

Shortest Path Algorithm with Heaps

Group number:1

3200103998 Haowei Cao

3200105787 Yunce Zhang

March 28, 2022

1 Introduction

Shortest path problems are ones of the most fundamental combinatorial optimization problems with many applications, both direct and as subroutines in other combinatorial optimization algorithms. Algorithms for these problems have been studied since 1950's and still remain an active area of research.

Dijkstra algorithm was proposed by Dutch computer scientist Dijkstra in 1959, so it is also called Dijkstra algorithm. It is the shortest path algorithm from one vertex to other vertices, which solves the shortest path problem in weighted graph. The main feature of Dijkstra algorithm is that it starts from the starting point and adopts the strategy of greedy algorithm. Each time it traverses the adjacent nodes of the vertices closest to the starting point and not visited until it extends to the end point.

This project suppose us to compute the shortest paths using Dijkstra's algorithm. The implementation shall be based on a min-priority queue, such as a Fibonacci heap. The goal of the project is to find the best data structure for the Dijkstra's algorithm.

In this project we will use three different data structures:

Fibonacci heap: Fibonacci heap is a kind of heap. Like binomial heap, it is also a mergeable heap; It can be used to merge priority queues. Fibonacci heap has better performance of spreading analysis than binomial heap, and the time complexity of its merging operation is $O(1)$. Like binomial heap, it is also composed of a set of heap minimum ordered trees, and it is a mergeable heap. Unlike the binomial heap, the trees in the Fibonacci heap are not necessarily binomial trees; And the trees in the binomial pile are arranged in order, but the trees in the Fibonacci pile are rooted and disordered.

Binomial heap: Binomial heap is a collection of binomial trees or a group of binomial trees. Binomial reactor has good properties. Two binomial heap merging operations can be completed in $O(\log n)$, so binomial heap is a mergeable heap, and just need $O(n)$ the insertion operation of binomial heap can be completed. Therefore, the priority queue and process scheduling algorithm based on binomial heap has good time performance. At the same time, due to the structural characteristics and properties of binomial tree, binomial tree is also widely used in many fields such as network optimization.

Pairing heap: Pairing heap is a heap data structure with simple implementation and superior sharing complexity. It was invented by Michael Fredman, Robert Sedgwick, Daniel Sleator and Robert Tayan in 1986. Pairing heap is a kind of multitree and can be considered as a simplified Fibonacci heap. Pairing heap is a better choice for implementing algorithms such as prim minimum spanning tree algorithm.

2 Data Structure / Algorithm Specification

We define the structure Edge common to three data structures. It is the core data of Dijkstra algorithm.

And we use three different heap structures: Binomial heap:

```
1 typedef struct BinomialNode *BinHeap, *BinNode;
2 struct BinomialNode {
3     int value;
4     int vertex;
5     int degree;
6     BinHeap parent;
7     BinHeap child;
8     BinHeap sibling;
9 };
10 BinHeap InitialBinNode(int value, int vertex);
11 BinHeap BinLinked(BinHeap binheap, BinHeap child);
12 BinHeap BinCombineOrder(BinHeap binheap1, BinHeap binheap2);
13 BinHeap BinMerge(BinHeap binheap1, BinHeap binheap2);
14 BinHeap* BinGetMin2(BinHeap binheap);
15 BinHeap BinDeleteMin(BinHeap binheap);
16 BinHeap BinGetMin(BinHeap binheap);
17 BinHeap BinInsert(BinHeap binheap, BinNode binnode);
18 void BinDecrease(BinHeap binheap, BinNode binnode,
19 int value, BinHeap* NodeArray);
20 bool IsBinEmpty(BinHeap binheap);
```

Fibonacci heap:

```
1 typedef struct FibonacciNode *FibNode;
2 struct FibonacciNode{
3     int vertex;
4     int degree;
5     int value;
6     FibNode right;
7     FibNode left;
8     FibNode parent;
9     FibNode child;
10    bool mark;
11 };
12 typedef struct FibonacciHeap *FibHeap;
13 struct FibonacciHeap{
14     FibNode min;
15     int maxDegree;
16     int keyNum;
17     FibNode *cons;
18 };
19 FibHeap InitialHeap(void);
20 FibNode InitialFibNode(int value, int vertex);
21 void FibInsertNode(FibHeap fibheap, FibNode fibnode);
22 void FibAddBefore(FibNode fibnode1, FibNode fibnode2);
23 FibHeap FibHeapMerge(FibHeap fibheap1, FibHeap fibheap2);
24 bool IsFibEmpty(FibHeap fibheap);
25 FibNode FibHeapMin(FibHeap fibheap);
26 void FibDecrease(FibHeap fibheap, FibNode fibnode, int value);
```

```

27 void FibTranstoRoot(FibHeap fibheap, FibNode fibnode);
28 void FibDeleteMin(FibHeap fibheap);
29 void FibNeaten(FibHeap fibheap);

```

Pairing heap:

```

1  typedef struct PairingNode* PairHeap,*PairNode;
2  struct PairingNode {
3      int vertex;
4      int value;
5      PairHeap child;
6      PairHeap sibling;
7      PairHeap Prev;
8  };
9  PairNode InitialPairHeap(int value, int vertex);
10 PairHeap PairMerge(PairHeap pairheap1, PairHeap pairheap2);
11 PairHeap PairInsert(PairHeap pairheap, PairNode pairnode);
12 PairHeap PairDeleteMin(PairHeap pairheap);
13 PairHeap CombineSiblings(PairHeap pairheap);
14 PairHeap PairDecrease(PairHeap pairheap, PairNode pairnode, int value);
15 bool IsPairEmpty(PairHeap pairheap);

```

Each heap structure has a corresponding basis function, such as initial,merge,insert,deletemin,decrease,isempty,each has its own special operation function

Because we use three different heap structures, we design three different Dijkstra functions, but the core idea of Dijkstra function is similar:

Step 1: find the unmarked point closest to the starting point (if not, stop the algorithm)

Step 2: update the points around the point with the point as the center

Step 3: execute repeatedly

Pseudocode of Dijkstra's algorithm (Heap-Based)

```

1  for every vertex  $v$  in  $V$  do
2       $d_v \leftarrow \infty$ ;  $p_v \leftarrow null$ 
3      Insert( $Q, v, d_v$ ) /* initialize vertex priority in the priority queue */
4   $d_s \leftarrow 0$ ; Decrease( $Q, s, d_s$ ) /* update priority of  $s$  with  $d_s$  */
5   $V_T \leftarrow \emptyset$ 
6  for  $i \leftarrow 0$  to  $|V| - 1$  do
7       $u^* \leftarrow DeleteMin(Q)$  /* delete the minimum priority element */
8       $V_T \leftarrow V_T \cup \{u^*\}$ 
9      for every vertex  $u$  in  $V - V_T$  that is adjacent to  $u^*$  do
10         if  $d_{u^*} + w(u^*, u) < d_u$ 
11              $d_u \leftarrow d_{u^*} + w(u^*, u)$ ;  $p_u \leftarrow u^*$ 
12         Decrease( $Q, u, d_u$ )

