Finding the Shortest Path Problem

Zichuan Liu
liux3445@umn.edu

Song Liu
liux4169@umn.edu

April 2019

Abstract

The shortest path problem has been one of the most important problem in computer science field, and it is specifically widely being researched in many crucial topics in artificial intelligence. The idea of solving shortest path problems are also widely being used in people's lives without their notices. In our research, we conducted a literature review on current approaches, tested several different searching algorithms and finished a report about their performances based on a map of Minneapolis and a random generated map using the function of "aima" python source code.

1 Introduction and Background

The shortest path problem is very fundamental and important in computer science and specifically in artificial intelligence. The meaning of the shortest path problem is that, if there is more than one path from one node (root node) to another node (goal node) in the graph, how to find a path so that the sum of the weights along each side of the path (path length) is minimized. With the continuous improvement of our social life, how to improve the efficiency of our life has become a significant consideration. Thus, the shortest path algorithm is becoming more and more important in our daily life. As students, when we take a taxi or drive to campus, the shortest path algorithm can find us a path that costs us the lowest of the taxi price or the time. On the other hand, with the development of industrial technology, there are more and more cars running on the roads, this puts a lot of pressure on the city's traffic. Therefore, the shortest path algorithm has a significant meaning for mitigating traffic. By doing this project, we can find the different efficiency of various searching algorithms and find the best one for solving the shortest path problems.

The shortest path problems we may encounter in real life could be complicated. So they can be classified into many different models. For example, there are directed graphs, undirected graphs or mixed graphs which means whether if there exists direction between nodes in the graphs. Also, the path costs between nodes could be recorded as weighted or unweighted. There also existed negative weights in the graphs. What's more, the problems could also be distinguished as single pair path problem, single destination path problem, single source path problem or all pairs path problem.

To solve the shortest path problem, many algorithms were built. All the search algorithm could be classified as informed search and uninformed search based on the information we already have. We could only use the informed search if we could get the heuristic value which is the estimation distance between the current node and the goal[3]. The uninformed search algorithm is used when we only have the information of current node. Such as breadth-first search (BFS), Depth-first search (DFS), Uniform-cost search (UCS), Depth limited search, Iterative deepening DFS, bidirectional search, etc. Dijkstra's algorithm is also uninformed search algorithm, but the difference between Dijkstra's algorithm and those mentioned is that Dijkstra's algorithm could not handle negative weights.

In this project, we implemented the code using A* search, Dijkstra's algorithm, greedy-best-first search, depth first search and breadth-first search (BFS)and analyzed the result from the perspective of run-time and quality of result about whether if the algorithm is optimal). The data source is from map.lisp file that Prof. Gini Maria provided. The first map we used if provided in the map.lisp file. It is a map of Minneapolis which contains the coordinates of locations so we could calculate using the coordinates. The second map we used is randomly generated by the function from aima-python.

2 Related Work

2.1 Improvement of Dijsktra's algorithm

Dijkstra's algorithm was published in 1959 and it is one of the most famous searching algorithms. This algorithm has many different versions, the original version found the shortest path between two nodes and a more common version is to fix a source node and finds the shortest paths from the source node to all other nodes in the graph. However, this algorithm has many disadvantages. For example, the algorithm does blind search, which means it wastes lots of time while processing. At the same time, it cannot deal with negative edge, which will cause errors on the acyclic graph. To improve this algorithm, many researchers provided valuable contributions.

In 1987, Michael L. Fredman and Robert Endre Tarjan developed a data structure called the "Fibonacci heaps" to implement heaps [5]. The Fibonacci heaps improved the run-time performance for random deletion from a heap to $O(\log n)$and $O(1)$for all other standard heap operations. By using the Fibonacci heaps, the run-time performance for the single source shortest path problem can be improved from $O(m \log_{m/n+2} n)$ to

$O(m + n \log n)$.

After the idea of Fibonacci heaps was published, Ravindra K Ahuja et al. created a data structure called the "radix heap" in 1990 [1]. They believed that using heap operations(or priority queue) is the key to improve the efficiency of Dijkstra's algorithm and the most significant property about heap operations is that successive delete min operations return vertices in nondecreasing order by tentative cost. The "radix heap" used these properties and improved the run-time efficiency of Dijkstra's algorithm. As a result, the run-time performance of combining the radix heap and Fibonacci heap is $O(m + n\sqrt{\log C})$.

Since the logic of Dijkstra's algorithm in solving single-source shortest path problem is to visit the node that has the smallest distance from the source node, it's necessary to sort all the nodes depends on their distance. However, the sorting process is not linear time, which will hurt efficiency. To deal with this problem, in 1999, Mikkel Thorup proved that it's possible to solve it in linear time and linear space for undirected single-source shortest path problem [11]. Thorup's algorithm uses the idea of the hierarchical bucketing structure to identify vertex pairs that may be visited in any order. In this way, the Dijkstra's algorithm does not need to visit all nodes in order of increasing distance, which avoids the limitation of sorting in linear time.

