Comparison and Optimization of the Dynamic Shortest Path Algorithms

Students Name :	TAO Zhongnian, YANG Junzhao, XU Zhuoyu
School :	Nanjing Foreign Language School
Province :	Jiangsu Province
Country :	China
Advisor :	Simon Fraser University WANG Jiannan

Contents

Comparison and Optimization of the Dynamic Shortest Path Algorithms

Tao Zhongnian, Yang Junzhao, XuZhuoyu

Abstract

Information Technology has enabled new levels of convenience to our daily life. While providing people with astonishingly good experiences, computer science has still a long way to meet the needs from the users for various scenarios. As a good example, path selection problem is an important and common problem which finds applications in location based services, internet routing and navigation systems. Choosing the most cost effective and time saving routines plays an essential role in both theory and engineering. Although a plethora of classical solutions have been made to the static shortest path problem, dynamic shortest path problem has not been thoroughly studied. To address this critical issue, this paper investigates the dynamic shortest path problem by optimizing existing work and conducting extensive experiments. The studies fall into three main variants of the problem: unweighted shortest paths, weighted shortest paths, and shortest paths on the map. We present a comprehensive comparison of our optimizations and existing solutions which demonstrates the effectiveness and efficiency of our algorithms.

Keywords

Dynamic graphs, the Shortest path, Algorithm, Optimization

Highlights of the Problem Selection:

The shortest path problem is a well-known issue in computer science. It is widely used in computer science, economics and many engineering problems. It is of great value both in theory and practice. Extremely rich studies have been made on it. We focus on the topic and try to improve and optimize the existing algorithms and apply them to solve real life problems.

Highlights of Our Solution

Today, the widespread use of GIS has made shortest path a reviving topic in research community. However, it is often implemented in a conventional and straightforward way such as BFS (Breadth

First Search) in unweighted graphs and Dijkstra Algorithm in weighted graphs.

The algorithms achieve good performance on small-scale datasets. However, in the situation with large-scale datasets, they may suffer from non-negligible inefficiency. Even on the medium-sized datasets, they still cannot establish efficiency guarantees. To solve these problems, we propose several optimizations and techniques to dramatically reduce the running time and outperform the baselines.

Contributions:

1. Present the basic methods and their improvements.

2. Make empirical studies on synthetic datasets. Moreover, we make our code available for reproducibility.

3. Give comprehensive time and space analysis and experimental comparisons with existing methods.

Novel aspect of the paper

1. We define the important problem of map shortest path. We discover the map properties and design dynamic shortest path algorithms on the map.

2. We present heuristic algorithms on dynamic map graphs. As verified by experiments, the proposed technique has good efficiency.

3. We propose the improvements when the query points are restricted in a region. The experimental results show the scalability of the method.

4. We compare our optimized algorithms against existing shortest path algorithms.

1. Introduction

1.1 Problem Significance

The point to point shortet path problem, i.e., finding the shortest path from a source to a target in a graph with many nodes, has drawn considerable attention in computer science. The pioneering work in this field is from the ACM Turing Award laureate, Edsgar Wybe Dijkstra, who solved the problem in $O(|E|+|V|\log|V|)$ time with effective implementation, where V,E are the sets of vertices and edges. Since then, the research community had sparked enthusiasm in studying this problem.

Motivated by commercial navigation systems, applications to large scale datasets require faster algorithms. On the other hand, the problem is still open especially when considering applying existing solutions to dynamic problem settings, which is close to real-world applications. Take the city road network management as an example, the administrators can give every road section a time weight according to real-time traffic status. By making dynamic shortest path computations, we can work out the shortest path for drivers and broadcast the information to the public by media. In this way, the shortest path algorithm may not only let the drivers reach their destinations in shorter path but also improve the efficiency of traffic, quality of traffic services, etc. In addition, this algorithm can also be applied to solve logistic address-selecting, cyber space construction, and many other problems, to make our lives more convenient.

1.2 Paper Motivations

Although fruitful results have been presented in the research community, real-world applications often adopted comparably simple solutions -- Breadth-First Search (BFS) and the Dijkstra algorithm, in an unweighted and a weighted graph, respectively. These algorithms are usually only fit for small-scale datasets, and large-scale datasets and some special cases may greatly reduce their efficiency. While in real scenarios, the researchers must wade through the difficulties of dynamicity and huge volume of data.

We consider the dynamic version of shortest path and propose several optimizations based on existing work. Studies in the paper fall into three main folds: unweighted shortest path, weighted shortest path, and shortest path on the map. We present a comprehensive comparison of our optimizations and existing solutions, demonstrating both effectiveness and efficiency

1.3 Paper Novelties

We define the important problem of map shortest path by discovering the map properties; and we design dynamic shortest path algorithms on the map.

We present a heuristic-based algorithm to the problem of finding the shortest path on dynamic map graph. Our experimental results show that the proposed technique achieves good efficiency.

In addition to the general setting of the classic problem, we also consider the case where query points are restricted in a region. Extensive empirical studies have been done and show that the

proposed method has a good scalability when applying on large-scale datasets.

We conduct extensive experiments on the comparison of our proposed methods and classic methods, demonstrating the advantage of our method both theoretically and empirically.

All code for the design and implementation of our algorithms, as well as the comparison and analysis of algorithms related to this paper is an open-source package written in c++. More information and code can be found at https://github.com/nflstyx/shortest-path-project to see the repeatability of our experiments.

2. Unweighted Dynamic Shortest Path

2.1 Problem description

In a weighted undirected graph G=(V,E) with n vertices and m edges, there are q operations and each of which is in one of the following three types:

Operation 1: Query how many edges need to be passed from S to T;

Operation 2: Adding an edge $\{u,v\}$;

Operation 3: Deleting the added *i-th* edge(The number in former graph is designated by the order of $1\ldots m$. The edge in operation 2 is numbered from m+1. The sequence number is not affected by the operation of deleting one edge.)

It is guaranteed that there are no overlapping edges and self-loop in this undirected graph.

2.2 Experimental Settings

2.2.1 Datasets

Data can be categorized into two types respectively in each of the two aspects——the density of edges and operation types. According to the density of edges, data can be categorized as sparse data and dense data. According to operation types, data can be categorized as the one with the same operation ratio (50% for query operations and 50% for two kinds of modification operations), and the other one with big differences in operation ratio (90%for query operations and 10% for modification operations).

Except for numbers of vertices, edges, queries and query ratio, which are decided factitiously, all other data are produced randomly.

For dense datasets, n=4000,q=10^6, m is in proportion to $O(n^2)$. We designate the density of dense graph as $k = \frac{m}{n^2}$.

For sparse datasets, n=10^5,q=10^6, m is $O(n)$. We designate the density of sparse graph as $\mathrm{k} = \frac{m}{n}$.

2.2.2 Platform Configurations

In this section, program running time is frequently mentioned. For clarity, all running time is measured by second and excludes the time of reading data. The running environment is 64 bits Windows 10 operating system, Intel Core i7-6600U CPU @ 2.60GHz, 2.81GHz，16GB RAM.

C++ Compiler is TDM-GCC 4.9.2 64-bit Release and the -O2 optimization is open. To reduce errors, all running times are given as the median of several running results.

2.3 Background

2.3.1 Breadth-First Search (BFS)

(1) Algorithm Description

Breadth-first search (BFS) is an algorithm for traversing or searching or graph data structures. It starts from the source vertex and explores the neighbor vertices first, before moving to the next vertex.

Algorithm 1 Find the shortest path in an unweighted and undirected graph using BFS

Require: Graph $G = (V, E)$, Source vertex S, Target vertex T.

Ensure: Distance from S to T, or a conclusion that no path exists from S to T.

```
1: function BFS(G, s, t)
2:     S ← ∅
3:     Q ← empty queue
4:     S ← S ∪ {s}
5:     Q.push(s)
6:     while Q is not empty do
7:         v ← Q.pop-front()
8:         for all {u, v} ∈ E do
9:             if u ∉ S then
10:                dist(u) ← dist(v) + 1
11:                if u == t then
12:                    return dist(u)
13:                end if
14:                S ← S ∪ {u}
15:                Q.push(u)
16:            end if
17:        end for
18:    end while
19:    return PathNotExist
20: end function
```

Figure 1: Pseudocode of BFS algorithm in unweighted and undirected graphs

(2) Time and Space Complexity

Under worst-case conditions, every vertex and every edge will be traversed, so the time complexity of single query is $O(|V| + |E|)$, in which ($|V|$ is the number of vertices and $|E|$ is the number of edges. The complexity of a single adding edge operation is $O(1)$O(1). The complexity of a single deleting edge operation is the average of $O(|E|/|V|)$, because alladjacent edges of a vertex need to be traversed.

The overall space complexity is $O(|V|+|E|)$ as well, because all edges and vertices of the graph need to be stored.

(3) Experiments on BFS

Running time is shown below:

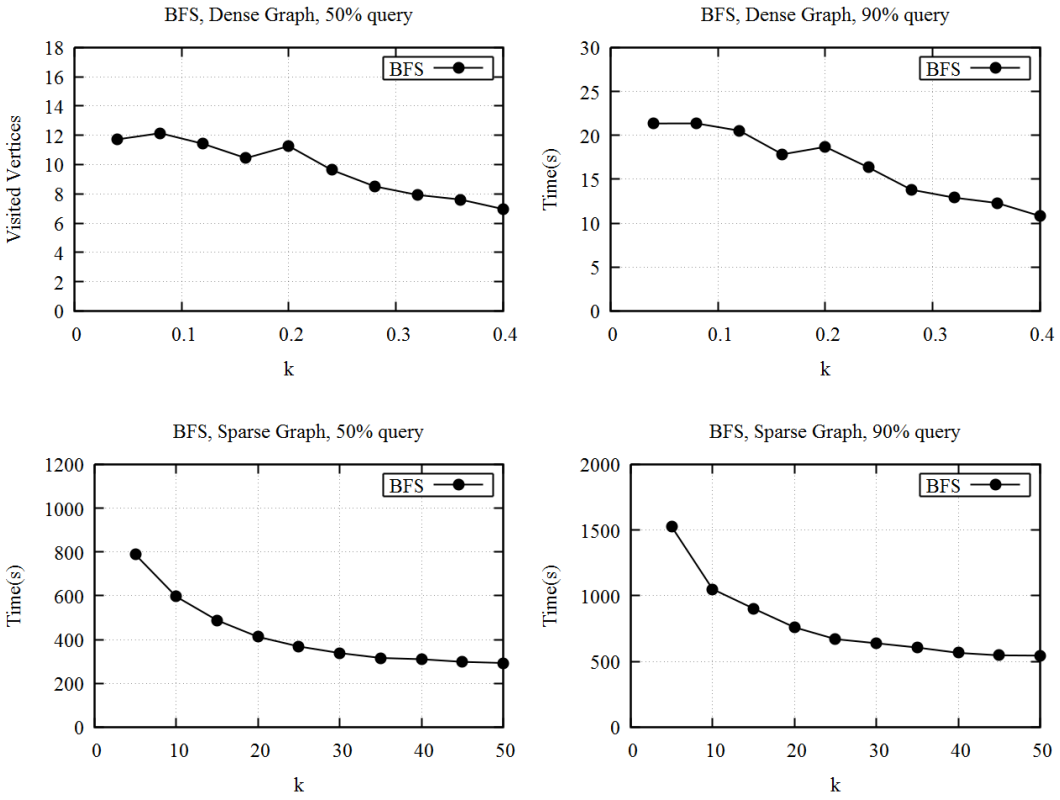
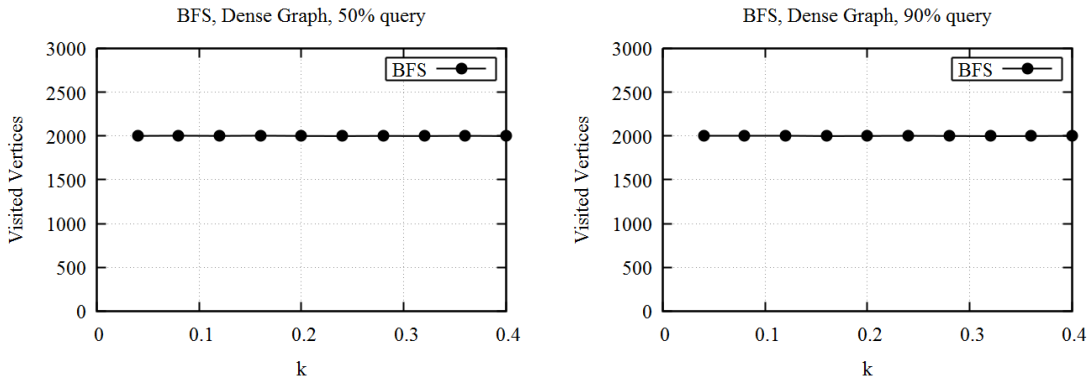


Figure 2: BFS running time in different graphs

The running time of BFS is mainly correlated to the number of vertices visited. The average number of vertices visited is listed below:



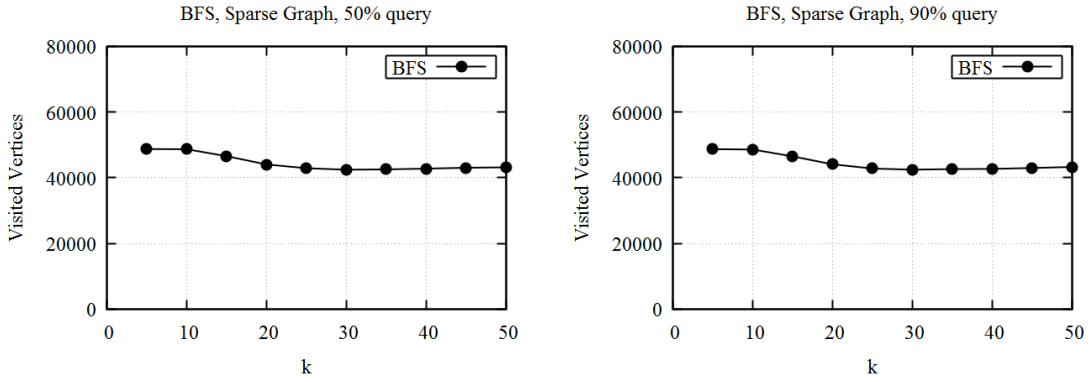


Figure 3: Average number of vertices BFS visited on different datasets

From the test results we can find out that in a dense graph, BFS algorithm visits approximately half of the vertices and this portion is kept fairly stable. While in a sparse graph, the number of vertices visited by BFS Algorithm is correlated to the density of graph. The larger the density is, the less the number of vertices visited is. Two types of different operation ratios have little impact on the average speed of dealing with a single query.