```

As for input of the map, we use readfile function to realize it, through this function, we can read the files in the same folder Txt file to read the map node information stored in the file.

```

1  Edge *ReadFile(int *maxNode)
2  {
3      FILE *fp;
4      char StrLine[150];
5      if (file has no data) Open file error;

```

```

6      Loading file...;
7      while (!feof(fp) && StrLine[0] != 'p') Read one line;
8      Split string;
9      for (int i = 0; i < 2; i++) Jump 'p' and 'sp';
10     Get the number of nodes;
11     create edge[];
12     Read one line;
13     while (!feof(fp))
14     {
15         if (StrLine[0] == 'a')
16         {
17             Split the string.
18             int departure = (int)atof(strtok(NULL, " "));
19             int destination = (int)atof(strtok(NULL, " "));
20             int weight = (int)atof(strtok(NULL, " "));
21             edge[departure] = AddEdge(edge[departure], weight, destination);
22         }
23         Read one line
24     }
25     Read completely!;
26     fclose(fp);
27     return edge;
28 }

```

Our program realizes this function:

Read map stored in file, calibrate the starting node, and then select the node to be found, the program will print the time of each way.

```

1  int main(void)
2  {
3      Read map from file;
4      if (edge != NULL)
5      {
6          Enter source node between 1 and %d:\n;
7          Choose source node;
8          Check if the source node is legal;
9          {
10             You can only choose sorNode between 1 and %d ;
11             scanf_s("%d", &sorNode);
12         }
13         while (true)
14         {
15             Enter the number of nodes you want to find;
16             scanf_s("%d", &numFind);
17             Check if the number of target node is legal;
18             {
19                 You can only choose numFind between 1 and %d\n;
20                 continue;
21             }
22             if (all nodes)
23                 numFind = maxNode - 1;
24             beginTime = clock();
25                 DijkstraWithFib;
26             endTime = clock();

```

```

27         printf
28         beginTime = clock();
29             DijkstraWithPair;
30         endTime = clock();
31         printf;
32         beginTime = clock();
33             DijkstraWithBin;
34         endTime = clock();
35         printf;
36     }
37 }
38 system("pause");
39 }

```

3 Testing Results

Table 1: time cost

nodes	edges	density	Pairing heap	Fibonacci Heap	Binomial Heap
100	400	M=4*N	121	44	640
100	400	M=4*N	102	44	488
100	400	M=4*N	102	42	501
100	1200	M=12*N	203	384	751
100	1200	M=12*N	212	475	762
100	1200	M=12*N	290	601	1320
100	2500	M=25*N	370	683	1121
100	2500	M=25*N	371	677	1139
100	2500	M=25*N	371	675	1128
100	200	M=N*N/50	72	6	365
100	200	M=N*N/50	70	5	352
100	200	M=N*N/50	70	5	372
100	1000	M=N*N/10	179	348	702
100	1000	M=N*N/10	177	350	697
100	1000	M=N*N/10	178	351	699
600	2400	M=4*N	1535	251	5214
600	2400	M=4*N	1613	250	5215
600	2400	M=4*N	1607	291	5202
600	7200	M=12*N	2174	3778	10313
600	7200	M=12*N	2182	3739	9720
600	7200	M=12*N	2168	3744	9656
600	15000	M=25*N	3105	2754	15319
600	15000	M=25*N	3265	2690	14335
600	15000	M=25*N	3111	2639	14561
600	7200	M=N*N/50	2211	2572	9633
600	7200	M=N*N/50	2197	2566	9526
600	7200	M=N*N/50	2182	2570	9625
600	36000	M=N*N/10	5784	10774	21225
600	36000	M=N*N/10	5746	10743	21454
600	36000	M=N*N/10	5950	10767	21860
1100	4400	M=4*N	5146	2904	12752
1100	4400	M=4*N	5155	2907	12789

1100	4400	$M=4*N$	5135	2904	12739
1100	13200	$M=12*N$	6223	5376	25651
1100	13200	$M=12*N$	6203	5359	26612
1100	13200	$M=12*N$	6254	5449	25865
1100	27500	$M=25*N$	7871	9202	39570
1100	27500	$M=25*N$	7967	9192	39072
1100	27500	$M=25*N$	8091	9325	39702
1100	24200	$M=N*N/50$	9501	12775	59749
1100	24200	$M=N*N/50$	7506	8520	35684
1100	24200	$M=N*N/50$	7573	8620	37161
1100	121000	$M=N*N/10$	19539	38524	80679
1100	121000	$M=N*N/10$	19531	38512	80873
1100	121000	$M=N*N/10$	19615	38358	79625
1600	6400	$M=4*N$	11181	4549	25441
1600	6400	$M=4*N$	11023	4704	23694
1600	6400	$M=4*N$	11581	4519	26689
1600	19200	$M=12*N$	12576	1338	50370
1600	19200	$M=12*N$	13029	1346	50582
1600	19200	$M=12*N$	12570	1330	49831
1600	40000	$M=25*N$	14946	14818	73491
1600	40000	$M=25*N$	14945	14818	75068
1600	40000	$M=25*N$	14823	14900	75071
1600	51200	$M=N*N/50$	16105	3973	84633
1600	51200	$M=N*N/50$	16310	3976	82460
1600	51200	$M=N*N/50$	16331	3925	82603
1600	256000	$M=N*N/10$	41383	51510	178432
1600	256000	$M=N*N/10$	41087	51755	178135
1600	256000	$M=N*N/10$	40932	51863	177724
2100	8400	$M=4*N$	19256	5960	35644
2100	8400	$M=4*N$	19063	5971	35638
2100	8400	$M=4*N$	19225	5961	35721
2100	25200	$M=12*N$	21613	1180	83637
2100	25200	$M=12*N$	21121	1163	80289
2100	25200	$M=12*N$	21466	1190	82004
2100	52500	$M=25*N$	24011	14889	119065
2100	52500	$M=25*N$	24041	14935	137691
2100	52500	$M=25*N$	25707	15378	136400
2100	88200	$M=N*N/50$	29351	33912	177945
2100	88200	$M=N*N/50$	38042	119082	209809
2100	88200	$M=N*N/50$	42769	63756	241128
2100	441000	$M=N*N/10$	70765	113691	317623
2100	441000	$M=N*N/10$	70913	113650	316929
2100	441000	$M=N*N/10$	70733	113222	317914
2600	10400	$M=4*N$	29375	4871	52130
2600	10400	$M=4*N$	29455	4852	51030
2600	10400	$M=4*N$	29558	4855	49868
2600	31200	$M=12*N$	31771	555	117147
2600	31200	$M=12*N$	31843	531	118705
2600	31200	$M=12*N$	31985	534	116291
2600	65000	$M=25*N$	35514	10013	180119
2600	65000	$M=25*N$	35402	10081	180163
2600	65000	$M=25*N$	35497	10044	178140
2600	135200	$M=N*N/50$	43164	10404	256604
2600	135200	$M=N*N/50$	43144	10360	255594

