the Dijkstra's algorithm is one of the main competitors of modified A* search algorithms, a modified Dijkstra's algorithm was examined in [11]. Noto and Sato [11] extended the conventional Dijkstras algorithm to reduce the search time to obtain a near-optimal solution. Since the conventional Dijkstra method requires a very long search time if the path is long, the authors proposed a new algorithm in which the Dijkstra method is applied from both directions: the starting point and the destination. Although this algorithm takes only 1/5 of the search time of the conventional Dijkstra method, it does not always return the optimal solution.

Besides approaches already explored, genetic algorithms can also be used to solve the shortest path problem. In the study presented in [6], the possibility of using a genetic algorithm to solve the shortest path problem was furthered explored. The most difficult task experienced by the researchers while conducting this study was encoding the path in a map into a chromosome. They used a priority based encoding method in order to represent all the paths in a map. In this method, a node ID represented the position of a gene on a chromosome, and the value of this ID was used to represent

the priority of this node for creating a unique path with all the nodes [6]. Instead of comparing this algorithm to current state-of-the-art algorithms, the authors simply tried to find efficient solutions for shortest path based optimization problems. This was because the performance of genetic algorithms cannot currently outperform any of the conventional algorithms [6]. Instead, this study sought to determine if the genetic algorithms were an area of study that was worth exploring for shortest path kinds of problems. To make this determination, these genetic algorithms were tested on three randomly generated shortest path problems [6]. The experimental results in [6] showed that these problems could be solved both quickly and with a high probability. Thus, genetic algorithms were concluded to hopefully be a new approach for solving shortest-path kinds of problems.

Ismael et al. [8] and Machado et al. [9] presented implementations of genetic algorithms as well their performances. In [8], Ismael et al. implemented a genetic algorithm to solve a mobile robot path planning problem in a static environment with predictable terrain. In this study, three different environments with various obstacles were proposed, including an indoor environment, a moderately scattered environment, and a more complexly scattered environment. The shortest path was found in all three environment using a genetic algorithm. It was also found that the shortest path can be found faster by increasing the number of generations. Machado et al. [9] presented the Real Time Pathfinding with Genetic Algorithm (RTP-GA), a method for real time path-finding algorithm based on genetic algorithms and A* search. This algorithm uses the classic A* inside a genetic algorithm and is claimed to be better than A* in some cases, especially since it was designed to work in dynamic environments. In this experiment, three types of maps were used: maps without obstacles, maps with patterns and maps without patterns. There was a tie when comparing the RTP-GA with A* in a map without obstacles, but RTP-GA outperformed A* in 90% of the cases in a map with patterns. In a map without patterns, however, RTP-GA was significantly worse than A* in most of the cases. This article claims that RTP-GA was proven to find a path to the maze exit in 100% of the cases.

In the case of genetic algorithms in [6], the probability of getting the optimal solution depended on the maximum generation size allowed and population size. Thus, in order to divulge deeper into the study of genetic algorithms, an equation that that aims to relate the size of a population, the desired quality of a solution, and other parameters would be of great assistance. Ahn and Ramakrishna [2] not only develops such an algorithm but also presents a genetic algorithm that aims to solve the shortest path problem. The experimental results from [2] show that the proposed algo-

rithm exhibits a higher quality solution when compared to other genetic algorithms. Furthermore, the developed equation can be scaled to larger networks and can be used to determine the necessary population size for any shortest path problem.

Most previously conducted research that compared different algorithms for finding the shortest paths usually used randomly generated networks in order to determine the algorithms performances [13]. However, the randomly generated networks might not represent the true properties of real roads. Thus, the paper [13] explored and researched the performances of varies algorithms on real road networks. The authors of this paper tested 15 possible algorithms on real road network data sets from 10 different states from the Midwest and Southeast parts of the United States. When considering the performances of these different algorithms, the researchers focused on the relative speeds of the algorithms, and memory requirements and possible implementation issues were not considered. Two of the incremental graph algorithms, Pape-Levit implementation and Pallottino implementation, performed the best when solving the one-to-all shortest path problems [13]. A one-to-all shortest path problem is one that requires the computation of the shortest paths from one node to all the nodes. In contrast, in the case of one-to-some shortest path problems, the Dijkstra implementations performed the best [13]. The results of [13] are also supported by work conducted in [5]. A dynamic road network model based on the Dijsktra's algorithm was build in [5] and its performance was tested against current cutting-edge algorithms in this field. The experimental results reinforced the same conclusion that was also reached by [13]: Dijsktra's is the the best algorithm for one-to-some shortest path problems in real road networks.

