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(logn),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:

```
typedef struct BinomialNode *BinHeap, *BinNode;
struct BinomialNode {
    int value;
    int vertex;
    int degree;
    BinHeap parent;
    BinHeap child;
    BinHeap sibling;
};
BinHeap InitialBinNode(int value, int vertex);
BinHeap BinLinked(BinHeap binheap, BinHeap child);
BinHeap BinCombineOrder(BinHeap binheap1, BinHeap binheap2);
BinHeap BinMerge(BinHeap binheap1, BinHeap binheap2);
BinHeap* BinGetMin2(BinHeap binheap);
BinHeap BinDeleteMin(BinHeap binheap);
BinHeap BinGetMin(BinHeap binheap);
BinHeap BinInsert(BinHeap binheap, BinNode binnode);
void    BinDecrease(BinHeap binheap, BinNode binnode,
int value, BinHeap* NodeArray);
bool    IsBinEmpty(BinHeap binheap);
```

Fibonacci heap:

```
typedef struct FibonacciNode *FibNode;
struct FibonacciNode{
    int vertex;
    int degree;
    int value;
    FibNode right;
    FibNode left;
    FibNode parent;
    FibNode child;
    bool mark;
};
typedef struct FibonacciHeap *FibHeap;
struct FibonacciHeap{
    FibNode min;
    int maxDegree;
    int keyNum;
    FibNode *cons;
};
FibHeap InitialHeap(void);
FibNode InitialFibNode(int value, int vertex);
void FibInsertNode(FibHeap fibheap, FibNode fibnode);
void FibAddBefore(FibNode fibnode1, FibNode fibnode2);
FibHeap FibHeapMerge(FibHeap fibheap1, FibHeap fibheap2);
bool IsFibEmpty(FibHeap fibheap);
FibNode FibHeapMin(FibHeap fibheap);
void FibDecrease(FibHeap fibheap, FibNode fibnode, int value);
```

```
void FibTranstoRoot(FibHeap fibheap, FibNode fibnode);
void FibDeleteMin(FibHeap fibheap);
void FibNeaten(FibHeap fibheap);
```

Pairing heap:

```
typedef struct PairingNode* PairHeap,*PairNode;
struct PairingNode {
    int vertex;
    int value;
    PairHeap child;
    PairHeap sibling;
    PairHeap Prev;
};
PairNode InitialPairHeap(int value, int vertex);
PairHeap PairMerge(PairHeap pairheap1, PairHeap pairheap2);
PairHeap PairInsert(PairHeap pairheap, PairNode pairnode);
PairHeap PairDeleteMin(PairHeap pairheap);
PairHeap CombineSiblings(PairHeap pairheap);
PairHeap PairDecrease(PairHeap pairheap, PairNode pairnode, int value);
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)

```
for every vertex $v$ in $V$ do
    $d_v \leftarrow \infty$; $p_v \leftarrow null$
    Insert($Q, v, d_v$)  /* initialize vertex priority in the priority queue */
$d_s \leftarrow 0$; Decrease($Q, s, d_s$)  /* update priority of $s$ with $d_s$ */
$V_T \leftarrow \emptyset$
for $i \leftarrow 0$ to $|V|-1$ do
    $u^* \leftarrow DeleteMin(Q)$  /* delete the minimum priority element */
    $V_T \leftarrow V_t \bigcup \{u^*\}$
    for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
        if $d_{u^*} + w(u^*, u) < d_u$
            $d_u \leftarrow d_{u^*} + w(u^*, u)$;  $p_u \leftarrow u^*$
            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.

```
Edge *ReadFile(int *maxNode)
{
    FILE *fp;
    char StrLine[150];
    if (file has no data) Open file error;
```

```
    Loading file...;
    while (!feof(fp) && StrLine[0] != 'p') Read one line;
    Split string;
    for (int i = 0; i < 2; i++) Jump 'p' and 'sp';
    Get the number of nodes;
    create edge[];
    Read one line;
    while (!feof(fp))
    {
        if (StrLine[0] == 'a')
        {
            Split the string.
            int departure = (int)atof(strtok(NULL, " "));
            int destination = (int)atof(strtok(NULL, " "));
            int weight = (int)atof(strtok(NULL, " "));
            edge[departure] = AddEdge(edge[departure], weight, destination);
        }
        Read one line
    }
    Read completely!;
    fclose(fp);
    return edge;
}
```