In 2013, Yizhen Huang et al. improved the search strategy of Dijkstra's algorithm by reducing the number of search nodes [7]. The main idea of the improvement is ignoring those uncorrelated nodes in the searching process and adding a constraint function with the weighted value. This value can be changed depends on the complexity of the graph. In this way, the constraint function will be suitable for different graphs.

Nabil Arman and Faisal Khamayseh developed an idea that can improve the Dijkstra's algorithm by reducing the number of nodes that needs to be traversed in the graph without changing the graph properties [2]. Their algorithm restructured the graph by only considered those critical nodes and put other nodes and weights into subpaths' properties. In this way, the graph was compressed and the algorithm performance will be improved dramatically.

2.2 Heuristic

There are plenty of famous searching algorithms that can handle the finding the shortest path problem. We classify these algorithms into the informed searching algorithm and uninformed searching algorithm according to what kinds of information we have. The major difference between these two kinds of algorithms is whether has the heuristic function. An informed searching algorithm has the heuristic function and the uninformed searching algorithm doesn't. The word heuristic was defined for strategies using readily accessible information to control the problem-solving process in people and machine [10].

How to find a good heuristic value has always been the main research purpose of the heuristic search algorithm. As we know, A* search algorithm is the heuristic search algorithm which requires the admissible heuristic value to be optimal. Raphael and others talked about this in his paper A formal basis for the heuristic determination of minimum cost paths.[4] They focus on how heuristic information can be used from the search problems and demonstrate an optimal way as a class of search strategy. They also talk about how real-life problems could be turned into pure mathematical graph problems and solved by algorithms.

2.3 Restricted Shortest Path Problem

Goldberg and Harrelson gave a kind of A* search algorithm called ALT algorithm in their paper.[6] This kind of algorithm mainly computes and stores the shortest path between all vertices and some chosen landmarks and then compute the result. Comparing to Dijkstra algorithm, A* algorithm. We could find that lower bound pruning does not significantly optimal the algorithm, instead, ALT is the most lower one among Dijkstra and A* algorithm. So, we do not implement ALT algorithm in our project.

Lorenz and Raz gave a fully polynomial approximation scheme for the restricted shortest path problem. [8]They improved Hassin's original result by a factor of n and the complexity of the bound they calculated can be used for all graph despite the cost. The complexity of ε-approximation they gave is $O(|E|n(loglog(n + 1/\varepsilon)))$.

2.4 Algorithms for the shortest path finding problem

Most of the work about shortest path problem are using real numbers to assign edge lengths. Okada et al. created an algorithm for solving shortest path problems on a network with fuzzy arc length. [9]Their algorithm is designed based on the multiple labeling method for a multicriteria shortest path problem. The proposed algorithm is numerically evaluated on large scale random networks. They proposed a method to reduce the number of paths according to a possible level for people to choose a preferable path.

2.5 Real-World Scenario

Church and Zeng showed how to solve the shortest path problem using A* search algorithm and Dijkstra algorithm. [13]They compared these two algorithms and gave the result that the A* algorithm could take advantage of spatial coordinates in trimming the search for the shortest path. Thus it allows the A* approach to significantly outperform Dijkstra for real road networks. In our experiment, we try to find the shortest path of a random map which is generated by aima-python function. So we could assume that A* search algorithm will perform better than Dijkstra algorithm in our study.

In a real-world situation, the status of problems changes rapidly. Will the heuristic be calculated correctly? Vanhove and Fack tested the heuristic function and the tests show that the running speed of heuristic is very high, and the quality of the heuristic result is mostly satisfactory.[12] Although this heuristic is still limited to some scenarios, the advantages of it should be fully made use of.

3 Approach

In our project, we used 5 different searching algorithms to find the shortest paths between two points on a map of Minneapolis and a random generated map. The searching algorithms we used are A* search, Dijkstra's algorithm, greedy-best-first search, depth first search and breadth-first search (BFS) on two maps which is described in the Experiment section. These 6 searching algorithms include both uninformed search strategies and informed (heuristic) search strategies. We used the search.py file in the aima-python directory as a reference and modify those algorithms inside this file to implement our project. For the heuristic function in the informed search strategies (greedy best first search and A* search), we designed the heuristic function by calculating the straight line distance between the current position and the goal position. In this way, the heuristic function is admissible since it can guarantee won't overestimate the minimal cost of the optimal path. To get the heuristic value, we used the Pythagorean theorem to calculate it. We will discuss the algorithms more specifically here and the code is attached in the Appendix section.

