Shortest Path Algorithm with Heaps

Group 1 Zhao Yilei, Wang Kedi, Shen Yunfeng

Date: 2022-03-27

Chapter 1: Introduction

In this project, we're required to optimize the algorithm of Dijkstra to find the shortest path. We downloaded our samples from http://www.dis.uniroma1.it/chalange9/download.shtml. To make it easy, we firstly use a program of Dijkstra's algorithm without optimizaiton. And then we use 2 different types of heap, Fibonacci heap and binomial heap.

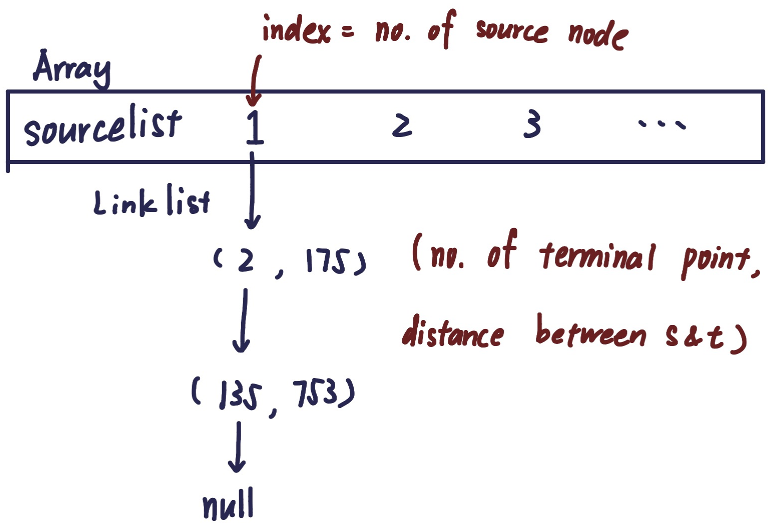
The heaps are used to deal with the operation of finding the minimum distance to the adjacent and not visited node. The heap has a much smaller time complexity in deleting the minimum, relaxing all nodes and so on.

We're supposed to use our samples to compare the function of Dijkstra of two kinds of optimizations. We will use the time spent as a criterion for comparison between Fibonacci heap and Binomial heap.

Chapter 2: Data Structure / Algorithm Specification

2.1 Dijkstra Algorithm

Data Structure: Adjacency List - Array & Link List



index = no. of source node
Array
sourcelist
1
2
3
...
Link list
(2 , 175)
(no. of terminal point, distance between s&t)
(135 , 753)
null

```
struct Node               //infomation about source point - array
{
    linknode *linklist;   //linklist of terminal points and distance

    int min_dis;          //shortest distance given point -》 this point
    int visited;          //0 - unvisited, 1 - visited

    pFibNode FNode;       //point to node in Febonacci Heap
    hNode BNode;          //point to node in binomial heap
}sourcelist[MAX_NODE];    //index is the No. of source point

struct LinkNode           //link list node of terminal point
{
    int t;                //No. of terminal point
    int dis;              //dsitance sorce - teminal
    struct LinkNode* next;   //next teminal
};
```

Specification:

Build Adjacency List

Before using Dijkstra Algorithm to find the shortest path between two given points, we should build the adjacency list in advance. By scanning the txt file row by row, we can insert the edge into the linklist by dividing the input into 3 cases —— all visited, 1 visited 1 unvisited, all unvisited.

```
void BuildLinkList()
{
    //initialization
    mark all the node in array sourcelist as unvisited;

    //scan the information file row by row
    while(is not the end of file)
    {
        scan No. of source & terminal, the distance;
        if(all unvisited)
        {
            mark s&t as visited, min distance as infinity;
            link s->t & t->s;
        }
        if(1 visited 1 unvisited)
        {
            mark unvisited one as visited, min distance as infinity;
            link visited to unvisited;
            add unvisited to visited's linklist;
        }
        if(all visited)
        {
            add to each other's linklist;
        }
    }
}
```

Dijkstra Algorithm

It can be divided into three points: initialization, find min and relax. In initialization, we mark every node as unvisited, and the min distance as infinity(except the source node, its min distance would be 0). In find min, we use heaps to lessen the cost of time, and after using the node, we should delete it from the heap. In relax, we traversal the linklist of the selected point to find all the nodes adjacent to it, and insert those unvisited one into heap & decrease the key value of heap node if we find a shorter path.

```
void Dijkstra(int source, int terminal)
{
    //initiallization
    mark every node as unvisited;
    make min distance as infinity(except source);
    make min distance of source as 0;
    insert the source point into heap;

    while(the heap is note empty)
    {
        find the node has minimum distance;
        if(visited) delete it from heap and continue;
        else
        {
            mrak it as visited;
            delete it from heap;

            //travel its linklist
            while(the linklist is not empty)
            {
                if(current node hasn't been visited) insert it into
heap;
                if(current node has been visited
                && we find a shorter path)
                   decrease its key value to shorter path length;
            }
        }
        visit next link node;
    }
}
```

2.2 Binomial-Heap

Data Structure

```
struct hNode
{
    int num;//the number of the node on the map
    int weight;
    int degree;//the number of edges
    struct hNode *p;
    struct hNode *child;//left-child
    struct hNode *sibling;
};
typedef struct hNode * hNode;
typedef struct heap
{
    struct hNode * root;
}* binomialHeap;
```