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.

```
int main(void)
{
    Read map from file;
    if (edge != NULL)
    {
        Enter sourse node between 1 and %d:\n;
        Choose source node;
        Check if the sourse node is legal;
        {
            You can only choose sorNode between 1 and %d ;
            scanf_s("%d", &sorNode);
        }
        while (true)
        {
            Enter the number of nodes you want to find;
            scanf_s("%d", &numFind);
            Check if the number of target node is legal;
            {
                You can only choose numFind between 1 and %d\n;
                continue;
            }
            if (all nodes)
                numFind = maxNode - 1;
            beginTime = clock();
                DijkstraWithFib;
            endTime = clock();
```

```
            printf
            beginTime = clock();
                DijkstraWithPair;
            endTime = clock();
            printf;
            beginTime = clock();
                DijkstraWithBin;
            endTime = clock();
            printf;
        }
    }
    system("pause");
}
```

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(vlogv+e).For the more important operations among the function,the time complexity of the findmin is Ω(lgn),the deletemin is Θ(lgn),the insertion is Ω(lgn),the decrease and merge operations are all Θ(lgn).After overall comprehensive calculation，the time complexity is O(vlogv+e).

As for binomial heap,the time complexity is O(elogv).For the more important operations among the function,the time complexity of the findmin is Θ(1),the deletemin is O(logn),the insertion,decrease and merge operations are all Θ(1). After overall comprehensive calculation，the time complexity is O(elogv).

As for pairing heap,the time complexity is O(vlogv+e).For the more important operations among the function,the time complexity of the findmin,merge and insert is Θ(1),the deletemin and decrease is O(logn).After overall comprehensive calculation，the time complexity is O(vlogv+e).

The space complexity is O(e+v).

5 Author list

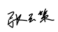
coding:Zhang Yunce
report:Cao Haowei

Declaration

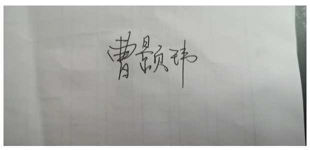
We hereby declare that all the work done in this project titled ”Shortest Path Algorithm with Heaps” is of our independent effort as a group.

6 Signatures

Zhang Yunce



Cao Haowei



A Source Code (if required)

main.c

```
#include "Dijkstra.h"
#include "Fibonacci heap.h"
#include "Pairing heap.h"
#include "Binomial heap.h"
#include <time.h>

int main(void)
{
    int maxNode, sorNode, numFind;
    int beginTime, endTime;
    Edge *edge = ReadFile(&maxNode); //Read map from file.
    if (edge != NULL)
    {
        printf("Enter sourse node between 1 and %d:\n", maxNode - 1);
        scanf_s("%d", &sorNode); //Choose source node.
        while (sorNode > maxNode - 1 || sorNode < 1)
        //Check if the sourse node is legal.
        {
            printf("You can only choose sorNode
            between 1 and %d \nPlease input again:", maxNode - 1);
            scanf_s("%d", &sorNode);
        }
        while (true)
        {
```

```
            printf("Enter the number of nodes you want to find
            (0 means all nodes):(Program will find numFind nearest nodes)
            \n");
            scanf_s("%d", &numFind);
            if (numFind > maxNode - 1  || numFind < 0)
            //Check if the number of target node is legal.
            {
                printf("You can only choose numFind
                between 1 and %d\n", maxNode - 1);
                continue;
            }
            if (numFind == 0) //0 means all nodes.
                numFind = maxNode - 1;
            beginTime = clock();
                DijkstraWithFib(edge, maxNode, sorNode, numFind);
            endTime = clock();
            printf("Fibonacci heap || Number of nodes
            : %d || Time: %dms\n", numFind, endTime - beginTime);
            beginTime = clock();
                DijkstraWithPair(edge, maxNode, sorNode, numFind);
            endTime = clock();
            printf("Pairing heap   || Number of nodes
            : %d || Time: %dms\n", numFind, endTime - beginTime);
            beginTime = clock();
                DijkstraWithBin(edge, maxNode, sorNode, numFind);
            endTime = clock();
            printf("Binomial heap  || Number of nodes
            : %d || Time: %dms\n", numFind, endTime - beginTime);
        }
    }
    system("pause");
}
```

Binomial heap.h

```
#pragma once
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
typedef struct BinomialNode *BinHeap, *BinNode;
struct BinomialNode {
    int value;
    int vertex;
    int degree;
    BinHeap parent;
    BinHeap child;
    BinHeap sibling;
};
BinHeap InitialBinNode(int value, int vertex); //Create a new node.
BinHeap BinLinked(BinHeap binheap, BinHeap child);
//Link child node with binomial heap.
BinHeap BinCombineOrder(BinHeap binheap1, BinHeap binheap2);
//Merge two lists in increase order.
BinHeap BinMerge(BinHeap binheap1, BinHeap binheap2);
```

```
//Merge two lists and combine subtrees with same degrees
BinHeap* BinGetMin2(BinHeap binheap);
//Get the min node and its previous node.
BinHeap BinDeleteMin(BinHeap binheap);
//Delete min node.
BinHeap BinGetMin(BinHeap binheap);
//Get the min node.
BinHeap BinInsert(BinHeap binheap, BinNode binnode);
//Insert a new node to heap.
void    BinDecrease(BinHeap binheap, BinNode binnode,
int value, BinHeap* NodeArray);
//Decrease the value of one certain node.
bool    IsBinEmpty(BinHeap binheap);
//Judge if the heap is empty.
```

