Route planning in static network

Shufeng Ren
Department of Mechanical Engineering
Northwesterm Umiversity
Evanston, IL 60208
Email: shufengren2019@u.northwestern.edu

Jiarui Li
Department of Mechanical Engineering
Northwesterm Umiversity
Evanston, IL 60208
Email: jiaruili2020@u.northwestern.edu

***Abstract*—In transportation networks, route planning algorithms have developed for decades since 1950's. Algorithms with faster speed and better performance have come up and have been applied to solve realistic problems include navigation, robotics control, logistics management etc. In this article, we did a survey on some popular classic shortest-path algorithm in the previous part. Based on our understanding of these algorithms, we implemented a visualization tools to show how these algorithms work under shortest-path finding scenarios.**

Keywords—Shortest-path, Route planning, Basic techniques, Goal directed techniques

I. INTRODUCTION

With the rapid economic growth, the city size is expanding continuously in many developing countries. This leads to growing scale of the transportation network. Route planning means scheduling a path between start point and goal point just as its literal meaning. In the transportation network, there are always multiply paths can be chose. However, under certain rules such as shortest distance or least cost, there only exist one optimal path. With the increasing nodes in the large-scale transportation network, the number of candidates for the reachable path grows exponentially. How to choose the optimal path with fast speed and high efficiency from transportation network containing millions of nodes is a problem attracts many researchers and technicians inputting time and energy to solve.

In the transportation aspects, the shortest-path algorithm can greatly improve the network working efficiency. People can plan their best path for outgoing. In the logistics service, shortest-path algorithm helps to schedule the reasonable vehicle path for distribution. So, the time for the product transmitting is reduced which ensures the quality and freshness of the product. In addition, the shortest-path algorithm can also be applied to other network and provide high performance service. For example, in the social network, the logical relationship between users can be interpreted as distance and shortest-path algorithm is able to enhance the query efficiency as well as improve the user experience. Considering the environmental protection issue, shortest-path algorithm can help company to decide an optimal distribution center. Shorter transportation distance will lead to lower energy consumption and greenhouse gases emission.

The following article contains three parts. In section 2, we looked through existing shortest-path algorithms with different approaches. Most popular and classic algorithms in static routing is presented in sequences. In section 3, we implemented five shortest-path algorithms to solve the static routing problem. A visualization tool is presented to show the animation of the path searching. Simulation of different algorithms in serval different maps are conducted for comparison, and simulation results are discussed. In section 4, we concluded the advantages and disadvantages of each algorithm.

II. SURVEY ON POPULAR ALGORITHMS

Shortest-path can be identified with various settings. When the nodes and the edges in the graph keep unchanged over time, the graph is static. Compared to this, in the dynamic graph, nodes and edges can be updated, deleted or introduced on the timeline. The edges in the graph can be either directed or undirected. The edges' weight can be either negative or non-negative.

Most of the shortest-path algorithms can be classified into two kinds depends on the number of the start points and goal points. The first kind is single-source shortest-path (SSSP), where the objective is to find the shortest-paths from a single-source node to all other nodes. The second kind is all-pairs shortest-path (APSP), where the objective is to find the shortest-paths between all pairs of nodes in a graph[1].

According to the work done by Amgad Madkour et al., the taxonomy of the shortest-path algorithm is shown in the Fig. 1. Each branch of the tree illustrates a special category of the problem.