Functions:

```
struct hNode{};
struct heap{};
typedef struct heap  *binomialHeap;
hNode create_hNode(int num, int weight);
binomialHeap createBH();
hNode bhMinimum(binomialHeap h);
```

```
void bhLink(hNode node1, hNode node2);//link node1 and node2 together,
node2 becomes a root.
hNode bhMerge(binomialHeap h1, binomialHeap h2);//merge h1,h2
binomialHeap bhUnion(binomialHeap * h1, binomialHeap * h2);//
binomialHeap bhInsert(binomialHeap * h1, int num, int weight);//insert
node x into bh
binomialHeap bhExtractMin(binomialHeap * h);//find the min
void bhDecreaseKey(binomialHeap h, hNode x, int key);
void bhDelete(binomialHeap h, hNode x);
void traverseBH(hNode t);//traverse the BH
```

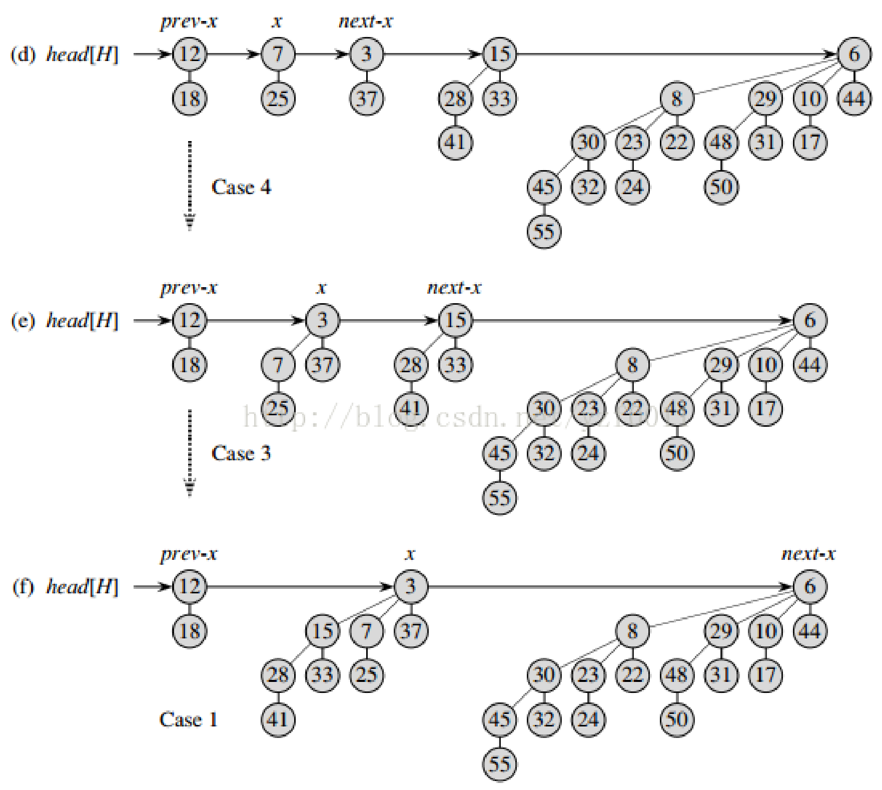
Insertion

We only insert the node **at the root of** the binomial heap, and then union and merge them together. For some trees of full degrees, they need to merge together in a decreasing order.

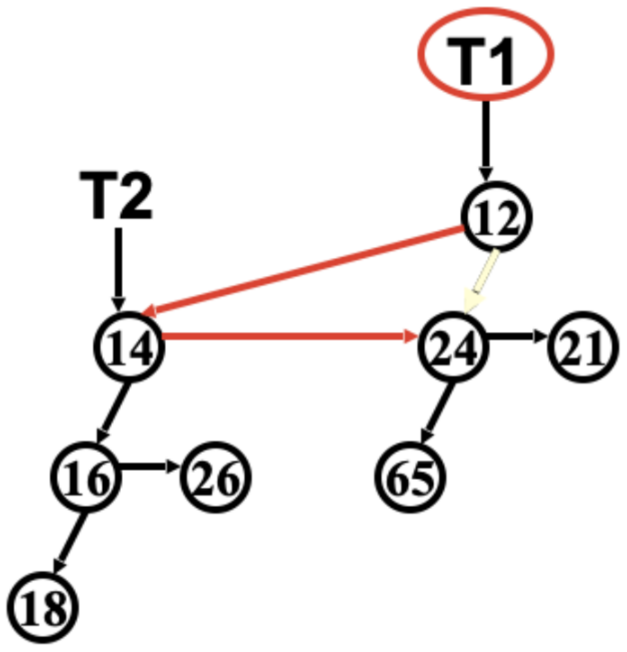
```
binomialHeap bhInsert(binomialHeap h1, int num, int weight)
{
    binomialHeap h = createBH();
    h->root = create_hNode(num, weight);
    h1 = bhUnion(&h1,&h);
    return h1;
}
```

Merge and Union

We will union the nodes from the root of heap. If the close trees have the same number of nodes, then union them together, and the one with small key(weight) will be the root of the new tree.