We modified the implementation of Dijkstra's algorithm based on uniform cost search algorithm in search.py in aima-python folder. At first, we assign a value to each node in the graph called distance. Then we created a empty set to save the result of the shortest path. Before the algorithm starts, we set the distance value of the starting point as zero and all other vertices as infinity. After that, we pick the vertices as starting point which has the minimum distance value and put into the result set. We then update the distance value of all vertices that are neighbors of the starting point. Then we iterate through all neighbors, and here comes the main idea of the Dijkstra algorithm. We update the distance value of the current vertices to be the sum of distance value of current node and the weight of the edge connecting it if the sum is less than the original distance value of this neighbor node.

The greedy best-first search algorithm is a informed search algorithm, unlike the bread first search algorithm and Dijkstra's algorithm. It depends on heuristic value to search for the destination which is a estimation value calculated the distance between current node and the goal. The heuristic value we used in our research is the straight line distance between current node and the destination node. The main idea of this algorithm is similar to Dijkstra's algorithm, but instead of updating the distance value of every node in the map and finally choose the minimum one, it choose the node with the minimum heuristic value.

The implementation of A* search combines the distance traveled and the heuristic value of estimation distance to the destination. We defined the current node as v and all its neighbours in a set s. Each node has a f value which is defined as the sum of g value and h value. The g value represents the distance traveled and the h value is the heuristic value of estimation distance to the destination. The algorithm will discover the nodes with the minimum f value iteratively.

The main idea of depth first search algorithm is to start from the start node and then search as deep as it can to find the destination. If it comes to a dead end, it will go back to another branch and go on. We implemented this algorithm by using a last in first out stack and storing the nodes in the stack. Also the disadvantage of depth first search algorithm is also obvious, the algorithm could easily lost in to a cycle route and it will never end. However, the space complexity is considerable.

The main idea of breadth first search algorithm is to check the nodes in the order of number of actions needed from the starting point. This means that breadth-first search checks the states that can be reached with the fewest steps first. Unlike the depth first search algorithm, it will search through the reachable nodes from the starting point first and then the second layer which could be described as "layer by layer". We implemented the algorithm using a first in first out queue to keep track of unexplored nodes. Breadth first search does not guarantee to find the optimal solution but is guaranteed to find solution if it exists.

Hypothetically, we will talk about the time complexities of some of these algorithms here after analyzing the pseudo-code: For breadth-first search:

$$O(b^d), \tag{1}$$

where b represents branching factor and d represents depth. For depth-first search:

$$O(b^m), \tag{2}$$

where b represents branching factor and C* represents cost of optimal solution. For Dijkstra's algorithm :

$$O(b^{1+C*/min(pathcost)}), \tag{3}$$

where b represents branching factor and C* represents cost of optimal solution. Below are the space complexities of some of the algorithms: For breadth-first search:

$$O(b^d), \tag{4}$$

where b represents branching factor and d represents depth.
For depth-first search:

$$O(b^m), \tag{5}$$

where b represents branching factor and m represents maximum depth of goal. For Dijkstra's algorithm :

$$O(b^{1+C*/min(pathcost)}), \tag{6}$$

where b represents branching factor and C* represents cost of optimal solution.

4 Experiment

4.1 Experiment environment

We implemented our searching algorithms by using Python 3.7.0. And all of our tests were performed on an Intel Iris Pro 1536 MB with 16 GB of RAM running macOS Mojave Version 10.14.4 at 2.5 GHz.

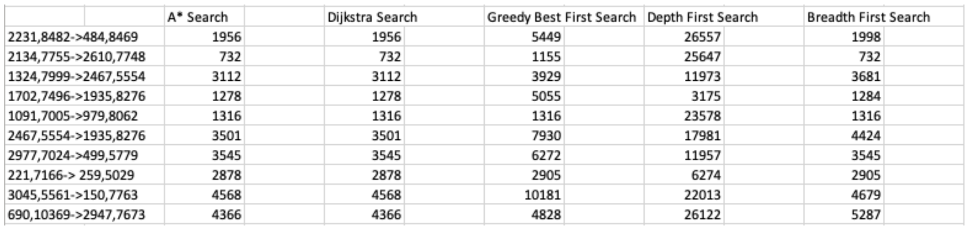
4.2 Experiment process

In our program, users can choose two different maps. The first map is a map of Minneapolis generated by using the given "map.lisp" file. Another map is generated by using the "RandomGraph()" function from aima-python. However, due to the original function can only produce a simple graph, it's very difficult to find the difference in the performance between each searching algorithm. So, we modified this function by adding the number of nodes from 10 to 300, adding the minimum links for each node from 2 to 4, increasing the size of the graph from 400*300 to 4000*3000. In this way, this function can generate a more complex map which can clearly show the performance difference between different searching algorithms.