In most emergency situations, the most important factor considered while choosing an path-finding algorithm is speed. Two different methods of improving the time complexities of the A* search and Dijkstras algorithms were analyzed in [3] and [10]. In most existing path-finding algorithms, the number of iterations required to find the shortest path is large. To overcome this, an algorithm that is claimed to reduce the number of iterations required to traverse the path is proposed in [3]. This algorithm is a hybrid of backtracking and a new technique named the modified 8-neighbour approach. It starts at the source node and traverses in all eight directions of the source, repeating the process until the destination node is reached. During each iteration, the cost of each node (distance of that node from the center node) is updated. The proposed algorithm is observed to have less time complexity when compared to A* and Dijkstras algorithm. In contrast, the partitioning of graphs is used as a method in [10] in order to speed up

the Dijkstras algorithm. The method of acceleration used for the Dijsktra algorithm is called the arc-flag approach. In this approach, a preprocessing of the network data is allowed to generate additional information, which can then be used to speedup shortest path queries [10]. The graph is divided into several segments in the preprocessing phase, and the information is gathered on whether the arc is the shortest path in a given region. Using this method, in addition to a bidirectional search, enables this approach to achieve a speedup factor of over 500 when compared to the standard Dijkstra's algorithm performance on large networks.

Abu-Ryash and Tamimi [1] discuss the four shortest path algorithms, namely Dijkstra's algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm, and Johnson's algorithm. The study investigates and compares the impacts of these shortest path algorithms and it is shown that the efficiency varies among algorithms. This paper states that Johnson's algorithm is only faster than Floyd-Warshall on sparser graphs. It also claims that Dijkstra's algorithm is more memory efficient for sparse graphs. Finally, although Dijkstra has a better time complexity than Bellman-Ford, it cannot be used to solve graphs containing negative weights.

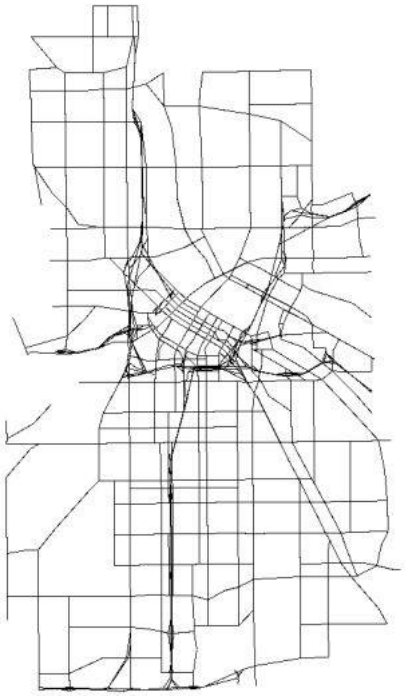
Lastly, all of these articles were quite varied from one another and had few similarities. The main similarity that they all had was that they all sought to solve the classic shortest path problem. The methods through which each of them approached this problem is where they differed. Goldberg [7], Botea [4], and Su [12] explored the performances of various modified A* search algorithms. In contrast, Noto [11] and Zhan [13] sought to prove that modified Dijkstras algorithms performed the best on real road networks. Finally, [6], [8], [9], and [2] aimed to present the advantages of genetic algorithms. Since speed is a vital aspect of effective route finding in emergency situations, [3] and [10] suggested acceleration methods for commonly used shortest-path finding algorithms. In conclusion, though these articles were very different from each other, they were a great way to learn the conventional methods of solving the shortest path problems, as well as, new areas of study that are worthy of exploration in this field.