2600	135200	$M=N*N/50$	43232	10335	256925
2600	676000	$M=N*N/10$	108672	190930	498486
2600	676000	$M=N*N/10$	108978	191662	499284
2600	676000	$M=N*N/10$	110941	191840	496657
3100	12400	$M=4*N$	42742	8616	70162
3100	12400	$M=4*N$	42269	8509	66178
3100	12400	$M=4*N$	42338	8523	67415
3100	37200	$M=12*N$	44890	14070	163224
3100	37200	$M=12*N$	44985	14098	164574
3100	37200	$M=12*N$	45340	14077	162299
3100	77500	$M=25*N$	49028	2097	289477
3100	77500	$M=25*N$	49152	2084	247327
3100	77500	$M=25*N$	49190	2221	246746
3100	192200	$M=N*N/50$	61586	22379	367170
3100	192200	$M=N*N/50$	61349	22528	365564
3100	192200	$M=N*N/50$	71689	23722	390660
3100	961000	$M=N*N/10$	155217	190369	775867
3100	961000	$M=N*N/10$	156157	190808	740477
3100	961000	$M=N*N/10$	155367	190393	738328
3600	14400	$M=4*N$	57233	10645	88708
3600	14400	$M=4*N$	56928	10649	88086
3600	14400	$M=4*N$	57306	10830	88907
3600	43200	$M=12*N$	60147	984	215981
3600	43200	$M=12*N$	60536	965	215154
3600	43200	$M=12*N$	61000	983	217740
3600	90000	$M=25*N$	64728	3039	320987
3600	90000	$M=25*N$	65434	2999	322667
3600	90000	$M=25*N$	65024	3028	323388
3600	259200	$M=N*N/50$	83163	83663	521061
3600	259200	$M=N*N/50$	83169	83599	524621
3600	259200	$M=N*N/50$	85394	85711	538314
3600	1296000	$M=N*N/10$	211497	367158	1066458
3600	1296000	$M=N*N/10$	216309	375757	1055858
3600	1296000	$M=N*N/10$	217179	378741	1062303
4100	16400	$M=4*N$	75436	12315	117747
4100	16400	$M=4*N$	75761	12214	114597
4100	16400	$M=4*N$	75494	12124	112015
4100	49200	$M=12*N$	78324	20742	273779
4100	49200	$M=12*N$	78254	20668	273517
4100	49200	$M=12*N$	81344	21585	290497
4100	102500	$M=25*N$	83802	32795	419177
4100	102500	$M=25*N$	86366	33755	418390
4100	102500	$M=25*N$	83130	32663	414462
4100	336200	$M=N*N/50$	108819	14836	687484
4100	336200	$M=N*N/50$	165930	18235	743406
4100	336200	$M=N*N/50$	124554	16079	709096
4100	1681000	$M=N*N/10$	282950	342160	1392555
4100	1681000	$M=N*N/10$	422153	408815	1619043
4100	1681000	$M=N*N/10$	276232	340881	1459948
4600	18400	$M=4*N$	94895	13136	133770
4600	18400	$M=4*N$	95499	13264	131488
4600	18400	$M=4*N$	96381	13060	131516
4600	55200	$M=12*N$	98641	11596	341402
4600	55200	$M=12*N$	98281	11489	346341

4600	55200	M=12*N	127785	26338	604580
4600	115000	M=25*N	112158	42201	618188
4600	115000	M=25*N	106597	36325	531903
4600	115000	M=25*N	112823	40233	670838
4600	423200	M=N*N/50	160558	140647	989018
4600	423200	M=N*N/50	142384	151648	1004729
4600	423200	M=N*N/50	157937	147852	977572
4600	2116000	M=N*N/10	348183	608216	1804076
4600	2116000	M=N*N/10	347272	607474	1809837
4600	2116000	M=N*N/10	369732	640647	2008660

4 Analysis and Comments

It can be seen from the above test table that the running time of binomial heap is much slower than Fibonacci heap and paired heap, and when the graph density is small, the running time of Fibonacci heap is faster than that of paired heap, while when the graph density is large, the running speed of paired heap is faster than that of Fibonacci heap. When the nodes and edges are large, the relationship between them will become more complex.

As for Fibonacci heap, the time complexity is $O(v \log v + e)$. For the more important operations among the function, the time complexity of the findmin is $\Omega(\lg n)$, the deletemin is $\Theta(\lg n)$, the insertion is $\Omega(\lg n)$, the decrease and merge operations are all $\Theta(\lg n)$. After overall comprehensive calculation, the time complexity is $O(v \log v + e)$.

As for binomial heap, the time complexity is $O(e \log v)$. For the more important operations among the function, the time complexity of the findmin is $\Theta(1)$, the deletemin is $O(\log n)$, the insertion, decrease and merge operations are all $\Theta(1)$. After overall comprehensive calculation, the time complexity is $O(e \log v)$.

As for pairing heap, the time complexity is $O(v \log v + e)$. For the more important operations among the function, the time complexity of the findmin, merge and insert is $\Theta(1)$, the deletemin and decrease is $O(\log n)$. After overall comprehensive calculation, the time complexity is $O(v \log v + e)$.

The space complexity is $O(e + v)$.

5 Author list

coding: Zhang Yunce

report: Cao Haowei

Declaration

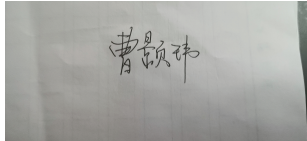
We hereby declare that all the work done in this project titled "Shortest Path Algorithm with Heaps" is of our independent effort as a group.