To find the shortest path in the map of Minneapolis, our program needs two inputs. The first input is the coordinate of the starting point, the second input is the coordinate of the destination point. Once the program has these two inputs, it will run all build-in searching algorithms and print out the chosen path, the path cost, the number of nodes in the path, and the searching time for each searching algorithms. And users can easily compare each algorithm and find the best one. We implemented five different searching algorithms include both informed and uninformed search strategies. They are A* search, Dijkstra algorithm, and Greedy best-first search as informed search strategies, Depth-first search and Breadth-first search as uninformed search strategies. As for the the heuristic function that the A* search needed, we defined it as the linear distance between the starting point and destination point. In this way, the heuristic function won't overestimate and it's admissible.

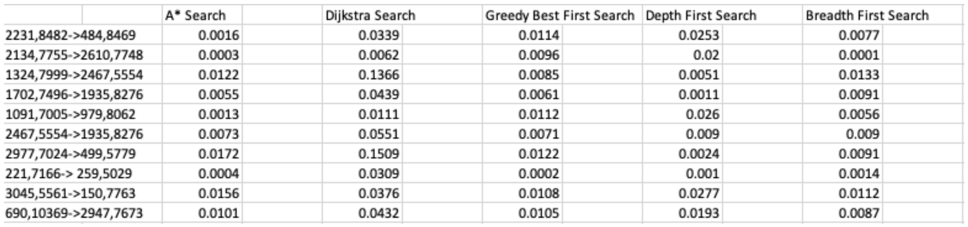
The process of finding the shortest path in the random map is the same as the map of Minneapolis. The only difference is that inputs of starting and destination points are a integer that between 0 and 99.
In this experiment, we chose ten pairs of starting and destination points in each map and ran all five searching algorithms on each pair. All point pairs were chosen randomly and contain different route network complexities.

4.3 Data Collection



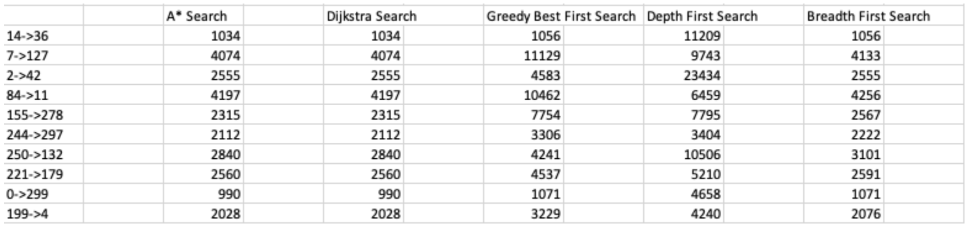
	A* Search	Dijkstra Search	Greedy Best First Search	Depth First Search	Breadth First Search
2231,8482->484,8469	1956	1956	5449	26557	1998
2134,7755->2610,7748	732	732	1155	25647	732
1324,7999->2467,5554	3112	3112	3929	11973	3681
1702,7496->1935,8276	1278	1278	5055	3175	1284
1091,7005->979,8062	1316	1316	1316	23578	1316
2467,5554->1935,8276	3501	3501	7930	17981	4424
2977,7024->499,5779	3545	3545	6272	11957	3545
221,7166-> 259,5029	2878	2878	2905	6274	2905
3045,5561->150,7763	4568	4568	10181	22013	4679
690,10369->2947,7673	4366	4366	4828	26122	5287

Figure 1: Total path cost in map of Minneapolis



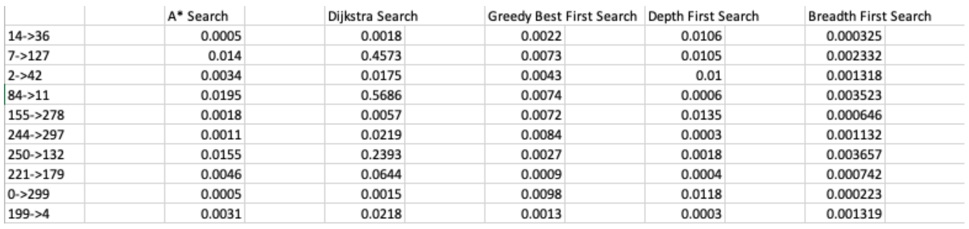
	A* Search	Dijkstra Search	Greedy Best First Search	Depth First Search	Breadth First Search
2231,8482->484,8469	0.0016	0.0339	0.0114	0.0253	0.0077
2134,7755->2610,7748	0.0003	0.0062	0.0096	0.02	0.0001
1324,7999->2467,5554	0.0122	0.1366	0.0085	0.0051	0.0133
1702,7496->1935,8276	0.0055	0.0439	0.0061	0.0011	0.0091
1091,7005->979,8062	0.0013	0.0111	0.0112	0.026	0.0056
2467,5554->1935,8276	0.0073	0.0551	0.0071	0.009	0.009
2977,7024->499,5779	0.0172	0.1509	0.0122	0.0024	0.0091
221,7166-> 259,5029	0.0004	0.0309	0.0002	0.001	0.0014
3045,5561->150,7763	0.0156	0.0376	0.0108	0.0277	0.0112
690,10369->2947,7673	0.0101	0.0432	0.0105	0.0193	0.0087