Binomial heap.c

```
#include "Binomial heap.h"


BinHeap InitialBinNode(int value, int vertex)
{
    BinHeap binheap = (BinHeap)malloc(sizeof(struct BinomialNode));
    binheap->child = NULL;
    binheap->parent = NULL;
    binheap->sibling = NULL;
    binheap->degree = 0;
    binheap->value = value;
    binheap->vertex = vertex;
    return binheap;
}

BinHeap BinLinked(BinHeap binheap, BinHeap child)
{
    child->parent = binheap;
    child->sibling = binheap->child;
    binheap->child = child;
    binheap->degree++;
    return binheap;
}

BinHeap BinCombineOrder(BinHeap binheap1, BinHeap binheap2)
{
    BinNode ptr1 = binheap1, ptr2 = binheap2;
    BinHeap binBegin = NULL;
    BinHeap *binheap = &binBegin;
    while (ptr1 != NULL && ptr2 != NULL)
    //Merge two lists in increase order.
    {
        if (ptr1->degree <= ptr2->degree)
        {
            *binheap = ptr1;
            ptr1 = ptr1->sibling;
        }
```

```
        else
        {
            *binheap = ptr2;
            ptr2 = ptr2->sibling;
        }
        binheap = &((*binheap)->sibling);
    }
    if (ptr1 == NULL)
        *binheap = ptr2;
    else
        *binheap = ptr1;
    return binBegin;
}

BinHeap BinMerge(BinHeap binheap1, BinHeap binheap2)
{
    BinHeap binheap = BinCombineOrder(binheap1, binheap2);
    //First step, merge them in increase order.
    if (binheap == NULL)
        return NULL;
    BinNode p_prev, p, p_sib;
    p_prev = NULL;
    p = binheap;
    p_sib = p->sibling;
    while (p_sib != NULL)
    {
        if (p->degree != p_sib->degree || p_sib->sibling != NULL
        && p_sib->degree == p_sib->sibling->degree)
        //In this case, forms a(p)-b or a(p)-a-a will be skipped.
        {
            p_prev = p;
            p = p_sib;
        }
        else  //Need to merge.
        {
            if (p->value <= p_sib->value)
            //In this case, a(p)-b, b will be a child of
            a.(value(a)<=value(b))
            {
                p->sibling = p_sib->sibling;  //p_sib will be romoved.
                p = BinLinked(p, p_sib);
            }
            else
            //In this case, a(p)-b, a will be a child of
            b.(value(a)>value(b))
            {
                if (p_prev == NULL)
                    binheap = p_sib;
                else
                    p_prev->sibling = p_sib;
                p_sib = BinLinked(p_sib, p);
                //p will be romeved.
                p = p_sib;
            }
```

```c
        }
        p_sib = p->sibling;
    }
    return binheap;
}

BinHeap* BinGetMin2(BinHeap binheap)
{
    if (binheap->sibling == NULL)
        return NULL;
    else
    {
        BinHeap* p = (BinHeap*)malloc(sizeof(BinHeap) * 2);
        //Used to storage min position.
        p[0] = NULL;
        p[1] = binheap;
        BinNode ptr, ptr_prev;  //Used to travel the link;
        ptr_prev = NULL;
        ptr = binheap;
        while (ptr!= NULL)
        {
            if (ptr->value < p[1]->value)
            {
                p[0] = ptr_prev;
                p[1] = ptr;
            }
            ptr_prev = ptr;
            ptr = ptr->sibling;
        }
        return p;
    }
}

BinHeap BinDeleteMin(BinHeap binheap)
{
    if (binheap == NULL)
        return NULL;
    else
    {
        BinHeap* min = BinGetMin2(binheap);
        //Get the min node and its previous node.
        if (min == NULL)
        //Check if the min node is the first node.
            return binheap->child;
        BinHeap binMin_prev = min[0], binMin = min[1];
        if (binMin_prev == NULL)
            binheap = binheap->sibling;
        else
            binMin_prev->sibling = binMin->sibling;
        BinNode childList = binMin->child;
        BinNode reverse = NULL;
        BinNode temp;
        while (childList != NULL)    //Reverse the list.
        {
```

```c
            temp = childList->sibling;
            if (reverse == NULL)
            {
                reverse = childList;
                reverse->sibling = NULL;
            }
            else
            {
                childList->sibling = reverse;
                reverse = childList;
            }
            childList = temp;
        }
        free(min);
        return BinMerge(binheap, reverse);
    }
}

BinHeap BinGetMin(BinHeap binheap)
{
    if (binheap == NULL)
        return NULL;
    else
    {
        BinNode ptr = binheap;
        BinNode temp = ptr;
        while (ptr != NULL)
        {
            if (ptr->value < temp->value)
                temp = ptr;
            ptr = ptr->sibling;
        }
        return temp;
    }
}

BinHeap BinInsert(BinHeap binheap, BinNode binnode)
{
    return BinMerge(binheap, binnode);
}

void BinDecrease(BinHeap binheap, BinNode binnode,
int value, BinHeap* NodeArray)
{
    binnode->value = value;
    BinNode parent, child;
    parent = binnode->parent;
    child = binnode;
    while (parent != NULL && parent->value > child->value)
    //Exchange the position between child and parent.
    {
        int temp_value, temp_vertex;
        BinNode temp = NodeArray[child->vertex];
        //Change the Distance array.
```

```
        NodeArray[child->vertex] = NodeArray[parent->vertex];
        NodeArray[parent->vertex] = temp;
        temp_value = child->value;
        temp_vertex = child->vertex;
        child->value = parent->value;
        child->vertex = parent->vertex;
        parent->value = temp_value;
        parent->vertex = temp_vertex;
        child = parent;
        parent = parent->parent;
    }
}

bool IsBinEmpty(BinHeap binheap)
{
    return binheap == NULL;
}
```