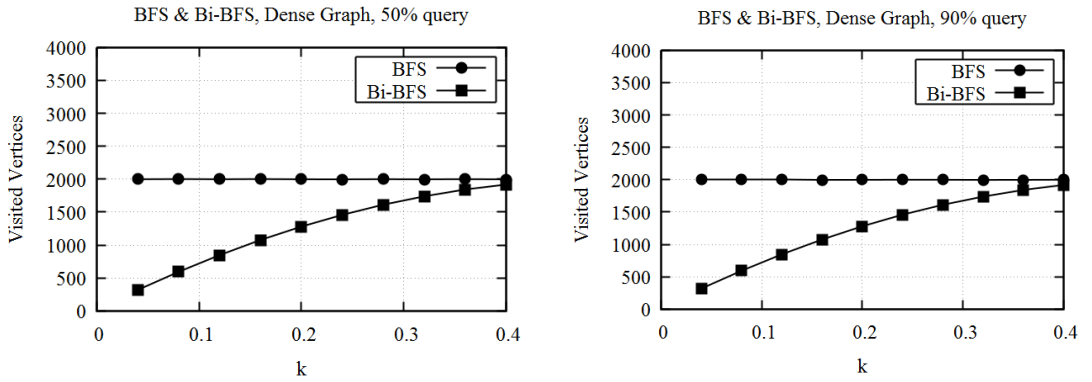
2.3.2 Bi-Directional Breadth First Search (Bi-directional BFS)

(1) Algorithm description

Now that we know the source and target of every query, we can start from two directions simultaneously so as to reduce the number of vertices passed. It conducts two search processes at the same time: one is a forward search from the source and the other is a backward search from the target. The two searches will not stop until both meet.

(2) Experiments on Bi-directional BFS

The advantage of bi-directional BFS is that it can reduce the number of vertices visited. In different datasets, the number of vertices visited is shown below:



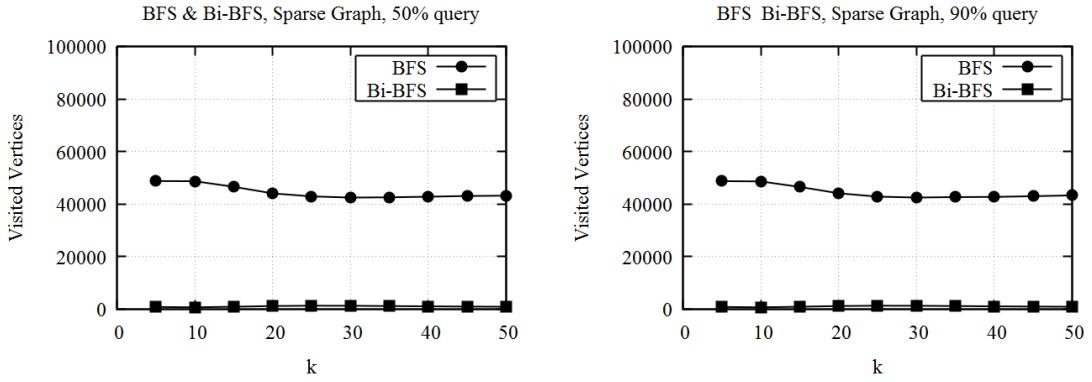


Figure 4: Number of vertices BFS and Bi-BFS visited on different datasets

We can see that in comparison with the basic BFS, this approach dramatically reduces the number of vertices visited.

Next, we move to the analysis of running time.

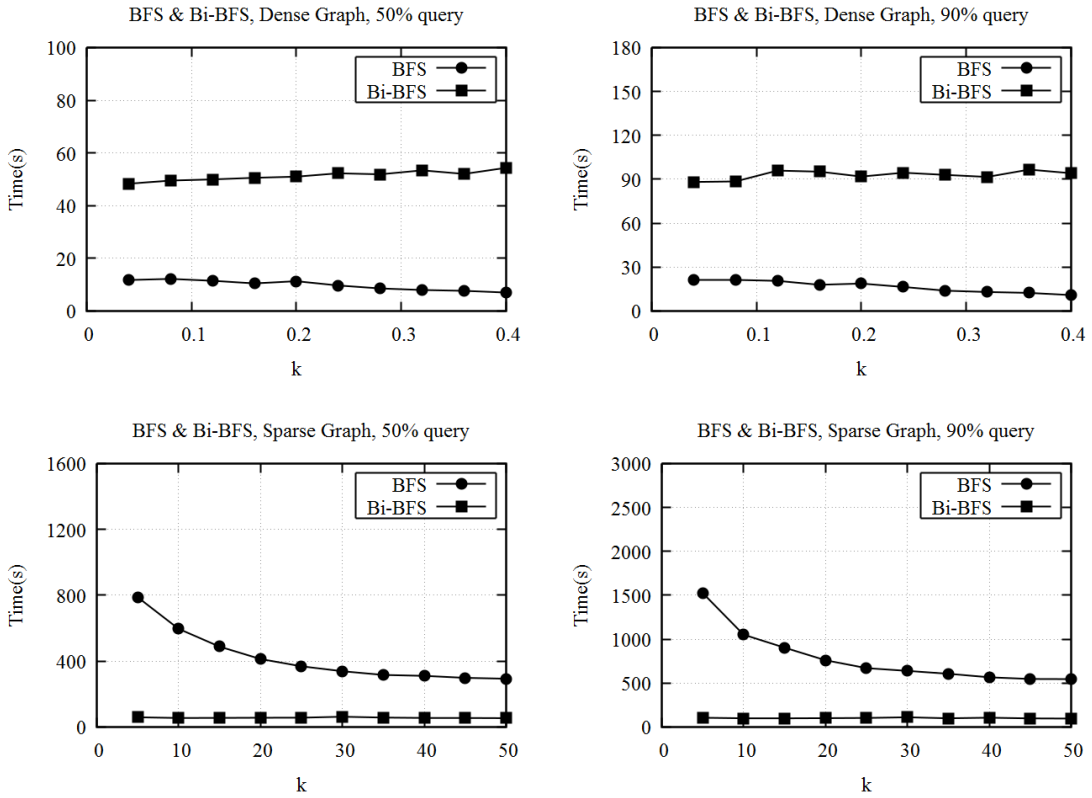


Figure 5: Running time of BFS and Bi-BFS in different graphs and queries

In a dense graph, although the number of vertices visited drops greatly, due to the shortness of the shortest path algorithm, the cost of maintaining two directions of BFS makes the program run at a relatively slow speed. This therefore makes the single direction BFS better. While in a sparse graph, bi-directional BFS is obviously better.

2.3.3Bit Compression

(1) The Essence of Bit Compression: treating each integer as a set

An integer with *w* binary digits can be regarded as a set of size *w* in which every digit indicates the existence of a corresponding element. The time complexity of merge, intersection, supplement and size-computation operation of a set can be done in $O(n/w)$, in which *n* stands for the size of the set. The space complexity of the set is also $O(n/w)$, or *n* bits. It is convenient to express a set of size 32 by **unsigned int** in C++.

In bit compression algorithm, we consider the use of **unsigned int** to store graphs (namely w=32). If the counterpart of the position $\{u,v\}$ in the form of **unsigned int** is binary 1,it means that edge $\{u,v\}$ exits, and vice versa.

(2) Algorithm Description

Algorithm 2 Store graphs using bit compression

```
1: function GetBlockIndex(u,BlockIndex)
2:     if BlockIndex unexists then
3:         Create new Block
4:     end if
5:     Return the Index of BlockIndex in u // use a hash table to maintain this
6: end function
7: function AddDirectedEdge(u, v)
8:     BlockIndex ← ⌊v/w⌋
9:     InBlockIndex ← v mod w
10:    Set the InBlockIndex-th binary digit of Graph[u][GetBlockIndex(u,BlockIndex)] to 1
11: end function
12: DirtRatio = p
13: function RemoveDirectedEdge(u, v)
14:    BlockIndex ← ⌊v/w⌋
15:    InBlockIndex ← v mod w
16:    Set the InBlockIndex-th binary digit of Graph[u][GetBlockIndex(u,BlockIndex)] to 0
17:    if the ratio of 0 in G[u] > DirtRatio then
18:        Delete All 0 in G[u]
19:        Update BlockIndex of u
20:    end if
21: end function
22: function AddEdge(u, v)
23:    AddDirectedEdge(u, v)
24:    AddDirectedEdge(v, u)
25: end function
26: function RemoveEdge(u, v)
27:    RemoveDirectedEdge(u, v)
28:    RemoveDirectedEdge(v, u)
29: end function
```

Figure 6: Pseudocode of storing graphs using bit compression

(3) Space Complexity

If using adjacency matrix to store the graph after bit compression, the space complexity will be $O(n^2/w)$. When n=500,000, it may occupy 232.8 GB. The space complexity is far too big! So we replace it with adjacency list. Comparing to adjacency matrix, it is required to store the sequence number of current vertex into adjacency list. This way, each element of adjacency list is expressed by two integers, standing for the number of the smallest point in current set and the information of the edge respectively. After that, the space complexity to store graphs is optimized to $O((n+m)/w)$.

(4) Experiments on Bi-Directional Bit Compression

The comparison of running time of bi-directional bit compression BFS and bi-directional BFS is shown below:

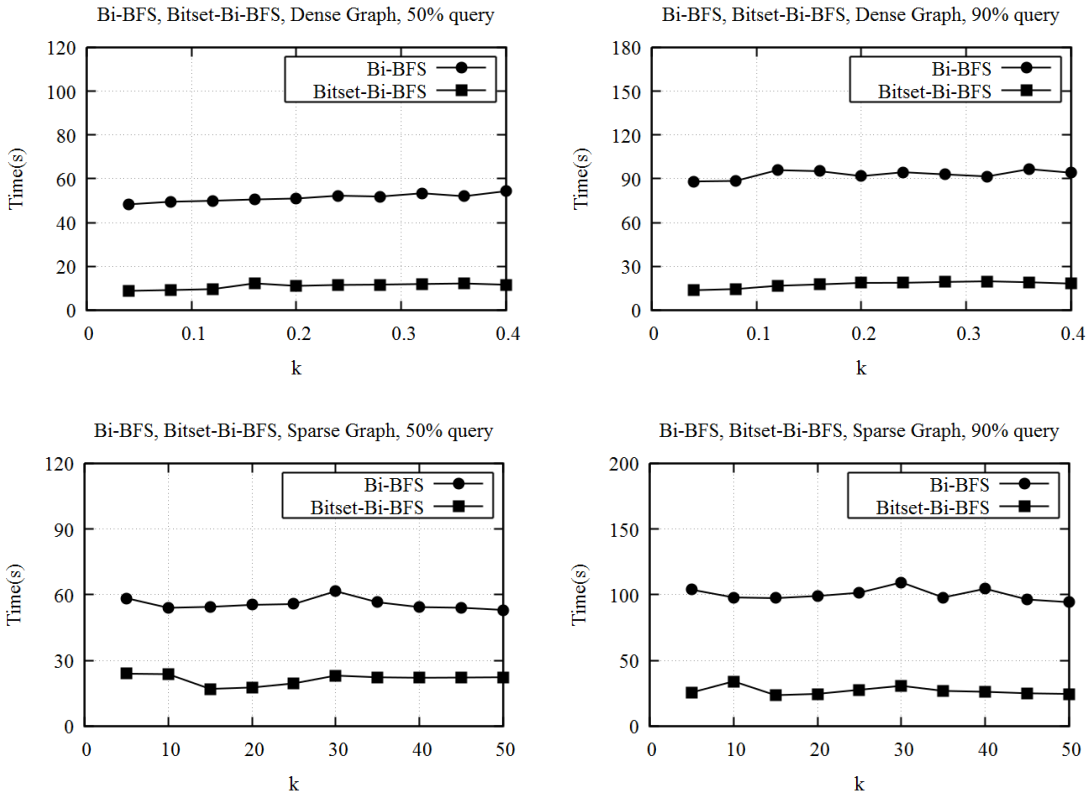


Figure 7: Comparison of running time of Bi-BFS and Bitset-Bi-BFS on different datasets

The optimization of bit compression is quite effective for both dense graphs and sparse graphs. Especially when applied to dense graphs, this optimization can compress adjacency list smaller, so its effect is better than that on sparse graphs. But in sparse graphs, with the increase of density, the running time of bit compression in 50% queries approximates that in 90% queries. It demonstrates that the modification operation costs a relatively long time on the data of 50% queries.