We can see from this picture that when we traverse the node and its siblings, we need to merge the trees with the same number of nodes. Make them in the father-son relation instead of sibling-relation.



```
binomialHeap bhUnion(binomialHeap * h1, binomialHeap * h2)
{
        hNode x = h->root;
    hNode prevNode = NULL;
    hNode nextNode = x->sibling;
    while(nextNode)
    {
        if(x->degree != nextNode->degree)//we need to find after these
nodes
        {
            prevNode = x;
            x = nextNode;
        }
        else if(x->key == nextNode->key)//union them and x will be the
root
        {
            x->sibling = nextNode->sibling;
            Link(nextNode, x);
        }
        else//union them and nextNode will be the root
        {
            pre->sibling = nextNode;
            Link(x, nextNode);
        }
      nextNode = x->sibling;
    }
    return h;
}
```

FindMin

Because for the trees, the key of the root is the smallest. However, the trees' order is random. Therefore, we need to traverse the siblings of the root node, and find the smallest. Like the picture above, the linked list will traverse all the nodes to find the miniest key.

```
hNode bhMinimum(binomialHeap h)
```

```
{
    int min;
    struct Node * p1 = h->root;
    struct Node * p2 = NULL;
    if(p1)//initializing the min
    {mark down the min.}
    while(p1)
    {
        if(p1->key < min)
        {update the min.}
        p1 = p1->sibling;
    }
    return p2;
}
```

Delete/Decrease Key

We will use the node to find it in the binomialHeap. We need to consider the position of the minimum node.

```
void bhDecreaseKey(binomialHeap h, hNode x, int key)
{
    hNode p = x;
    hNode p1 = x->p;//p1 is the parent of x
    if(x->key < key)
        Error!
    x->key = key;//decrease the key of x.
    while(p && p->key < p1->key)
    {
        p->key = p1->key;
        p1->key = key;
        p = p1;
        p1 = p->p;
    }
}
void bhDelete(binomialHeap h, hNode x)
{
    int key = -1000;//You can set it up to any number.
    bhDecreaseKey(h,x,key);//decrease the x to the minimum
    bhExtractMin(&h);//
}
void bhExtractMin(binomialHeap * h)
{
    if (x == (*h)->root)//x is the root of the heap, just delete x
          (*h)->root = x->sibling;
    else if (x->sibling == NULL)//x is at the end of the sibling
          x_prev->sibling = NULL;
    else //delete node x
          x_prev->sibling = x->sibling;
}
```

2.3 Fibonacci Heap

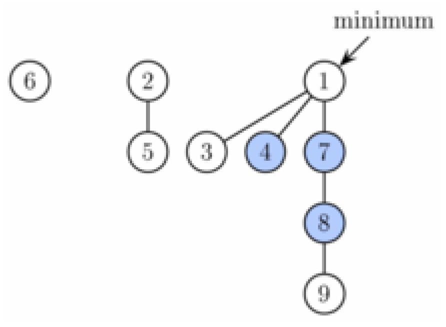
Data Structure: Fibonacci Heap

```
typedef struct FibNode{
    int key;    //value of the node
    int degree; //the number of children of the node
    int position;   //the position of the vertice in the Dijkstra map
    bool mark;      //whether the node has lost its child after the 
last time it becomes a child
    pFibNode p;     //pointer towards the node's parent
    pFibNode left;  //pointer towards the node's left sibling
    pFibNode right; //pointer towards the node's right sibling
    pFibNode child; //pointer towards one of the node's children
} FibNode;
typedef struct FibHeap{
    int totalNode;  //total number of the nodes
    pFibNode min;   //minimum node of the whole heap
    pFibNode *degreeArray;  //array stores trees of each degree
} FibHeap;
```

Specification:

Fibonacci heap is a collection of heap-ordered trees, which is often used to implement the operation of priority queue. Compared with Binary heap, it provides better amortized time complexity when dealing with operations which are not associated with deletion. What's more, when applied in Dijkstra algorithm, the amortized time complexity of "Decrease key" will be improved to O(1), which is much better than O(logn) in binary heap, especially in a dense graph.

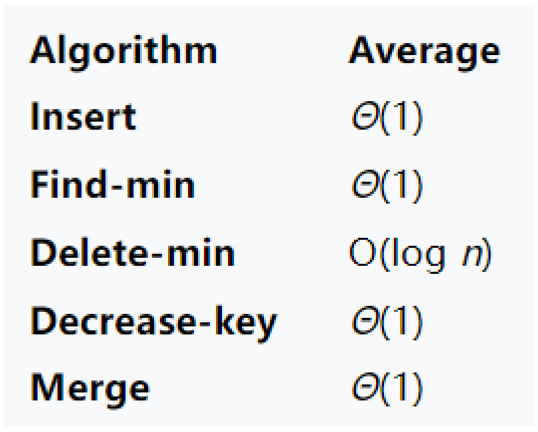
When it comes to the detailed implementation, Fibonacci heap is a forest, each of the tree follows minimum-heap property. The roots of the trees are linked by a linked list, while children of each node are connected with linked list as well. In the graph below, node 1/2/6 are connected as "roots" of Fibonacci heap, while 3/4/7 are connected as 1's children. And obviously, each of the tree follows minimum-heap property.



Graph 1: Example of Fibonacci heap (from Wikipedia)

In Fibonacci heap, there's a pointer showing the min node, while the total number of nodes are recorder as well. And in each node, we store the key we need, and pointers towards its parents/siblings/(one of its)child. What's more, the degree, which means the number of children a node has, is stored as well. That causes us to store an array "dArray[]" in the heap pointer to store the trees of each degree while deleting min. After all, in each node, we need a bool variable "mark" to record whether it has lost child after it becomes a child.

Here comes operations of Fibonacci heap. In real practice in Dijkstra algorithm, we don't need "Merge" operation because all the nodes are inserted one by one.