6 Signatures

Zhang Yunce

曹豪威

Cao Haowei



A Source Code (if required)

main.c

```
1  #include "Dijkstra.h"
2  #include "Fibonacci heap.h"
3  #include "Pairing heap.h"
4  #include "Binomial heap.h"
5  #include <time.h>
6
7  int main(void)
8  {
9      int maxNode, sorNode, numFind;
10     int beginTime, endTime;
11     Edge *edge = ReadFile(&maxNode); //Read map from file.
12     if (edge != NULL)
13     {
14         printf("Enter source node between 1 and %d:\n", maxNode - 1);
15         scanf_s("%d", &sorNode); //Choose source node.
16         while (sorNode > maxNode - 1 || sorNode < 1)
17             //Check if the source node is legal.
18         {
19             printf("You can only choose sorNode
20             between 1 and %d \nPlease input again:", maxNode - 1);
21             scanf_s("%d", &sorNode);
22         }
23         while (true)
24         {
```

```

25     printf("Enter the number of nodes you want to find
26     (0 means all nodes):(Program will find numFind nearest nodes)
27     \n");
28     scanf_s("%d", &numFind);
29     if (numFind > maxNode - 1 || numFind < 0)
30         //Check if the number of target node is legal.
31     {
32         printf("You can only choose numFind
33         between 1 and %d\n", maxNode - 1);
34         continue;
35     }
36     if (numFind == 0) //0 means all nodes.
37         numFind = maxNode - 1;
38     beginTime = clock();
39     DijkstraWithFib(edge, maxNode, sorNode, numFind);
40     endTime = clock();
41     printf("Fibonacci heap || Number of nodes
42     : %d || Time: %dms\n", numFind, endTime - beginTime);
43     beginTime = clock();
44     DijkstraWithPair(edge, maxNode, sorNode, numFind);
45     endTime = clock();
46     printf("Pairing heap || Number of nodes
47     : %d || Time: %dms\n", numFind, endTime - beginTime);
48     beginTime = clock();
49     DijkstraWithBin(edge, maxNode, sorNode, numFind);
50     endTime = clock();
51     printf("Binomial heap || Number of nodes
52     : %d || Time: %dms\n", numFind, endTime - beginTime);
53 }
54 }
55 system("pause");
56 }

```

Binomial heap.h

```

1  #pragma once
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<stdbool.h>
5  typedef struct BinomialNode *BinHeap, *BinNode;
6  struct BinomialNode {
7      int value;
8      int vertex;
9      int degree;
10     BinHeap parent;
11     BinHeap child;
12     BinHeap sibling;
13 };
14 BinHeap InitialBinNode(int value, int vertex); //Create a new node.
15 BinHeap BinLinked(BinHeap binheap, BinHeap child);
16 //Link child node with binomial heap.
17 BinHeap BinCombineOrder(BinHeap binheap1, BinHeap binheap2);
18 //Merge two lists in increase order.
19 BinHeap BinMerge(BinHeap binheap1, BinHeap binheap2);

```

```

20 //Merge two lists and combine subtrees with same degrees
21 BinHeap* BinGetMin2(BinHeap binheap);
22 //Get the min node and its previous node.
23 BinHeap BinDeleteMin(BinHeap binheap);
24 //Delete min node.
25 BinHeap BinGetMin(BinHeap binheap);
26 //Get the min node.
27 BinHeap BinInsert(BinHeap binheap, BinNode binnode);
28 //Insert a new node to heap.
29 void BinDecrease(BinHeap binheap, BinNode binnode,
30 int value, BinHeap* NodeArray);
31 //Decrease the value of one certain node.
32 bool IsBinEmpty(BinHeap binheap);
33 //Judge if the heap is empty.

```

Binomial heap.c

```

1 #include "Binomial heap.h"
2
3
4 BinHeap InitialBinNode(int value, int vertex)
5 {
6     BinHeap binheap = (BinHeap)malloc(sizeof(struct BinomialNode));
7     binheap->child = NULL;
8     binheap->parent = NULL;
9     binheap->sibling = NULL;
10    binheap->degree = 0;
11    binheap->value = value;
12    binheap->vertex = vertex;
13    return binheap;
14 }
15
16 BinHeap BinLinked(BinHeap binheap, BinHeap child)
17 {
18     child->parent = binheap;
19     child->sibling = binheap->child;
20     binheap->child = child;
21     binheap->degree++;
22     return binheap;
23 }
24
25 BinHeap BinCombineOrder(BinHeap binheap1, BinHeap binheap2)
26 {
27     BinNode ptr1 = binheap1, ptr2 = binheap2;
28     BinHeap binBegin = NULL;
29     BinHeap *binheap = &binBegin;
30     while (ptr1 != NULL && ptr2 != NULL)
31         //Merge two lists in increase order.
32         {
33             if (ptr1->degree <= ptr2->degree)
34             {
35                 *binheap = ptr1;
36                 ptr1 = ptr1->sibling;
37             }

```

```

38         else
39         {
40             *binheap = ptr2;
41             ptr2 = ptr2->sibling;
42         }
43         binheap = &((*binheap)->sibling);
44     }
45     if (ptr1 == NULL)
46         *binheap = ptr2;
47     else
48         *binheap = ptr1;
49     return binBegin;
50 }
51
52 BinHeap BinMerge(BinHeap binheap1, BinHeap binheap2)
53 {
54     BinHeap binheap = BinCombineOrder(binheap1, binheap2);
55     //First step, merge them in increase order.
56     if (binheap == NULL)
57         return NULL;
58     BinNode p_prev, p, p_sib;
59     p_prev = NULL;
60     p = binheap;
61     p_sib = p->sibling;
62     while (p_sib != NULL)
63     {
64         if (p->degree != p_sib->degree || p_sib->sibling != NULL
65             && p_sib->degree == p_sib->sibling->degree)
66             //In this case, forms a(p)-b or a(p)-a-a will be skipped.
67             {
68                 p_prev = p;
69                 p = p_sib;
70             }
71         else //Need to merge.
72         {
73             if (p->value <= p_sib->value)
74                 //In this case, a(p)-b, b will be a child of
75                 a.(value(a)<=value(b))
76                 {
77                     p->sibling = p_sib->sibling; //p_sib will be removed.
78                     p = BinLinked(p, p_sib);
79                 }
80             else
81                 //In this case, a(p)-b, a will be a child of
82                 b.(value(a)>value(b))
83                 {
84                     if (p_prev == NULL)
85                         binheap = p_sib;
86                     else
87                         p_prev->sibling = p_sib;
88                     p_sib = BinLinked(p_sib, p);
89                     //p will be removed.
90                     p = p_sib;
91                 }

```

```

92     }
93     p_sib = p->sibling;
94 }
95 return binheap;
96 }
97
98 BinHeap* BinGetMin2(BinHeap binheap)
99 {
100     if (binheap->sibling == NULL)
101         return NULL;
102     else
103     {
104         BinHeap* p = (BinHeap*)malloc(sizeof(BinHeap) * 2);
105         //Used to storage min position.
106         p[0] = NULL;
107         p[1] = binheap;
108         BinNode ptr, ptr_prev; //Used to travel the link;
109         ptr_prev = NULL;
110         ptr = binheap;
111         while (ptr!= NULL)
112         {
113             if (ptr->value < p[1]->value)
114             {
115                 p[0] = ptr_prev;
116                 p[1] = ptr;
117             }
118             ptr_prev = ptr;
119             ptr = ptr->sibling;
120         }
121         return p;
122     }
123 }
124
125 BinHeap BinDeleteMin(BinHeap binheap)
126 {
127     if (binheap == NULL)
128         return NULL;
129     else
130     {
131         BinHeap* min = BinGetMin2(binheap);
132         //Get the min node and its previous node.
133         if (min == NULL)
134             //Check if the min node is the first node.
135             return binheap->child;
136         BinHeap binMin_prev = min[0], binMin = min[1];
137         if (binMin_prev == NULL)
138             binheap = binheap->sibling;
139         else
140             binMin_prev->sibling = binMin->sibling;
141         BinNode childList = binMin->child;
142         BinNode reverse = NULL;
143         BinNode temp;
144         while (childList != NULL) //Reverse the list.
145         {

