UROP Project Report

Chen Shr En

December 15, 2024

Abstract

Shortest path on a dynamic spatial network has been an important topic in the field of computer science. To answer the dynamic shortest path effiicently, one may want to precompute some results and stored them in an oracle. However, since the graph is dynamic in the sense the weights changes overtime, the oracles have to be updated. Therefore, it's difficult to balance the trade-off between the precomputation and the update. My project investigate this problme based on the approach in Wei, Wong, and Long (2024) [1]. It solves the dynamic shortest path problem efficiently with high probability. In this report, I will present some attempts to improve the result to be deterministically.

1 Introduction

The dynamic spatial network is prevelent in many real world applications. One of the most familiar application is "Google maps", which compute the best route in real-time. We can model the real world map as a graph $G = (V, E)$ with non-negative edge weights. The weights can represent the cost of traveling time. Users may want to know the fastest path from the starting position s to terminate position t. Then the goal is to find the shortest path from s to t in G. The graph is dynamic in the sense that the edge weights may change over time. Furthermore, we will have the assumption that the nodes and edges will not disappear or appear since that senario usually corresponed to a massive change, such as closing down a road or intersection. The change may take several days, and meanwhile, we can simple reconstruct our oracles. Therefore, we assume there is no edge or node change. Furthermore, we restrict ourself to road network, where the weights of edges can be the cost of traveling time.

One naive solution is running Dijkstra's algorithm given a time query. However, it's not applicable in practice since the running time is $O(|V|^2)$ or $O((|V| + |E|) \log |V|)$ which is a very large number in real road network, and hence the computation is heavy for only a query. Some existing algorithms [2, 3, 4] have been proposed to solve the dynamic shortest path problem. However, they either not having good enough runtime for a query or suffering from long updating time. In Wei, Wong, and Long (2024) [1], they propose a architecutre-intact oracles to deal with dhynamic shortest path problem. By constructing sufficient amount but not too many shortcuts, they reach a great balance in answering query and upodating orcales efficiently in polynomial time of $O(\log |V|)$. It's a break through in the field, but the runtime is still probabilistic. In this project, I tried to improve the result to be deterministic.

2 Literature Review

In this section, I will go through the main ideas in Wei, Wong, and Long (2024) [1], identify its limitation, and propose some possible improvements.

2.1 Architeture-intact oracles

Different others algorithms which create and remove the shortcuts on the fly, this paper propose an architeture-intact oracles. The shortcuts are created in the preprocessing stage and will not be removed or added afterwards. Those shortcuts speed up the query time by reducing the number of nodes to be visited. Furthermore, while the weights of edges change, only the weights in the oracles change, instead of the graph architeture. We call our oracles G' which is the original graph G plus some created shortcuts.

2.2 Constructing shortcuts

The construction of shortcuts will help us prune the search space in the query time. Before we define shortcuts, we define *rank*, which is an oredering on the vertices given a permutation of $\{1,2,...,n\}$. The rank value of v is denoted as $R_I(v)$, where I is a permutation of $\{1,2,...,n\}$. One can interpret it as the "height" of the vertices. Then we define *valley path* from s to t as a path that the intermediate vertices have ranks less than $\min\{R(s),R(t)\}$. The shortest one among all vally paths from s to t is called the *shortest valley path*. The algorithm will construct shortcuts so that for every pair of vertices s and t which have valley paths connecting them, there is a shortcut connecting them with weights as the length of the shortest valley path. This helps us prune out the search space of all valley paths.
The algorithm for shortcut construction is as follows:

1. Generate a random ordering (the ranks) of the vertices.

2. Process the vertices in the order of the ranks (from small to large). At the i-th vertex, we will create shortcuts for (u,w) if (u,v_i) and (v_i,w) are in the graph.

The Property 1 and Property 2 in the paper together show the correctness of the algorithm. The complexity will be $O(n\log^2 n)$ with high probability. *Remark*: The architeture of oracles is completely dependent on the given ordering of ranks only.