Fibonacci heap.h

```
#pragma once
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<math.h>

typedef struct FibonacciNode *FibNode;
struct FibonacciNode{
    int vertex;
    int degree;
    int value;
    FibNode right;
    FibNode left;
    FibNode parent;
    FibNode child;
    bool mark;
};

typedef struct FibonacciHeap *FibHeap;
struct FibonacciHeap{
    FibNode min;
    int maxDegree;
    int keyNum;
    FibNode *cons;
};

FibHeap InitialHeap(void);  //Initial the heap.
FibNode InitialFibNode(int value, int vertex);
//Create a new node.
void FibInsertNode(FibHeap fibheap, FibNode fibnode);
//Insert a node into fibonacci heap.
void FibAddBefore(FibNode fibnode1, FibNode fibnode2);
//add node 1 before node2
FibHeap FibHeapMerge(FibHeap fibheap1, FibHeap fibheap2);
```

```c
bool IsFibEmpty(FibHeap fibheap);
FibNode FibHeapMin(FibHeap fibheap);
//Return the least node.
void FibDecrease(FibHeap fibheap, FibNode fibnode, int value);
//Change the value of a certain node.
void FibTranstoRoot(FibHeap fibheap, FibNode fibnode);
//Move the node to root.
void FibDeleteMin(FibHeap fibheap);
void FibNeaten(FibHeap fibheap);
//Merge the nodes of same degrees.
```

