Shortest Path Algorithm with Heaps

xxx xxx xxx

Date:2020-04-07

CONTENT

Chapter 1 Introduction

1.1 Background

Shortest Path Algorithm, which is the algorithm to find the shortest distance from one point to another, is one of the most classic and basic algorithms in computer science; and it is also the basis of many other algorithms, so we It is necessary to study how to optimize this basic algorithm.

1.2 Problem Description

In this project, we are to solve the “Shortest Path Algorithm with Heaps” problem. We are supposed 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.

We are to solve 3 sub-problems.

1. Implement Dijkstra’s algorithm based on a min-priority queue.
2. Repeat the algorithm with random source and target for 1000 times.
3. Count the total time of 1000 runs.

Here are the heaps we test.

1. Fibonacci heap, invented by Michael L. Fredman and Robert E. Tarjan.
2. Binomial heap, invented by Jean Vuillemin.

Chapter 2 Algorithm Specification

2.1 Dijkstra

It is used to find the shortest path between two points specified in the figure, or the shortest path between one point and all other points. It is essentially a greedy algorithm.

Basic idea:

1. Consider the initial point on the graph as a set S, and other points as another set
2. According to the initial point, find the distance $d[i]$ from other points to the initial point
3. Select the smallest $d[i]$ (denoted as $d[x]$), and add the point corresponding to this $d[i]$edge (denoted as x) to the set S
4. According to x, update the $d[y]$ value of the point y adjacent to x: $d[y] = \min(d[y], d[x] + \mathit{edge\ weight}\ w[x][y])$, because the distance may be reduced, so this update operation is called a slack operation.
5. Repeat steps 3 and 4 until the target point is also added to the set. At this time, $d[i]$ corresponding to the target point is the shortest path length.

```
distance[s] ← 0
for i ← 1 to n
    t ← unmarked point that has the minimum distance // (use heap!!)
    for every edge (t, v)
        update the distance of v
    mark vertex t
```

2.2 Introduction of Heaps

2.2.1 Binomial heap

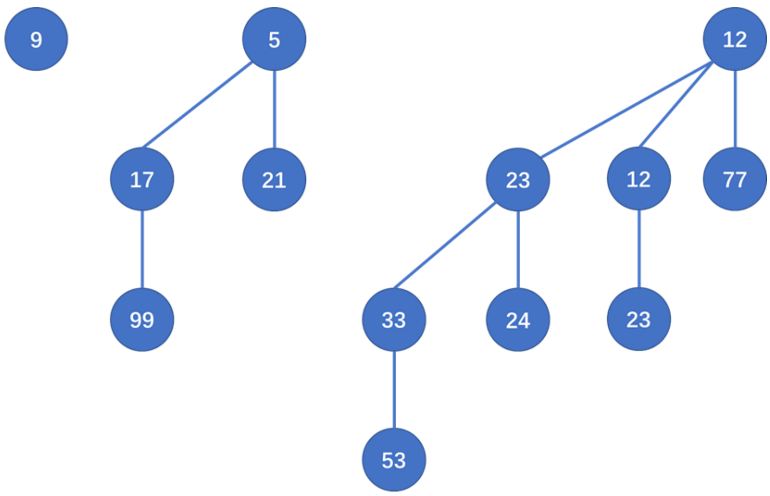


Figure1 Sample of Binomial Heap

The binomial heap is different from the common heap. Instead of being an unordered heap, it is better to say that it is a collection of ordered heaps, or a forest of multi-fork trees. Each of these trees is a binomial tree, and trees of the same height can only appear once. If we mark the tree of height k with the symbol B_k and specify the height of a single node to be 0, it is not difficult to find that B_k is actually composed of two trees B_{k-1}, so when two trees of the same height appear, we will proceed Merge to generate a new tree, where the root with a small root node value will continue to be the root, and the root of another tree will be hung in the form of a son, and the height of the new tree will be increased by one; the new tree If you encounter trees of the same height again,This process will be repeated. The process of merging this tree is also similar to the carry process of addition of binary numbers. Therefore, we call this data structure a binomial heap.

The overall structure and internal nodes of the binomial heap are defined as follows:

```
struct node{
    ElementType data;
    node* left;
    node* next;
};
struct binHeap{
    int cursize;
    node* tree[treenum];
};
```

Each node is connected to other nodes in the tree by "left son, right brother", and the root nodes of all trees are stored in the array *trees* of the structure *BinomialHeap*.

In the binomial heap, almost all operations are achieved through merging, and the process of merging is exactly the same as binary addition:

1. Directly merge the two forests.
2. Starting from the height $h = 0$, check whether there is only one tree at the height h, if not, merge two of them to generate a height of $h+1$ and check the height of the same height.
3. Repeat until all heights have been checked.

```
function merge(BinomialHeap H1,BinomialHeap H2)
begin
    for height :=0 to MaxHeight begin
        if(hasTwoTrees(height)) begin
            combineTwoTrees()
        end if
    end for
end
```

With the merge operation, the heap operation can be regarded as merging the original heap and the heap with only one node.

The top element of the heap, which is our commonly used *top*() operation, needs to be searched on the roots of all trees every time, because the elements on the root, and the prime element is the smallest element in the entire tree.