In the second algorithm, because graph is read frequently, we use vector as the storing structure. In

this way, the complexity of reading operation is $O(1)$. But elements in vector cannot be deleted efficiently, it is time-consuming to delete edges.

Now, we can consider a *"lazy"* deleting operation. When setting a set adjacent to *v* as 0, although there is no edge in the set, we do not delete it at once. When over a certain proportion of numbers in the sets correlated to v are value 0, we delete all sets with value 0.

We defineDirtRatioas the threshold, and later we enumerate different DirtRatio to test our program in sparse graphs:

DirtRatio	**0**	**0.05**	**0.1**	**0.15**	**0.2**	**0.4**	**0.6**	**0.8**
Average running time in 50% query(s)	21.344	16.606	15.498	15.792	16.015	16.146	16.009	16.002
Average running time in 90% query(s)	26.808	23.795	24.705	24.999	25.722	26.068	25.663	25.785

Figure 8: Average running time in different queries

Therefore, the value of DirtRatio should be around 0.1.

(5) Data Collected from Programs

All data is shown below:

①Data collected from unweighted graph programs:

Data No.	n	m	q	k	Query Ratio (%)	BFS Running Time (s)	BFS Average Visited Vertices	Bi-BFS Running Time (s)	Bi-BFS Average Visted Vertices	Bi-BFS+Bit Compression Running Time (s)
1	4000	640000	1000000	0.04	50	11.720	1999.978	48.283	318.587	8.765
2	4000	640000	1000000	0.04	90	21.346	1999.918	88.113	318.656	13.586
3	4000	1280000	1000000	0.08	50	12.144	2002.439	49.474	594.279	9.131
4	4000	1280000	1000000	0.08	90	21.350	2000.026	88.462	593.981	14.383
5	4000	1920000	1000000	0.12	50	11.418	1999.953	49.923	847.980	9.545
6	4000	1920000	1000000	0.12	90	20.522	2000.093	95.882	847.698	16.612
7	4000	2560000	1000000	0.16	50	10.439	2002.490	50.531	1077.860	12.183
8	4000	2560000	1000000	0.16	90	17.831	1997.894	95.137	1076.932	17.569
9	4000	3200000	1000000	0.2	50	11.265	1999.449	50.951	1280.876	11.042
10	4000	3200000	1000000	0.2	90	18.681	1999.527	91.792	1281.545	18.598
11	4000	3840000	1000000	0.24	50	9.629	1998.287	52.236	1459.549	11.474
12	4000	3840000	1000000	0.24	90	16.348	2000.804	94.443	1461.212	18.673
13	4000	4480000	1000000	0.28	50	8.510	2000.233	51.859	1613.814	11.624
14	4000	4480000	1000000	0.28	90	13.800	1999.684	92.924	1613.138	19.243
15	4000	5120000	1000000	0.32	50	7.924	1998.687	53.352	1740.136	11.887
16	4000	5120000	1000000	0.32	90	12.894	1997.679	91.541	1739.870	19.710
17	4000	5760000	1000000	0.36	50	7.609	2000.866	51.992	1844.401	12.140
18	4000	5760000	1000000	0.36	90	12.284	1998.644	96.558	1842.416	19.047
19	4000	6400000	1000000	0.4	50	6.955	1998.775	54.320	1919.125	11.557
20	4000	6400000	1000000	0.4	90	10.800	2000.293	94.084	1920.398	18.131
21	100000	500000	1000000	5	50	787.555	48721.295	58.403	803.419	23.906
22	100000	500000	1000000	5	90	1525.106	48697.135	103.829	803.858	25.545
23	100000	1000000	1000000	10	50	595.554	48664.584	54.023	651.282	23.699
24	100000	1000000	1000000	10	90	1049.854	48581.033	97.909	651.547	34.045
25	100000	1500000	1000000	15	50	487.408	46538.939	54.442	881.275	16.969
26	100000	1500000	1000000	15	90	902.045	46477.388	97.377	880.405	23.497
27	100000	2000000	1000000	20	50	412.054	44022.034	55.357	1144.049	17.604
28	100000	2000000	1000000	20	90	758.932	44077.459	98.998	1145.617	24.488
29	100000	2500000	1000000	25	50	368.037	42892.222	55.725	1269.943	19.473
30	100000	2500000	1000000	25	90	669.207	42787.675	101.497	1269.005	27.620
31	100000	3000000	1000000	30	50	337.895	42431.780	61.607	1241.248	23.049
32	100000	3000000	1000000	30	90	637.822	42440.415	109.358	1240.354	30.614
33	100000	3500000	1000000	35	50	315.577	42529.303	56.548	1143.016	22.261
34	100000	3500000	1000000	35	90	604.964	42615.961	97.880	1142.620	26.797
35	100000	4000000	1000000	40	50	309.960	42766.248	54.329	1037.118	22.056
36	100000	4000000	1000000	40	90	564.562	42679.738	104.549	1036.705	26.085
37	100000	4500000	1000000	45	50	297.830	43033.559	54.038	947.990	22.150
38	100000	4500000	1000000	45	90	545.755	42950.565	96.348	946.789	24.943
39	100000	5000000	1000000	50	50	292.471	43163.133	53.032	878.108	22.275
40	100000	5000000	1000000	50	90	544.022	43277.000	94.281	880.890	24.445

Figure 9: Data collected from unweighted graph programs

② Data collected when testing bit compression parameters:

Data No.	n	m	q	k	Query Ratio (%)	DirtRatio= 0	DirtRatio= 0.05	DirtRatio= 0.1	DirtRatio= 0.15	DirtRatio= 0.2	DirtRatio= 0.4	DirtRatio= 0.6	DirtRatio= 0.8
1	100000	500000	1000000	5	50	23.906	19.879	20.083	20.616	20.922	21.368	21.492	21.514
2	100000	500000	1000000	5	90	25.545	32.969	35.227	31.537	33.985	35.395	33.991	33.957
3	100000	1000000	1000000	10	50	23.699	15.127	15.300	13.940	15.570	15.491	14.864	15.443
4	100000	1000000	1000000	10	90	34.045	21.472	25.547	22.426	23.550	25.674	23.657	23.667
5	100000	1500000	1000000	15	50	16.969	16.143	16.314	15.848	16.377	15.912	15.779	15.876
6	100000	1500000	1000000	15	90	23.497	22.644	23.287	23.114	25.210	25.084	25.054	25.079
7	100000	2000000	1000000	20	50	17.604	18.312	16.736	15.049	16.821	17.316	16.987	16.857
8	100000	2000000	1000000	20	90	24.488	24.998	25.644	23.660	27.162	27.293	27.143	27.092
9	100000	2500000	1000000	25	50	19.473	17.487	16.815	15.065	17.077	17.161	17.117	17.265
10	100000	2500000	1000000	25	90	27.620	25.756	26.175	28.610	28.317	28.310	28.097	29.302
11	100000	3000000	1000000	30	50	23.049	16.329	15.911	18.863	16.371	16.321	16.681	16.309
12	100000	3000000	1000000	30	90	30.614	23.871	24.004	26.795	26.655	27.054	26.596	26.635
13	100000	3500000	1000000	35	50	22.261	15.565	14.455	16.242	15.453	16.109	15.460	15.418
14	100000	3500000	1000000	35	90	26.797	22.415	23.128	26.042	25.164	25.188	25.055	25.102
15	100000	4000000	1000000	40	50	22.056	17.789	13.548	14.407	14.527	14.571	14.579	14.555
16	100000	4000000	1000000	40	90	26.085	23.535	21.440	23.954	23.478	23.508	23.438	23.545
17	100000	4500000	1000000	45	50	22.150	15.471	13.173	13.649	13.784	13.822	13.798	13.602
18	100000	4500000	1000000	45	90	24.943	20.972	22.020	21.831	22.325	21.805	22.187	22.262
19	100000	5000000	1000000	50	50	22.275	13.955	12.641	14.243	13.247	13.385	13.330	13.177
20	100000	5000000	1000000	50	90	24.445	19.315	20.582	22.024	21.370	21.368	21.416	21.212

Figure 10: Data collected when testing bit compression parameters

3. The Weighted Dynamic Shortest Path Problem

3.1 Problem Description

Next, we introduce weight into the dynamic shortest path problem. Every edge is assigned an integer weight in the range of [1,1000]. Query of "Operation 1"in section 2.1 is transformed into how to compute the shortest distance between two vertices.

3.2 Background

3.2.1 Dijkstra Algorithm

Dijkstra algorithm solves the shortest path problem by computing the shortest path from a vertex to all of the other vertices in a graph.

Algorithm 3 Dijkstra Algorithm

Require: Weighted graph $G = (V, E)$, source vertex T, target vertex T.

Ensure: The shortest path from S to T, or judge no path exists from S to T.

```
1: function DIJKSTRA(G, s, t)
2:     Q ← empty priority queue
3:     for all v ∈ G do
4:         dist(v) ← ∞
5:     end for
6:     dist(s) ← 0
7:     Q.insert-key(s, 0)
8:     while Q is not empty do
9:         v ← Q.extract-min()
10:        if dist(v) == the extracted minimum distance in Q then
11:            if v == t then
12:                return dist(v)
13:            end if
14:            for all {u, v} ∈ E do
15:                if dist(u) > dist(v) + weight(u, v) then
16:                    dist(u) ← dist(v) + weight(u, v)
17:                    Q.insert-key(u, dist(u))
18:                end if
19:            end for
20:        end if
21:    end while
22:    return PathNotExist
23: end function
```

Figure 11: Pseudocode of Dijkstra algorithm

If we employ priority_queue in-built in C++ to implement heap optimization, every vertex and edge will be visited once. The time complexity to pop each vertex is $O(\log n)$ and the time complexity to

update the corresponding vertex's value by each edge is $O(\log n)$ as well, so the overall time complexity of each shortest path query is $O((n+m)\log n)$.

3.2.2 Bi-directional Dijkstra Algorithm

Applying the same idea used in dealing with unweighted shortest path, we get the bi-directional Dijkstra Algorithm.

(1) Algorithm Description

We update the vertices with the shortest distance from the corresponding source in two directions respectively. Unlike the situation in unweighted graphs, when a vertex is visited twice, we cannot take the sum of shortest path from the source and from the target as the result. For example, in the figure below, v_1 is visited by s and t respectively. We can see that its sum of distance from two vertices is 200, but obviously the shortest path is 150 instead of 200.

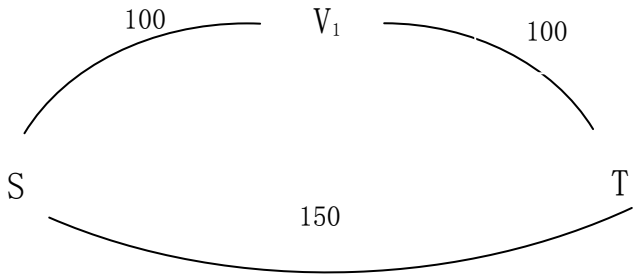


Figure 12: A counterexample

If the running program does not stop until all of the vertices are visited, which means lots of vertices need to be visited during the process, the advantage of bi-directional Dijkstra algorithm can't be fully demonstrated. This time, we can keep on updating the shortest path *x* from *S* to *T* based on the current situation. For the Dijkstra process starting from *S*, when we update the distance from vertex *S* to vertex *v* with the distance from *S* to *u*, we can update *x* with *dist*(*S*,*v*)+*dist*(*T*,*v*). The process of computing distance from *T* can be executed in the same way.

Theorem 1: Denote the weight of the top elements of the two heaps starting from s and t as y_1 and y_2, respectively. When $x \leqslant y_1+y_2$, *x* is the distance of the shortest path.