Fibonacci heap.c

```c
#include"Fibonacci heap.h"
#include<time.h>

FibHeap InitialHeap(void)
{
    FibHeap fibheap = (FibHeap)malloc(sizeof(struct FibonacciHeap));
    fibheap->min = NULL;
    fibheap->maxDegree = 0;
    fibheap->keyNum = 0;
    fibheap->cons = NULL;
    return fibheap;
}

FibNode InitialFibNode(int value, int vertex)
{
    FibNode fibnode = (FibNode)malloc(sizeof(struct FibonacciNode));
    fibnode->child = NULL;
    fibnode->parent = NULL;
    fibnode->left = fibnode;
    fibnode->right = fibnode;
    fibnode->vertex = vertex;
    fibnode->value = value;
    fibnode->degree = 0;
    fibnode->mark = false;
    return fibnode;
}

void FibInsertNode(FibHeap fibheap, FibNode fibnode)
{
    if (fibnode == NULL)  //Fibnode is illeagal
        return;
    else if (fibheap->keyNum == 0)
    //Fibnode is the first member of the fibonacci heap.
        fibheap->min = fibnode;
    else
    {
        FibNode curMin = fibheap->min;
        FibAddBefore(fibnode, curMin);
        //Insert fibnode to the left of the min node.
        if (fibnode->value < curMin->value)
            fibheap->min = fibnode;
```

```
    }
    fibheap->keyNum++; //Update
}

void FibAddBefore(FibNode fibnode1, FibNode fibnode2)
//-fibnode1- and -fibnode2- become -fibnode1-fibnode2-
{
    fibnode1->left = fibnode2->left;
    fibnode2->left->right = fibnode1;
    fibnode1->right = fibnode2;
    fibnode2->left = fibnode1;
}

FibHeap FibHeapMerge(FibHeap fibheap1, FibHeap fibheap2)
{
    if (fibheap1 == NULL) //Check if fibheap1 or fibheap2 is null.
        return fibheap2;
    else if (fibheap2 == NULL)
        return fibheap1;
    else
    {
        if (fibheap1->maxDegree < fibheap2->maxDegree)
        //Let fibheap1's maxDegree bigger than fibheap2.
        {
            FibHeap temp = fibheap1;
            fibheap1 = fibheap2;
            fibheap2 = temp;
        }
        if (fibheap1->min == NULL)
        {
            fibheap1->keyNum = fibheap2->keyNum;
            fibheap1->maxDegree = fibheap2->maxDegree;
        }
        else if (fibheap2->min != NULL) //Merge two fibonacci heap.
        {
            FibNode temp;
            FibNode fibnode1 = fibheap1->min, fibnode2 = fibheap2->min;
            temp = fibnode1->right;
            fibnode1->right = fibnode2->right;
            fibnode2->right->left = fibnode1;
            fibnode2->right = temp;
            temp->left = fibnode2;
            if (fibnode1->value > fibnode2->value)
                fibheap1->min = fibheap2->min;
            fibheap1->maxDegree += fibheap2->maxDegree;
        }
        free(fibheap2->cons);
        free(fibheap2);
        return fibheap1; //fibheap2 has been merged into fibheap1.
    }
}

bool IsFibEmpty(FibHeap fibheap)
{
```

```c
    if (fibheap->min  == NULL)
        return true;
    else
        return false;
}

FibNode FibHeapMin(FibHeap fibheap)
{
    return fibheap->min;
}

void FibDecrease(FibHeap fibheap, FibNode fibnode, int value)
{
    if (fibheap == NULL || fibheap->min == NULL || fibnode == NULL)
        return;
    fibnode->value = value;
    FibNode parent = fibnode->parent;
    if (parent != NULL && parent->value > fibnode->value)
    {
        FibTranstoRoot(fibheap, fibnode);
        //if fibnode's value is bigger than its parent, move it to root.
    }
    if (fibheap->min->value > fibnode->value)
        fibheap->min = fibnode;
}

void FibTranstoRoot(FibHeap fibheap, FibNode fibnode)
{
    FibNode parent = fibnode->parent;
    if (parent == NULL)
        return;
    fibnode->right->left = fibnode->left;
    //Delete fibnode from left-fibnode-right
    fibnode->left->right = fibnode->right;
    if (fibnode->right == fibnode)
    //Update parent node
        parent->child = NULL;
    else
        parent->child = fibnode->right;
    parent->degree--;
    FibAddBefore(fibnode, fibheap->min);
    //Update fibnode information
    fibnode->parent = NULL;
    fibnode->mark = false;
    if (parent->mark == false)
    //Determine whether the fibnode needs to be moved to the root
        parent->mark = true;
    else
        FibTranstoRoot(fibheap, parent);
}

void FibDeleteMin(FibHeap fibheap)
{
    if (fibheap == NULL || fibheap->min == NULL)
```

```
        return;
    FibNode min = fibheap->min;
    while (min->child != NULL)  //Add all children of min to root-link
    {
        FibNode child = min->child;
        child->right->left = child->left;
        //Delete min from left-fibnode-right
        child->left->right = child->right;
        if (child->right == child)
            min->child = NULL;
        else
            min->child = child->right;
        FibAddBefore(child, min);
        child->parent = NULL;
    }
    min->right->left = min->left;  //Delete min
    min->left->right = min->right;
    fibheap->min = (fibheap->keyNum == 1) ? NULL : min->right;
    fibheap->keyNum--;
    FibNeaten(fibheap);
}

void FibNeaten(FibHeap fibheap)
{
    if (fibheap == NULL || fibheap->min == NULL)
        return;
    fibheap->maxDegree = (int)(log2((double)(fibheap->keyNum)) + 1);
    fibheap->cons = (FibNode*)malloc(sizeof(FibNode)*
    (fibheap->maxDegree + 1));
    //Allocate an array to combine heaps with the same degrees.
    for (int i = 0; i < fibheap->maxDegree + 1; ++i)
        fibheap->cons[i] = NULL;
    while (fibheap->min != NULL)  //Combine heaps with the same degrees.
    {
        FibNode curMin = fibheap->min;  //Get the current heap.
        int curDegree = curMin->degree;
        if (curMin->right == curMin)
            fibheap->min = NULL;
        else
        {
            fibheap->min = curMin->right;
            curMin->right->left = curMin->left;
            curMin->left->right = curMin->right;
        }
        curMin->left = curMin->right = curMin;  //Isolate this node.

        while (fibheap->cons[curDegree] != NULL)
        //Add current heap to array cons. If there is a collision.
        {
            FibNode SameDegree = fibheap->cons[curDegree];
            if (SameDegree->value < curMin->value)
            {
                FibNode temp = SameDegree;
                SameDegree = curMin;
```

```c
                curMin = temp;
            }
            SameDegree->right->left = SameDegree->left;
            SameDegree->left->right = SameDegree->right;
            if (curMin->child == NULL)
            //Make SameDegree be curMin's children
                curMin->child = SameDegree;
            else
                FibAddBefore(SameDegree, curMin->child);
            SameDegree->parent = curMin;
            SameDegree->mark = false;
            curMin->degree++;
            fibheap->cons[curDegree] = NULL;
            curDegree++;
        }
        fibheap->cons[curDegree] = curMin;
    }
    fibheap->min = NULL;
    for (int i = 0; i < fibheap->maxDegree + 1; ++i)
    //Copy the heap from cons to fibheap.
    {
        if (fibheap->cons[i] != NULL)
        {
            if (fibheap->min == NULL)
                fibheap->min = fibheap->cons[i];
            else
            {
                FibAddBefore(fibheap->cons[i], fibheap->min);
                if (fibheap->cons[i]->value < fibheap->min->value)
                    fibheap->min = fibheap->cons[i];
            }
        }
    }
    free(fibheap->cons); //Free memory.
}
```