The operation to delete the top element actually only needs the following steps:

1. Find the top of the pile, the smallest element
2. Remove the tree from the forest, delete its roots, and leave the remaining nodes as a new forest
3. The two forests are merged and you get a heap with the top element removed

The pseudo code of the delete operation is as follows:

```
function pop(BinomialHeap H)
begin
    MinTree := findMin();
    H1 := H - MinTree;
    H2 := MinTree - MinTree.top();
    merge(H1,H2);
end
```

2.2.2 Fibonacci heap

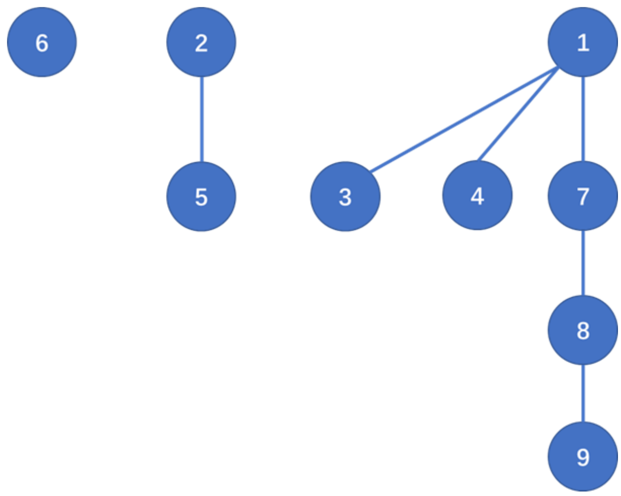


Figure2 Sample of Fibonacci Heap

The initial idea of the Fibonacci heap is similar to the binomial heap, and it is also to maintain a forest. Each tree in the forest meets the most basic nature of the heap, that is, the value of the parent node must be greater than the value of all nodes in its subtree.

The difference with the binomial heap is that the Fibonacci heap does not force every root degree to meet certain requirements, but only stores all root nodes in a doubly linked list, and uses a pointer *minNode* to maintain the minimum node information. . And the Fibonacci heap does not maintain the balanced nature of the heap when inserting, merging, and finding the smallest node. The structure of the heap is only modified when the smallest node is deleted.

In addition, the child nodes of each node of the Fibonacci heap are also stored through a doubly linked list. The information stored in the node includes the key value of the node, the pointer to the node's parent node, the node on the left and the right The two pointers of the node, the pointer to the starting node of the doubly linked list, the tag variable (the variable used to maintain and modify the value of the node), and the degree of the node. The structure is defined as follows:

```
typedef struct fib_node* ptr_fib_node;
struct fib_node {
    int distance;           //The length of path from the source node to this node
    int degree;             //The number of its children
    int index;              //The index of this node
    bool mark;              //Whether its child had been removed or not.
    ptr_fib_node parent;    //The pointer to its parent
    ptr_fib_node child;     //The pointer to its child
    ptr_fib_node left;      //The pointer to its left sibling
    ptr_fib_node right;     //The pointer to its right sibling
};

//This structure is used in fibonacci method
struct {
    int tree_num;                                   //The number of trees in the fibonacci heap
    ptr_fib_node fib_temp_node;                     //Only used as a temporary pointer in variable deliver
    ptr_fib_node fib_min_node;                      //The pointer to the node with minimum distance
    ptr_fib_node fib_temp_stack[MAX_NODE/2 + 1];    //A temp array, used to reconstruct fib_heap
    struct fib_node fib_node[MAX_NODE];             //All the fibonacci nodes
}fib_heap;
```

The insertion operation of the Fibonacci heap is very simple. You only need to insert the node to be inserted as a root node with no child nodes into the doubly linked list of the root node, and update the *minNode* if necessary. In the same way, the merging operation is also very simple. Just merge the linked lists of the root nodes of the two Fibonacci heaps, and remember to update *minNode* as well. The operation to find the minimum value is to return the key value of the node pointed to by *minNode* .

None of the above operations change the overall structure of the heap, but simply operate on the root doubly linked list. This makes the insertion of the Fibonacci heap and the time complexity of the operation of finding the minimum value both linear, and the time complexity of a merge is linearly related to the number of roots in the heap.

The difference between the Fibonacci heap and the binomial heap is that the Fibonacci heap does not necessarily have at most two nodes with the same degree, so the code must also be adjusted. We call this operation **consolidate**. There are many ways to achieve consolidation, but the mainstream method is to use $A[deg]$ to record the node with the smallest key value of degree deg that has been found so far. When a new node is found, if $A[deg]$ is empty, assign $A[deg]$ to this node, if not, combine this node with the node stored in $A[deg]$ into one node, merge $A[deg]$ is set to null and the merged nodes are used as new nodes to operate with $A[deg+1]$.

The pseudo code is as follows:

```
function conslidate()
begin
    pointer ptr := minNode
    pointer startP := ptr
    do begin
        deg := ptr->degree
        while A[deg] is not null begin
            ptr2 := A[deg]
            if ptr->key > ptr2->key begin
                swap ptr ptr2
            end if
            if ptr2 is the minimum node so far begin
                minNode := ptr
            end if
            if ptr2 == startP begin
                startP point to the next node of *startP
            end if
            Link ptr2 to ptr's child list
            A[deg] := null
            deg := deg + 1
        end while
        A[deg] := ptr
        ptr point to the right node of *ptr
    end while ptr != startP
    update minNode with the minimum node in heap
end
```

The operation to be described below is the modified value operation of the Fibonacci heap. If the value after the increase is smaller than the previous value, all the child nodes of the added node will be linked to the root linked list, and the added node will also be connected Go to the root linked list and cascade its parent node.