2.3 Answerng query

Given the oracles G' with the shortcuts created above, we can answer the query efficiently. We run bidirectional Dijkstra's algorthms with *directional constraints*. Suppose we want to query the shortest path from s to t. We run Dijkstra's algorithm from s and t simultaneously. When we expand a vertex v in the forward search, we will only consider the edges (v,w) that $R(v) < R(w)$. Similarly, in the backward search, we will only consider the edges (u,v) that $R(v) > R(u)$. This helps us to prune the search space. The correctness and time complexity of the algorithm can be found in the tecnical report of the paper. The complexity is $O(log^4 n \log\log n)$ for distance query and $O(log^4 n \log\log n + l)$ for shortest path query, where l is the length of the longest shortest path.

2.4 Updating oracles

The updating part is the distinguishing part of this paper from the others. It shows us that although we've created sufficient amount of shortcuts, but while updating a weight from an edge, the amount of shortcuts weights we need to update will not be too many. Suppose we update weights of (u,v) from w to w', where $R(u) < R(v)$ with out loss of generality. Then all shortcuts which will be affected are the ones whose real path is a shortest valley path containing the edge (u,v). The algorithm, *update(G', (u,v), w')* for updating the oracles is as follows:

1. If there is some shortest path from (u,v), then update the weights of the shortcuts to be $\min\{w,w'\}$. If there is no shortcut, then assign the weight to be w'.

2. If there is a change of the weight of (u,v), we will call an algorithm, *subUpdate*, for further subupdating.

The algoirthm *subUpdate(G', (u,v))* is as follows: It looks for all vertex x which has an in-edge to u, check whether the edge (x,v) has to be updated. If so, call recursion on $G(G',(x,v))$. The correctness of the algorithm can be found in the paper. The complexity will be $O(\log^3 n)$ with high probability, which is significantly less than the state-of-the-art algorithms.

2.5 Limitation

In this subsection we look at some simple bad cases in the algorithm. In other words, the complexity of algorithms will be polynomial in n instead of $\log n$ for those bad spatial networks.

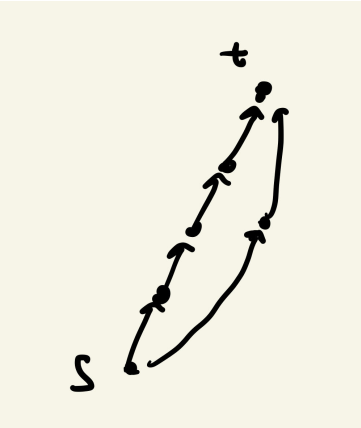


Figure 1: Bad case 1

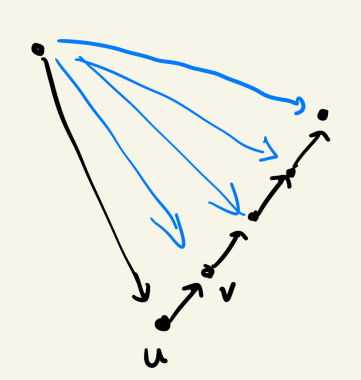


Figure 2: Bad case 2

- Bad case 1: A graph with acending paths only. Then no shortcuts will be created since there is no valley path. Hence, the query of shortest path between s and t will take $O(n)$ time.
- Bad case 2: A graph with a valley path only. Then when updating the weights of (u,v) there might involves all blue shrotcuts, which contribute $O(n)$ runtime also.

These two simple examples show that for any given random ordering, we can find some bad spatial networks which make the algoirthm slow. It's worth knowing that although it's easy to come up with some simple examples that the algorithms will have high complexity, but those cases have very low probability to happen since most of them require long consecutive acending/desencding path. Furthermore, the two examples are bad when there are no connections between other virtices, which is also unlikely in the real spatial network. Therefore, we hope that fixing these problems will be sufficient to improve the algorithm to be deterministically efficient.

3 Approaches

As discussed in the previous section, the main issue for the algorithm to fail is that we only have one ordering of vertices and it's easy to construct counter examples. However, those examples are somehow extreme cases, such as having long ascending/desencding paths and has few connections with other vecrtices. Therefore, we can try to introduce multiple (k) ordering and correspondingly create multiple oracles. Hopefully, the bad cases in some particular ordering will no longer be the bad cases in other ordering. Additionally, to guarantee that the runtime is still in polynomial of $log(n)$, we try to find $k \in O(\log^c n)$ for some constant c.