Pairing heap.h

```c
#pragma once
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

typedef struct PairingNode* PairHeap,*PairNode;
struct PairingNode {
    int vertex;
    int value;
    PairHeap child;
    PairHeap sibling;
    PairHeap Prev;
};

PairNode InitialPairHeap(int value, int vertex);
//Initial the heap(Create a node).
```

```c
PairHeap PairMerge(PairHeap pairheap1, PairHeap pairheap2);
//Merge two pairing heap.
PairHeap PairInsert(PairHeap pairheap, PairNode pairnode);
PairHeap PairDeleteMin(PairHeap pairheap);
//Delete the min node and call CombineSiblings().
PairHeap CombineSiblings(PairHeap pairheap);
//Merge the nodes in one lists.
PairHeap PairDecrease(PairHeap pairheap, PairNode pairnode, int value);
bool IsPairEmpty(PairHeap pairheap);
```

Pairing heap.c

```c
#include "Pairing heap.h"

PairNode InitialPairHeap(int value, int vertex)
{
    PairNode pairnode = (PairNode)malloc(sizeof(struct PairingNode));
    pairnode->child = NULL;
    pairnode->Prev = NULL;
    pairnode->sibling = NULL;
    pairnode->value = value;
    pairnode->vertex = vertex;
    return pairnode;
}

PairHeap PairMerge(PairHeap pairheap1, PairHeap pairheap2)
{
    if (pairheap1 == NULL)
        return pairheap2;
    else if (pairheap2 == NULL)
        return pairheap1;
    else  //Add pairnode2 to pairnode1.
    {
        if(pairheap1->value > pairheap2->value)
        //Let node1_pre linked to node2
        {
            PairNode heap1_prev = pairheap1->Prev;
            pairheap2->Prev = heap1_prev;
            if (heap1_prev != NULL)
            {
                if (heap1_prev->child == pairheap1)
                    heap1_prev->child = pairheap2;
                else
                    heap1_prev->sibling = pairheap2;
            }
            PairNode temp = pairheap1;
            //Exchange node1 and node2 to unified form with "else" part.
            pairheap1 = pairheap2;
            pairheap2 = temp;
        }
        else
        //Let node2->sibling linked to node1
        {
            PairNode heap2_sibling = pairheap2->sibling;
```

```
            pairheap1->sibling = heap2_sibling;
            if (pairheap1->sibling != NULL)
                heap2_sibling->Prev = pairheap1;
        }
        PairNode heap1_child = pairheap1->child;
        pairheap1->child = pairheap2;
        pairheap2->Prev = pairheap1;
        pairheap2->sibling = heap1_child;
        if (pairheap2->sibling != NULL)
            heap1_child->Prev = pairheap2;
        return pairheap1;
    }
}

PairHeap PairInsert(PairHeap pairheap, PairNode pairnode)
{
    if (pairheap == NULL)
        return pairnode;
    else
        return PairMerge(pairheap, pairnode);
        //Call PairMerge to merge them.
}

PairHeap PairDeleteMin(PairHeap pairheap)
{
    if (pairheap == NULL || pairheap->child == NULL)
        return NULL;
    else
    {
        PairHeap firstSibling = pairheap->child;
        firstSibling->Prev = NULL;
        return CombineSiblings(firstSibling);
        //After we delete the min node, we need to Combine the sibling nodes.
    }
}

PairHeap CombineSiblings(PairHeap pairheap)
{
    PairNode heap_next = pairheap->sibling;
    PairNode porign = pairheap;
    while (pairheap != NULL)  //The first round from left to right.
    {
        porign = PairMerge(pairheap, heap_next);
        pairheap = porign->sibling;
        heap_next = (pairheap == NULL ? NULL : pairheap->sibling);
        if (heap_next == NULL)
            break;
    }
    if (pairheap != NULL)
        porign = pairheap;
    PairNode heap_prev = porign->Prev;
    while (heap_prev != NULL) //The second round from ight to left.
    {
        porign = PairMerge(heap_prev, porign);
```

```
        heap_prev = porign->Prev;
    }
    return porign;
}

PairHeap PairDecrease(PairHeap pairheap, PairNode pairnode, int value)
{
    pairnode->value = value;
    if (pairnode != pairheap)
    {
        PairNode node_prev = pairnode->Prev;
        if (node_prev->child == pairnode)
            node_prev->child = pairnode->sibling;
        else
            node_prev->sibling = pairnode->sibling;
        if (pairnode->sibling != NULL)
            pairnode->sibling->Prev = node_prev;
        pairnode->Prev = pairnode->sibling = NULL;
        pairheap = PairMerge(pairnode, pairheap);
    }
    return pairheap;
}

bool IsPairEmpty(PairHeap pairheap)
{
    if (pairheap == NULL)
        return true;
    else
        return false;
}
```