If the modification value is a decrease value operation, it is necessary to judge whether the value decrease is smaller than the value of its parent node, if it is smaller than the value of the node, connect it to the root linked list and cascade the parent node. And update $minNode$ at any time.

2.3 Sketch of Algorithms

2.3.1 Sketch of Algorithm Using Binomial Heap

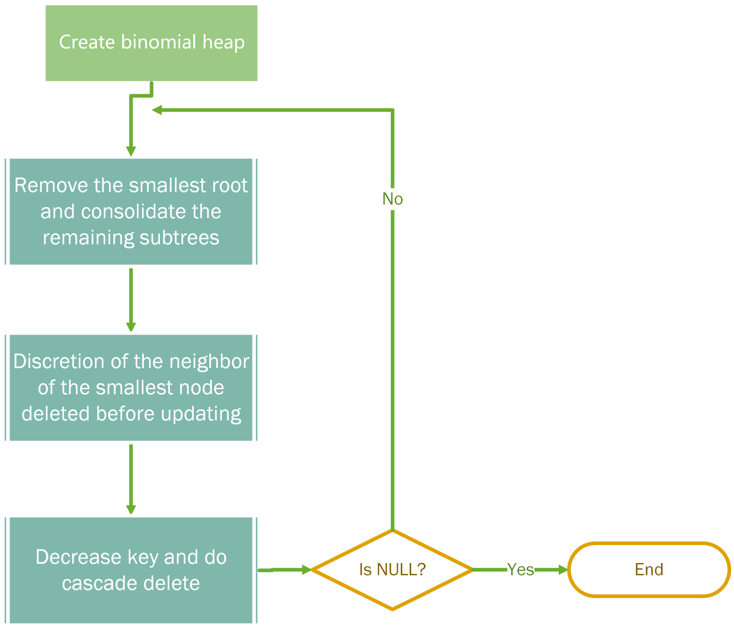


Figure3 Sketch of Algorithm Using Binomial Heap

2.3.2 Sketch of Algorithm Using Fibonacci Heap

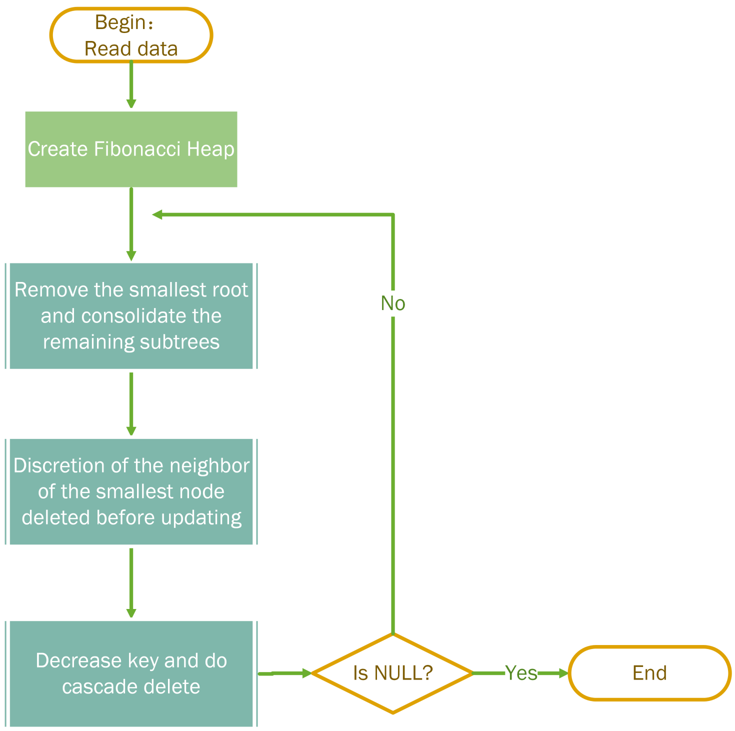


Figure4 Sketch of Algorithm Using Fibonacci Heap

Chapter 3 Testing Results

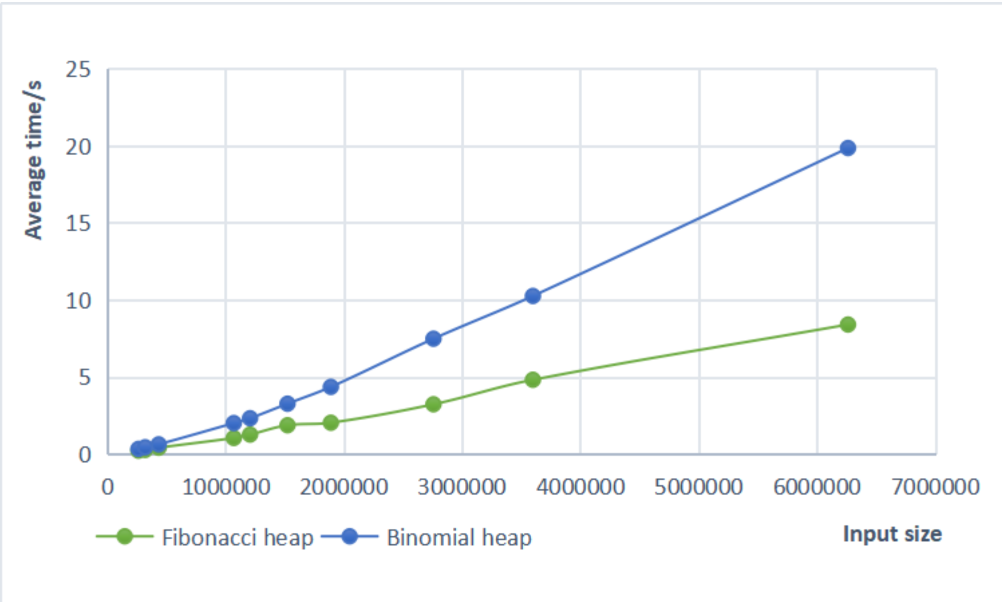
3.1 Testing Results in Table

Table1 Average Time with Queries=10000

Case	Name	Nodes	Arcs	Fibonacci heap/s	Binomial heap/s
0	test.txt	264,346	733,846		
1	USA-road-d.NY.gr	264,346	733,846	0.239	0.353
2	USA-road-d.BAY.gr	321,270	800,172	0.287	0.460
3	USA-road-d.COL.gr	435,666	1,057,066	0.433	0.649
4	USA-road-d.FLA.gr	1,070,376	2,712,798	1.062	2.026
5	USA-road-d.NW.gr	1,207,945	2,840,208	1.283	2.340
6	USA-road-d.NE.gr	1,524,453	3,897,636	1.887	3.282
7	USA-road-d.CAL.gr	1,890,815	4,657,742	2.049	4.371
8	USA-road-d.LKS.gr	2,758,119	6,885,658	3.245	7.493
9	USA-road-d.E.gr	3,598,623	8,778,114	4.833	10.263
10	USA-road-d.W.gr	6,262,104	15,248,146	8.411	19.849

3.2 Testing Results in Graph

Figure5 Run times vs. Input sizes



Chapter 4 Analysis and Comments

4.1 Analysis on Time Complexity

4.1.1 Binomial Heap

Merge

The time complexity of merging two binomial heaps is $T = O(\log N)$, which can be very easily proven because the merging of Binomial Heaps is exactly the same as the addition of two binary numbers. Each bit of the binary number corresponds to a tree, and the two trees are merges. So the complexity of merging two trees is $O(1)$ and the number of merging operations is no more than the bits of the binary number. Therefore, the time complexity of merging is $T = O(\log N)$.

Inset

Insertion is essentially merging the original heap with a heap with only one node, so the worst-case time complexity is $O(\log N)$. However, this worst-case scenario does not occur every time, so we analyze the amortized complexity below.

Define the potential energy function of the binomial reactor as:

$$\Phi(D_i) = \text{The number of trees after the i-th insertion}$$

We assume that the cost of creating a new forest (obviously this is a constant) is 1, so we can get the cost per operation:

$$c_i = 1 + \text{Rounding times}$$

Each carry will trigger a tree merge, the cost of tree merge is $O(1)$, so the total carry cost is proportional to the number of carry.