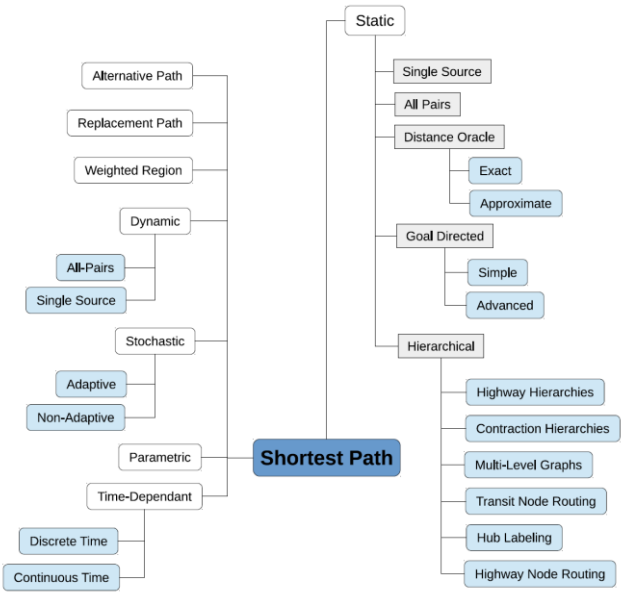

Fig. 1 Taxonomy of Shortest-Path Algorithms1

For this part of the article, we basically focus on the understanding and realization of some classic algorithms with basic techniques or goal direct techniques for the static problem.

A. *Dijkstra's algorithm*

Dijkstra's algorithm[2] has come up for about 60 years and it is the standard algorithm for solving single source shortest-

path problem. The algorithm first assign infinite to all the distance and set the start node as visited. Then it continuously updates the tentative distance between start node and each node by checking all the edges. If the algorithm finds the shortest edge, it will put the corresponding node in the visited list. The algorithm ends when all nodes are visited. The weakness of Dijkstra's algorithm is that the weights of edges have to be non-negative and it only solves static problem.

Assume the number of nodes is n and the number of edges is m, the time complexity of Dijkstra's algorithm is $O(n^2)$. If using binary heap[3] in the algorithm, the time complexity can be improved to $O((m+n)logn)$. Fredman and Tarjan combined Fibonacci heap with Dijkstra's algorithm[4] and the time complexity achieved $O(nlogn+m)$.

The pseudocode of Dijkstra's algorithm

```
1.  Input: A graph graph and a starting node
2.  Output: Tables of node distances dist and predecessors
    pred
3.
4.  for every node v in graph do
5.      dist[v] ← ∞; pred[v] ← −1;
6.  end
7.  dist[start] ← 0;
8.  todo ← the set of nodes in graph;
9.  while todo is not empty do
10.     v ← remove the element of todo with minimal dist[v]
    ;
11.     for every outgoing edge (v,u) with weight w do
12.         if dist[v] + w < dist[u] then
13.             dist[u] ← dist[v] + w;
14.             pred[u] ← v
15.         end
16.     end
17. end
```

If execute Dijkstra's algorithm forward from the start node and backwards from the goal node at the same time, bidirectional search is formed. The shortest path is derived once some node has been visited from both direction[5]. The search space can be reduced compared to Dijkstra's algorithm.

B. *Breadth-First Search (BFS) and Depth-First Search (DFS)*

Both the BFS and DFS are the algorithms for traversing or searching graph data structures which are suitable for shortest-path searching. From the start node, they transverse all the nodes through different approach. For BFS, it explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. For DFS, it explores as far as possible along each branch before backtracking. They were used as the strategy for solving maze.[6][7][8]

As every node and edge will be explored in the worst situation, the time complexity of both algorithms is $O(n+m)$ and the space complexity is $O(n)$

The pseudocode of BFS algorithm

```
1.  Procedure BFS(graph, start) is
2.      preds ← new array (same size as graph, filled with fa
    lse);
3.      todo ← new queue;
4.      preds[start] ← true;
5.      Enqueue(todo, start);
6.      while todo is not empty do
7.          v ← Dequeue(todo);
8.          for u in Successors(graph, v) do
9.              if not preds[u] then
10.                 preds[u] ← v;
11.                 Enqueue(todo, u)
12.             end
13.         end
14.     end
15.     return preds
16. end
```

The pseudocode of DFS algorithm

```
1.  Procedure DFS(graph)is
2.      preds ← new array (same size as graph, filled with fa
    lse);
3.      Procedure Visit(pred, v)is
4.          if not preds[v] then
5.              preds[v] ← pred;
6.              for u in Successors(graph, v) do
7.                  Visit(v, u)
8.              end
9.          end
10.     end
11.     for v in Nodes(graph) do
12.         Visit(true, v)
13.     end
14.     return preds
15. end
```

C. *A* algorithm*

Goal directed techniques aim to lead the search direction towards the goal and avoid waste resources on the aimless searching. This requires additional information of the graph nodes or edges to determine how to prune the graph in the search space.