```

```

146         temp = childList->sibling;
147         if (reverse == NULL)
148         {
149             reverse = childList;
150             reverse->sibling = NULL;
151         }
152         else
153         {
154             childList->sibling = reverse;
155             reverse = childList;
156         }
157         childList = temp;
158     }
159     free(min);
160     return BinMerge(binheap, reverse);
161 }
162 }
163
164 BinHeap BinGetMin(BinHeap binheap)
165 {
166     if (binheap == NULL)
167         return NULL;
168     else
169     {
170         BinNode ptr = binheap;
171         BinNode temp = ptr;
172         while (ptr != NULL)
173         {
174             if (ptr->value < temp->value)
175                 temp = ptr;
176             ptr = ptr->sibling;
177         }
178         return temp;
179     }
180 }
181
182 BinHeap BinInsert(BinHeap binheap, BinNode binnode)
183 {
184     return BinMerge(binheap, binnode);
185 }
186
187 void BinDecrease(BinHeap binheap, BinNode binnode,
188 int value, BinHeap* NodeArray)
189 {
190     binnode->value = value;
191     BinNode parent, child;
192     parent = binnode->parent;
193     child = binnode;
194     while (parent != NULL && parent->value > child->value)
195         //Exchange the position between child and parent.
196     {
197         int temp_value, temp_vertex;
198         BinNode temp = NodeArray[child->vertex];
199         //Change the Distance array.

```

```

200     NodeArray[child->vertex] = NodeArray[parent->vertex];
201     NodeArray[parent->vertex] = temp;
202     temp_value = child->value;
203     temp_vertex = child->vertex;
204     child->value = parent->value;
205     child->vertex = parent->vertex;
206     parent->value = temp_value;
207     parent->vertex = temp_vertex;
208     child = parent;
209     parent = parent->parent;
210 }
211 }
212
213 bool IsBinEmpty(BinHeap binheap)
214 {
215     return binheap == NULL;
216 }

```

Fibonacci heap.h

```

1  #pragma once
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<stdbool.h>
5  #include<math.h>
6
7  typedef struct FibonacciNode *FibNode;
8  struct FibonacciNode{
9      int vertex;
10     int degree;
11     int value;
12     FibNode right;
13     FibNode left;
14     FibNode parent;
15     FibNode child;
16     bool mark;
17 };
18
19 typedef struct FibonacciHeap *FibHeap;
20 struct FibonacciHeap{
21     FibNode min;
22     int maxDegree;
23     int keyNum;
24     FibNode *cons;
25 };
26
27 FibHeap InitialHeap(void); //Initial the heap.
28 FibNode InitialFibNode(int value, int vertex);
29 //Create a new node.
30 void FibInsertNode(FibHeap fibheap, FibNode fibnode);
31 //Insert a node into fibonacci heap.
32 void FibAddBefore(FibNode fibnode1, FibNode fibnode2);
33 //add node 1 before node2
34 FibHeap FibHeapMerge(FibHeap fibheap1, FibHeap fibheap2);

```

```

35 bool IsFibEmpty(FibHeap fibheap);
36 FibNode FibHeapMin(FibHeap fibheap);
37 //Return the least node.
38 void FibDecrease(FibHeap fibheap, FibNode fibnode, int value);
39 //Change the value of a certain node.
40 void FibTranstoRoot(FibHeap fibheap, FibNode fibnode);
41 //Move the node to root.
42 void FibDeleteMin(FibHeap fibheap);
43 void FibNeaten(FibHeap fibheap);
44 //Merge the nodes of same degrees.

```

Fibonacci heap.c

```

1  #include "Fibonacci heap.h"
2  #include <time.h>
3
4  FibHeap InitialHeap(void)
5  {
6      FibHeap fibheap = (FibHeap)malloc(sizeof(struct FibonacciHeap));
7      fibheap->min = NULL;
8      fibheap->maxDegree = 0;
9      fibheap->keyNum = 0;
10     fibheap->cons = NULL;
11     return fibheap;
12 }
13
14 FibNode InitialFibNode(int value, int vertex)
15 {
16     FibNode fibnode = (FibNode)malloc(sizeof(struct FibonacciNode));
17     fibnode->child = NULL;
18     fibnode->parent = NULL;
19     fibnode->left = fibnode;
20     fibnode->right = fibnode;
21     fibnode->vertex = vertex;
22     fibnode->value = value;
23     fibnode->degree = 0;
24     fibnode->mark = false;
25     return fibnode;
26 }
27
28 void FibInsertNode(FibHeap fibheap, FibNode fibnode)
29 {
30     if (fibnode == NULL) //Fibnode is illegal
31         return;
32     else if (fibheap->keyNum == 0)
33         //Fibnode is the first member of the fibonacci heap.
34         fibheap->min = fibnode;
35     else
36     {
37         FibNode curMin = fibheap->min;
38         FibAddBefore(fibnode, curMin);
39         //Insert fibnode to the left of the min node.
40         if (fibnode->value < curMin->value)
41             fibheap->min = fibnode;

```



```

42     }
43     fibheap->keyNum++; //Update
44 }
45
46 void FibAddBefore(FibNode fibnode1, FibNode fibnode2)
47 // -fibnode1- and -fibnode2- become -fibnode1-fibnode2-
48 {
49     fibnode1->left = fibnode2->left;
50     fibnode2->left->right = fibnode1;
51     fibnode1->right = fibnode2;
52     fibnode2->left = fibnode1;
53 }
54
55 FibHeap FibHeapMerge(FibHeap fibheap1, FibHeap fibheap2)
56 {
57     if (fibheap1 == NULL) //Check if fibheap1 or fibheap2 is null.
58         return fibheap2;
59     else if (fibheap2 == NULL)
60         return fibheap1;
61     else
62     {
63         if (fibheap1->maxDegree < fibheap2->maxDegree)
64             //Let fibheap1's maxDegree bigger than fibheap2.
65             {
66                 FibHeap temp = fibheap1;
67                 fibheap1 = fibheap2;
68                 fibheap2 = temp;
69             }
70         if (fibheap1->min == NULL)
71         {
72             fibheap1->keyNum = fibheap2->keyNum;
73             fibheap1->maxDegree = fibheap2->maxDegree;
74         }
75         else if (fibheap2->min != NULL) //Merge two fibonacci heap.
76         {
77             FibNode temp;
78             FibNode fibnode1 = fibheap1->min, fibnode2 = fibheap2->min;
79             temp = fibnode1->right;
80             fibnode1->right = fibnode2->right;
81             fibnode2->right->left = fibnode1;
82             fibnode2->right = temp;
83             temp->left = fibnode2;
84             if (fibnode1->value > fibnode2->value)
85                 fibheap1->min = fibheap2->min;
86             fibheap1->maxDegree += fibheap2->maxDegree;
87         }
88         free(fibheap2->cons);
89         free(fibheap2);
90         return fibheap1; //fibheap2 has been merged into fibheap1.
91     }
92 }
93
94 bool IsFibEmpty(FibHeap fibheap)
95 {