Figure 2: Run-time performance in map of Minneapolis



	A* Search	Dijkstra Search	Greedy Best First Search	Depth First Search	Breadth First Search
14->36	1034	1034	1056	11209	1056
7->127	4074	4074	11129	9743	4133
2->42	2555	2555	4583	23434	2555
84->11	4197	4197	10462	6459	4256
155->278	2315	2315	7754	7795	2567
244->297	2112	2112	3306	3404	2222
250->132	2840	2840	4241	10506	3101
221->179	2560	2560	4537	5210	2591
0->299	990	990	1071	4658	1071
199->4	2028	2028	3229	4240	2076

Figure 3: Total path cost in random map



	A* Search	Dijkstra Search	Greedy Best First Search	Depth First Search	Breadth First Search
14->36	0.0005	0.0018	0.0022	0.0106	0.000325
7->127	0.014	0.4573	0.0073	0.0105	0.002332
2->42	0.0034	0.0175	0.0043	0.01	0.001318
84->11	0.0195	0.5686	0.0074	0.0006	0.003523
155->278	0.0018	0.0057	0.0072	0.0135	0.000646
244->297	0.0011	0.0219	0.0084	0.0003	0.001132
250->132	0.0155	0.2393	0.0027	0.0018	0.003657
221->179	0.0046	0.0644	0.0009	0.0004	0.000742
0->299	0.0005	0.0015	0.0098	0.0118	0.000223
199->4	0.0031	0.0218	0.0013	0.0003	0.001319

Figure 4: Run-time performance in random map

4.4 Data Analyze

In the map of Minneapolis, the comparison of each searching algorithm on path cost and run-time are shown in Figure 5 and Figure 6.

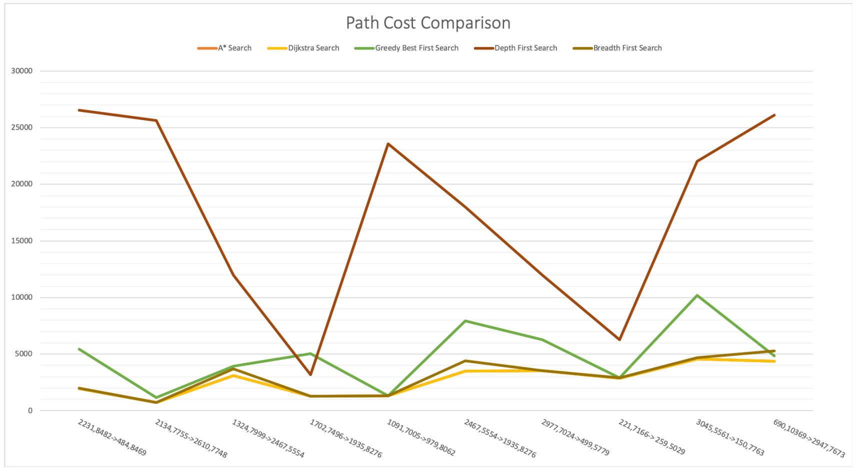


Figure 5: Total path cost in map of Minneapolis

In Figure 5, because A* search and Dijkstra search can find the optimal solution, the path of their solutions are the same in all 10 pairs of testing data, which cause their lines to appear as one(yellow line). By comparing with other lines, we can find out that the A* search and Dijkstra search has the best performance, their path costs are the lowest in all tests. The Breadth-first search also has very good performance, the brown line is very similar and close to the yellow line. The performance of Greedy best-first search(green line) is not as good as the previous three algorithms. The performance of Depth-first search(red line) is the worst, the path cost of it is the longest in most of tests.

In Figure 6, we can find out that the A* search(orange line) has the best run-time performance. The Breadth-first search(brown line) and