Algorithm	Average
Insert	$\Theta(1)$
Find-min	$\Theta(1)$
Delete-min	$O(\log n)$
Decrease-key	$\Theta(1)$
Merge	$\Theta(1)$

Graph 2: Operations of Fibonacci heap (from Wikipedia)

Insertion: When we insert, we add it into the roots list directly, which is exactly easy.
Pseudo Code:

```
Insert-Fibheap (heap H, node x)
{
    link x into H’s roots list;
    if(x->key < H->min->key)
        H->min = x;
    H->totalnode++;
}
```

Find-min: Because we have a pointer towards the min element, it’s very easy.
Pseudo Code:

```
find-min (heap H)
{   return H->min;  }
```

Delete-min: When it comes to deletion, we need to remove the min node, link all its children to the roots, and then merge the trees with the same degree until none of their degree is the same. The process is shown in the pseudo code.
Pseudo Code:

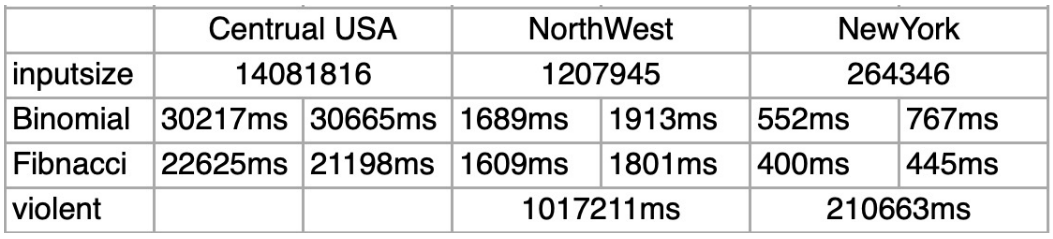
```
Delete-min(heap H)
{
    z = H->min;
    H->totalnode--;c
    if (z == NULL)  return NULL;
    remove all of z’s children to H->roots;
    remove z from H->roots;
    if(z == z->right)   //H has only one node
        H->min = NULL;
    else
    {
        H->min = z->right;  //we’ll fix the min pointer soon
        Consolidate(H);     //merge the trees with the same degree
    }
    return z;
}
Consolidate(heap H)
{
    H->dArray = (fibNode**) calloc (sizeof(fibNode*) * maxDegree);
//here the maxDegree is to be discussed sooner
    while(H->min != NULL)
    {
        ptr = H->min;
        d = H->min->degree;
        remove ptr from H->roots;
        while (dArray[d] != NULL)
        {
            ptr2 = dArray[d];   //merge two trees of degree d
            link ptr with ptr2 with the one having smaller root being
the new root, and make the subtree’s mark being false at the same
time;
            ptr = the new tree we get just now;
            dArray[d++] = NULL; //move to merge the next degree
        }
        dArray[d] = ptr;
    }
link all trees in dArray[] and update the min node while linking;
}
```

Decrease key: The process is not so difficult as well. We just move the node to the roots if the node’s value becomes less than its parent, and continue to move its parent until the node’s mark becomes false of the node is a root. Each time we move a node to root, we cmake its parent’s mark true.

Pseudo code:

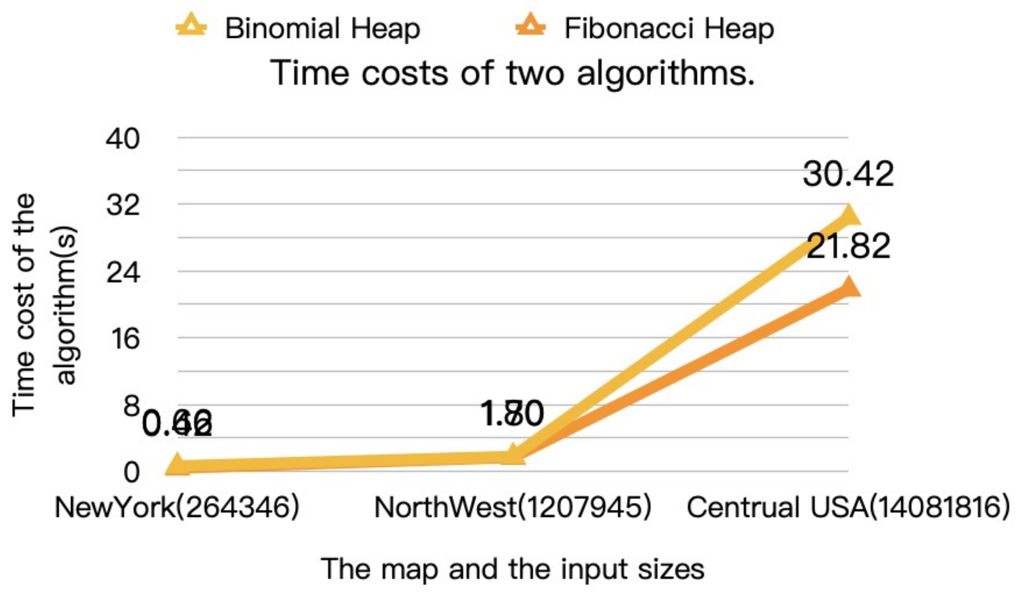
```
Decrease-key(heap H, node x, value k)
{
    if(x->key > k)  x->key = k;
    y = x->parent;
    if(y != NULL && y->value > x->value)
    {
        Cut(H, y, x);   //move x to the root
        CascadingCut(H, y); //continue to cut until we get false value
    }
    if(H->min->key > k) H->min = x;
}
Cut(heap H, node parent, node child)
{
    move child to the roots of H;
    parent->degree--;
    child->mark = false;    //child becomes a root, and never loses
child after it becomes a root THIS TIME
}
CascadingCut(heap H, node y)
{
    z = y->parent;
    if(z == NULL)   //y is root
        return;
    if(y->mark == false)    //we come to false node
        y->mark = true; //it lost child in the Cut before, and no need
to continue cutting
    else        //true, and continue to cut
    {
        Cut(H, y, z);
        CascadeCut(H, z);
    }
}
```

Chapter 3: Testing Results



	Centrual USA		NorthWest		NewYork	
inputsize	14081816		1207945		264346	
Binomial	30217ms	30665ms	1689ms	1913ms	552ms	767ms
Fibnacci	22625ms	21198ms	1609ms	1801ms	400ms	445ms
violent			1017211ms		210663ms	

run time table



run time vs. input size

We can draw the conclusion from the table and graph that, in shortest path searching problem, Fibonacci Heap performs better than Binomial Heap, and the bigger the input size is, the more obvious the advantage shows.