Dijkstra.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include<string.h>
#include<limits.h>
#include"Fibonacci heap.h"
#include"Pairing heap.h"
#include"Binomial heap.h"

#define Infinity INT_MAX

typedef struct EdgeNode* Edge;
struct EdgeNode{
    int weight;   //The distance between two nodes.
    int destination;
    Edge next;
};

Edge AddEdge(Edge start, int weight, int destination);
//Create n list to storage information from map.(n is the number of nodes.)
Edge *ReadFile(int *maxNode);
//Read map from a file.
```

```
void DijkstraWithFib(Edge *edge, int maxNode,
int sorNode, int numFind);
//Implement Dijstra algorithm with Fibonacci heap.
void DijkstraWithPair(Edge *edge, int maxNode,
int sorNode, int numFind);
//Implement Dijstra algorithm with Pairing heap.
void DijkstraWithBin(Edge *edge, int maxNode,
int sorNode, int numFind);
//Implement Dijstra algorithm with Binomail heap.
```

Dijkstra.c

```
#include"Dijkstra.h"
Edge AddEdge(Edge start, int weight, int destination)
{
    Edge newEdge = (Edge)malloc(sizeof(struct EdgeNode));
    newEdge->weight = weight;
    newEdge->destination = destination;
    if (start == NULL)  //The orignal  list is empty.
    {
        newEdge->next = NULL;
        start = newEdge;
    }
    else
    //The orignal  list is not empty. Insert the new node in the head.
    {
        newEdge->next = start;
        start = newEdge;
    }
    return start;
}

Edge *ReadFile(int *maxNode)
{
    FILE *fp;
    char StrLine[150];
    if ((fp = fopen("map.txt", "r")) == NULL)
    {
        printf("Open file error!\n");
        return NULL;
    }
    printf("Loading file...");
    while (!feof(fp) && StrLine[0] != 'p')
    {
        fgets(StrLine, 1024, fp);  //Read one line
    }
    char *p = strtok(StrLine, " ");  //Split string.
    for (int i = 0; i < 2; i++)
    {
        p = strtok(NULL, " "); //Jump 'p' and 'sp'.
    }
    *maxNode = (int)atof(p) + 1; //Get the number of nodes.

    Edge *edge = (Edge*)malloc(sizeof(Edge)*(*maxNode));
```

```
    for (int i = 0; i < *maxNode; i++)
        edge[i] = NULL;
    fgets(StrLine, 1024, fp);  //Read one line
    while (!feof(fp))
    {
        if (StrLine[0] == 'a')
        {
            p = strtok(StrLine, " "); //Split the string.
            int departure = (int)atof(strtok(NULL, " "));
            int destination = (int)atof(strtok(NULL, " "));
            int weight = (int)atof(strtok(NULL, " "));
            edge[departure] = AddEdge(edge[departure], weight, destination);
        }
        fgets(StrLine, 1024, fp);  //Read one line
    }
    printf("\nRead completely!\n");
    fclose(fp);
    return edge;
}



void DijkstraWithFib(Edge *edge, int maxNode, int sorNode, int numFind)
{
    int curFind = 0;
    FibHeap fibheap = InitialHeap();
    FibNode *Distance = (FibNode*)malloc(sizeof(FibNode)*maxNode);
    for (int i = 1; i < maxNode; i++)
    {
        Distance[i] = InitialFibNode(Infinity, i);
    }
    Edge pedge = edge[sorNode];
    while (pedge != NULL)  //Deal with the sourse node.
    {
        Distance[pedge->destination]->value = pedge->weight;
        FibInsertNode(fibheap, Distance[pedge->destination]);
        //Insert the nodes linked to sourse node to heap.
        pedge = pedge->next;
    }
    while (IsFibEmpty(fibheap) == false)
    {
        int minNode = fibheap->min->vertex;
        // min is the shortest node
        //printf("%d is found with distance %d\n", minNode, Distance[minNode]->value);
        curFind++;
        if (curFind == numFind)
            break;
        FibDeleteMin(fibheap); //Get the nearest node.
        pedge = edge[minNode];

        while (pedge != NULL)
        {
            if (Distance[pedge->destination]->value == Infinity)
            //Check if the node is known.
```