At the same time, it is noted that each carry will cause the number of trees to increase by one, so each time the number of trees decreases is equal to the number of carry -1, it needs to be reduced by 1, This is because the newly inserted node will initially introduce a tree. If we replace the number of carry with c_i, it can be known that the reduction in the number of trees is $2 - c_i$.

$$c_i + (\phi_i - \phi_{i-1}) = 2$$

Accumulate the equation to get:

$$\sum_{i=1}^{N} c_i + \phi_N - \phi_0 = 2N$$

$$\sum_{i=1}^{N} c_i = 2N + \phi_0 - \phi_n \leq 2N = O(N)$$

Therefore, the amortized complexity of the insert operation is $O(1)$.

FindMin

The minimum value will only appear at the root of the tree, so you only need to traverse the tree root and update the minimum value when inserting and deleting elements. So the time complexity of finding the minimum value is $O(1)$.

Delete

Deletion consists of the above three steps, so the time complexity of the delete operation is $O(\log N)$.

4.1.2 Fibonacci Heap

FindMin

When finding the minimum, since the minimum heap property is satisfied, it is only necessary to find the root node of the binomial tree. Since there are $\log N$ subtrees in total, the time taken is $O(\log N)$. We can save a pointer to the smallest element, so it takes $O(1)$ to find the node where the smallest keyword is. This pointer needs to be modified when performing other operations.

Delete Min

When deleting the minimum, first find the node where the smallest keyword is locates, then delete it from its binomial tree and get its subtree. Think of there subtrees as separated binomial heap, and then merge this heap into the original heap. Since each tree has at most $\log N$ subtrees, the time to create a new heap is $O(\log N)$. At the same time, the time to merge the heap is also $O(\log N)$, so the time required for the entire operation is $O(\log N)$.

Decrease

When decreasing a certain key, after reducing the value of a specific node, the minimum stacking property may not be satisfied. At this time, the node where it is located is exchanged with the parent node, and if the minimum heap property is not satisfied, then it is exchanged with the grandfather node until the minimum heap property is satisfied. The time required for the operation is $O(\log N)$.

Insert

In Binomial Heap, when inserting, create a binomial heap containing only the elements to be inserted, and then merge this heap with the original binomial heap to get the inserted heap. Because of the need to merge, the insert operation takes $O(\log N)$ time. The time complexity of the amortized analysis is $O(1)$.

Extracting Min

In Binomial Heap, when inserting, we have $t(H') = t(H) + 1, m(H') = m(H)$, so there exists:

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$$

As a consequence, we know that the real complexity is $O(1)$, and the amortized complexity is $O(1)$.