A* is a simple goal directed algorithm proposed by Hart et al.[9] The algorithm uses a heuristic approach (h(n)) in finding the shortest-path. In the searching iteration, the algorithm determines which path to extend based on the total cost f(n) which consists two parts g(n) and h(n). g(n) refers to the cost of the path from start node to current node while h(n) estimates the cost from current node to goal. If the heuristic function used is admissible which means that it never overestimates the actual cost to get to the goal, a least-cost path is guaranteed.

Bidirectional search can also be combined with A*. In the standard case, the forward and the backwards use same heuristic function. In the work of Ikeda et al.[10], they combined two functions $h_f(n)$ and $h_r(n)$ into consistent by using $(h_f(n)-h_r(n))/2$ in the forward search as well as $(h_r(n)-h_f(n))/2$ in the backward search. Change the stopping criterion instead of using two same functions is another way used in practice such as the work done by Goldberg[11], Kaindl[12] and Pohl[13].

Based on the A* algorithm, Goldberg and Harrelson proposed ALT algorithm which combines A* search,

landmarks and the triangle inequality[11]. In the pre-processing stage, the algorithm carefully chooses a few landmarks. Then it computes and stores shortest-path distances between all nodes and each of these landmarks. Lower bounds are computed using the stored distance and the triangle inequality. Their experimental results shown that ALT is efficient on several important graph class. Work from Efentakis and Pfoser[14] shows that if landmarks are well-spaced and close to the boundary of the graph in the road network, results can be good with acceptable average query times.

III. Implementation

In this part, several path planning algorithms simulation are demonstrated under python 3.7.1, and the comparison of the algorithms is shown here. Firstly, the map representation is introduced, and the methods we use to build the simulation word are also presented. Then, the breadth first search (BFS) and depth first search (DFS) methods are implemented in an empty map, and the difference between stack data structure and queue data structure can be clearly found. Finally, A* search, Dijkstra search and Greedy best first search are implemented in two maze maps, and the performance are explained.

A. Map Representations

The map representation could make a huge difference in the performance of every algorithms. A map with less nodes, could decrease both the time complexity and space complexity when implementing path search algorithm. Simultaneously, high precision map lead to higher path quality but more calculations. A trade-off exists when choosing map representation.

Fig. 2 shows a simple square grid map we used in path search simulation since it's easier to visualize the concepts and design such map with Matplotlib package in Python.

There are two key elements in a grid map:

1) Nodes: Store the location information attributes of location of every grid, including (x,y) coordinates, whether it is on the map boundary, and whether it is an obstacle.

2) Edges: Store the available directions from a grid location and the move costs on these directions, the initial edges for every node contains four directions shown by the arrows in the Fig. 2. The directions point to obstacles or boundaries should be deleted in later graph preprocessing.

Even within same grid map, how the point moves make things different. Here we use tile movement as shown in Fig. 2, the green tile could move to any of the gray tiles. There are also edge movement and node movement to choose when the tiles are large and doesn't consider the size of move unit.

To simplify the model, a uniform movement costs is considered, which set the move cost as 1 on all the edges.

B. BFS and DFS

BFS and DFS use the different data structure, where BFS uses queue data structure and DFS uses stack data structure. Queue is a "first in first out" (FIFO) structure with two operations: enqueue and dequeue, and queue one by one stores nodes of different level from start point until the goal point is visited (early exit) or all nodes are visited. However, stack is a "first in first out" (LIFO) structure with two operations: push and pop, and stack visits all nodes from one

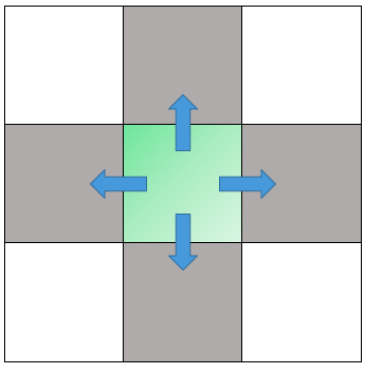

Fig. 2 Tile movement on a square grid map

of four directions from the start points, then jumps up and visit one of left directions until goal points are found or all nodes are visited.