3.1 Rabin-Carp-like rotations

We want to "break down" the long acending/desending paths in the bad cases. We consider a Rabin-Carp like rotation on the rank of the vertices and construct multiple graphs.
In this section, I will go through some approaches I've tried and discussed whether it's feasible and present graudal approach to attack this problem although I haven't found the final solution.

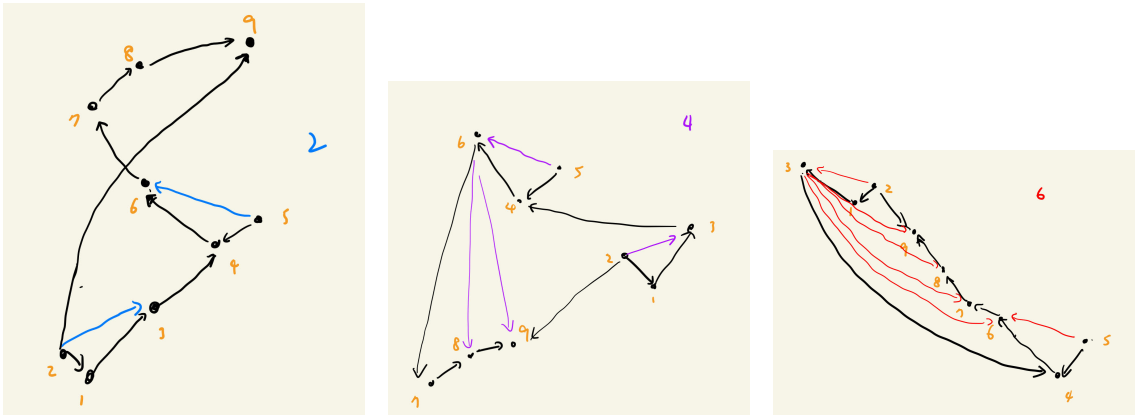
1. Pick a random ordering of vertices

2. Partition the vertices into k groups with the i-th group $g_i = \{v : 1 + i \cdot \frac{n}{k} \leq R(v) \leq (i+1) \cdot \frac{n}{k}\}$ for $i = 0, 1, ..., k-1$.

3. Do k times Rabin-Karp-like rotations of the k groups and get k different graphs. Denote the graph resulted from the i-th rotations as G_i for $i = 0, 1, ..., k-1$.

4. Construct shortcuts for each graph

Here is an example (Figure 3) with $n = 9$ and $k = 3$: The black arrows are the edge in the original spatial networks and the colored arrows are the shortcuts created under a specific ordering of vertices.



(a) G_0 (b) G_1 (c) G_2

Figure 3: Rabin-Carp like rotations

We can see that in the initial graph G_0, the query $(1,9)$ has to go through the long ascending path. However, after the rotation G_1, G_2, the path is broken down and the query will be faster. However, this is not always the case. Since k is small, the size of each group is nearly large $\sim O(n/\log n)$. If those bad cases have no connections with other groups, then there is still no shortcuts created for them. This indicates that with **only one ordering** is not enough.

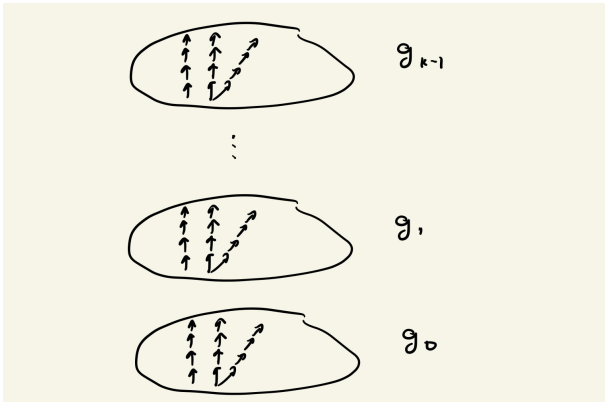


Figure 4: Bad case for rotations

3.2 Multiple Ordering

Although Rabin-Carp-like rotations can help us break down bad cases crossing different groups, we've seen it may end up with those bad cases inside each group. This gives motivation to

have multipler ordering principles for differerent groups. Firstly, let's see discuss how will the shortcuts change when we change an oredering in a particular group. For simplicity, we discuss the case in G_0 for any random ordering. We notice that when we change the ordering inside g_i for $1 \leq i \leq k-1$, some shortcuts will not be changed. Let's characterize those shortcuts. Notice that each shortcut represents a shortest valley path.