Chapter 4: Analysis and Comments

4.1 Dijkstra Algorithm

Time Complexity:

```
As we should travel all the nodes to do the initialization, the time
complexity of initialization would be O(n).
```

When finding the shortest path between the given points, the worst case is that every two points are connected, and every time we do the relax, we need to decrease the key of every node adjacent to the current node. Thus we should do n times of insertion(every node should be insert into the heap once) and n times of Find-Min & Delete-Min(every node should be deleted from the heap and marked as visited), which would cost n * （O(logn) + O(1) + O(logn)） = O(n logn) And in relax part, if every two point are connected and we have to decrease key for every adjacent nodes, then we need to do (n-1 + n-2 + ... + 1) = n(n-1)/2 times of Decrease-Key, and that would cost n(n-1)/2 * O(c) = O(n^2). But in most case the number of edges e is

By adding up all those parts, the total time complexity of Dijkstra Algorithm would be O(n^2) (n is the number of the vertices), while in normal cases, the time complexity is O(e) (e is the number of edges, which is no more than n^2.

Space Complexity:

Each direction of edge would take a single space in linklist, so if every two nodes in the graph is biconnected, it would take O(n^2), normally it would take O(e) (e is the number of directed edge). As the space complexity of heap is O(n), the total Space Complexity would be O(max(n, e)).

4.2 Binomial Heap

Time Complexity:

	Find-Min	Insert	Delete-Min	Merge
Time-Complexity	O(1)	O(log n)	O(log n)	O(log n)

Space Complexity

Each node would take a single place and the worst case is that every node is in the heap, thus the space complexity is O(n) (n is the number of vertices).

4.3 Fibonacci Heap

Time Complexity:

Honestly, the time complexity is difficult to analyze, and details can be seen in the “Introduction to Algorithms”. Here my analysis might (honestly, must) be worse than that in the book, but I’ll try my best to analyze.

First, insertion and find-min is an easy O(1), obviously. The difficult part is Delete-min and Decrease-key. Here we define the potential function Φ(H) = t(H) + 2m(H), where H is a certain Fibonacci heap, t(H) is the number of trees it has, and m(H) is the number of nodes whose “mark” is true.

Let’s discuss Delete-min. After one delete, there’s most D(n)+1 trees, where D(N) is the max degree of the heap. What’s more, it won’t add more true “mark”, so after delete, the potential is at most D(n)+1+2m(H).

Next is the real cost. The cost of removing children to roots and linking dArray[] into new roots is obviously O(D(n)), so we need to analyze the process of merging trees of the same degree. We know that after removing the min node’s children to roots, each time we merge two trees, a tree is removed from new roots. So here the cost is O(number of nodes in new roots). Meanwhile, there’s at most t(H)+D(n)-1 nodes, where t(H) is original roots, D(n) is children of min, and -1 is the min node deleted. So we get the real cost O(t(H)+D(n)).

After Delete-min, the amortized cost is O(t(H)+D(n)) + D(n)+1+2m(H) - (t(H)+2m(H)) = O(D(n)) + O(t(H)) - t(H) = O(D(n)). Meanwhile, we know D(n) is O(logn). So Delete-min is O(logn) totally.

Now let's discuss Decrease-key. Suppose we need to operate CascadingCut for c times in one Decrease-key, and each CascadingCut need O(1) time, so real cost is O(c).

After Decrease-key, node x becomes a new tree, and it's mark becomes false. Following, each cascading-cut will reduce a true "mark" and add a new tree except the last one. So heap H has t(H)+c trees, and at most m(H)-c+2 true marked node (m(H) originally, c-1 reduced by cascading-cut, and last cascading-cut might add one).

So after Decrease-key, the amortized time cost is O(c) + (t(H)+c + 2*(m(H)-c+2)) - (t(H)+2m(H)) = O(c)-c+4 = O(1).

Space Complexity:

The space complexity is easy to analyze. Each node takes a place, so in the worst case, all the nodes are in the heap (let's imagine the case where node 1 is source and all nodes are connected to node 1), so the space complexity is O(V), where V is the number of vertices.

Appendix: Source Code

Fibonacci Heap

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#include "Fibonacci.h"

#define INF 0xEFEFEF
#define MAX_NODE 14081900        //max No. of vertice

struct LinkNode                 //linklist of terminal points
{
    int t;                      //No. of terminal point
    int dis;                    //distance between source and terminal
points
    struct LinkNode* next;      //point to next adjacent point of
source point
};
typedef struct LinkNode linknode;

struct Node                     //array of source points
{
    linknode *linklist;         //point to adjacent points of current
node

    int min_dis;                //shortest path length between given
point and current point
    int visited;                //has current point been visited

    pFibNode FNode;             //point to according node in Fibonacci
Heap
}sourcelist[MAX_NODE];          //index is the no. of source point
typedef struct Node node;