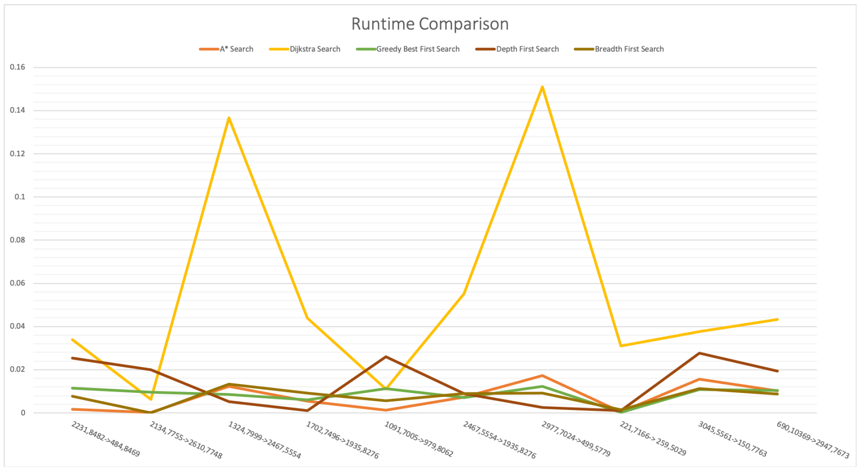


Figure 6: Run-time performance in map of Minneapolis

Greedy Best-first search(green line) also has good run-time performance but not as good as A* search. The Depth-first search(red line) is not stable, sometimes it has the best run-time performance, but sometimes it has the worst. The Dijkstra search(yellow line) has the worst run-time performance, the yellow line is higher than all other lines in most of tests.

In the random map, the comparison of each searching algorithm on path cost and run-time are shown in Figure 7 and Figure 8.

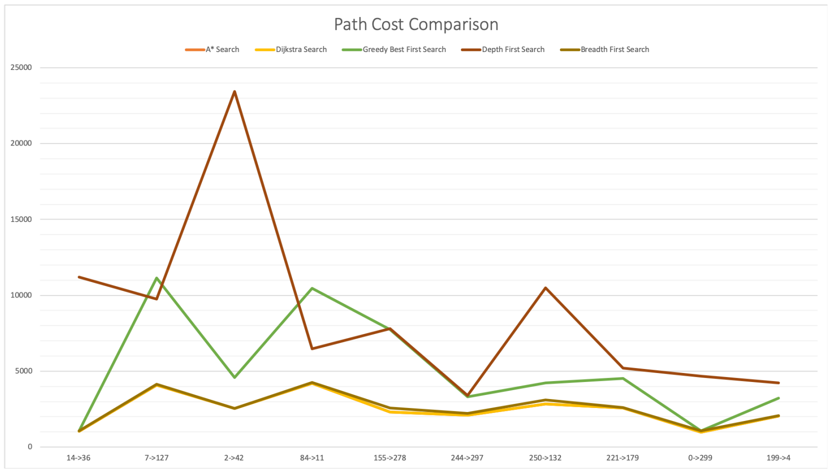


Figure 7: Total path cost in map of Minneapolis

The data from Figure 7 is very similar to the result of path cost in the map of Minneapolis. The A* search and Dijkstra search has the smallest path cost, the breadth-first search also has small path cost, but not as good as A* search and Dijkstra search. The Greedy best-first search has pretty big path costs and the Depth-first search has the biggest path cost.

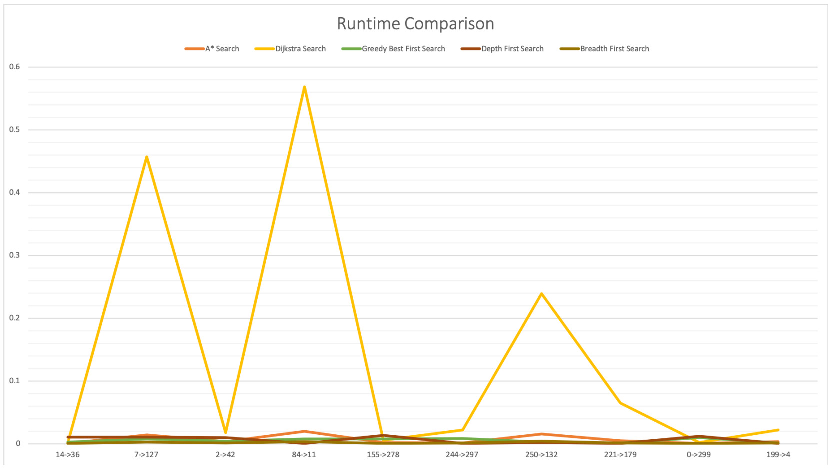


Figure 8: Run-time performance in map of Minneapolis

The data from Figure 8 is kind of difficult to analyze due to the run-time of the Dijkstra search is too long. But we can still find that the Breadth-first search(brown line) has the best run-time performance. The Greedy best-first search also has good run-time performance, but not as good as the breadth-first search. The run-time performance of A* search(orange line) and Depth-first search(red line) are good but still worst than the previous two searching algorithms. The Dijkstra search(yellow line) has the worst run-time performance.

5 Limitation

There are some limitations in our experiment. One of them is that we did not choose different heuristic functions for A* search. We only used the linear distance between starting and destination points as the heuristic value. If we changed the heuristic function, the performance of A* search might become different. Another limitation in our experiment is that our random map might be too simple compared to the real-world scenario, which may limit the performance of our searching algorithms.