- Case 1: Shortcuts inside $\bigcup_{j=0}^{i-1} g_j$ (between the vertices inside the subset)

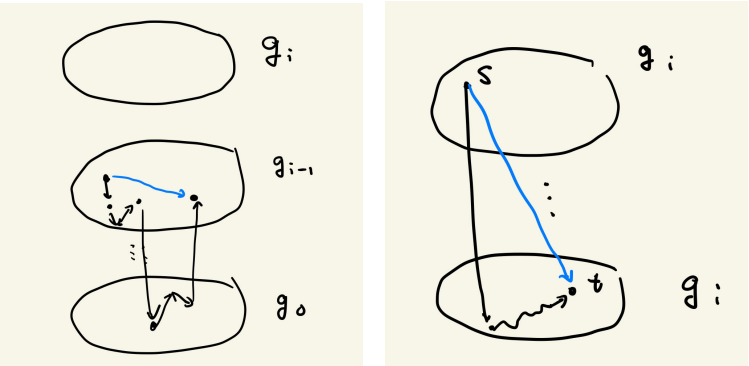
  It's because that the valley path inside g_j will not used any vertices in g_i so there will be no new shortcuts nor the old shortcuts will be removed. Furthermore, the weights of such shortcuts will not change since it doesn't involve any edge from g_i.

- Case 2: Shortcuts starting from $s \in g_i$ and ending in $t \in g_j$ $(0 \leq j \leq i-1)$

  The valley path from s to t will only use the vertices having smaller rank than t. In other words, those intermediate vertices will be in $g_0 \cup g_1 \cup \cdots \cup g_j$. Then after reordering the g_i, the valley path will still exist. Furthermore, the weights of such shortcuts will not change since it doesn't involve any edge from g_i.

- Case 3: Shortcuts starting from $s \in g_j$ and ending in $t \in g_i$ $(0 \leq j \leq i-1)$

  The same argument as above applies.



(a) Case 1 (b) Case 2

Figure 5: Shortcuts remain after reordering

By the above arguments, it's feasible to change the ordering with bottom-up order since while changing the order in the upper groups, the "lower" shortcuts won't be affected.

3.3 Recursion

Another approach is to use Recursion with Rabin-Carp rotation to reorder each group. However, due to time limitation, I haven't explored deeply on this approach yet.

4 Conclusion

In this report, I've discussed the dynamic shortest path problem and the state-of-the-art algorithm proposed in Wei, Wong, and Long (2024) [1]. I pointed out the limitation of the algorithm and proposed some possible improvements, such as constructing different graph with Rabin-Carp rotation and introducing multiple ordering. There are still more to explore in this field. For example, how to construct the reordering principles in each group exactly, and furthermore, analyzing whether the efficient updating property still holds.

References

[1] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long, *Architecture-Intact Oracle for Fastest Path and Time Queries on Dynamic Spatial Networks*, DR-NTU (Nanyang Technological University), May 2020, doi: 10.1145/3318464.3389718.

[2] J. Dibbelt, B. Strasser, and D. Wagner, *Customizable Contraction Hierarchies*, Journal of Experimental Algorithmics (JEA), 2016.

[3] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, *Exact Routing in Large Road Networks Using Contraction Hierarchies*, Transportation Science, 2012.

[4] T. Akiba, Y. Iwata, and Y. Yoshida, *Dynamic and Historical Shortest-Path Distance Queries on Large Evolving Networks by Pruned Landmark Labeling*, In Proceedings of the International World Wide Web Conference (WWW), 2014.