3 Approach

Though there have been a number of algorithms proposed for finding the shortest paths in a map, we chose to compare three different path-finding algorithms that were the most interesting according to the work done in [7], and [13]. The algorithms that were chosen were the A* search algo-

rithm, Dijkstras algorithm, and the Bellman-Ford algorithm. Most of the prior testing has been on synthetically generated data sets. These synthetic sets yield different results than the real road data sets when tested for performances of shortest pathfinding algorithms according to [13]. Thus, the performances of above mentioned algorithms are compared in this paper using real road data of Minneapolis to find the shortest node-to-node paths in a map. While these algorithms are often used in shortest path problems, they differ in terms of their trade-off between precision and speed. For this paper, the aim is to find the algorithm that will find the optimal shortest path from a start point to a goal point in the shortest amount of time.

Figure 1: Map of Minneapolis



Firstly, the Minneapolis road data set that was used was acquired from the professors project suggestions document. The road data was presented in a lisp file with each line representing a segment of a road in the Minneapolis map. The format of each line in this file was that the first integer represented whether this segment was a one-way or a two-way road. The next two integers were the x and y coordinates of the start node of the segment.

Finally, the last two nodes were the x and y coordinates of the end node of this segment. For example, the line (2, 1127 4523 1042 5423) would represent a line segment that is a two way road starting at point [1127,4523] and ending at point [1042,7545]. The lisp file has 1357 lines in it, and each of these lines represents a segment of a road in Minneapolis. The map that this data represents is shown in Figure 1. The parsing of this data into a usable format was conducted in the following way. First, a Node class was created that contained the information about the x and y coordinate of a Node, and also information in a list about which nodes were connected to this node. So, for each point in the map, a Node was created and the parsing also provided the necessary information about the other points this node was connected to. Then, a hash set was created that contained all the nodes in main.

The A* search algorithm was the first algorithm that was chosen to be implemented since the work done in [7], [4], and [12] pointed towards the modified versions of this algorithm being the best options for path finding algorithms. A* search is an algorithm in which the cost associated with a node is f(n)=g(n)+h(n), where the h(n) is the estimated heuristic of the distance from the n node to the goal node, and g(n) is the cost of the path from the start node to the node n. The heuristic function used in an A* search has to be admissible, which means that it should never overestimate the actual distance from the node n to the goal node. Thus, for our implementation, the heuristic function was chosen to be the Euclidean distance between the node n and the goal node. The A* search algorithm that was tested for performance in this paper was implemented in the following way. First, a Hash Set was created to represent the nodes that are open (meaning that they have yet to be explored), and another Hash Set was created to represent the nodes that were closed (meaning they were already explored). Then, the start node is added to the open set. Then, a loop was written that first set the node with the lowest f-cost to be the current node. Then, the current node is removed from the open set and added to the closed set. If the current node is the goal node then we return the path through which it was reached. Else, for each neighbor of the current node, if the neighbor is in the closed set then we skip to the next neighbor, and if the new path to the neighbor is shorter or the neighbor is not in the open set, then the f-cost of the neighbor node is set, and the parent of the neighbor is set to the current node. Through this iterative method, the A* search algorithm was implemented.

Dijkstra was the second algorithm that was chosen to be implemented and tested for performance on the Minneapolis road data. The reason it was

chosen was because [5] stated the various advantages of using this algorithm in path finding. The main benefit that was noted was that since Dijkstra implementations have the advantage of being terminated as soon as they read the destination node is permanently labeled, this results in computational time saved when trying to find destination nodes that are fairly close to the start node. This algorithm was implemented in the following manner. There are two hash sets, one that is the open set and the other that is the closed set. Initially, the value of each node is set to infinity, and the source is the only one in the open set. Then, a loop was written that considers all of the neighbors of the current node that are not in the closed set and calculates the distances to the current node from the start node combined with the distance from the current node to the neighbor. If this is less than the current distance, it is replaced with the new calculated value. After all the neighbors are considered, the current node is moved to closed set from the open set. When the goal node is the one that is moved to the closed set, the algorithm terminates. Else, the node in the open set that has the smallest set becomes the current node and the loop iterates again. Through this method, the Dijkstra algorithm was implemented.