Proof. Use reduction to absurdity. If there exists a $S-T$ path whose weight is less than x, there must exist two vertices u, v on the path which make$\text{dist}(S,u) < y_1, \text{dist}(T,v) < y_2$, and there must exist a group u, v which makes an edge exist between u, v.

Assume vertices passed by the paths whose weights are less than *x* as $p_1, p_2, \cdots, p_k$

Apparently, eligible u, v respectively correspond to a sequence of consecutive index set $S_u = [1, i], S_v = [j, k]$.

If $i+1 < j$, at least one vertex is not in the path. From the fact that $i+1$ and $j-1$ do not belong to S_u, S_v respectively, we can easily deduce that the distance of the path is not less than x.

If $+1 \geq$ j , the adjacent edge $\{u, v\}$ has been updated before, so the distance of the path is not less than x.■

(2) Experiments on Dijkstra Algorithm and Bi-directional Dijkstra Algorithm

As all of the storing structure of each query in this section is identical, we only need to consider the datasets of 50% queries. Because the running speed of weighted query is relatively slow, let $q = 10^5$ in this section.The contrast of running time between Dijkstra and bi-directional Dijkstra Algorithm is illustrated below:

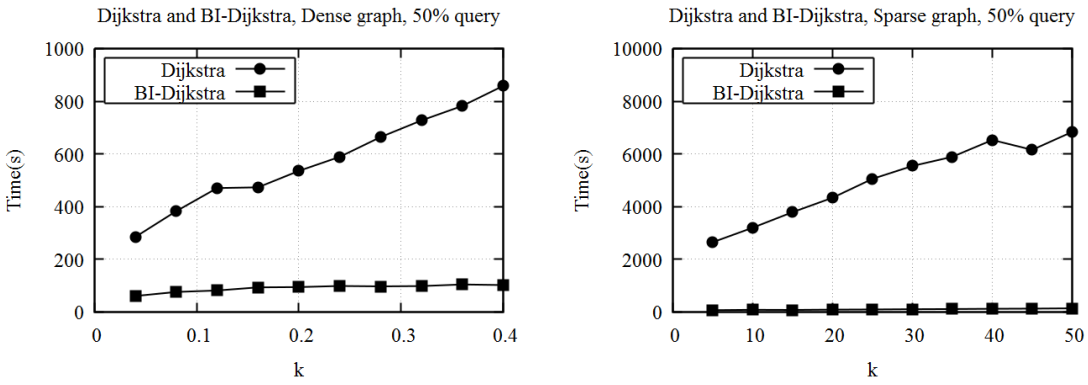


Figure 13: Running time of Dijkstra and Bi-Dijkstra algorithms on different datasets

	Average running time of sparse graph (s)	**Average running time of dense graph (s)**
Dijkstra	576	4995
Bi-directional Dijkstra	90	97

Figure 14: Running time of Dijkstra and Bi-Dijkstra algorithms in sparse and dense graphs

We can conclude that theorem 1 is a very efficient pruning algorithm, which drastically reduces calculations.

3.2.3 SPFA Algorithm

(1) Algorithm Description

The essence of SPFA (Shortest Path Faster Algorithm) is an optimization based on Bellman-Ford algorithm. It is a highly-efficient shortest path algorithm.

(2) Pseudocode

Algorithm 4 SPFA Algorithm

Require: Weighted graph $G = (V, E)$, source vertex S, target vertex T.

Ensure: The shortest path from S to T, or judge no path exists from S to T.

```
1: function SPFA(G, s, t)
2:     S ← ∅
3:     Q ← empty queue
4:     S ← S ∪ {s}
5:     Q.push(s)
6:     dist(s) ← 0
7:     while Q is not empty do
8:         v ← Q.pop-front()
9:         S ← S − {v}
10:        for all u, {u, v} ∈ E do
11:            dist(u) ← dist(v) + 1
12:            if u ∉ S then
13:                S ← S ∪ {u}
14:                Q.push(u)
15:            end if
16:        end for
17:    end while
18:    return PathNotExist
19: end function
```

Figure 15:Pseudocode of SPFA algorithm

Please refer to *Reference*[13] for the proof of the correctness of the algorithm.

(3) Two Optimizations of SPFA Algorithm

Optimization 1(Small Label First, SLF): In the process of inserting queue, if the shortest distance to the vertex is less than the shortest distance to the head of queue, insert the vertex into the head of queue, otherwise insert it into the tail of the queue.

Optimization 2(Last Label Last, LLL): When pop out a vertex, if the shortest distance of the vertex is bigger than the average of shortest distances of all vertices in the queue, then insert the vertex into the tail of queue and pop out a vertex over again.

(4) Experimental Results

Data collected from weighted graph programs is listed below:

Data No.	n	m	q	k	Query Ratio (%)	Dijkstra	Bi-Dijkstra	SPFA	SPFA-SLF	SPFA-LLL	SPFA-SLF-LLL
1	4000	640000	100000	0.040	50	284.631	60.377	379.526	292.717	407.626	267.584
2	4000	1280000	100000	0.080	50	382.557	75.602	576.840	445.347	660.440	411.162
3	4000	1920000	100000	0.120	50	470.058	81.217	701.704	588.956	858.164	587.615
4	4000	2560000	100000	0.160	50	472.879	93.012	859.220	740.017	975.486	703.670
5	4000	3200000	100000	0.200	50	535.369	94.117	1013.756	912.426	1180.167	901.376
6	4000	3840000	100000	0.240	50	588.646	98.317	1113.980	996.647	1323.778	1060.092
7	4000	4480000	100000	0.280	50	663.819	96.863	1231.640	1140.247	1503.447	1065.443
8	4000	5120000	100000	0.320	50	726.926	98.015	1403.671	1249.024	1673.652	1197.554
9	4000	5760000	100000	0.360	50	781.977	104.057	1516.220	1433.134	1925.385	1493.578
10	4000	6400000	100000	0.400	50	858.134	101.728	1740.966	1543.531	2063.343	1411.621
11	100000	500000	100000	5.000	50	2645.790	63.489	2562.990	1786.964	2824.041	1710.550
12	100000	1000000	100000	10.000	50	3196.648	78.179	4308.069	2950.647	4264.452	2938.073
13	100000	1500000	100000	15.000	50	3786.417	73.435	5140.185	3658.214	5104.661	3651.835
14	100000	2000000	100000	20.000	50	4333.667	84.306	5952.570	4237.729	5805.614	4625.841
15	100000	2500000	100000	25.000	50	5051.126	91.750	6047.001	4710.522	6334.118	5242.813
16	100000	3000000	100000	30.000	50	5540.735	100.670	6504.154	5215.255	7331.015	5580.581
17	100000	3500000	100000	35.000	50	5889.195	106.751	7444.714	5819.885	8365.950	6205.657
18	100000	4000000	100000	40.000	50	6525.290	116.856	7311.508	5900.586	8026.262	5987.309
19	100000	4500000	100000	45.000	50	6152.225	123.070	8401.937	6601.891	11273.654	6827.370
20	100000	5000000	100000	50.000	50	6832.847	134.019	8878.453	7201.036	9439.362	7302.294

Figure 16: Data collected from weighted graph programs

(5) Experiments on SPFA Algorithm and its Optimization

Running time of optimizations of SPFA algorithm in dense& sparse graphs is shown below:

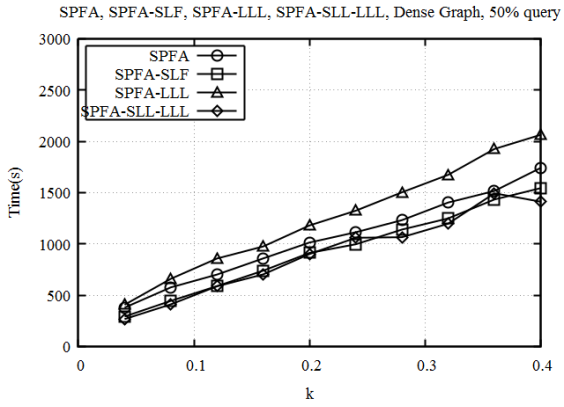


Figure 17: Running time of optimizations of SPFA algorithm in dense graphs

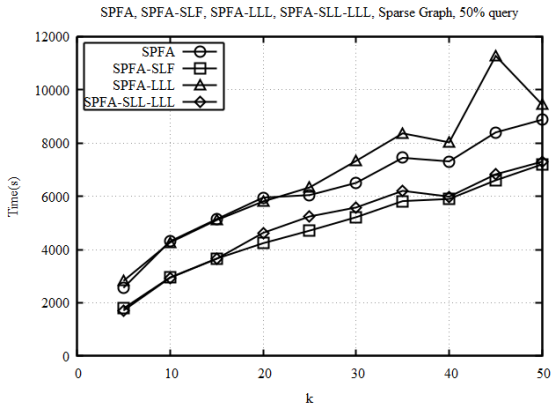


Figure 18: Running time of optimizations of SPFA algorithm in sparse graphs

The comparison on overall running time of SPFA algorithm, its different optimizations, and Dijkstra algorithm is listed below:

	Average running time in sparse graph(s)	**Average running time in dense graph(s)**
Dijkstra	576	4995
SPFA	1053	6255
SPFA+SLF	934	4808
SPFA+LLL	1257	6876
SPFA+SLF+LLL	909	5007

Figure 19: Average running time of different algorithms in sparse and dense graphs

From the table above, we can find out that the optimization of SLF can achieve 12% increase in efficiency; plus the optimization of LLL, the efficiency can be increased by 16%. But the optimization of LLL alone is not so good and its efficiency performance is also not stable.

Although the efficiency of SPFA algorithm is much lower compared with that of bi-directional Dijkstra algorithm, SPFA algorithm has already improved a lot in comparison with the efficiency of frequently-used Bellman-Ford Algorithm. However, there are two limitations in SPFA. Firstly, it cannot be used in certain graphs, such as dense grid graphs. In that case, its complexity will degrade to $O(n^2)$. Secondly, the time complexity of SPFA algorithm has not been proved strictly. When SPFA algorithm is applied, its running time cannot be estimated accurately and its efficiency changes greatly with various patterns and densities of graphs. For the reasons given above, we do not recommend the application of SPFA algorithm.

4. Dynamic Shortest Paths on the Map

4.1 Problem Description

We have already discussed the unweighted graph shortest path problem and weighted graph shortest path problem. The definitions of "unweighted graphs" and "weighted graphs" are so broad that their structures are uncertain and elusive. This means that just vertices and the relation among vertices can construct a graph, and almost all of the regular shortest path problem can be converted to unweighted graph problem or weighted graph problem. On the one hand it shows that our algorithms can be applied widely, on the other hand it means that our algorithms are lack of pertinence, which makes further optimization difficult. When encountering a practical shortest path problem, we will find lots of characteristics, such as structure of graph, type of query, graph features, etc. Only after mastering the essences of the characteristics we can do better in optimization.

Next we will discuss the application on the map. Because many shortest path problems are related to path finding in two-dimensions or can be abstracted to it, we focus on the shortest paths on the map. Please note that the map here does not necessarily refer to a real map such as a city map, a similar structure will be ok.

In order to confirm the similarity between a graph and a map, we list two features. We define them as **map properties**:

① every vertex in the graph will have another property, namely position. This position corresponds to a point in coordinate system, and the distance between positions is defined as the distance between corresponding points. The positions of different points should be different. In practice, we can delete redundant positions by merging points of the same position.

② an edge (*u,v,d*), setting *d'* as the distance of position *u* between position v, the closer d to d' is, the more similar to a map the graph is. Strictly speaking, suppose that the maximum of *d/d'* is r, the minimum of *d/d'* is l, Then r/l can indicate the degree to which current graph is similar to a map.

Next we put the graph into the Cartesian coordinate system, in order to study the graph by the means of studying a map. Considering a dynamic graph, we only study how to inquire a group of shortest path queries. In this section, we denote the Source as *S*, and Target as *T*.

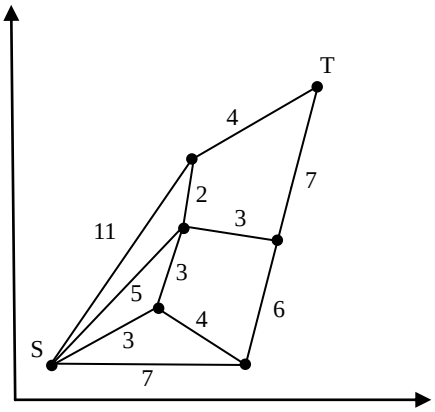


Figure 20: Put a graph into plane coordinate system

4.2 Proposed Methods

4.2.1 Background of Shortest Path Algorithms

(1) Algorithm Description

Searching a shortest path on map by all the algorithms discussed so far is somewhat aimless, because target is 'unknown' and you do not know whether the paths you choose can reach the target quickly. But when applied on a map, it is utterly different. Observing the graph above, you can easily find that the shortest path does not deviate target too much. The path we search should try the best to aim at the target. If it goes in the opposite direction, you can drop the path. Detour may occur in the shortest path, but it will not detour too much. But if we just aim at the direction of target, the path we found may not be the best one in most circumstances. Sometimes it even goes to impasse.