While merging:

$$\phi(H) - (\phi(H_1) + \phi(H_2)) = (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2)))$$

This equation equals to 0, which is also fits the prediction.

When extracting the smallest node, we have potential:

$$\phi(pre) = t(H) + 2m(H)$$

And then the potential increases to $D(n) + 1 + 2m(H)$. Therefore, there are at most $D(n) + 1$ roots after operation with no nodes to be marked. So:

$$O(D(n) + t(H)) + (D(n) + 1 + 2m(H)) - (t(H) + 2m(H)) = O(D(n)) + O(t(H)) - t(H) = O(D(n))$$

As we can know, $D(m) = O(\log n)$, so the time complexity of extracting minimum element is $O(\log N)$
.

4.2 Analysis on Space Complexity

4.2.1 Binomial Heap

Each tree in the binomial heap is stored using a link structure, each key value is stored in a node, and all the trees are passed a linear number which is based on the array structure, so the space complexity is $O(N)$

4.2.2 Fibonacci Heap

The space required for the Fibonacci heap is $O(N)$ to store the contents of all nodes and $O(\log N)$ to store the auxiliary array required for consolidation operation. So the total time complexity is $O(N)$.

4.3 Time and Space Complexity of Dijkstra Algorithm

In Dijkstra's algorithm, the number of inserting nodes into the heap, querying the smallest node, and deleting the smallest node are the same. The number of insert operations determines the number of the last two operations. Below we explore how many nodes can be inserted into the heap in the algorithm.

Although a node can be taken out of the heap multiple times, it can only relax other nodes when it is taken out for the first time. How many nodes a node can relax depends on how many edges are connected to it. If the relaxation is successful, a node will be inserted into the heap.

An edge is connected to two nodes, therefore, an edge can correspond to at most two insertion operations, so the number of insertion operations is $O(M)$, and M is the number of edges in the graph. Therefore, the total number of nodes in the heap is at most $O(M)$, and the number of various heap operations is also $O(M)$.

Time Complexity

As we know, the upper bound of the time complexity of a heap operation is $O(\log M)$, so we can deduce the worst case of the combined time complexity of the algorithm is $M \log M$. The best case is that each node will only be added to the heap once, so the overall time complexity is $O(N \log N)$. According to common sense, M and N are of the same order, so the best time complexity and the worst time complexity are of the same order.

Space Complexity

The space required by the algorithm is mainly divided into four parts:

1. $O(M)$ to store the information on the edges in the graph
2. $O(N)$ to store the value of distance
3. $O(N)$ to store whether the node is in the white dot set or black dot set
4. $O(M)$ to store the heap

So the total space complexity is $O(M + N)$

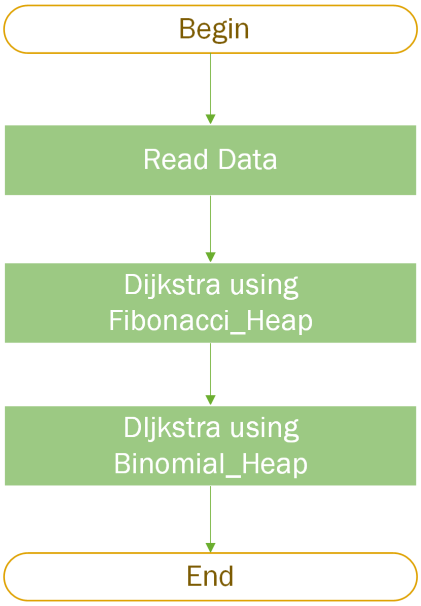
4.4 Comments

Based on the previous analysis and the actual test results, the Fibonacci heap is more suitable for Dijkstra's algorithm. We find that although the time complexity constant of the Fibonacci heap may be relatively large, it is still the highest.

Our guess may be that the deletion of the smallest node in the shortest path algorithm is used more frequently, and the unused merge operation and node value reduction operation that are likely to cause instability factors. This makes the Dijkstra heap the most time-consuming operation-the average time complexity of each deletion, and will not face the situation of consolidating multiple unconsolidated nodes at the same time. The operation without reducing the node value also makes the actual internal structure of the Fibonacci heap very similar to the binomial heap, without imbalances and complicated merges.

Appendix

Appendix I Sketch of the Program



Appendix II Source Code in C

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>
#define MAX_NODE 265000    //Must be greater than the nodes number
#define MAX_TEMP_STACK 50  //Used in bin_heap, this means 2^50 - 1 nodes are allowed
typedef enum{false, true}bool;

//This structure stores the information of edges, used in both Fibonacci and adjacent list
typedef struct adj_list_node* ptr_adj_list_node;
struct adj_list_node {
    int to;                         //The start node index
    int weight;                     //The destination node index
    ptr_adj_list_node next;         //The pointer to next adjacent node
};

//This structure maintains a visited array
typedef struct adj_found_node* ptr_adj_found_node;
struct adj_found_node{
    int index;                      //The index of the node
    int distance;                   //The distance from the source node
    bool visited;                   //Whether this node is visited or not
};

//This structure stores the information of edges, used in both Fibonacci and adjacent list
struct {
    int nodes;                                          //The number of nodes in the graph
    int edges;                                          //The number of edges in the graph
    int found_num;                                      //The number of valid items in the visit array
    struct adj_list_node adj_list_node[MAX_NODE];       //Contains each node
    struct adj_found_node adj_found_list[MAX_NODE];     //Contains possible next-nodes, used in Dijkstra using adj list.
}adj_list;