The Bellman-Ford algorithm computes the shortest paths from a source node to all other nodes in the graph. However, unlike Dijkstra's algorithm, Bellman Ford algorithm can also work correctly in graphs containing negative weights. Although we do not have negative weights in our Minneapolis map data, we decided to compare the run time of this algorithm with the other two algorithms. The Bellman-Ford algorithm first constructs a Hash Map from the nodes to their distances. The source distance is assigned to 0, while every other vertex distance is assigned to infinity. We also create a temporary map so that we can flip back and forth between the result map and the temporary map during each iteration of the algorithm, to avoid needlessly reallocating maps. The total number of iterations is the size of the graph minus one. For each iteration, we copy all of the mappings from our result map to the temporary map since we assume that each node in the iteration will have a cost equal to its cost on the previous iteration. Then for each node in the graph, we scan across all edges and update the costs of each node's paths at their endpoints. By using Math.min function, we make sure that the new cost of the shortest path to the current node is less than or equal to the cost of the shortest path to the node's neighbor plus the cost of the edge from that neighbor into this node. At the end of each iteration, we exchange the temporary map holding the new result with the result map holding last iteration's results. Finally, after we finish all the iterations, we return the result map.Through this method, the Bellman-Ford algorithm

was implemented.

4 Experiment Design and Results

For the experiment, ten pairs of start-end nodes were chosen and tested on all three algorithms to calculate the run times. A mixture of long and short paths were chosen to ensure that the algorithm works correctly. The longest path chosen has a distance of 4439.9 ((1384, 5051) → (1073, 9480)), while the shortest path chosen has a distance of 530.4 ((1106, 7568) → (0581, 7492)). A pair was also included where it should be impossible for the algorithms to successfully find a path as the path does not exist ((0405, 7732) → (2925, 7406)). In the tables below, it can be seen that all algorithms return null for path 7 since there is no path that leads from that start to end node.

Three trials for each algorithm were run and the averages was calculated for greater accuracy. Table 1 shows the ten paths and their respective distances. Tables 2, 3 and 4 show the run times of the A*, Dijkstra, and Bellman-Ford algorithms respectively from all three trials and their averages. A graph of distance versus run time for all three algorithms was also plotted and shown in Figure 1.

Table 1: Paths

Path	Start → End Nodes	Distance
1	(1534, 7253) → (0, 6379)	1765.5
2	(1009, 10500) → (0202, 10252)	844.2
3	(1121, 7568) → (2224, 8426)	1397.4
4	(1042, 7545) → (690, 10500)	2975.9
5	(1384, 5051) → (1073, 9480)	4439.9
6	(1106, 7568) → (0581, 7492)	530.4
7	(0405, 7732) → (2925, 7406)	2541
8	(1174, 7954) → (1759, 7830)	598
9	(1522,7879) → (3041, 6544)	2022.3
10	(0994,5779) → (3075,7024)	2425

Table 2: A* Algorithm

Path	Trial 1(ms)	Trial 2(ms)	Trial 3(ms)	Average(ms)
1	7	7	7	7
2	5	5	5	5
3	19	21	21	20.3
4	11	11	10	10.7
5	24	25	23	24
6	5	4	4	4.3
7	null	null	null	null
8	6	7	7	6.7
9	12	15	13	13.4
10	11	10	11	10.7

Table 3: Dijkstra's Algorithm

Path	Trial 1(ms)	Trial 2(ms)	Trial 3(ms)	Average(ms)
1	16	15	21	17.3
2	6	6	7	6.3
3	30	31	30	30.3
4	45	47	47	46.3
5	61	59	60	60
6	3	3	2	2.7
7	null	null	null	null
8	12	11	12	11.7
9	25	27	31	27.7
10	23	24	24	23.7

Table 4: Bellman-Ford Algorithm

Path	Trial 1(ms)	Trial 2(ms)	Trial 3(ms)	Average(ms)
1	21	20	21	20.7
2	13	12	12	12.3
3	57	54	58	56.3
4	27	33	32	30.7
5	79	81	72	77.3
6	15	17	17	16.3
7	null	null	null	null
8	20	20	19	19.7
9	36	35	33	34.7
10	32	31	31	31.3