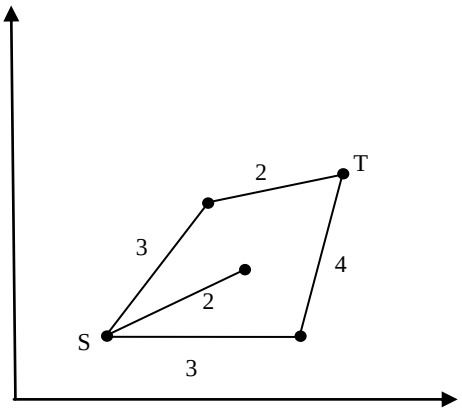


Figure 21: The example of impasse

Next we introduce A* algorithm brought forth in 1968.It makes use of the concept of estimation function. Vertex *v* in estimation function *h*(*v*) corresponds to a real number, which indicates the estimated shortest path from vertex *v* to the target. With estimation function, we can estimate the shortest path from vertex *v* to the target. Furthermore, we can get whether the path of target v is worth searching for. When our searching path is deviating from target, estimation function will increase, then we can see how far we detour.

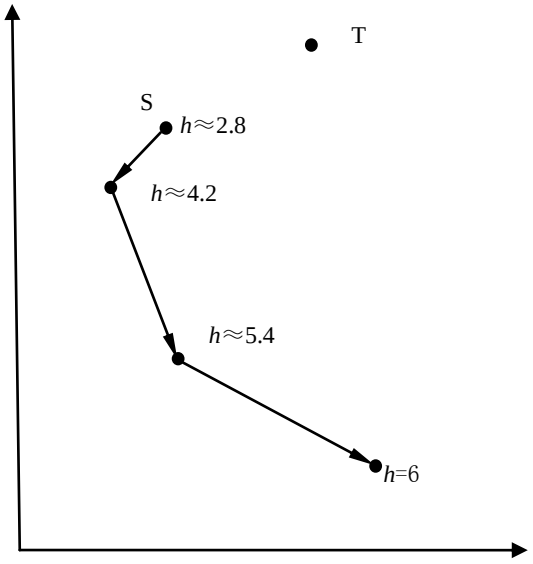


Figure 22: The indications of estimation function and detour

A* algorithm is a mixture of Dijkstra algorithm and estimation function. Dijkstra algorithm will keep on updating currently found shortest path. Now we have the estimation function. Let function $f(v)=h(v)+dis(v)$ (dis *here* is $dis(S,v)$). When choosing the vertex to update in Dijkstra algorithm, we choose the minimal $f(v)$ instead of the minimal $dis(v)$. Other steps are the same as Dijkstra algorithm.

(2) Pseudo code

The pseudo code of A* algorithm is listed below:

Algorithm 5 A* Algorithm

Input: Weighted Graph G=(V,E), Source vertex S, Target vertex T

Output: The shortest path from S to T, or indicate that no path exists from S to T

Algorithm 5 A* Algorithm

Require: Weighted Graph $G = (V, E)$, source vertex S, target vertex T.

Ensure: The shortest path from S to T, or judge no path exists from S to T.

```
1: function ASTAR(G, s, t)
2:     Q ← empty priority queue
3:     for all v ∈ G do
4:         dist(v) ← ∞
5:     end for
6:     dist(s) ← 0
7:     Q.insert-key(s, h(s))
8:     while Q is not empty do
9:         v ← Q.extract-min()
10:        if dist(v) == the extracted minimum distance in Q then
11:            if v == t then
12:                return dist(v)
13:            end if
14:            for all {u, v} ∈ E do
15:                if dist(u) > dist(v) + weight(u, v) then
16:                    dist(u) ← dist(v) + weight(u, v)
17:                    Q.insert-key(u, dist(u) + h(s))
18:                end if
19:            end for
20:        end if
21:    end while
22:    return PathNotExist
23: end function
```

Figure 23: A* algorithm

(3) A discussion on related algorithms

How can A* algorithm find the shortest path? It is easy to find that if $h(v)$=0, then A* algorithm is the same as Dijkstra algorithm. In other words, Dijkstra algorithm is a special case of A* algorithm. How about the estimation functions which are not zero?

We will discuss the correctness of A* algorithm by creating a new graph. Every step we take the

minimal $f(v)$ to update, so we can modify edge weights to make $f(v)$ become the distance. We need to modify the quondam weight of edges. For an edge(u,v,d), assume the path distance which visited the edge before is $dis0$,then the path distance will become $dis0+d$ after visiting the edge. While the distance in the new graph will in fact change from$dis0+h(u)$ to$dis0+d+h(v)$,so we need to subtract the weight of edge $h(u)$ and add $h(v)$. Then in the new graph, $dis0+h(u)$ becomes $dis0+d+h(v)$. Namely $d'=d-h(u)+h(v)$. On that condition, former A* algorithm is equivalent to Dijkstra algorithm in the new graph.

Now we study the relationship between the distance in the new graph and that in the former graph. Suppose path P corresponds to path P' in the new graph, and its size is n. The distance in the former graph is $dis(P)=\sum d_i$ (d_i stands for the distance of i-th edge, and the same as below), and the distance in the new graph is $dis(P')=\sum d'_i$. Because $d'_i=d_i-h(u_i)+h(v_i)$ (u_i,v_i stands for the starting and ending points of the i-th edge), $dis(P')=\sum (d_i - h(u_i) + h(v_i))$

$=d_1-h(u_1)+h(v_1)+d_2-h(u_2)+h(v_2)+\ldots+d_n-h(u_n)+h(v_n)=d_1+d_2+..+d_n-h(u_1)+h(v_n)$. (the target of the previous path is the source of the next path). Then $dis(P')=dis(P)-h(st(P))+h(ed(P))$. So computing the shortest path from S to T in the new graph is equivalent to finding the shortest path in the former graph plus a constant $-h(S)+h(T)$. Since $h(T)=0$, the shortest path in the new graph is obtained by just deducting an additional $h(S)$. Then the shortest path in the new graph is almost equivalent to that in the former graph. The difference is just a constant.

We can see that the correctness of A* algorithm depends on the correctness of Dijkstra algorithm in the new graph. But Dijkstra algorithm in the new graph is not always correct, because the weight of edge$d-h(u)+h(v)$ in the new graph may be a negative value. As mentioned earlier, the correctness of Dijkstra depends on the consecutive increase of distance of the shortest path. If there are negative weights of edge in the shortest path, the given shortest path by Dijkstra algorithm may be in question. Since Dijkstra algorithm in the new graph is equivalent to A* algorithm, we can draw a conclusion on the correctness of A* algorithm.

Theorem 2: Let the estimation function be h, A* algorithm can work out the shortest path if and only if for each edge (u,v,d),$h(u)<=h(v)+d$. (We call this correctness inequation)

Proof. When $h(u)<=h(v)+d$, $h(v)+d-h(u)=d'>=0$and there exist no negative weighted edges in the new graph. A* algorithm can work out the shortest path. Contrariwise, if there are negative weighted edges in the new graph, the algorithm does not guarantee to work out the shortest path. ■

Someone may think that when estimation function is less than practical distance, A* algorithm is guaranteed to find the shortest path. But there are some counterexamples for that.

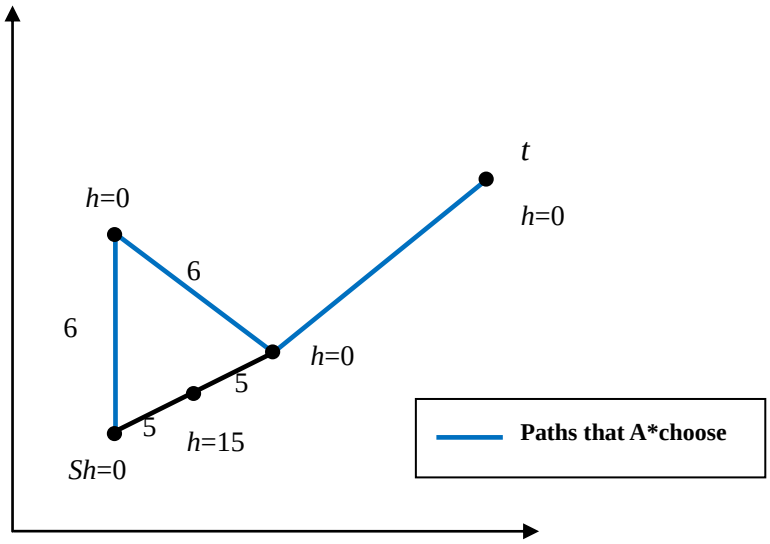


Figure 24: A counterexample

In fact, the equation above is similar to the condition of the shortest path *dis*(*u*)<=*dis*(*v*)+*d*. Its estimation function can be treated as one whose source is T and whose direction of edge is the reverse of the *dis* starting from S. It shows that estimation function is related to yet different from the real shortest path. The shortest path must make the inequation maximum, but not for estimation function. Under extreme condition, if estimation function h is maximum when meeting the requirement of correctness inequation, then h(v) is exactly the shortest distance *dis*(*v*,*T*) from v to T. At that time A* algorithm will only traverse the shortest path (if there are more than one, it will traverse all of them). Under another extreme condition when *h*(*v*)=0, A* algorithm will degrade to Dijkstra algorithm. We can see that choosing the right h function can reduce the number of traversing vertices. Moreover, meeting the requirement of correctness inequation, the more h function is (namely the more accurately you estimate), the less the number of vertices A* will traverse and the more efficient the algorithm will be.

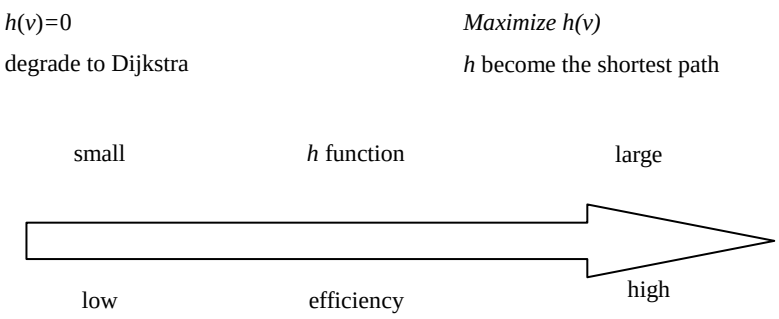


Figure 25: Relationship between *h* and *h*(*v*)

Figure 26 illustrates the efficiency of A* algorithm as compared to Dijkstra algorithm.

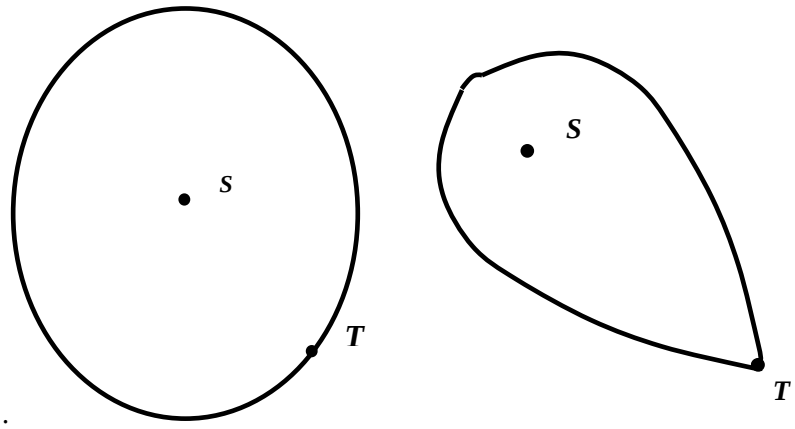


Figure 26: The efficiency of A* algorithm as compared to Dijkstra algorithm

If $h(u)>h(v)+d$ occurs in the edge of (u,v,d), then the correctness will not be assured and the efficiency is also difficult to analyze. But we can see that although the shortest path is not sure to be found, the number of vertices visited is indeed decreased. Since Dijkstra is inclined to visit negative weight edge, the efficiency of algorithm has important relationships with the negative weight edge in the new graph which is caused by *h* function. Under certain conditions that *h* function is relatively easy, we can still analyze the efficiency of the algorithm.

Since the number of calling *h* function is at least as many as that of visited vertices, the efficiency of computing *h* function is also very important. When *h*is maxima, we can compute *h* function only after the shortest path starting from *T*(namely inverse graph) is found. It is utterly unnecessary to do that, because it needs to solve another shortest path problem which is exactly the same as our originally problem. Obviously it is the easiest and the most useful to adopt Euclidean distance. Since the edges have weights, we can let $h(v)$ be l*the distance from v to T (the definition of the distance of positions and l can be referred in the property of map) . Because the line distance between two points is the shortest and the constant l of edge is the minimal of the interval (see the definition of map property), it meets the requirement of correctness inequation. But sometimes the distance calculated by estimation function deviates from the real distance too much, in this case we can increase estimation function. It does not meet the requirement of correctness inequation, nevertheless the efficiency of algorithm may increase. This treatment is helpful to those who do not need the shortest path but a relatively short one. Of course, there exist other algorithms with pre-calculation, but we do not discuss them because we are dealing with dynamic graphs.