//This structure is a formal one of a binomial heap node
typedef struct bin_node* ptr_bin_node;
struct bin_node {
    int index;                      //The index of the node
    int distance;                   //The distance from the source node
    ptr_bin_node parent;            //The pointer to its parent
    ptr_bin_node left_child;        //The pointer to its child
    ptr_bin_node right_sibling;     //The pointer to its right sibling
};

//This structure is used in binomial method
struct {
```

```
    ptr_bin_node bin_temp_node;                     //Only used as a temporary pointer
    ptr_bin_node bin_stack[MAX_TEMP_STACK];         //Pointer array of all trees
    ptr_bin_node bin_temp_stack[MAX_TEMP_STACK];    //Used when merging
    ptr_bin_node bin_node_ptr[MAX_NODE];            //Pointer to every node
    struct bin_node bin_node[MAX_NODE];             //All the binomial nodes
}bin_heap;

//This structure is a formal one of a fibonacci heap node
typedef struct fib_node* ptr_fib_node;
struct fib_node {
    int distance;           //The length of path from the source node to this node
    int degree;             //The number of its children
    int index;              //The index of this node
    bool mark;              //Whether its child had been removed or not.
    ptr_fib_node parent;    //The pointer to its parent
    ptr_fib_node child;     //The pointer to its child
    ptr_fib_node left;      //The pointer to its left sibling
    ptr_fib_node right;     //The pointer to its right sibling
};

//This structure is used in fibonacci method
struct {
    int tree_num;                                   //The number of trees in the fibonacci heap
    ptr_fib_node fib_temp_node;                     //Only used as a temporary pointer in variable deliver
    ptr_fib_node fib_min_node;                      //The pointer to the node with minimum distance
    ptr_fib_node fib_temp_stack[MAX_NODE/2 + 1];    //A temp array, used to reconstruct fib_heap
    struct fib_node fib_node[MAX_NODE];             //All the fibonacci nodes
}fib_heap;

void read_data();

void dijkstra_using_fibonacci_heap();

void fib_construct(int source);

int fib_delete_min();

void fib_reconstruct();

ptr_fib_node fib_combine_two_nodes_in_temp_stack(ptr_fib_node new_node);

void fib_update_distance(int index);

void fib_decrease(int index);

void dijkstra_using_adjacent_list();

void adj_construct(int source);

ptr_adj_found_node adj_find_next();

void adj_update_distance(ptr_adj_found_node found_node);

void dijkstra_using_binomial_heap();

void bin_construct(int source);

void bin_merge();

void bin_decrease(int source);

void bin_combine_two_nodes(ptr_bin_node *node_adder_1, ptr_bin_node *node_adder_2);

int bin_delete_min();

void bin_update_distance(int index);

int main() {
    //load information from xxx.gr. NOTE THAT #define MAX_NODE must be greater than the number of nodes!
    read_data();

    //traverse the whole graph using fibonacci heap
    dijkstra_using_fibonacci_heap();

    //traverse the whole graph using binomial heap
    dijkstra_using_binomial_heap();

    //traverse the whole graph using adjacent list
//    dijkstra_using_adjacent_list();
    return 0;
}

/*
 * This function will using one input as source node, traverse the whole graph.
```

```c
 * After that, the total time will be printed to the console.
 */
void dijkstra_using_binomial_heap() {
    clock_t start, total;
    int source;
    int index;
    printf("Enter Source x (1 <= x <= %d):\n",adj_list.nodes);
    scanf("%d",&source);
    start = clock();
    //Build the initial heap.
    bin_construct(source);
    for (int i = 0; i < adj_list.nodes; i++) {
        //Delete the node that contains the minimal distance.
        index = bin_delete_min();
        //Update that node's neighbors.
        bin_update_distance(index);
    }
    total = clock() - start;
    printf("Using Binomial Heap to do Dijkstra needs %.3f seconds\n",(double)total / (double)CLOCKS_PER_SEC);
}

/*
 * This function will update all the node which are connected to the given node.
 */
void bin_update_distance(int index) {
    int neighbor_index;
    ptr_adj_list_node temp;
    temp = &adj_list.adj_list_node[index];
    //This loop will traverse all the nodes that connect to the given node
    do {
        neighbor_index = temp->to;
        //If the updated distance is smaller than its parent, then swap them iteratively
        if (bin_heap.bin_node_ptr[neighbor_index]->distance > bin_heap.bin_node_ptr[index]->distance + temp->weight) {
            bin_heap.bin_node_ptr[neighbor_index]->distance = bin_heap.bin_node_ptr[index]->distance + temp->weight;
            bin_decrease(neighbor_index);
        }
        temp = temp->next;
    }while (temp != NULL);
}