```
            {
                Distance[pedge->destination]->value =
                Distance[minNode]->value + pedge->weight;
                FibInsertNode(fibheap, Distance[pedge->destination]);
            }
            else
            {
                if (Distance[minNode]->value + pedge->weight
                < Distance[pedge->destination]->value)
                //Relaxation operation
                    FibDecrease(fibheap, Distance[pedge->destination],
                    Distance[minNode]->value + pedge->weight);
            }
            pedge = pedge->next;
        }
    }
    for (int i = 1; i < maxNode; i++) //Free memory.
        free(Distance[i]);
    free(Distance);
}

void DijkstraWithPair(Edge *edge, int maxNode, int sorNode, int numFind)
{
    int curFind = 0;
    PairHeap pairheap = NULL;
    PairHeap *Distance = (PairHeap*)malloc(sizeof(PairHeap)*maxNode);
    for (int i = 1; i < maxNode; i++)
    {
        Distance[i] = InitialPairHeap(Infinity, i);
    }
    Edge pedge = edge[sorNode]; //Get the nearest node.
    while (pedge != NULL)
    {
        Distance[pedge->destination]->value = pedge->weight;
        pairheap = PairInsert(pairheap, Distance[pedge->destination]);
        //Insert the nodes linked to sourse node to heap.
        pedge = pedge->next;
    }
    while (IsPairEmpty(pairheap) == false)
    {
        int minNode = pairheap->vertex; // min is the shortest node
        //printf("%d is found with distance %d\n", minNode, Distance[minNode]->value);
        curFind++;
        if (curFind == numFind)
            break;
        pairheap = PairDeleteMin(pairheap);
        pedge = edge[minNode];

        while (pedge != NULL)
        {
            if (Distance[pedge->destination]->value == Infinity)
            //Check if the node is known.
            {
                Distance[pedge->destination]->value =
```

```
                Distance[minNode]->value + pedge->weight;
                pairheap = PairInsert(pairheap, Distance[pedge->destination]);
            }
            else
            {
                if (Distance[minNode]->value + pedge->weight
                < Distance[pedge->destination]->value)
                    pairheap = PairDecrease(pairheap,
                    Distance[pedge->destination], Distance
                    [minNode]->value + pedge->weight);
                    //Relaxation operation
            }
            pedge = pedge->next;
        }
    }
    for (int i = 1; i < maxNode; i++) //Free memory.
        free(Distance[i]);
    free(Distance);
}

void DijkstraWithBin(Edge *edge, int maxNode, int sorNode, int numFind)
{
    int curFind = 0;
    BinHeap binheap = NULL;
    BinHeap *Distance = (BinHeap*)malloc(sizeof(BinHeap)*maxNode);
    for (int i = 1; i < maxNode; i++)
    {
        Distance[i] = InitialBinNode(Infinity, i);
    }
    Edge pedge = edge[sorNode]; //Deal with the sourse node.
    while (pedge != NULL)
    {
        Distance[pedge->destination]->value = pedge->weight;
        binheap = BinInsert(binheap, Distance[pedge->destination]);
        pedge = pedge->next;
    }
    while (IsBinEmpty(binheap) == false)
    {
        int minNode = BinGetMin(binheap)->vertex;
        // min is the shortest node
        //printf("%d is found with distance %d\n", minNode, Distance[minNode]->value);
        curFind++;
        if (curFind == numFind)
            break;
        binheap = BinDeleteMin(binheap); //Get the neareast node.
        pedge = edge[minNode];

        while (pedge != NULL)
        {
            if (Distance[pedge->destination]->value == Infinity)
            //Check if the node is known.
            {
                Distance[pedge->destination]->value =
                Distance[minNode]->value + pedge->weight;
```

```
                    binheap = BinInsert(binheap, Distance[pedge->destination]);
                }
                else
                {
                    if (Distance[minNode]->value + pedge->weight
                    < Distance[pedge->destination]->value)
                        BinDecrease(binheap, Distance[pedge->destination],
                        Distance[minNode]->value + pedge->weight,Distance);
                        //Relaxation operation
                }
                pedge = pedge->next;
            }
        }
    for (int i = 1; i < maxNode; i++)  //Free memory.
        free(Distance[i]);
    free(Distance);
}
```