```

```

96     if (fibheap->min == NULL)
97         return true;
98     else
99         return false;
100 }
101
102 FibNode FibHeapMin(FibHeap fibheap)
103 {
104     return fibheap->min;
105 }
106
107 void FibDecrease(FibHeap fibheap, FibNode fibnode, int value)
108 {
109     if (fibheap == NULL || fibheap->min == NULL || fibnode == NULL)
110         return;
111     fibnode->value = value;
112     FibNode parent = fibnode->parent;
113     if (parent != NULL && parent->value > fibnode->value)
114     {
115         FibTranstoRoot(fibheap, fibnode);
116         //if fibnode's value is bigger than its parent, move it to root.
117     }
118     if (fibheap->min->value > fibnode->value)
119         fibheap->min = fibnode;
120 }
121
122 void FibTranstoRoot(FibHeap fibheap, FibNode fibnode)
123 {
124     FibNode parent = fibnode->parent;
125     if (parent == NULL)
126         return;
127     fibnode->right->left = fibnode->left;
128     //Delete fibnode from left-fibnode-right
129     fibnode->left->right = fibnode->right;
130     if (fibnode->right == fibnode)
131         //Update parent node
132         parent->child = NULL;
133     else
134         parent->child = fibnode->right;
135     parent->degree--;
136     FibAddBefore(fibnode, fibheap->min);
137     //Update fibnode information
138     fibnode->parent = NULL;
139     fibnode->mark = false;
140     if (parent->mark == false)
141         //Determine whether the fibnode needs to be moved to the root
142         parent->mark = true;
143     else
144         FibTranstoRoot(fibheap, parent);
145 }
146
147 void FibDeleteMin(FibHeap fibheap)
148 {
149     if (fibheap == NULL || fibheap->min == NULL)

```

```

150         return;
151     FibNode min = fibheap->min;
152     while (min->child != NULL) //Add all children of min to root-link
153     {
154         FibNode child = min->child;
155         child->right->left = child->left;
156         //Delete min from left-fibnode-right
157         child->left->right = child->right;
158         if (child->right == child)
159             min->child = NULL;
160         else
161             min->child = child->right;
162         FibAddBefore(child, min);
163         child->parent = NULL;
164     }
165     min->right->left = min->left; //Delete min
166     min->left->right = min->right;
167     fibheap->min = (fibheap->keyNum == 1) ? NULL : min->right;
168     fibheap->keyNum--;
169     FibNeaten(fibheap);
170 }
171
172 void FibNeaten(FibHeap fibheap)
173 {
174     if (fibheap == NULL || fibheap->min == NULL)
175         return;
176     fibheap->maxDegree = (int)(log2((double)(fibheap->keyNum)) + 1);
177     fibheap->cons = (FibNode*)malloc(sizeof(FibNode)*
178     (fibheap->maxDegree + 1));
179     //Allocate an array to combine heaps with the same degrees.
180     for (int i = 0; i < fibheap->maxDegree + 1; ++i)
181         fibheap->cons[i] = NULL;
182     while (fibheap->min != NULL) //Combine heaps with the same degrees.
183     {
184         FibNode curMin = fibheap->min; //Get the current heap.
185         int curDegree = curMin->degree;
186         if (curMin->right == curMin)
187             fibheap->min = NULL;
188         else
189         {
190             fibheap->min = curMin->right;
191             curMin->right->left = curMin->left;
192             curMin->left->right = curMin->right;
193         }
194         curMin->left = curMin->right = curMin; //Isolate this node.
195
196         while (fibheap->cons[curDegree] != NULL)
197             //Add current heap to array cons. If there is a collision.
198         {
199             FibNode SameDegree = fibheap->cons[curDegree];
200             if (SameDegree->value < curMin->value)
201             {
202                 FibNode temp = SameDegree;
203                 SameDegree = curMin;

```

```

204         curMin = temp;
205     }
206     SameDegree->right->left = SameDegree->left;
207     SameDegree->left->right = SameDegree->right;
208     if (curMin->child == NULL)
209         //Make SameDegree be curMin's children
210         curMin->child = SameDegree;
211     else
212         FibAddBefore(SameDegree, curMin->child);
213     SameDegree->parent = curMin;
214     SameDegree->mark = false;
215     curMin->degree++;
216     fibheap->cons[curDegree] = NULL;
217     curDegree++;
218 }
219 fibheap->cons[curDegree] = curMin;
220 }
221 fibheap->min = NULL;
222 for (int i = 0; i < fibheap->maxDegree + 1; ++i)
223     //Copy the heap from cons to fibheap.
224 {
225     if (fibheap->cons[i] != NULL)
226     {
227         if (fibheap->min == NULL)
228             fibheap->min = fibheap->cons[i];
229         else
230         {
231             FibAddBefore(fibheap->cons[i], fibheap->min);
232             if (fibheap->cons[i]->value < fibheap->min->value)
233                 fibheap->min = fibheap->cons[i];
234         }
235     }
236 }
237 free(fibheap->cons); //Free memory.
238 }

```

Pairing heap.h

```

1  #pragma once
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<stdbool.h>
5
6  typedef struct PairingNode* PairHeap,*PairNode;
7  struct PairingNode {
8      int vertex;
9      int value;
10     PairHeap child;
11     PairHeap sibling;
12     PairHeap Prev;
13 };
14
15 PairNode InitialPairHeap(int value, int vertex);
16 //Initial the heap(Create a node).