/*
 * This function will remove the minimal node from the binomial heap,
 * push all its child to temp_stack, and then merge back.
 */
int bin_delete_min() {
    int position = 0;
    int index;
    bin_heap.bin_temp_node = NULL;
    //This loop will save the node with minimal distance in the temp_node
    for (int i = 0; i < MAX_TEMP_STACK; i++) {
        if (bin_heap.bin_stack[i] == NULL) continue;
        //If temp_node is NULL, then fill it with a node in the stack
        if (bin_heap.bin_temp_node == NULL) {
            bin_heap.bin_temp_node = bin_heap.bin_stack[i];
            position = i;
        }
        //This procedure maintains previous property that temp_node saves the minimal node
        if (bin_heap.bin_temp_node->distance > bin_heap.bin_stack[i]->distance) {
            bin_heap.bin_temp_node = bin_heap.bin_stack[i];
            position = i;
        }
    }
    //Count the number of children of this tree
    int count = 0;
    ptr_bin_node temp;
    temp = bin_heap.bin_temp_node->left_child;
    while (temp != NULL) {
        count++;
        temp = temp->right_sibling;
    }
    //Delete minimal node from the stack, push all its children to temp stack, and then merge them.
    temp = bin_heap.bin_temp_node->left_child;
    index = bin_heap.bin_temp_node->index;
    while (count > 0) {
        count--;
        bin_heap.bin_temp_stack[count] = temp;
        temp = temp->right_sibling;
        bin_heap.bin_temp_stack[count]->parent = NULL;
        bin_heap.bin_temp_stack[count]->right_sibling = NULL;
    }
    //Set this slot NULL to remove the minimal node
    bin_heap.bin_stack[position] = NULL;
    bin_merge();
```

```
    return index;
}

/*
 * This function will initiate the binomial heap and build up index array in bin_node_ptr
 * By doing so, it will be much more easier to locate one node in key-decrease part.
 */
void bin_construct(int source) {
    for (int i = 1; i <= adj_list.nodes; i++) {
        bin_heap.bin_node[i].index = i;
        bin_heap.bin_node[i].distance = INT_MAX;
        bin_heap.bin_node[i].parent = NULL;
        bin_heap.bin_node[i].left_child = NULL;
        bin_heap.bin_node[i].right_sibling = NULL;
        bin_heap.bin_node_ptr[i] = &bin_heap.bin_node[i];
        bin_heap.bin_temp_stack[0] = &bin_heap.bin_node[i];
        //Use merge to do "insertion"
        bin_merge();
    }
    bin_heap.bin_node[source].distance = 0;
    bin_heap.bin_temp_node = NULL;
    //Find the entance
    bin_decrease(source);
}

/*
 * This function will swap ALL information of two nodes.
 */
void bin_decrease(int source) {
    ptr_bin_node temp;
    temp = bin_heap.bin_node_ptr[source];
    int temp_int_for_swap;
    //This loop will swap a node whose distance is smaller than its parent iteratively
    while (temp->parent != NULL && temp->distance < temp->parent->distance) {
        //Keep the property that bin_node_ptr is an index array
        bin_heap.bin_node_ptr[temp->index] = temp->parent;
        bin_heap.bin_node_ptr[temp->parent->index] = temp;
        //swap data
        temp_int_for_swap = temp->distance;
        temp->distance = temp->parent->distance;
        temp->parent->distance = temp_int_for_swap;

        temp_int_for_swap = temp->index;
        temp->index = temp->parent->index;
        temp->parent->index = temp_int_for_swap;
        //go up to the parent
        temp = temp->parent;
    }
}

/*
 * This function will merge two binomial heaps into one.
 */
void bin_merge() {
    //Denote :
    // bin_heap.bin_stack[i]        => A      Original stack
    // bin_heap.bin_temp_node       => B      Stores temp addition result
    // bin_heap.bin_temp_stack[i]   => C      Another stack, a temp stack
    bin_heap.bin_temp_node = NULL;
    for (int i = 0; i < MAX_TEMP_STACK; i++) {
        if (bin_heap.bin_temp_node == NULL) {
            if (bin_heap.bin_temp_stack[i] == NULL) continue;  //A = null or not null | B = null | C = null;
            if (bin_heap.bin_stack[i] == NULL) {  //A = null | B = null | C = not null
                bin_heap.bin_stack[i] = bin_heap.bin_temp_stack[i];
                bin_heap.bin_temp_stack[i] = NULL;
                continue;
            }
            //A = not null | B = null | C = not null
            bin_combine_two_nodes(&bin_heap.bin_stack[i], &bin_heap.bin_temp_stack[i]);
            bin_heap.bin_stack[i] = NULL;
            bin_heap.bin_temp_stack[i] = NULL;
        } else {
            if (bin_heap.bin_stack[i] == NULL) {
                if (bin_heap.bin_temp_stack[i] == NULL) {  //A = null | B = not null | C = null
                    bin_heap.bin_stack[i] = bin_heap.bin_temp_node;
                    bin_heap.bin_temp_node = NULL;
                } else {  //A = null | B = not null | C = not null
                    bin_combine_two_nodes(&bin_heap.bin_temp_stack[i], &bin_heap.bin_temp_node);
                    bin_heap.bin_temp_stack[i] = NULL;
                }
            } else {
                if (bin_heap.bin_temp_stack[i] == NULL){  //A = not null | B = not null | C = null
                    bin_combine_two_nodes(&bin_heap.bin_stack[i], &bin_heap.bin_temp_node);
                    bin_heap.bin_stack[i] = NULL;
                } else {  //A = not null | B = not null | C = not null
```

```
                    bin_combine_two_nodes(&bin_heap.bin_temp_stack[i], &bin_heap.bin_temp_node);
                    bin_heap.bin_temp_stack[i] = NULL;
                }
            }
        }
    }
}