//build adjacency list
```

```
void BuildLinkList();
//return the address of the last node in linklist
linknode* GetLastLinknode(linknode* root);
//using Dijkstra Algorithm to find the shortest path between source
and terminal
void Dijkstra(int source, int terminal);
//find whether node X is adjacent with given point
linknode *FindXinLink(int x, linknode* linkroot);

pFibHeap FHeap = NULL;


int main()
{

 SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_BL
UE|FOREGROUND_GREEN); //output software reminder in blue
    FHeap = makeFibHeap();
    //Build Adjacency List
    BuildLinkList();

    printf("build linklist fin\n");
    int source, terminal;
    printf("input '-1' to quit\n");
    printf("Please input the query: ");

 SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_IN
TENSITY|FOREGROUND_RED|FOREGROUND_GREEN|FOREGROUND_BLUE);//output
user's input in white
    scanf("%d", &source);
    //query
    while(source != -1)
    {
        scanf("%d", &terminal);
        if(source < MAX_NODE && terminal < MAX_NODE)
        {
            Dijkstra(source, terminal);
        }
        else
        {

 SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_RE
D);//output error in red
            printf("#ERROR# input is too big\n");
        }

 SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_BL
UE|FOREGROUND_GREEN); //output software reminder in blue
        printf("Please input the query: ");

 SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_IN
TENSITY|FOREGROUND_RED|FOREGROUND_GREEN|FOREGROUND_BLUE);//output
user's input in white
        scanf("%d", &source);
    }
 SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_BL
UE|FOREGROUND_GREEN); //output software reminder in blue
    printf("Bye!\n");
    system("pause");
    return 0;
}

//build adjacency list
void BuildLinkList()
{
    //initialization
    for(int i = 0; i < MAX_NODE; i++) sourcelist[i].visited = -1;
    int maxx = -1;

    printf("Building linklist...\n");
    FILE *fp = fopen("..\\documents//B_connect.txt", "r");
    if(!fp)
    {
        printf("#ERROR# open B_connect.txt failed!\n");
        return;
    }

    int s;    //source
    int t;    //terminal
    int dis;  //distance between source & terminal

    //scan the Map File
    while(!feof(fp))
    {
        fscanf(fp, "%d %d %d\n", &s, &t, &dis);
        if(s > maxx) maxx = s;
        if(t > maxx) maxx = t;
        if(sourcelist[s].visited && sourcelist[t].visited)
        //s & t are not in adjacency list
        {
            //s->t
            sourcelist[s].visited = 0;
            sourcelist[s].min_dis = INF;
            //first point in linklist
            linknode *tmp = (linknode*)malloc(sizeof(linknode));
            tmp->t = t;
            tmp->dis = dis;
            tmp->next = NULL;
            sourcelist[s].linklist = tmp;

            //t->s
            sourcelist[t].visited = 0;
            sourcelist[t].min_dis = INF;
            //first point in linklist
            tmp = (linknode*)malloc(sizeof(linknode));
            tmp->t = s;
            tmp->dis = dis;
            tmp->next = NULL;
            sourcelist[t].linklist = tmp;
        }
        else if(!sourcelist[s].visited && !sourcelist[t].visited)
        //s & t are both in the adjacency list
        {
            //s->t
            linknode *linklast =
GetLastLinknode(sourcelist[s].linklist);
            linknode *subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = t;
            subtmp->dis = dis;
            subtmp->next = NULL;
            linklast->next = subtmp;

            //t->s
            linklast = GetLastLinknode(sourcelist[t].linklist);
            subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = s;
            subtmp->dis = dis;
            subtmp->next = NULL;
            linklast->next = subtmp;
        }
        else if(sourcelist[s].visited && !sourcelist[t].visited)
        //s not in, t in
        {
            //t->s
            linknode *linklast =
GetLastLinknode(sourcelist[t].linklist);
            linknode *subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = s;
            subtmp->dis = dis;
            subtmp->next = NULL;
            linklast->next = subtmp;

            //s->t
            sourcelist[s].visited = 0;
            sourcelist[s].min_dis = INF;
            //first point in linklist
            subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = t;
            subtmp->dis = dis;
            subtmp->next = NULL;
            sourcelist[s].linklist = subtmp;
        }
        else if(!sourcelist[s].visited&& sourcelist[t].visited)
        //s in, t not in
        {
            //s->t
            linknode *linklast =
GetLastLinknode(sourcelist[s].linklist);
            linknode *subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = t;
            subtmp->dis = dis;
            subtmp->next = NULL;
            linklast->next = subtmp;
            //t->s
            sourcelist[t].visited = 0;
            sourcelist[t].min_dis = INF;
            //first point in linklist
            subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = s;
            subtmp->dis = dis;
            subtmp->next = NULL;
            sourcelist[t].linklist = subtmp;
        }
    }
    return ;
}

//return the address of the last node in linklist
linknode* GetLastLinknode(linknode* root)
{
    if(root->next == NULL) return root;
    else                   return GetLastLinknode(root->next);
}

//using Dijkstra Algorithm to find the shortest path between source
and terminal
void Dijkstra(int source, int terminal)
{
    time_t start = clock();//timer start

    pFibNode curnode;

    //initialization
    int i = 1;
    while(i < MAX_NODE)
    {
        sourcelist[i].visited = 0;

        if(i == source)//the source point itself
        {
            sourcelist[i].min_dis = 0;
            sourcelist[i].FNode = getFibNode(0, source);
        }
        else
        {
            sourcelist[i].min_dis = INF;
        }
        i++;
    }