Table 5: Averages of all three algorithms

Path	A*	Dijkstra	Bellman-Ford
1	7	17.3	20.7
2	5	6.3	12.3
3	20.3	30.3	56.3
4	10.7	46.3	30.7
5	24	60	77.3
6	4.3	2.7	16.3
7	null	null	null
8	6.7	11.7	19.7
9	13.4	27.7	34.7
10	10.7	23.7	31.3

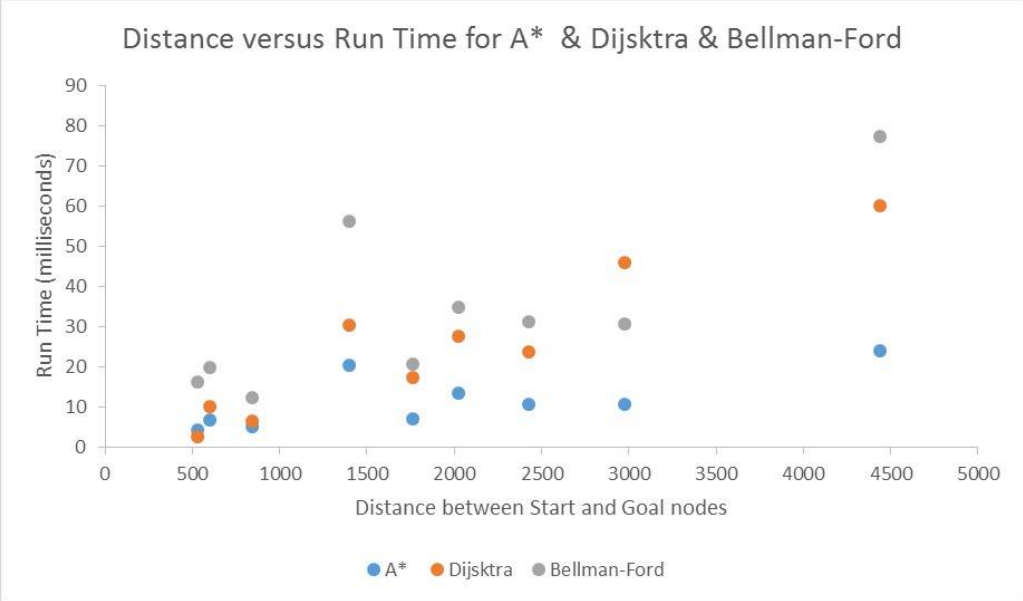


Figure 2: Distance versus Run Time

5 Analysis

The performances of the three algorithms were tested on ten different paths. As mentioned in the experimental design section, these paths were chosen so that the Euclidean distances between some of them were short and some

of them were longer. The Euclidean distances of each of these paths was calculated and noted in Table 1. From Table 2, it can be seen that the run times for A* search for each of the paths remains consistent throughout all of the trials. Similarity, Table 3 and Table 4 also show that the run times are fairly consistent throughout all three trials for the Dijkstra and Bellman-Ford algorithm. Before further analysis, it is important to note that the paths were not numbered based on increasing distances between them. Thus, Figure 2 was constructed in order to get a better visual representation of the relationship between the distance between the start node and end node and the run time.

As can be seen from Figure 1 and Table 5, the run time for path finding was between 4.3-16.7 seconds for the paths that were tested for the A* search algorithm. It had also correctly returned null for the path 7 since there does not exist a path that leads from its start node to its end node. Based on the results, though there seems to be a slight increasing relationship between the distance and the run time of the paths, the run time was not always increasing when the distance increased. This might have been because even though the distances were larger for some paths, some might have required going through several road segments while others did not require as many. When it came to the Dijkstra algorithm the relationship between the distance and the run time was stronger as can be seen from Figure 2. This algorithm also returned null for a non-existing path. Furthermore, the run times for path finding with the Dijkstra algorithm were higher than A* search run times. The reason that the run times increased more drastically with the Dijkstra algorithm is probably because this algorithm works best when the goal nodes are closer to the start nodes. The reason for this behaviour is because the Dijkstra algorithm terminates as soon as the destination node is reached, and if that node is close to the start node, that means that computation time is a lot quicker. Finally, the results show that the run times from the Dijkstra algorithm are higher than the run times for the A* search algorithm for all the paths. A* search is faster since it uses a best first search approach and utilizes a heuristic. Dijkstra on the other hand uses a greedy approach to searching, and does a blind search which is a disadvantage in a data set such as the Minneapolis data since it is large. Finally, as can be seen from Table 5 and Figure 2, the Bellman-Ford algorithm performed the worst out of the three that were tested. Its run times were higher than the Dijkstra and A* search algorithms for every path that was tested. However, it did return the correct outputs such as returning null for a path that did not exist. The Minneapolis road data also did not contain any negative edge weights, and since the Bellman-Ford algorithm is designed to handle

