A study on shortest path routing algorithms

Chang-Heng Liou

1 Introduction

I study on the several one-to-one shortest path routing algorithms and experiment on different DIMAC USA datasets. All of the experiments run on OS X with Intel i7-8750H CPU, and only single thread is used. While traditional dijkstra algorithm [1] could find the shortest path in few hundred milliseconds on the graph of New York with roughly 260,000 nodes and 730,000 arcs, it takes up to 5 secs to find the optimal path in the graph of Western USA, which is not desirable for real application. The fastest method I have studied could find shortest path in less than 1 seconds, which is pretty close to the state-of-the-art implentation in the routing software, such as Google map. I also study on other apporaches: bidirectional search, A* [4] and ALT [3]. Bidirectional search speeds up the process by starting search from from both the starting and target points. A* adds heuristic function to enforce the algorithm to choose vertice that are closer to target nodes. Finally, ALT uses pre-defined landmarks and pre-calculate the distance between all nodes and landmarks and use them as heuristic function.

2 Routing algorithms

In this section, I discuss the following 4 methods to find shortest path: Bidirectional Dijkstra, A* [4], ALT [3], Contraction Hierarchy [2] and I will show our experiment results in the section 3.

2.1 Bidirectional search

The search space of Dijkstra is exactly the same as breadth first search; they both search their neighbors, so their search spaces are naturally in a ball shape. Bidirection search accelerates the process by reducing the diameter of the circle. It starts searching from both the starting and the ending vertex, which traverses 2 smaller circles rather than 1 huge circle. Consider the traditional Dijkstra, I assume its search space has radius $2r$ and its counterpart use 2 circle with radius r; then the Bidirectional Dijkstra search only half of the vertices compares to traditional Dijkstra.

```
while (forward.emtpy? or backward.empty?)
  if (forward.top.cost + backward.top.cost > u)
    return // found shortest path
  forwardSearch()
  backwardSearch()
```

Both the forward and backward search are just normal Dijkstra. The only difference is that I have to update u when there is a vertex that is processed by both forward and backward search.

$$u = min(u, d(s,u) + l(u,w) + d(w,t)) \tag{1}$$

Notice how the stop condition works. The first vertex that the both searches meet is not necessarily in the optimal shortest path; thus, I keep searching until the minimal path is found.

2.2 Heuristic function and A*

Another classical optimization technique of Dijkstra's algorithm is A*. It uses a heuristic function to estimate the cost of each path. A common choice of the heuristic function would be euclidean distance or manhattan distance.

$$\begin{aligned} estimateCost &= euclideanDist(V, target) \\ nextCost &= currCost + cost(V) + estimateCost(V) \end{aligned} \tag{2}$$

The intuition is that the algorithm would choose the vertex that is more closer to the target vertex, which means the less euclidean distance. Generally speaking, the shortest routes are often toward position of the target vertex, and the potential function rewards the correct direction and punish the other directions, which decreases the number of vertices being processed.

2.3 Landmark-based search

ALT is also a goal-directed method. It selects a subset of vertices as landmarks L in the graph G and precompute the distance between each landmark L and all the other vertices in G. It uses triangle inequality to find the lower bound of the possible distance between vertices.

$$\begin{aligned} d(v,w) \geq & d(L,w) - d(L,v) \\ d(v,w) \geq & d(v,L) - d(w,L) \end{aligned} \tag{3}$$

So the heuristic function can be derived as follows:

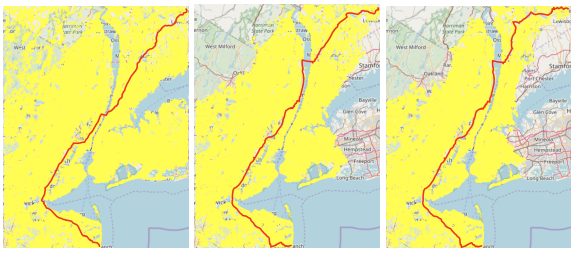
$$d(v,w) \geq max(d(L,w) - d(L,v), d(v,L) - d(w,L)) \tag{4}$$

Another critical issue is how to choose the subset of landmarks. We can surely randomly pick random number of landmarks from the graph. However, we should select the vertices that are farest from L to other vertices. Intuitively, in the regular road map, the vertices near the continent boader are good choices.

2.4 Contraction Hierarchy

Contraction Hierarchy also requires preprocessing. The algorithm involves 2 stages: contraction and query. At the contraction stage, a node v is contracted by removing it from the network and replaced it with the shortcuts that doesn't change the relation of each vertex. For example, path (u,v,w) is replaced by (u,w) while v is deleted. Also, vertices have different importance and the least important node is contracted first. Finally, at the query stage, it only searches neighbors with higher importance and it utilizes those shortcuts to speed up queries.

3 Experiments



(a) Dijkstra (b) A* (c) ALT

I implement several algorithms on DIMAC road datasets and the visualize my results on the routing challenges in New York city. Figure (a) almost traverse all nodes and Figure (b) and (c) based on goal-directed routing algorithms clearly skip many vertices that are unlikely on the shortest routes. I choose to use Manhattan distance as our heuristic function when running A*. Only 1 vertex at the top corner is used as a landmark. Surprisingly, when running my implementation on larger networks, bidirectional search doesn't show us performance gains. The possible reason might be there is no clear relations between weights in datasets and the real distances on map. I direcly use coordinates to calculate Manhattan distance and I realize that when running goal-directed algorithms, $nextCost = cost + estimateCost$. The $estimatedCost$ might be quite influential, as the distance are much bigger than weights provided by datasets. Therefore, I set an hyperparameter a to control the influence of $estimatedCost$. The hyperparameter a is currently optimized for NYC datasets, therefore the results on other datasets are unexpected.

	Dijkstra	B-Dijkstra	A*	ALT
NY	113.498	79.9082	74.8182	56.2518
US-E	1539.25	1551.82	1102.66	1203.71
US-W	2481.8	2620.68	2642.6	3072.37

Table 1: Experiment results in datasets, results are derived from average 100 random queries time in milliseconds
NYC - 263,436 nodes, 733,846 arcs
US-East - 3,598,623 nodes, 8,778,114 arcs
US-West - 6,262,104 nodes, 15,248,146 arcs

Admittedly, this project is still far from completion. The landmark selection procedure is not optimized; therefore, bizzard result is derived on US-west datasets. In addition, the implementation of Contraction Hierarchy algorithm still has bugs on Nov 2, 2019. Finally, all the codes can be found on https://github.com/qq52184962/routing-algo.

References

[1] DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numer. Math. 1*, 1 (Dec. 1959), 269–271.

[2] GEISBERGER, R., SANDERS, P., SCHULTES, D., AND DELLING, D. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms* (Berlin, Heidelberg, 2008), WEA'08, Springer-Verlag, pp. 319–333.

[3] GOLDBERG, A., AND HARRELSON, C. Computing the shortest path: A* search meets graph theory. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms* (04 2003).

[4] HART, P. E., NILSSON, N. J., AND RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC-4(2)* (1968), 100–107.