6 Conclusion

According to the result of our experiment, we believed that the A* search is the best searching algorithm to deal with problems about find the shortest path. This is because that the A* search with admissible heuristic function can guarantee to find the optimal solution. Although Dijkstra search can also find the optimal solution, A* search has a better run-time performance. The breadth-first search has very good performance in our experiment but it cannot guarantee to always find the optimal solution. The Depth-first search and Greedy best-first search did not have good performance in our experiment so we don't consider these two algorithms. As a result, we chose the A* search as the best searching algorithm in all these five algorithms.

7 Future work

There are a lot of aspects that we can make further progress on this project. Firstly, we can add more test cases, because some corner cases might be missed in our experiment. Secondly, we can try different heuristic functions fro A* search and analyze the difference between their performance to find the best one. Thirdly, we can try running our algorithms on a more complex graph. The more complicated the map, the more convincing our results are. Finally, we can try more searching algorithms, such as bidirectional search, depth-limited search, and iterative-deepening search.

8 Contribution

Zichuan Liu(liux3445):

1 Literature review in Improvement of Dijkstra's Algorithm and Heuristic.

2 Finished the majority of experiment and Data Analyze section

3 Data analysis

4 Final revision

Song Liu(liux4169):

1 Literature review in the Restricted Shortest Path Problem, Algorithms for the shortest path finding problem and Real-World Scenario.

2 Finished the majority of introduction, backgound and approach section

3 Data analysis

4 Final revision

Both of the group members spent more than 50 hours in this project individually.

References

[1] Ravindra K. Ahuja et al. “Faster Algorithms for the Shortest Path Problem”. In: *J. ACM* 37.2 (Apr. 1990), pp. 213–223. ISSN: 0004-5411. DOI: 10.1145/77600.77615. URL: http://doi.acm.org/10.1145/77600.77615.

[2] Nabil Arman and Faisal Khamayseh. “A Path-Compression Approach for Improving Shortest-Path Algorithms”. In: 2015.

[3] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://dx.doi.org/10.1007/BF01386390.

[4] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *Intelligence/sigart Bulletin - SIGART* 37 (Dec. 1972), pp. 28–29. DOI: 10.1145/1056777.1056779.

[5] Michael L. Fredman and Robert Endre Tarjan. “Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms”. In: *J. ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: 10.1145/28869.28874. URL: http://doi.acm.org/10.1145/28869.28874.

[6] Andrew V. Goldberg and Chris Harrelson. “Computing the Shortest Path: A Search Meets Graph Theory”. In: *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’05. Vancouver, British Columbia: Society for Industrial and Applied Mathematics, 2005, pp. 156–165. ISBN: 0-89871-585-7. URL: http://dl.acm.org/citation.cfm?id=1070432.1070455.

[7] Yizhen Huang, Qingming Yi, and Min Shi. “An Improved Dijkstra Shortest Path Algorithm”. In: (Mar. 2013). DOI: 10.2991/iccsee.2013.59.

[8] Dean H. Lorenz and Danny Raz. “A Simple Efficient Approximation Scheme for the Restricted Shortest Path Problem”. In: *Oper. Res. Lett.* 28.5 (June 2001), pp. 213–219. ISSN: 0167-6377. DOI: 10.1016/S0167-6377(01)00069-4. URL: http://dx.doi.org/10.1016/S0167-6377(01)00069-4.

[9] Shinkoh Okada and Timothy Soper. “A Shortest Path Problem on a Network with Fuzzy Arc Lengths”. In: *Fuzzy Sets Syst.* 109.1 (Jan. 2000), pp. 129–140. ISSN: 0165-0114. DOI: 10.1016/S0165-0114(98)00054-2. URL: http://dx.doi.org/10.1016/S0165-0114(98)00054-2.

[10] J. Pearl. “Heuristics: Intelligent search strategies for computer problem solving”. In: (Jan. 1984).

[11] Mikkel Thorup. "Undirected Single-source Shortest Paths with Positive Integer Weights in Linear Time". In: *J. ACM* 46.3 (May 1999), pp. 362–394. ISSN: 0004-5411. DOI: 10.1145/316542.316548. URL: http://doi.acm.org/10.1145/316542.316548.

[12] Stéphanie Vanhove and Veerle Fack. "An effective heuristic for computing many shortest path alternatives in road networks". In: *International Journal of Geographical Information Science* 26.6 (2012), pp. 1031–1050. DOI: 10.1080/13658816.2011.620572. eprint: https://doi.org/10.1080/13658816.2011.620572. URL: https://doi.org/10.1080/13658816.2011.620572.