Fig. 3 and Fig. 4 shows the simulation results in an empty map given same start, goal points. The plots obviously reflect the characteristics of stack and queue that the path of DFS is along one direction, however, BFS traverse each layer and the visited points in search animation spread toward goal position like ripple. The iteration number of BFS is about 2.5 times that of DFS, and the visited points (gray blocks) almost cover the whole map for BFS. Even if BFS is less efficient than DFS, the path length of BFS is 0.08 time that of B, and we can conclude that BFS have higher time complexity but lower space complexity than DFS.

To show the influence of obstacle on these two search algorithms, a map with obstacles is utilized. The Fig. 5 and Fig. 6 shows the simulation results in a maze map given same start, goal points. In this case, the length of path of BFS is also shorter than that of DFS, and the result of number of iterations

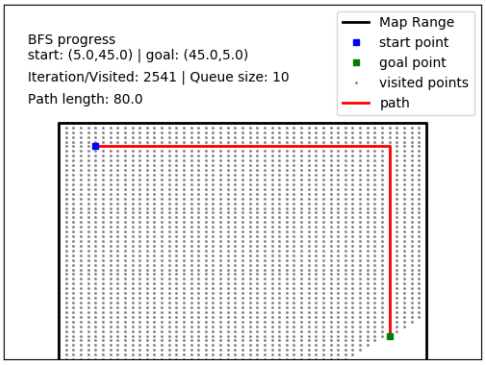

Fig. 3 Breadth first search on an empty map

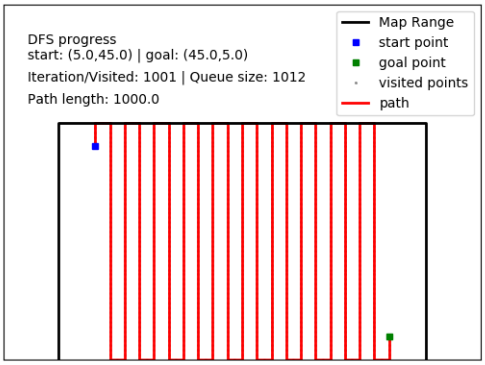

Fig. 4 Depth first search on an empty map

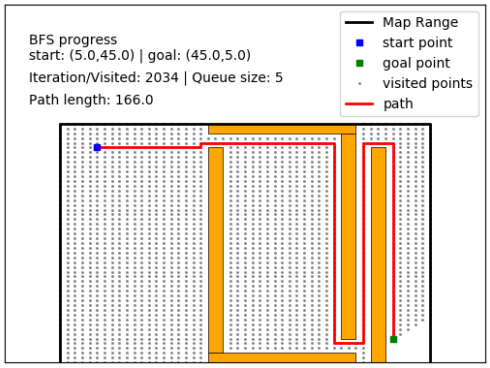

Fig. 5 Breadth first search on a maze map

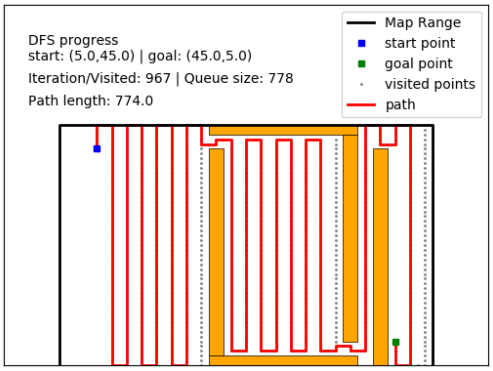

Fig. 6 Depth first search on a maze map

and queue size also match the conclusion in the empty map.

No matter BFS or DFS, the movement cost is not considered, and both methods are exhaustive to enumerate all situations. Based on the BFS, if we turn queue to priority queue, which assign "priority" to the elements in visited queue and the order of dequeue is based on the priority, it's the essence of Dijkstra shortest path, A* shortest path and Greedy best first.