negative edge weights, it performs worse than others in the situation where all the edges are non-negative.

6 Conclusion

In conclusion, the performances of the A* search, Dijkstra, and the Bellman-Ford algorithms were tested using the real road data of Minneapolis. The experimental results showed that the A* search algorithm performed the best out of all the algorithms that were tested. The Dijkstra algorithm came in second, and the Bellman-Ford algorithm performed the worst. However, the Dijkstra algorithm also has its advantages in that it was less complex to program than A* search. And, though Bellman-Ford performed the worst, if it was used with a data set that had negative edges, it could be a very useful algorithm.

In terms of future work in this field, there are several interesting possibilities that could be explored. First, the algorithms that were implemented in this paper could be modified to perform more efficiently. For example, the A* search algorithm could have has a better heuristic function. Or, it could be modified in order to perform better such as the algorithms in [7], [4], or [12]. Future work could include experiments that would be run in order to determine how much better these modified algorithms are when compared to the A* algorithm implemented in this paper. As learned from research explored in the related work section of this paper, genetic algorithms are also an area worthy of exploration for future work in the field of path finding algorithms.

7 Contributions of team members

Siddeswari Thunuguntla: description of our approach to solve the problem including algorithms details, related work, analysis of the results, conclusion/summary and future work, A* and Dijkstras algorithm

Ren Xian Thian: abstract, introduction, related work, description of experiment design and results, Dijkstras and Bellman-Ford algorithm

References

[1] H. M. Abu-Ryash and A. A. Tamimi. Comparison studies for different shortest path algorithms.

[2] C. W. Ahn and R. S. Ramakrishna. A genetic algorithm for shortest path routing problem and the sizing of populations. *Evolutionary Computation, IEEE Transactions on*, 6(6):566–579, 2002.

[3] A. Ansari, M. A. Sayyed, K. Ratlamwala, and P. Shaikh. An optimized hybrid approach for path finding. *arXiv preprint arXiv:1504.02281*, 2015.

[4] A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical pathfinding. *Journal of game development*, 1(1):7–28, 2004.

[5] Y.-z. Chen, S.-f. Shen, T. Chen, and R. Yang. Path optimization study for vehicles evacuation based on dijkstra algorithm. *Procedia Engineering*, 71:159–165, 2014.

[6] M. Gen, R. Cheng, and D. Wang. Genetic algorithms for solving shortest path problems. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 401–406. IEEE, 1997.

[7] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[8] A. Ismail, A. Sheta, and M. Al-Weshah. A mobile robot path planning using genetic algorithm in static environment. *Journal of Computer Science*, 4(4):341–344, 2008.

[9] A. F. d. V. Machado, U. O. Santos, H. Vale, R. Gonçalvez, T. Neves, L. S. Ochi, and E. W. G. Clua. Real time pathfinding with genetic algorithm. In *Games and Digital Entertainment (SBGAMES), 2011 Brazilian Symposium on*, pages 215–221. IEEE, 2011.

[10] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 11:2–8, 2007.

[11] M. Noto and H. Sato. A method for the shortest path search by extended dijkstra algorithm. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2316–2320. IEEE, 2000.

[12] P. Su, Y. Li, Y. Li, and S. C.-K. Shiu. An auto-adaptive convex map generating path-finding algorithm: genetic convex a*. *International Journal of Machine Learning and Cybernetics*, 4(5):551–563, 2013.

[13] F. B. Zhan and C. E. Noon. Shortest path algorithms: an evaluation using real road networks. *Transportation Science*, 32(1):65–73, 1998.