```

```

17 PairHeap PairMerge(PairHeap pairheap1, PairHeap pairheap2);
18 //Merge two pairing heap.
19 PairHeap PairInsert(PairHeap pairheap, PairNode pairnode);
20 PairHeap PairDeleteMin(PairHeap pairheap);
21 //Delete the min node and call CombineSiblings().
22 PairHeap CombineSiblings(PairHeap pairheap);
23 //Merge the nodes in one lists.
24 PairHeap PairDecrease(PairHeap pairheap, PairNode pairnode, int value);
25 bool IsPairEmpty(PairHeap pairheap);

```

Pairing heap.c

```

1  #include "Pairing heap.h"
2
3  PairNode InitialPairHeap(int value, int vertex)
4  {
5      PairNode pairnode = (PairNode)malloc(sizeof(struct PairingNode));
6      pairnode->child = NULL;
7      pairnode->Prev = NULL;
8      pairnode->sibling = NULL;
9      pairnode->value = value;
10     pairnode->vertex = vertex;
11     return pairnode;
12 }
13
14 PairHeap PairMerge(PairHeap pairheap1, PairHeap pairheap2)
15 {
16     if (pairheap1 == NULL)
17         return pairheap2;
18     else if (pairheap2 == NULL)
19         return pairheap1;
20     else //Add pairnode2 to pairnode1.
21     {
22         if(pairheap1->value > pairheap2->value)
23             //Let node1_pre linked to node2
24             {
25                 PairNode heap1_prev = pairheap1->Prev;
26                 pairheap2->Prev = heap1_prev;
27                 if (heap1_prev != NULL)
28                 {
29                     if (heap1_prev->child == pairheap1)
30                         heap1_prev->child = pairheap2;
31                     else
32                         heap1_prev->sibling = pairheap2;
33                 }
34                 PairNode temp = pairheap1;
35                 //Exchange node1 and node2 to unified form with "else" part.
36                 pairheap1 = pairheap2;
37                 pairheap2 = temp;
38             }
39     else
40         //Let node2->sibling linked to node1
41         {
42             PairNode heap2_sibling = pairheap2->sibling;

```

```

43         pairheap1->sibling = heap2_sibling;
44         if (pairheap1->sibling != NULL)
45             heap2_sibling->Prev = pairheap1;
46     }
47     PairNode heap1_child = pairheap1->child;
48     pairheap1->child = pairheap2;
49     pairheap2->Prev = pairheap1;
50     pairheap2->sibling = heap1_child;
51     if (pairheap2->sibling != NULL)
52         heap1_child->Prev = pairheap2;
53     return pairheap1;
54 }
55 }
56
57 PairHeap PairInsert(PairHeap pairheap, PairNode pairnode)
58 {
59     if (pairheap == NULL)
60         return pairnode;
61     else
62         return PairMerge(pairheap, pairnode);
63     //Call PairMerge to merge them.
64 }
65
66 PairHeap PairDeleteMin(PairHeap pairheap)
67 {
68     if (pairheap == NULL || pairheap->child == NULL)
69         return NULL;
70     else
71     {
72         PairHeap firstSibling = pairheap->child;
73         firstSibling->Prev = NULL;
74         return CombineSiblings(firstSibling);
75         //After we delete the min node, we need to Combine the sibling nodes.
76     }
77 }
78
79 PairHeap CombineSiblings(PairHeap pairheap)
80 {
81     PairNode heap_next = pairheap->sibling;
82     PairNode porign = pairheap;
83     while (pairheap != NULL) //The first round from left to right.
84     {
85         porign = PairMerge(pairheap, heap_next);
86         pairheap = porign->sibling;
87         heap_next = (pairheap == NULL ? NULL : pairheap->sibling);
88         if (heap_next == NULL)
89             break;
90     }
91     if (pairheap != NULL)
92         porign = pairheap;
93     PairNode heap_prev = porign->Prev;
94     while (heap_prev != NULL) //The second round from ight to left.
95     {
96         porign = PairMerge(heap_prev, porign);

```

```

97     heap_prev = porign->Prev;
98 }
99 return porign;
100 }
101
102 PairHeap PairDecrease(PairHeap pairheap, PairNode pairnode, int value)
103 {
104     pairnode->value = value;
105     if (pairnode != pairheap)
106     {
107         PairNode node_prev = pairnode->Prev;
108         if (node_prev->child == pairnode)
109             node_prev->child = pairnode->sibling;
110         else
111             node_prev->sibling = pairnode->sibling;
112         if (pairnode->sibling != NULL)
113             pairnode->sibling->Prev = node_prev;
114         pairnode->Prev = pairnode->sibling = NULL;
115         pairheap = PairMerge(pairnode, pairheap);
116     }
117     return pairheap;
118 }
119
120 bool IsPairEmpty(PairHeap pairheap)
121 {
122     if (pairheap == NULL)
123         return true;
124     else
125         return false;
126 }

```

Dijkstra.h

```

1  #pragma once
2  #define _CRT_SECURE_NO_WARNINGS
3  #include <string.h>
4  #include <limits.h>
5  #include "Fibonacci heap.h"
6  #include "Pairing heap.h"
7  #include "Binomial heap.h"
8
9  #define Infinity INT_MAX
10
11 typedef struct EdgeNode* Edge;
12 struct EdgeNode{
13     int weight;    //The distance between two nodes.
14     int destination;
15     Edge next;
16 };
17
18 Edge AddEdge(Edge start, int weight, int destination);
19 //Create n list to storage information from map.(n is the number of nodes.)
20 Edge *ReadFile(int *maxNode);
21 //Read map from a file.

```

```

22 void DijkstraWithFib(Edge *edge, int maxNode,
23 int sorNode, int numFind);
24 //Implement Dijkstra algorithm with Fibonacci heap.
25 void DijkstraWithPair(Edge *edge, int maxNode,
26 int sorNode, int numFind);
27 //Implement Dijkstra algorithm with Pairing heap.
28 void DijkstraWithBin(Edge *edge, int maxNode,
29 int sorNode, int numFind);
30 //Implement Dijkstra algorithm with Binomail heap.

```

Dijkstra.c

```

1  #include "Dijkstra.h"
2  Edge AddEdge(Edge start, int weight, int destination)
3  {
4      Edge newEdge = (Edge)malloc(sizeof(struct EdgeNode));
5      newEdge->weight = weight;
6      newEdge->destination = destination;
7      if (start == NULL) //The original list is empty.
8      {
9          newEdge->next = NULL;
10         start = newEdge;
11     }
12     else
13         //The original list is not empty. Insert the new node in the head.
14     {
15         newEdge->next = start;
16         start = newEdge;
17     }
18     return start;
19 }
20
21 Edge *ReadFile(int *maxNode)
22 {
23     FILE *fp;
24     char StrLine[150];
25     if ((fp = fopen("map.txt", "r")) == NULL)
26     {
27         printf("Open file error!\n");
28         return NULL;
29     }
30     printf("Loading file...\n");
31     while (!feof(fp) && StrLine[0] != 'p')
32     {
33         fgets(StrLine, 1024, fp); //Read one line
34     }
35     char *p = strtok(StrLine, " "); //Split string.
36     for (int i = 0; i < 2; i++)
37     {
38         p = strtok(NULL, " "); //Jump 'p' and 'sp'.
39     }
40     *maxNode = (int)atof(p) + 1; //Get the number of nodes.
41
42     Edge *edge = (Edge*)malloc(sizeof(Edge)*(*maxNode));