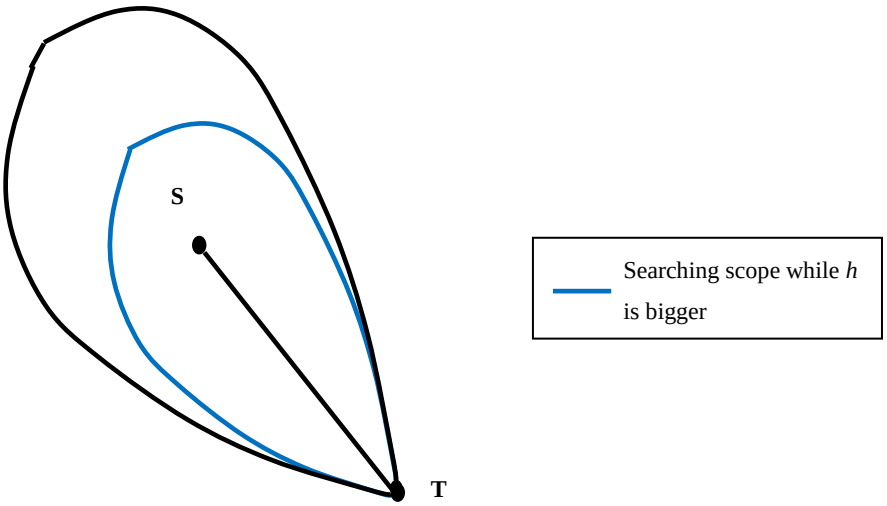


Figure 27: Relationship between Searching scope and *h*

We have discussed above that if *h* function does not meet the requirement of correctness inequation, it is difficult to analyze its efficiency. But if *h* function is taken as the multiple of Euclidean distance from that vertex to the target, we can easily observe the relationship between the scale and efficiency of *h* function. The more the *h* function is, the larger the weight of the h function becomes. Therefore it reduces the number of visited vertices and increases the algorithm efficiency. So in some cases, we can sacrifice a little correctness for efficiency. The relationship between *h* function and the algorithm efficiency will be further discussed in another section of this chapter.

Notice that *h* function here is just for maps concerning weight based on Euclidean distance. Some other *h* functions, such as Manhattan distance, Chebyshev distance, which will run better in some specific cases.

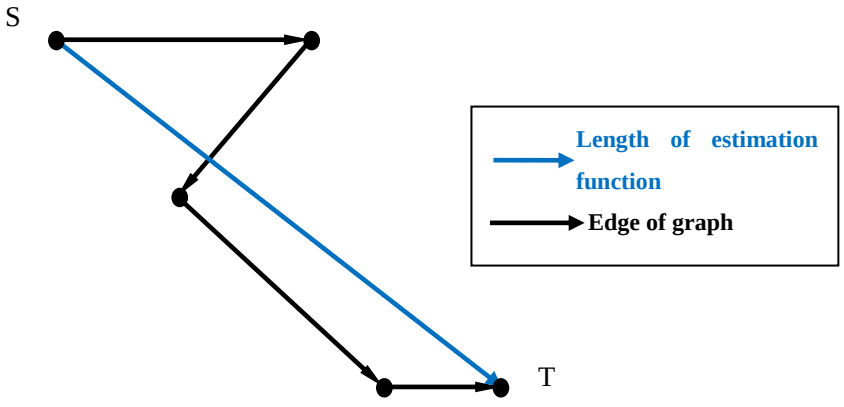


Figure 28: convergence

4.2.2 Bi-directional A* algorithm

Can A* algorithm be applied to solving the shortest path problem on weighted directed graph? Because of no map properties, it is difficult for us to locate every position. Even so, the region in

map property 2 will be too big to optimize.

Can A* algorithm be bi-directional? The answer is yes. Because the nature of A* algorithm is Dijkstra algorithm, if we can construct a new graph, we will certainly apply bi-directional Dijkstra algorithm in that graph in theory. To some extent it is a class of bi-directional A* algorithm as well, and its correctness and efficiency utterly depends on bi-directional Dijkstra algorithm.

We cannot employ the method of bi-directional in A* algorithm as that of Dijkstra algorithm. The reason is that the estimation function of forward A* is different from that of reverse A*, namely their constructed new graphs are different from each other too. If that, the correctness cannot be guaranteed, and the case finding a shorter path while jumping off the shortest path may even happen.

In summary, A* algorithm optimizes the general Dijkstra algorithm by the map properties. It usually traverses less vertices and has the same complexity even in the worst case. Because the essence of A* algorithm is the same as that of Dijkstra, its complexity with heap optimization is still $O(|V|\log|V|+|E|)$. The A* algorithm meeting the requirement of correctness inequation can still solve the shortest path problem.

4.2.3 Restricted Path Finding Algorithm

The algorithm below, to some extent, is an optimization. It is based on a very general observation that the shortest path of two vertices on a map is mainly restricted in a region. In other words, the shortest path of two vertices on a map will not deviate from the l segment between S and T too much. We just select a region whose size is related to the size of map, we can ignore the vertices and edges outside the region and solve the problem in a subgraph. Though this procedure will sacrifice some correctness, it indeed reduces the number of visited vertices. Even for A* algorithm, it will still visit many unnecessary vertices.

We define **restricted region** is a vertex set which consists of S, T and other vertices. Restricted path finding algorithm is based on one shortest path algorithm and excludes the vertices not in the restricted region. Take Dijkstra algorithm as (same with A* algorithm) an example, if current vertex is not in the restricted region, then skip the vertex.

We have some different restricted regions:

①The restricted region contains vertices in the outer rectangle. The inner rectangle contains S and T as its corners.

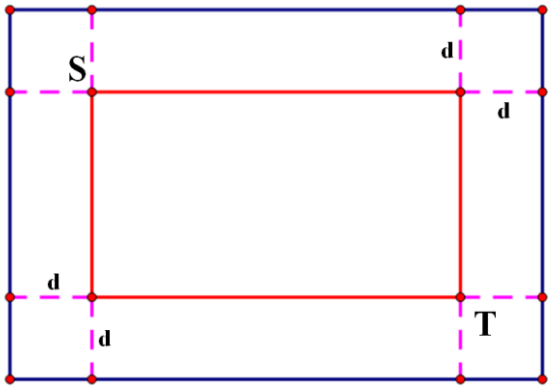


Figure 29: The restricted region contains vertices in the outer rectangle

②The restricted region contains the vertices whose distance to the segment between S and T is less than or equal to a certain distance *d*.

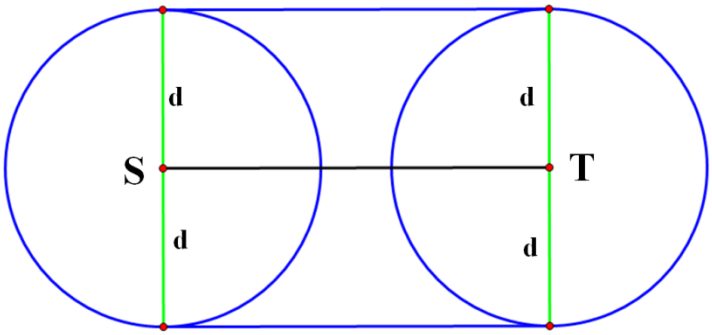


Figure 30: Distance to the segment between S and T is less than or equal to a certain distance *d*

③The restricted region contains vertices whose sum of distance to S and T is less than and equal to a certain distance *d*.

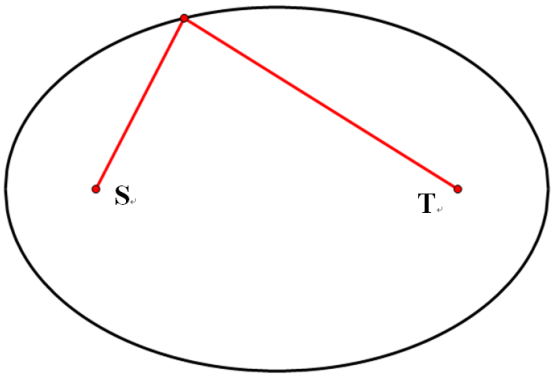


Figure 31: Sum of distance to S and T is less than and equal to a certain distance *d.*

Note: the sum of distances between the two red lines is d. This figure is an ellipse

These restricted regions have different optimization effect. For example, the decision criterion used for region 1 is very simple, but the region is large for us. Obviously, the other two corners in the rectangle except corner S and corner T can shrink the region. The judges of region 2 and region 3 are not so easy, but their region is more appropriate. If the edge weight is based on Manhattan distance, we should select region 1, because region 2 and region 3 are adapted to optimization of Euclidian

distance. We should select different d from different graphs, different requirement of correctness and efficiency. In the implementation and analysis of algorithms, we will analyze the relationship between the correctness and efficiency.

Though the region of restricted path finding sacrifices some correctness, it does improve the efficiency. Most of all, it can speed up efficiency according to the requirement. One can always find a relatively shorter path by keeping on expanding the search region until the shortest one is found at last. It is impossible for some other algorithms to do so, because they can only take a long time to find the shortest path before knowing any path from *S* to *T*.

4.3 Experiments on Algorithm Comparisons

Next we will implement different A* algorithms and test their practical performance.

4.3.1 Datasets

Three kinds of graphs are used in the test:

①Sparse map. Coordinates are generated randomly in [0,100000). Edges are generated randomly between 2 vertices whose distance is less than 2**sqrt*(100000), and weight of edge is randomly chosen that is 1~3 times of distance.

②Relatively dense map. The number of vertices/the whole area=1%. Any other conditions are the same as ①except the region of coordinates. Dense here refers to that the number of vertices is relatively dense compared with area.

③Sparse map. The weight of edge is equal to distance. Other conditions are the same as ①.

There are 100,000 vertices, 400,000 edges and 10,000commandsin all 3 kinds of map above. Among commands, $\text{adding edges}:\text{deleting edges}:\text{query} = 1:1:2$.

All standard answers are given by the trival Dijkstra algorithm.

4.3.2 Implementation Details

We have implemented the following algorithms:

①Dijkstra algorithm

②A* algorithm which satisfies the correctness inequation

③A* algorithm which does not satisfy the correctness inequation

④Restricted path finding algorithm (Region 1)

⑤Restricted path finding algorithm (Region 2)

⑥Restricted path finding algorithm (Region 3)

Restricted path finding algorithms have different parameters .The selection of d is related with the average edge weight.

4.3.3 Evaluation Criteria

Because approximation is used in the algorithms, we assess the performance of each test point in each program by 3 different criteria: running time, the rate of correctness and accuracy.

Running time: the whole time from the moment before the program begins to the moment the program ends. Its unit is millisecond(ms). The running time includes input time of reading file (We use 'fread' to accelerate the speed of reading file in all of the programs. The time is about 60 ms, which has little effect on running time.)
Rate of correctness: the percentage of output correct answers of program to the whole answers. Its unit is %.

Accuracy: for the output answers of program ans0 and standard answers ans0, if standard answer finds the shortest path while program not, then the score of that point is -1, otherwise the score is $\frac{|ans0-ans|}{max(ans0,ans)}$. Accuracy is the average of score of all of answers.

4.3.4 Test Environment

CPU: Intel® Core(TM) i5-5250U cpu@1.60GHz 1.60 GHz
RAM: 4GB
OS: 64 bits x64 CPU

4.3.5 Compile Option

g++ -o run run.cpp -std=c++11 -O2 -Wl,--stack=268435456

(-Wl,--stack=268435456 stands for enough stack space)

4.3.6 Test Method

Batch test uses a program written by us (see attached file: batchtestforall.cpp). The whole test time is 3634.212 s.

Note: Bi-directional Dijkstra is tested alone and excluded from the whole time.

4.3.7 Practical Running Status

Figure 32 is the contrast of algorithm types.

sqr	Region 1（rectangle）
el	Region 2（ellipse）
lim	Region 3
ma	Estimation function is based on Manhattan distance

Figure 32: The contrast of algorithm types

Three parameters in parentheses of algorithm types respectively stand for algorithm type, times of estimation function, parameter of restricted routing *d*.

All performance of implemented algorithms are listed below: (Because the table is too big, the picture is divided into 2 parts.)