    //insert source into the heap
    FHeap = insertToFibHeap(FHeap, sourcelist[source].FNode);

    while(FHeap->totalNode)
    //when the heap is not empty
    {
        curnode = deleteMinFibHeap(FHeap);

        int index = curnode->position;         //curnode->position - 
index of xurnode

        if(sourcelist[index].visited)
        //curnode has been visited
        {
            free(curnode);
            continue;
        }

        sourcelist[index].visited = 1;          //mark curnode as 
visited
        free(curnode);                          //delete curnode from 
heap

        //relax
        linknode* cur_update = sourcelist[index].linklist;
        while(cur_update!=0)
        //there are still point adjacent to curnode
        {
            int t = cur_update->t;     //index of adjacent node
            if(!sourcelist[t].visited && cur_update-
>dis+sourcelist[index].min_dis < sourcelist[t].min_dis)
            {
                if(sourcelist[t].min_dis == INF)
                //new unvisited point - insert into heap
                {
                    sourcelist[t].FNode = getFibNode(cur_update-
>dis+sourcelist[index].min_dis, t);
                    FHeap = insertToFibHeap(FHeap, 
sourcelist[t].FNode);
                }
                else
                //old unvisited point - update its key value in heap
                {
                    FHeap = decreaseFibHeap(FHeap, 
sourcelist[t].FNode, cur_update->dis+sourcelist[index].min_dis);
                }
                //update the shortest path length
                sourcelist[t].min_dis = cur_update-
>dis+sourcelist[index].min_dis;
            }
            //visit next adjacent node
            cur_update = cur_update->next;
        }
    }

    time_t stop = clock();//timer stop
    //output query result

 SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_GR
EEN);//output in green color
```

```
    if(sourcelist[terminal].min_dis == INF) printf("%d and %d are
unconnected!\n", source, terminal);
    else printf("The shortest path between %d and %d is %d.\n",
source, terminal, sourcelist[terminal].min_dis);
    printf("cost time %.0fms\n", difftime(stop, start));
    return;
}

//find whether node X is adjacent with given point
linknode *FindXinLink(int x, linknode* linkroot)
{
    if(!linkroot) return NULL;                                //no
point in the linklist