```



```

43     for (int i = 0; i < *maxNode; i++)
44         edge[i] = NULL;
45     fgets(StrLine, 1024, fp); //Read one line
46     while (!feof(fp))
47     {
48         if (StrLine[0] == 'a')
49         {
50             p = strtok(StrLine, " "); //Split the string.
51             int departure = (int)atof(strtok(NULL, " "));
52             int destination = (int)atof(strtok(NULL, " "));
53             int weight = (int)atof(strtok(NULL, " "));
54             edge[departure] = AddEdge(edge[departure], weight, destination);
55         }
56         fgets(StrLine, 1024, fp); //Read one line
57     }
58     printf("\nRead completely!\n");
59     fclose(fp);
60     return edge;
61 }
62
63
64
65 void DijkstraWithFib(Edge *edge, int maxNode, int sorNode, int numFind)
66 {
67     int curFind = 0;
68     FibHeap fibheap = InitialHeap();
69     FibNode *Distance = (FibNode*)malloc(sizeof(FibNode)*maxNode);
70     for (int i = 1; i < maxNode; i++)
71     {
72         Distance[i] = InitialFibNode(Infinity, i);
73     }
74     Edge pedge = edge[sorNode];
75     while (pedge != NULL) //Deal with the source node.
76     {
77         Distance[pedge->destination]->value = pedge->weight;
78         FibInsertNode(fibheap, Distance[pedge->destination]);
79         //Insert the nodes linked to source node to heap.
80         pedge = pedge->next;
81     }
82     while (IsFibEmpty(fibheap) == false)
83     {
84         int minNode = fibheap->min->vertex;
85         // min is the shortest node
86         //printf("%d is found with distance %d\n", minNode, Distance[minNode]->value);
87         curFind++;
88         if (curFind == numFind)
89             break;
90         FibDeleteMin(fibheap); //Get the nearest node.
91         pedge = edge[minNode];
92
93         while (pedge != NULL)
94         {
95             if (Distance[pedge->destination]->value == Infinity)
96                 //Check if the node is known.

```

```

97         {
98             Distance[pedge->destination]->value =
99             Distance[minNode]->value + pedge->weight;
100             FibInsertNode(fibheap, Distance[pedge->destination]);
101         }
102         else
103         {
104             if (Distance[minNode]->value + pedge->weight
105                 < Distance[pedge->destination]->value)
106                 //Relaxation operation
107                 FibDecrease(fibheap, Distance[pedge->destination],
108                     Distance[minNode]->value + pedge->weight);
109         }
110         pedge = pedge->next;
111     }
112 }
113 for (int i = 1; i < maxNode; i++) //Free memory.
114     free(Distance[i]);
115 free(Distance);
116 }
117
118 void DijkstraWithPair(Edge *edge, int maxNode, int sorNode, int numFind)
119 {
120     int curFind = 0;
121     PairHeap pairheap = NULL;
122     PairHeap *Distance = (PairHeap*)malloc(sizeof(PairHeap)*maxNode);
123     for (int i = 1; i < maxNode; i++)
124     {
125         Distance[i] = InitialPairHeap(Infinity, i);
126     }
127     Edge pedge = edge[sorNode]; //Get the nearest node.
128     while (pedge != NULL)
129     {
130         Distance[pedge->destination]->value = pedge->weight;
131         pairheap = PairInsert(pairheap, Distance[pedge->destination]);
132         //Insert the nodes linked to source node to heap.
133         pedge = pedge->next;
134     }
135     while (IsPairEmpty(pairheap) == false)
136     {
137         int minNode = pairheap->vertex; // min is the shortest node
138         //printf("%d is found with distance %d\n", minNode, Distance[minNode]->value);
139         curFind++;
140         if (curFind == numFind)
141             break;
142         pairheap = PairDeleteMin(pairheap);
143         pedge = edge[minNode];
144
145         while (pedge != NULL)
146         {
147             if (Distance[pedge->destination]->value == Infinity)
148                 //Check if the node is known.
149                 {
150                     Distance[pedge->destination]->value =

```

```

151         Distance[minNode]->value + pedge->weight;
152         pairheap = PairInsert(pairheap, Distance[pedge->destination]);
153     }
154     else
155     {
156         if (Distance[minNode]->value + pedge->weight
157             < Distance[pedge->destination]->value)
158             pairheap = PairDecrease(pairheap,
159                                     Distance[pedge->destination], Distance
160                                     [minNode]->value + pedge->weight);
161         //Relaxation operation
162     }
163     pedge = pedge->next;
164 }
165 }
166 for (int i = 1; i < maxNode; i++) //Free memory.
167     free(Distance[i]);
168 free(Distance);
169 }
170
171 void DijkstraWithBin(Edge *edge, int maxNode, int sorNode, int numFind)
172 {
173     int curFind = 0;
174     BinHeap binheap = NULL;
175     BinHeap *Distance = (BinHeap*)malloc(sizeof(BinHeap)*maxNode);
176     for (int i = 1; i < maxNode; i++)
177     {
178         Distance[i] = InitialBinNode(Infinity, i);
179     }
180     Edge pedge = edge[sorNode]; //Deal with the source node.
181     while (pedge != NULL)
182     {
183         Distance[pedge->destination]->value = pedge->weight;
184         binheap = BinInsert(binheap, Distance[pedge->destination]);
185         pedge = pedge->next;
186     }
187     while (IsBinEmpty(binheap) == false)
188     {
189         int minNode = BinGetMin(binheap)->vertex;
190         // min is the shortest node
191         //printf("%d is found with distance %d\n", minNode, Distance[minNode]->value);
192         curFind++;
193         if (curFind == numFind)
194             break;
195         binheap = BinDeleteMin(binheap); //Get the nearest node.
196         pedge = edge[minNode];
197
198         while (pedge != NULL)
199         {
200             if (Distance[pedge->destination]->value == Infinity)
201                 //Check if the node is known.
202             {
203                 Distance[pedge->destination]->value =
204                 Distance[minNode]->value + pedge->weight;

```

```

205         binheap = BinInsert(binheap, Distance[pedge->destination]);
206     }
207     else
208     {
209         if (Distance[minNode]->value + pedge->weight
210             < Distance[pedge->destination]->value)
211             BinDecrease(binheap, Distance[pedge->destination],
212                         Distance[minNode]->value + pedge->weight, Distance);
213         //Relaxation operation
214     }
215     pedge = pedge->next;
216 }
217 }
218 for (int i = 1; i < maxNode; i++) //Free memory.
219     free(Distance[i]);
220 free(Distance);
221 }

```