[13] W. Zeng and R. L. Church. "Finding shortest paths on real road networks: the case for A*". In: *International Journal of Geographical Information Science* 23.4 (2009), pp. 531–543. DOI: 10.1080/13658810801949850. eprint: https://doi.org/10.1080/13658810801949850. URL: https://doi.org/10.1080/13658810801949850.

9 Appendix

```
# Analyze the map.lisp file and store all the roads information
def split_info(info):
    last_char = None
    info_norm = []
    for curr_char in info:
        if curr_char.isalnum():
            if last_char.isalnum():
                info_norm[-1] += curr_char
            else :
                info_norm.append(curr_char)
        elif not curr_char.isspace():
            info_norm.append(curr_char)
        last_char = curr_char
    return info_norm


def change_form(info_norm):
    i = 0
    f_pare = 0
    map_info = []
    while f_pare < 3:
        curr_char = info_norm[i]
        if curr_char == '(':
            f_pare += 1
        i += 1
    i -= 1
    pare_pair = 0
    while i < len(info_norm):
        curr_char = info_norm[i]
        if curr_char == '(':
            pare_pair += 1
            curr_content = []
            while pare_pair != 0:
                i += 1
                curr_char = info_norm[i]
                if curr_char == ')':
                    pare_pair -= 1
                if pare_pair != 0:
                    curr_content.append(int(curr_char))
            map_info.append(curr_content)
        if curr_char == ')' and info_norm[i+1] == ')':
            break
        i += 1
    return map_info


# Calculate the linear distance between two points
def linear_distance(x, y):
    dis = int(math.sqrt(x**2+y**2))
    return dis


# Build the map of Minneapolis According to the given data
def build_map(data):
    ug = UndirectedGraph()
    for i in data:
        distance = linear_distance((i[3]-i[1]), (i[4]-i[2]))
        if i[0] == 1:
            ug.connect1(str(i[1])+","+str(i[2]), str(i[3])+","+str(i[4]), distance
        if i[0] == 2:
            ug.connect(str(i[1])+","+str(i[2]), str(i[3])+","+str(i[4]), distance)
    return ug


# Build a random map with 300 nodes
def RandomGraph(nodes=list(range(300)), min_links=4, width=4000, height=3000,
                curvature=lambda: random.uniform(1.1, 1.5)):
    g = UndirectedGraph()
    g.locations = {}
    for node in nodes:
        g.locations[node] = (random.randrange(width), random.randrange(height))
    for i in range(min_links):
        for node in nodes:
            if len(g.get(node)) < min_links:
                here = g.locations[node]
                def distance_to_node(n):
                    if n is node or g.get(node, n):
                        return infinity
                    return distance(g.locations[n], here)
                neighbor = argmin(nodes, key=distance_to_node)
                d = distance(g.locations[neighbor], here) * curvature()
                g.connect(node, neighbor, int(d))
    return g


# main function
def main():
    filename = 'mapinfo.lisp'
    with open(filename) as f_obj:
        contents = f_obj.read()
    res = split_info(contents)
    info = change_form(res)
    mn_map = build_map(info)
    rand_map = RandomGraph()
    print(mn_map.nodes())
    idea = input("enter m for Mn map, enter r for random map:")
    flag = 1
    if idea == 'm':
        while flag == 1:
            start_p = input("Starting point:")
            end_p = input("Destination point:")
            mp = GraphProblem(start_p, end_p, mn_map)
            print("In Minneapolis map\n")
            print_result(mp)
            stop = input("Enter q to stop: ")
            if stop == 'q':
                flag = 0
    if idea == 'r':
        while True:
            start = input("enter start:")
            end = input("enter end:")
            rp = GraphProblem(int(start), int(end), rand_map)
            print("In random map\n")
            print_result(rp)
            stop = input("Enter q to stop: ")
            if stop == 'q':
                break


# Breadth-first search
def breadth_first_graph_search(problem):
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return node
    frontier = deque([node])
    explored = set()
    while frontier:
        node = frontier.popleft()
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                if problem.goal_test(child.state):
                    return child
                frontier.append(child)
    return None


# Depth-first search
def depth_first_graph_search(problem):
    frontier = [(Node(problem.initial))]
    explored = set()
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)
    return None


# Greedy Best-first search
def best_first_graph_search(problem, f):
    f = memoize(f, 'f')
    node = Node(problem.initial)
    frontier = PriorityQueue('min', f)
    frontier.append(node)
    explored = set()
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                incumbent = frontier[child]
                if f(child) < f(incumbent):
                    del frontier[incumbent]
                    frontier.append(child)
    return None
```

```
# A* search
def astar_search(problem, h=None):
    h = problem.h
    return best_first_graph_search(problem, lambda n: n.path_cost + h(n))


# Dijkstra search
def dijkstra_search(problem):
    return best_first_graph_search(problem, lambda n: n.path_cost)
```