    if(linkroot->t == x) return linkroot;
 //find the given point
    else                    return FindXinLink(x, linkroot->next);
 //visit next point in linklist
}
```

Binomial Heap

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "binomial.h"
#include "binomial.c"
#define INF 0xEFEFEF
#define MAX_NODE 14082000       //5k行测试文件的最大结点编号

struct LinkNode//邻接点记录
{
    int t;     //终点的index
    int dis;   //距离
    struct LinkNode* next;    //指向下一个指针
};
typedef struct LinkNode linknode;

struct Node//邻接表结构
{
    linknode *linklist;   //指向终点组成的链表

    int min_dis;          //从指定点到该点的最短距离
    int visited;          //是否被访问过

    //pFibNode FNode;     //指向斐波那契堆结点
    hNode BNode;
}sourcelist[MAX_NODE];             //下标为起点编号
typedef struct Node node;

//建立邻接表，结果存储在node数组s[]中
void BuildLinkList();
//返回子邻接表最后一个结点的地址
```

```c
linknode* GetLastLinknode(linknode* root);
//找到编号为source的点到所有点的shortestpath
void Dijkstra(int source, int terminal);
//查找子邻接表中是否存在编号为x的点
linknode *FindXinLink(int x, linknode* linkroot);

binomialHeap BHeap = NULL;

int main()
{
    BHeap = createBH();
    //建立邻接表
    BuildLinkList();
    printf("Build linklist successfull!\n");
    int source, terminal;
    printf("input '-1' to quit\n");
    printf("Please input the query:");
    scanf("%d", &source);
    while(source != -1)
    {
        scanf("%d", &terminal);
        if(source < MAX_NODE && terminal < MAX_NODE)
        {
            Dijkstra(source, terminal);
        }
        else
        {
            printf("#ERROR# input is too big\n");
        }
        printf("Please input the query:");
        scanf("%d", &source);
    }

    printf("Bye!\n");
    system("pause");
    return 0;
}

//建立邻接表，结果存储在node数组s[]中
void BuildLinkList()
{
    //初始化node数组
    for(int i = 0; i < MAX_NODE; i++) sourcelist[i].visited = -1;
    int maxx = -1;

    printf("Building linklist...\n");
    FILE *fp = fopen("..\\documents//B_connect.txt", "r");
    if(!fp)
    {
        printf("打开B_connect.txt失败!\n");
        return;
    }

    printf("100%%[▧▧▧▧▧▧▧▧▧▧▧▧▧▧▧▧▧▧▧▧]");
    int s;    //起点
    int t;    //终点
    int dis;  //s-t的距离

    //建立邻接表
    while(!feof(fp))
    {
        fscanf(fp, "%d %d %d\n", &s, &t, &dis);
        if(s > maxx) maxx = s;
        if(t > maxx) maxx = t;
        //printf("handling: %d %d %d\n", s, t, dis);
        if(sourcelist[s].visited && sourcelist[t].visited)//两点均不存在
邻接表中，在结尾插入新的结点
        {
            //printf("00\n");
            //s->t
            sourcelist[s].visited = 0;
            sourcelist[s].min_dis = INF;
            //初始化首条邻边
            linknode *tmp = (linknode*)malloc(sizeof(linknode));
            tmp->t = t;
            tmp->dis = dis;
            tmp->next = NULL;
            sourcelist[s].linklist = tmp;

            //t->s
            sourcelist[t].visited = 0;
            sourcelist[t].min_dis = INF;
            //初始化首条邻边
            tmp = (linknode*)malloc(sizeof(linknode));
            tmp->t = s;
            tmp->dis = dis;
            tmp->next = NULL;

            sourcelist[t].linklist = tmp;
        }
        else if(!sourcelist[s].visited && !sourcelist[t].visited)//s,t
均在邻接表中
        {
            //printf("11\n");
            //s->t
            linknode *linklast =
GetLastLinknode(sourcelist[s].linklist);
            linknode *subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = t;
            subtmp->dis = dis;
            subtmp->next = NULL;
            linklast->next = subtmp;

            //t->s
            linklast = GetLastLinknode(sourcelist[t].linklist);
            subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = s;
            subtmp->dis = dis;
            subtmp->next = NULL;
            linklast->next = subtmp;
        }
        else if(sourcelist[s].visited && !sourcelist[t].visited)//s不
在，t在
        {
            //printf("01\n");
            //t->s
            linknode *linklast =
GetLastLinknode(sourcelist[t].linklist);
            linknode *subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = s;
            subtmp->dis = dis;
            subtmp->next = NULL;
            linklast->next = subtmp;

            //s->t
            sourcelist[s].visited = 0;
            sourcelist[s].min_dis = INF;
            //初始化首条邻边
            subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = t;
            subtmp->dis = dis;
            subtmp->next = NULL;
            sourcelist[s].linklist = subtmp;
        }
        else if(!sourcelist[s].visited&& sourcelist[t].visited)//s在，t
不在
        {
            //s->t
            linknode *linklast =
GetLastLinknode(sourcelist[s].linklist);
            linknode *subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = t;
            subtmp->dis = dis;
            subtmp->next = NULL;
            linklast->next = subtmp;

            //t->s
            sourcelist[t].visited = 0;
            sourcelist[t].min_dis = INF;
            //初始化首条邻边
            subtmp = (linknode*)malloc(sizeof(linknode));
            subtmp->t = s;
            subtmp->dis = dis;
            subtmp->next = NULL;
            sourcelist[t].linklist = subtmp;
        }
    }
    //printf("max node index is %d\n", maxx);
    return ;
}

//返回子邻接表最后一个结点的地址
linknode* GetLastLinknode(linknode* root)
{
    if(root->next == NULL) return root;
    else                       return GetLastLinknode(root->next);
}

//找到编号为source的点到所有点的shortestpath
void Dijkstra(int source, int terminal)
{
    time_t start = clock();
    hNode curnode;
    BHeap = createBH();

    //初始化 - 将所有点标为unvisited
    //若邻接，min_dis标为边长；否则表为INF
    int i = 1;
    while(i < MAX_NODE)
    {
        sourcelist[i].visited = 0;
        sourcelist[i].BNode = NULL;
        //查找curnode是否在sourcenode的子邻接表中
        linknode *pos = FindXinLink(i, sourcelist[source].linklist);

        if(i == source)
        {
            sourcelist[i].min_dis = 0;
            sourcelist[i].BNode = create_hNode(0, source);
        }
        else
        {
            sourcelist[i].min_dis = INF;
        }
        i++;
    }
    //起点入堆, value - min_dis, position - 点编号
    BHeap = bhInsert(BHeap,sourcelist[source].BNode);

    //for(i = 1; i < MAX_NODE; i++)
    while(BHeap->root)//堆非空时
    {
        curnode = bhMinimum(BHeap);                  //curnode-
>position为该点编号
        int index = curnode->position;
        if(sourcelist[index].visited)
        {
            bhExtractMin(&BHeap);
            continue; //已访问则跳过
        }
        sourcelist[index].visited = 1;          //指定点标为已访问
        bhExtractMin(&BHeap);               //释放min结点空间

        //更新所有和min_node相连且unvisited的点
        linknode* cur_update = sourcelist[index].linklist;
        //if(!cur_update) printf("nothing...\n");
        while(cur_update)
        {
            int t = cur_update->t;
            if(!sourcelist[t].visited && cur_update-
>dis+sourcelist[index].min_dis < sourcelist[t].min_dis)
            {
                if(sourcelist[t].min_dis == INF)
                {
                    sourcelist[t].BNode = create_hNode(cur_update-
>dis+sourcelist[index].min_dis, t);
                    BHeap = bhInsert(BHeap, sourcelist[t].BNode);
                }
                else
                {
                    bhDecreaseKey(BHeap, sourcelist[t].BNode,
cur_update->dis+sourcelist[index].min_dis);
                }
                sourcelist[t].min_dis = cur_update-
>dis+sourcelist[index].min_dis;
            }
            cur_update = cur_update->next;
        }
        //if(sourcelist[terminal].visited)
         //   break;
    }
    time_t stop = clock();
    if(sourcelist[terminal].min_dis == INF) printf("%d and %d are
unconnected!\n", source, terminal);
    else printf("The shortest path between %d and %d is %d.\n",
source, terminal, sourcelist[terminal].min_dis);
    free(BHeap);
    printf("cost time %.0fms\n", difftime(stop, start));
    return;
}

//查找子邻接表中是否存在编号为x的点
linknode *FindXinLink(int x, linknode* linkroot)
{
    if(!linkroot) return NULL;

    if(linkroot->t == x) return linkroot;
    else                 return FindXinLink(x, linkroot->next);
}
```

References

None.

Author List

Name	Contribution
Wang Kedi	Fibonacci Heap According part in Chap 2,3 in Report
Zhao Yilei	Binomial Heap Chap 1 and according part in Chap 2,3 in report
Shen Yunfeng	Build Adjacency List and Dijkstra Algorithm According part in Chap 2,3 in report

Declaration

We here by declare that all the work done in this project titled "pj2" is of our independent effort as a group.

Signature

Each author must sign his/her name here.