	Sparse graph.weight1~3			**Dense graph.weight1~3**		
	Running time/ms	Correctness %	Accuracy %	Running time/ms	Correctness %	Accuracy %
A*(- 1 -)	160466	100	100	155815	100	100
A*(- 1.5 -)	64595	6.90144	99.636	65352	12.6444	99.687
A*(- 2 -)	4040	0.38454	94.017	3375	0.72948	94.092
A*(- 2.5 -)	2957	0.22263	88.337	2589	0.34448	88.281
A*(- 3 -)	2875	0.20239	84.641	2341	0.18237	84.512
A*(lim 1.5 6)	19315	5.60615	98.805	19983	9.78723	99.105
A*(lim 1.5 7)	21774	6.23356	99.076	22966	10.6788	99.313
A*(lim 1.5 8)	24795	6.37523	99.253	25954	11.3273	99.441
A*(el 1 4)	41489	95.0617	99.892	42525	97.9129	99.992
A*(el 1 6)	50931	99.2916	99.957	52008	99.8582	100
A*(el 1 8)	59874	99.8583	100	59824	100	100
A*(el 1 10)	66508	100	100	69672	100	100
A*(lim 1 6)	18734	36.2882	99.154	19775	43.7893	99.405
A*(lim 1 8)	24645	57.3366	99.608	25908	65.1672	99.748
A*(lim 1 10)	30932	73.6491	99.805	33437	80.8916	99.894
A*(lim 1 12)	37226	84.0113	99.901	40515	90.1722	99.954
A*(lim 1 18)	57088	97.5106	99.988	59380	98.8652	99.997
A*(lim 1 24)	76391	99.6964	99.999	80670	99.9392	100
A*(sqr 1 6)	87543	93.6652	99.946	88558	95.2786	99.967
A*(sqr 1 10)	99052	98.3404	99.99	100126	99.0881	99.996
A*(ma 1 -)	111379	18.0328	99.853	112554	27.4772	99.864
A*(ma 1.5 -)	19430	0.38454	92.598	18689	0.68896	92.85
A*(ma 2 -)	4172	0.16191	83.947	3607	0.28369	84.019
Dijkstra	308274	100	100	305093	100	100
Bi-directional Dijkstra	80072	100	100	80326	100	100

Figure 33: All performance of implemented algorithms part 1

	Sparse graph. weight 1		
	Running time/ms	Correctness %	Accuracy%
A*(- 1 -)	52912	100	100
A*(- 1.5 -)	2558	0.34205	94.575
A*(- 2 -)	2352	0.26157	91.857
A*(- 2.5 -)	2374	0.26157	90.43
A*(- 3 -)	2337	0.22133	89.504
A*(lim 1.5 6)	2072	0.34205	94.524
A*(lim 1.5 7)	2102	0.34205	94.552
A*(lim 1.5 8)	2037	0.34205	94.555
A*(el 1 4)	37495	99.8592	99.839
A*(el 1 6)	43619	100	100
A*(el 1 8)	47836	100	100
A*(el 1 10)	50612	100	100
A*(lim 1 6)	17107	60.5835	99.819
A*(lim 1 8)	21803	80.8249	99.94
A*(lim 1 10)	26689	91.7907	99.98
A*(lim 1 12)	31818	96.7002	99.993
A*(lim 1 18)	40981	99.9396	100
A*(lim 1 24)	48586	100	100
A*(sqr 1 6)	47365	98.169	99.994
A*(sqr 1 10)	50928	99.7787	99.999
A*(ma 1 -)	12721	0.4829	93.875
A*(ma 1.5 -)	2829	0.28169	87.834
A*(ma 2 -)	2712	0.24145	86.061
Dijkstra	277166	100	100
Bi-directional Dijkstra	69637	100	100

Figure 34: All performance of implemented algorithms part 2

4.3.8 Result Analysis

We analyze correctness guaranteed algorithms: A*(- 1 -), Dijkstra algorithm, and bi-directional Dijkstra algorithm. (The following table omits correctness and accuracy.)

	MAP 1	**MAP 2**	**MAP 3**
A*(- 1 -)	160466	155815	52912
Dijkstra	308274	305093	277166
Bi-directional Dijkstra	80072	80326	69637

Figure 35: Time-costing of different algorithm in different maps

On map 1 and map 2, A* algorithm runs almost twice as fast as the Dijkstra, while bi-directional

runs the fastest. On map 3 of the same size, A* algorithm runs the fastest. The reason is that the weight of map 1 and map 2 is 1~3 times of distance, but estimation function only select 1 time in order to keep correctness, which slows down its speed. But the weight of map 3 equal to distance, the selection of estimation is comparatively appropriate, which reduces running time greatly.

We also observe that although el type (and A *(lim 1 24)) of A* algorithm does not guarantee correctness, they get 100% on the rate of correctness. Especially, A*(1 el 10) algorithm on all three maps get 100% on the rate of correctness and are more efficient than the above mentioned three algorithms .

	Sparse graph.weight1~3			**Dense graph.weight1~3**			**Sparse graph.weight 1**		
	Running time/ms	Correct-ness %	Accu-racy%	Running time/ms	Correct-ness %	Accu-racy%	Running time/ms	Correct-ness%	Accu-racy%
A*(el 1 6)	50931	99.2916	99.957	52008	99.8582	100	17107	60.5835	99.819
A*(el 1 8)	59874	99.8583	100	59824	100	100	21803	80.8249	99.94
A*(el 1 10)	66508	100	100	69672	100	100	26689	91.7907	99.98

Figure 36: Results of three algorithms

This demonstrates that if we could select appropriate parameters, restricted path finding algorithm can both optimize algorithm greatly and keep high rate of correctness.

Next we analyze the approximation algorithm.

The statistics figure for the accuracy and running time of all algorithms on map 1. The second figure is the selected one whose accuracy is about 99%. The followings are the same.

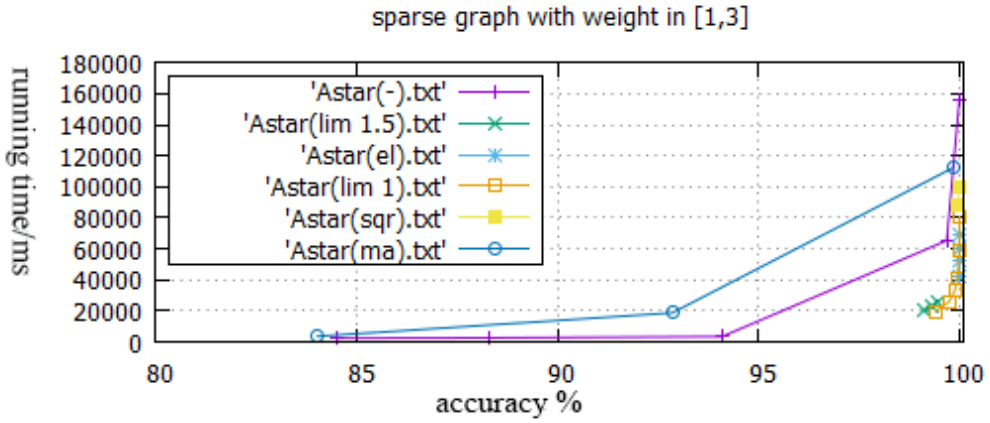


Figure 37: Accuracy and running time of all algorithms on map 1

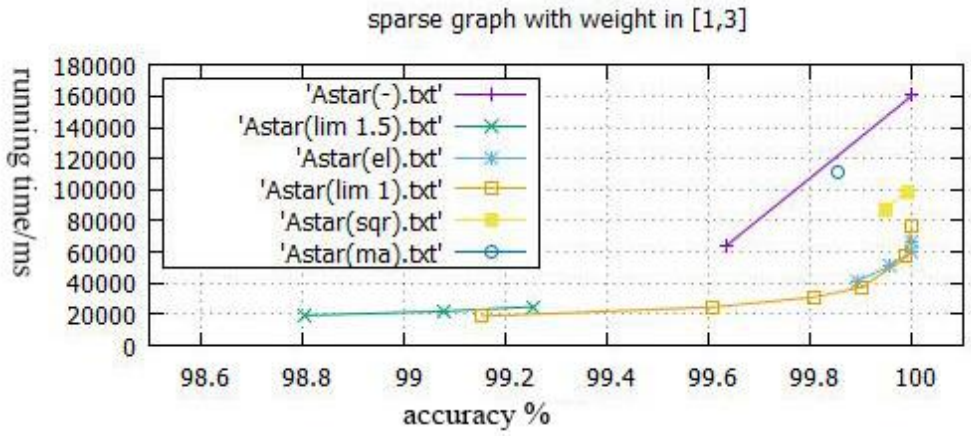


Figure 38: Selected algorithm whose accuracy is about 99% on map 1

The statistics figure for the accuracy and running time of all algorithms on map 2:

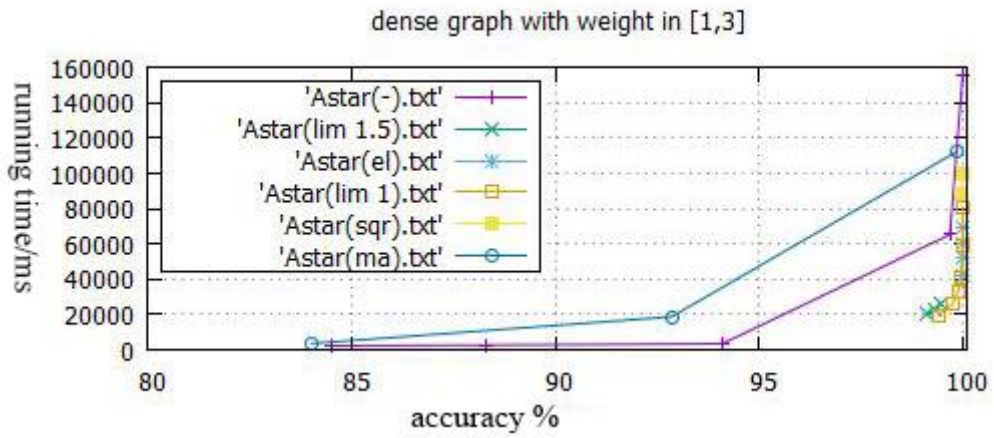


Figure 39: The statistics figure for the accuracy and running time of all algorithms on map 2

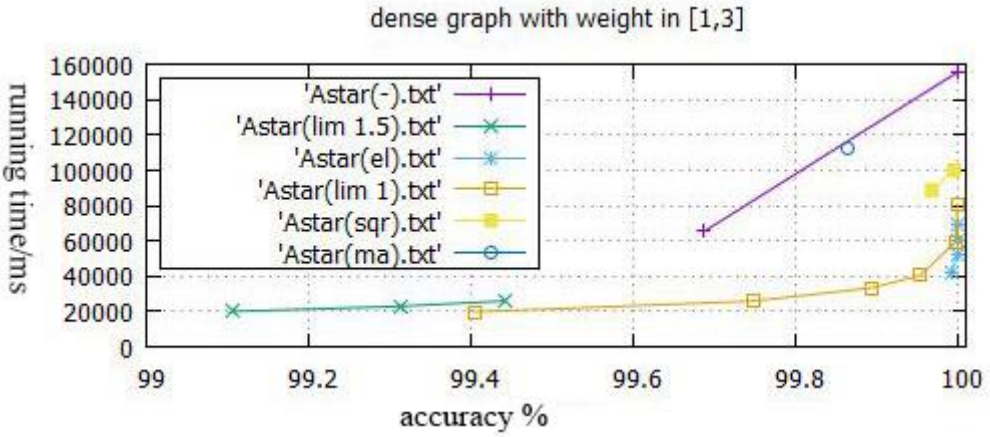


Figure 40: Selected algorithm whose accuracy is about 99% on map 2

The statistics figure for the accuracy and running time of all algorithms on map 3 sees to figure 41.