C. A, Dijkstra and Greedy*

In these three algorithms, priority queue by binary heap is utilized with different rules to evaluate cost of path (called priority). As we have introduced, Dijkstra algorithm selecting the next node closest to the start point, which use the distance from start point as the cost of path, however, Greedy best first algorithm use the distance from goal point (called a heuristic) as the cost value. A* algorithm works like the combination of Dijkstra and Greedy, it calculates the sum of distance to start point and distance to goal point as the evaluation of the path cost.

Fig. 7, Fig. 8 and Fig. 9 illustrate the performance of these three algorithms with two different maze maps. The followings compare these three algorithms from two aspect.

1) Number of Iterations: Greedy best-first search iterates much fewer than Dijkstra and A* algorithms which means it runs much fatser. The reason is that it uses the heuristic function to guide its way towards the goal, and the nodes around the start point are not visited.

2) Path length: In the second map, all these three algorithms have the same performance in path length, and this path length is the shortest path if we calculate by hand. However, situation changed in the first map, where the path length of Greedy is almost two times of other two algorithms. The first map set the obstacle on the way from start point to goal, and the exit is in the opposite direction start-goal connection. Once the Greedy move towards the goal point

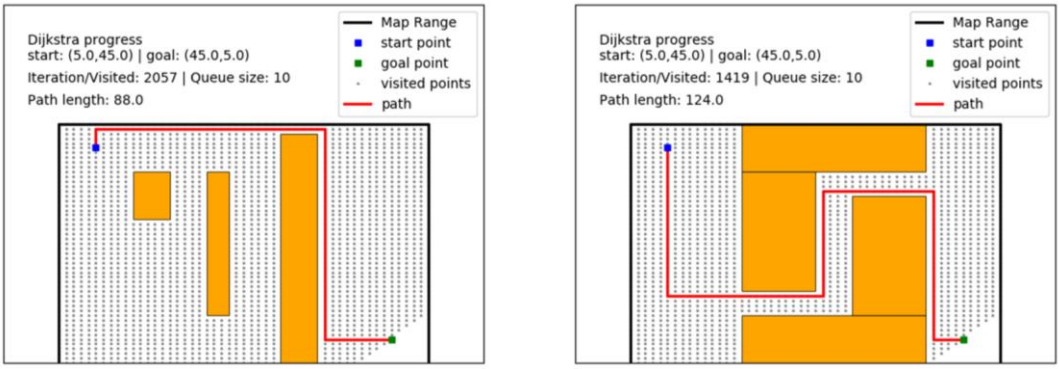

Fig. 7 Dijkstra shortest path search on two maps

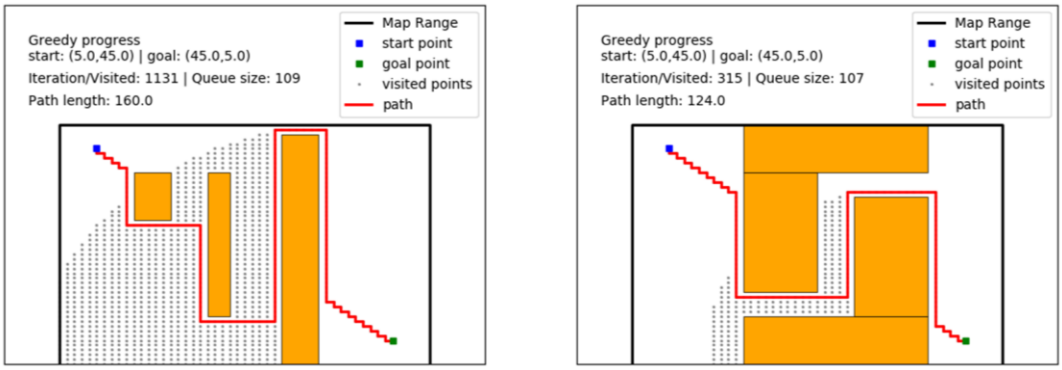

Fig. 8 Greedy best first search on two maps

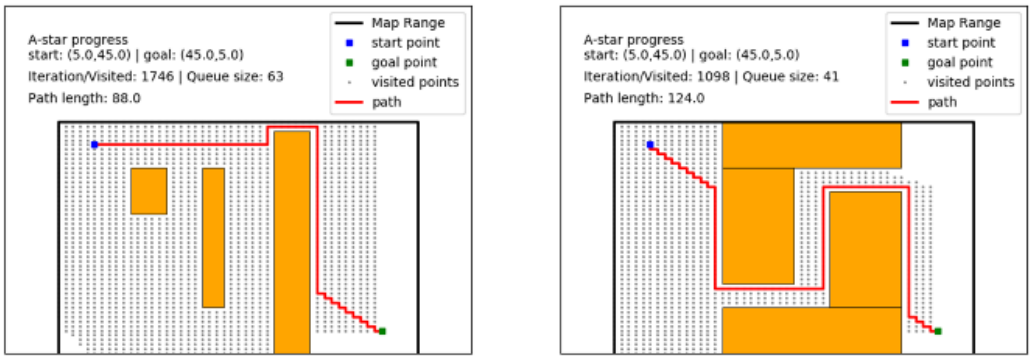

Fig. 9 A* shortest path search on two maps

and meet the obstacle, it have to move on the side of the obstacle. In contrast, the A* and Dijkstra find the shotest path in both maps.

IV. Conclusion

Through reading related literature, a survey on shortest-path algorithm for static network which contains basic techniques and goal direct techniques is completed. Then five classic path search algorithms are implemented with different maps. The BFS and DFS algorithms is inefficient since BFS would traverse most nodes while DFS cannot guarantee to find the path. The Dijkstra algorithm could guarantee to a shortest path, and through simulation, it shows that Greedy best first search can find paths very quickly compared to Dijkstra algorithm. A* algorithm combine the advantage of Greedy and Dijkstra, which could guarantee to a