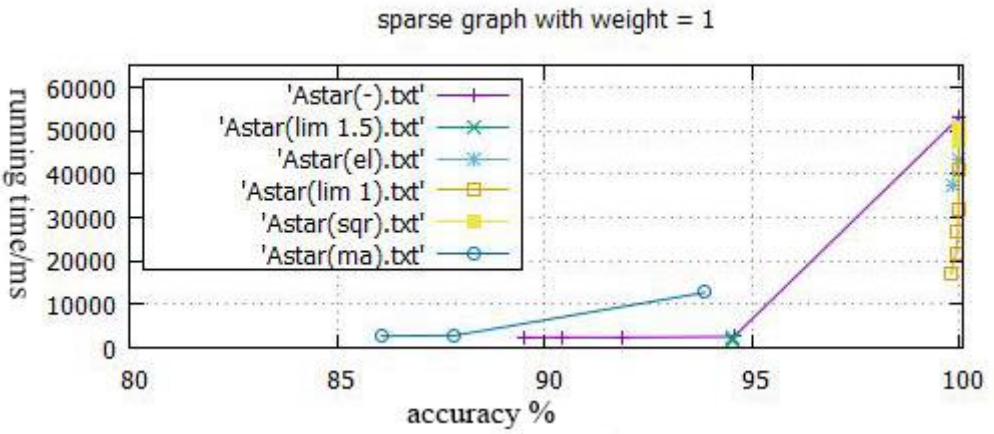


Figure 41: The statistics figure for the accuracy and running time of all algorithms on map 3

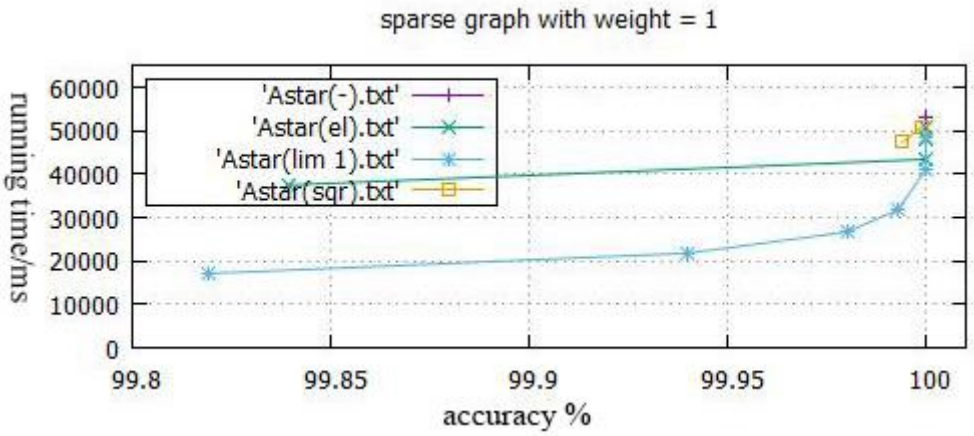


Figure 42: Selected algorithm whose accuracy is about 99% on map 3

There are five types of approximation algorithm format in this paper. There are 2 A* algorithms meeting the requirement of correctness inequation and 3 restricted path finding algorithms with different regions.

For "-"(without optimization), the A* algorithms meeting the requirement of correctness inequation is not efficient. Other A* algorithms using distinct estimation function with an accuracy above 99% work faster, but compared to other A* algorithms with optimization, it is still slow. In a whole, the efficiency of A* algorithm without other optimizations is not so good.

For "sqr" type restricted path finding A* algorithm, because rectangular region restriction is comparatively big for this kind of map, they keep the high accuracy of 99.9%. But their efficiency is quite ordinary. The running time on map 1 and map 2 is 90~100s and on map 3 is 50s. For map 1 and map 2, its efficiency is still better than that of A*(- 1 -).

For "ma" type A* algorithm not meeting correctness inequation, because Manhattan distance estimation function is not appropriate for maps whose edge weight is based on Euclidian distance,

its efficiency is not good. The efficiency of A*(ma 1 -) algorithm on map 1 and map 2 is about 99.9%. But both its accuracy and efficiency are not as good as those of "sqr" type A* algorithm. The accuracy of the others is too low for the sake of very big estimation function and their efficiency is even lower than that of "-" type A* algorithm.

For "el" restricted path finding A* algorithm, both its accuracy and efficiency are better than those of former algorithm. Its accuracy is so high as about 99.9% and its efficiency is also very good. Especially, it runs 1 time faster than "sqr" A* algorithm on map 1 and map 2. It also runs faster than the other algorithms. It shows that this kind of restricted region is rather appropriate.

For "lim" restricted path finding A* algorithm, its efficiency is very high and its accuracy is also as good as over 99%. A*(lim 16) can reach the accuracy of 99.1%, 99.4%, 99.8% in 20s. A*(lim 1 12) can even reach the accuracy of 99.9% and run more quickly than A*(el 1 4) whose accuracy failed to reach 99.9% in one term. Especially, "lim" restricted path finding A* algorithm performs excellently on map 3. A*(lim 1 8)even reached the accuracy of 99.93% in 21.8s. A*(lim 1 24) cannot perform the superior accuracy (all answers are right)of "e1"under the condition of guaranteeing high efficiency. But it is excellent considering that an approximation algorithm is used here.

Retrospectively compared with the trival Dijkstra algorithm, our optimizations reduce running time from 300s to about 20s with only less than 1% sacrifice in accuracy. The efficiency of our optimization is 15 times that of the former. We can answer 10000 queries in 20s, namely taking on average 2ms to answer a query.

In fact, it is not significant to compute those test points of low accuracy in 10s, because it is almost a greedy algorithm which finds a path directly to the target. Of course, if the practical requirement of accuracy is very low, this algorithm can also be used.

Next we analyze the different algorithm performances on three types of maps. It is easy to find that the running time of all algorithms on map 1 and map 2 is almost the same. Although the density of the two maps is different, their structures are similar. We can also find that the accuracy and the rate of correctness on map 2 are better than those of map 1. It is because that the density of map 2 is big and the weight is small.

Compared with map 1 and map 3, all algorithms (including Dijkstra algorithm itself) on map 3 are more efficient than those on map 1. It is because that map properties of map 3 is comparatively better than those of map 1, the efficiency of A* algorithm is comparatively high. Furthermore, because the shortest path on map 3 is shorter than that on map 1, the efficiency of Dijkstra algorithm is also higher. But some algorithms, such as 1.5 times estimation function or the estimation function using Manhattan distance, whose estimation function is much larger than the practical distance, are less accurate on map 3. The accuracy and efficiency of the other algorithms on map 3 have all been improved.

A summary of algorithms is listed below. "Correctness" stands for meeting correctness inequation

and “incorrectness ” for not meeting correctness inequation.

Algorithm	Advantages	Disadvantages
correctness A* -	Optimized Dijkstra	
incorrectness A* -	efficiency improved again	High estimation function easily leads to low accuracy
A* ma	Easy to compute estimation function	Not applicable to map concerning Euclidian distance
A* sqr	Easy to compute limited region, high accuracy	Region too large, not applicable to map concerning Euclidian distance
A* el	More accurate than lim, good efficiency	Less efficient than lim
A* lim	Some extent more efficient than el, Good accuracy	Some extent less efficient than el Some extent difficult to compute limited region
Bi-directional Dijkstra	Can apply to common graphs Good efficiency	Performance in maps with good map properties is worse than A*

Figure 43: A summary of algorithms

We can conclude that the better the map properties are, the better the performance of A* algorithm is.

5. Conclusions

5.1 Conclusions

In this paper, we implemented algorithms from three different types of shortest path problems. We studied the computational cost of these algorithms empirically, on large-scale datasets and showed the advantages and disadvantages of them. Our studies covered a wide range of algorithms: from plain, widely used BFS algorithm and Dijkstra, with or without optimization, to more sophisticated ones such as A* algorithm. We also proposed a new algorithm to improve existing methods and evaluated through a thorough experimental comparison with all previously mentioned counterparts. Moreover, we demonstrate that our proposed new methods can be used in real-word applications.

This paper has also discussed the shortest path algorithm on maps, and proposed a heuristic restrict path finding algorithm. In next steps, we plan to work on constant optimization and multiple threads optimization, which can improve the efficiency as well. As for the implementation and analysis, bi-directional A* algorithm has not been dealt with and it is a pity that we have not applied our program to real maps. Regarding special shortest path problems, such as grid graph with obstacles, labyrinthetc, we have some ideas, but not have time to discuss and implement them. Even so, our dynamic shortest path algorithm can still optimize many practical problems greatly. We believe that dynamic shortest path problem on maps can be further improved.

5.2 Applications and Future Perspective

Dynamic shortest paths problem is an important class of problems, solutions to which can be applied in many real-world applications. For example:

Scenario 1: There are many GISs such as Baidu Map, Google Map, which collect huge volumes of data. When a customer gives the source and the target, the GIS will activate the shortest path algorithm and find the most suitable route according to the customer's preference. It is a seemingly simple but very complicated problem. On the base of well-prepared fundamental theories, GISs can be widely applied to navigation, transportation, logistics, delivery services and so on.

Scenario 2: the complexity of business management increases rapidly with the quick development of the society. More and more companies introduce engineering approaches, which convert many requirements according to time schedule, work load, quality of tasks etc. to mathematical problems. Many of these problems can be solved by the shortest path finding algorithms. Consequently, the overall management and administration can be enhanced to facilitate further development of national economy.

In conclusion, the shortest path problem, having a wide range of applications is a problem worth further investigations.

Acknowledgement

The authors would like to thank Dr. Jiannan Wang for his advice, and Nanjing Foreign Language School, Nanjing University, and our parents for their support.

References

1.T. Cormen, C.Leiserson. Introduction to Algorithm. The MIT Press. 2001

2.V. Kalavri, T. Simas, D. Logothetis . The Shortest Path is not Always a Straight Line. VLDB 2016,

3.L. Wu, X. Xiao, D.Deng, G. Cong. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. VLDB 2012.

4.https://www.microsoft.com/en-us/research/publication/reach-for-a-efficient-point-to-point-shortest-paths-algorithms/

5.http://www.cnblogs.com/anrainie/p/4923817.html

6.http://101.96.10.64/www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20paths%20algorithms.pdf

7.https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

8.https://en.wikipedia.org/wiki/Shortest_Paths_Faster_Algorithm

9. Bi-directional search algorithm. https://en.wikipedia.org/wiki/Bidirectional_search

10. Breadth first search algorithm. https://en.wikipedia.org/wiki/Breadth-first_search

11.https://en.wikipedia.org/wiki/A*_search_algorithm

12.王逸松，《浅谈计算机系统结构与竞赛中的实用编程技巧》，2017 年全国青少年信息学奥林匹克冬令营

13.段凡丁，“关于最短路径的 SPFA 快速算法”，西南交通大学学报，1994-2

Contributions of Authors

Tao Zhongnian: Participated in constructing the overall framework of the paper; proposed applications of shortest paths on maps; conceived, designed, and implemented improvements, wrote all codes, generated all test data, compared related algorithms of Chapter 2 and Chapter 3; wrote major portions of the two chapters.

Yang Junzhao: Participated in discussing the paper framework; proposed the optimizations in chapter 4; conceived, designed, and implemented improvements, wrote all codes, generated all test data, compared related algorithms of Chapter 4; wrote Chapter 4;set up testing environment.

Xu Zhuoyu: Participated in discussing the paper framework; collected materials, maintained program and data; coordinated the paper writing; participated in writing chapter 1, chapter 5 and some parts of the other chapters; responsible for the integration of paper and English translation of most of the paper.

Biography of Authors

Tao Zhongnian, male, born in 2000, is a student in Senior 2 (1) Science Experiment Class of Nanjing Foreign Language School. Tao loves computer science and has won National First Prize of National Olympiad in Informatics in Provinces (Senior Group) from Junior 2 to Senior 2 consecutively. Other honors received by Tao include: ranked 1st among old medalists of National Olympiad in Informatics 2017 Winter Camp, gold medalist of China Team Selection Competition2017, silver medalist of Asia Pacific Informatics Olympiad 2017, and Provincial First Prize winner of National Olympiad in Mathematics in Provinces 2017.

Yang Junzhao, male, born in 2002, is a student in Senior 1 (1) Science Experiment Class of Nanjing Foreign Language School. Yang loves computer science and in his Junior 2 grade, he has got the contract of admission by Tsinghua university if passing the first batch of undergraduate. He was National First Prize winner in National Olympiad in Informatics in Provinces (NOIP), silver medalist in National Olympiad in Informatics(NOI) 2016, and silver medalist in China Team Selection Competition(CTSC) 2016. Other honors received by Yang include: Codeforces International Grandmaster, ranked global 1% in American Mathematics Competition (AMC 10): (144/150), American Invitational Mathematics Examination(AIME) : (12/15).

Xu Zhuoyu, female, born in 2001, is a student in Senior 2 (1) Science Experiment Class of Nanjing Foreign Language School. Xu loves computer science and other STEM subjects. In Junior 1She obtained Provincial First Prize in National Olympiad in Informatics in Provinces (Senior Group) 2013, then National First Prize in National Olympiad in Informatics in Provinces (Senior Group). Xu is the team leader of computer team in Jiangsu Province in “Elite Plan” of Ministry of Education, and was the Captain of China team and Bronze medalist of 5th the International Young Naturalists' Tournament (IYNT 2017). She ranked global 1% in American Mathematics Competition (AMC 10).

Biography of Advisor

Dr. Wang Jiannan is an assistant professor at Simon Fraser University in Canada. He obtained his Ph.D. degree from Department of Computer Science and Technology in Tsinghua University in 2013, and his thesis received the honor of Meritorious Doctor’s Degree thesis of China Computer Federation 2013.He conducted his postdoctoral research at University of California at Berkeley from 2013 to 2016. Dr. Wang’s research area covers database systems, big data science, etc. His research papers are published in SIGMOD, VLDB, ICDE, SIGKDD, TODS, VLDB and other top meeting proceedings.

本参赛团队声明所提交的论文是在指导老师指导下进行的研究工作和取得的研究成果。尽本团队所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果。若有不实之处，本人愿意承担一切相关责任。

参赛队员：[illegible] 杨骏昭 许倬昱 指导老师：[illegible]

2017 年 9 月 15 日