shortest path and improves the efficiency compared to Dijkstra. There are many optimizations to the algorithm that can be done, such as using other data structure and quick sort rather than binary heaps, assigning weights to edges to make it close to real world, setting depth limit to DFS to avoid waste resource, implementing other type of grid (E.g. hexagon which provides 6 directions of movement) and using “string pulling” algorithm (E.g. Funnel algorithm) to straighten the path.

V. CONtribution

For this path search project, Shufeng Ren contributes to the literature research, the implementation of A*, Dijkstra and Greedy Best First algorithms, and the animated display interface. Jiarui Li contributes to the literature research, the implementation of BFS and DFS, and organization of this report.

References

[1] Madkour, Amgad, et al. "A survey of shortest-path algorithms." arXiv preprint arXiv:1705.02044 (2017).

[2] Dijkstra, Edsger W. "A note on two problems in connexion with graphs." Numerische mathematik 1.1 (1959): 269-271.

[3] Williams, John William Joseph. "Algorithm 232: heapsort." Commun. ACM 7 (1964): 347-348.

[4] Fredman, Michael L., and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms." Journal of the ACM (JACM) 34.3 (1987): 596-615.

[5] Dantzig, George. Linear programming and extensions. Princeton university press, 2016.

[6] Moore, Edward F. "The shortest path through a maze." Proc. Int. Symp. Switching Theory, 1959. 1959.

[7] Skiena, Steven S. "Sorting and Searching." The Algorithm Design Manual. Springer, London, 2012. 103-144.

[8] Sedgewick, Robert. "Algorithms in C++ Part 5: Graph Algoritms (Pt. 5)." (2002).

[9] Hart, Peter E., 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 (1968): 100-107.

[10] Ikeda, Takahiro, et al. "A fast algorithm for finding better routes by AI search techniques." Proceedings of VNIS'94-1994 Vehicle Navigation and Information Systems Conference. IEEE, 1994.

[11] Goldberg, Andrew V., and Chris Harrelson. "Computing the shortest path: A search meets graph theory." Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2005.

[12] Kaindl, Hermann, and Gerhard Kainz. "Bidirectional heuristic search reconsidered." Journal of Artificial Intelligence Research 7 (1997): 283-317.

[13] Pohl, Ira. Bidirectional and heuristic search in path problems. No. SLAC-R-104. 1969.

[14] Efentakis, Alexandros, and Dieter Pfoser. "Optimizing landmark-based routing and preprocessing." Proceedings of the Sixth ACM SIGSPATIAL International Workshop on Computational Transportation Science. ACM, 2013