/*
 * This function is used while doing merging in the way of adding two binary numbers.
 */
void bin_combine_two_nodes(ptr_bin_node *node_adder_1, ptr_bin_node *node_adder_2) {
    ptr_bin_node big_node, small_node;
    big_node = ((*node_adder_1)->distance > (*node_adder_2)->distance) ? (*node_adder_1) : (*node_adder_2);
    small_node = ((*node_adder_1)->distance <= (*node_adder_2)->distance) ? (*node_adder_1) : (*node_adder_2);
    //Let the big node become the left_child of the small one.
    big_node->right_sibling = small_node->left_child;
    small_node->left_child = big_node;
    big_node->parent = small_node;
    bin_heap.bin_temp_node = small_node;
}

/*
 * This function will using one input as source node, traverse the whole graph.
 * After that, the total time will be printed to the console.
 */
void dijkstra_using_adjacent_list() {
    clock_t start, total;
    start = clock();
    int source ;
    ptr_adj_found_node next = NULL;
    printf("Enter Source x (1 <= x <= %d):\n",adj_list.nodes);
    scanf("%d",&source);
    adj_construct(source);
    for (int i = 0; i < adj_list.nodes; i++) {
        //Find the tree whose root node contains the minimal distance.
        next = adj_find_next();
        //Update all the nodes that connect to that node.
        adj_update_distance(next);
    }
    total = clock() - start;
    printf("Using Adjacent List to do Dijkstra needs %.3f seconds\n",(double)total / (double)CLOCKS_PER_SEC);
}

/*
 * This function will update all the nodes that connect to the given node.
 */
void adj_update_distance(ptr_adj_found_node found_node) {
    int find;
    int i;
    ptr_adj_list_node temp_node;
    temp_node = &adj_list.adj_list_node[found_node->index];
    do { // The loop will traverse all the nodes that connect to this node.
        find = temp_node->to;
        for (i = 1; i <= adj_list.found_num; i++) {
            if (adj_list.adj_found_list[i].index == find) {
                //If the node that to be updated is already been found, then update in the array.
                if (adj_list.adj_found_list[i].distance > found_node->distance + temp_node->weight)
                    adj_list.adj_found_list[i].distance = found_node->distance + temp_node->weight;
                break;
            }
        }
        //If the node that to be updated hasn't been found before, then add it to the array.
        if (i > adj_list.found_num) {
            adj_list.adj_found_list[i].distance = found_node->distance + temp_node->weight;
            adj_list.adj_found_list[i].index = find;
            adj_list.adj_found_list[i].visited = false;
            adj_list.found_num++;
        }
        temp_node = temp_node->next;
    }while (temp_node != NULL);
}

/*
 * This function will find the node with the minimal distance in the visited list, then return it's pointer.
 */
ptr_adj_found_node adj_find_next() {
    int min = INT_MAX;  //This will make "Find minimum" more convenient.
    int index_in_found_list = -1;
    ptr_adj_found_node index = NULL;
```

```c
    //This loop will find the minimal distance and corresponding node.
    for (int i = 1; i <= adj_list.found_num; i++) {
        //If the node has been visited before, the skip it.
        if (adj_list.adj_found_list[i].visited == true) continue;
        //Otherwise, record its index.
        if (adj_list.adj_found_list[i].distance <  min) {
            min = adj_list.adj_found_list[i].distance;
            index_in_found_list = i;
        }
    }
    //Mark this newly found node.
    adj_list.adj_found_list[index_in_found_list].visited = true;
    index = &adj_list.adj_found_list[index_in_found_list];
    return index;
}

/*
 * This function is used to initiate the adjacent list
 */
void adj_construct(int source) {
    adj_list.found_num = 1;
    adj_list.adj_found_list[1].visited = false;
    adj_list.adj_found_list[1].index = source;
    adj_list.adj_found_list[1].distance = 0;
}

/*
 * This function will using one input as source node, traverse the whole graph.
 * After that, the total time will be printed to the console.
 */
void dijkstra_using_fibonacci_heap() {
    clock_t start, total;
    int source;
    int index;
    printf("Enter Source x (1 <= x <= %d):\n",adj_list.nodes);
    scanf("%d",&source);
    start = clock();
    //Build the initial heap.
    fib_construct(source);
    for (int i = 0; i < adj_list.nodes; i++) {
        //Delete the node that contains the minimal distance.
        index = fib_delete_min();
        //Update that node's neighbors.
        fib_update_distance(index);
    }
    total = clock() - start;
    printf("Using Fibonacci Heap to do Dijkstra needs %.3f seconds\n",(double)total / (double)CLOCKS_PER_SEC);
}

/*
 * This function will update all the nodes that connect to the previously deleted node.
 */
void fib_update_distance(int index) {
    ptr_adj_list_node temp = &adj_list.adj_list_node[index];
    int fib_index;
    do {
        //Traverse all the neighbors of that node.
        fib_index = temp->to;
        if (fib_heap.fib_node[fib_index].distance > fib_heap.fib_node[index].distance + temp->weight) {
            fib_heap.fib_node[fib_index].distance = fib_heap.fib_node[index].distance + temp->weight;
            //More details will be done in the fib_decrease function
            fib_decrease(fib_index);
        }
        temp = temp->next;
    }while (temp != NULL);
    //Fresh the min_node
    fib_heap.fib_temp_node = fib_heap.fib_min_node;
    for (int i = 0 ;i < fib_heap.tree_num; i++) {
        if (fib_heap.fib_min_node->distance > fib_heap.fib_temp_node->distance) {
            fib_heap.fib_min_node = fib_heap.fib_temp_node;
        }
        fib_heap.fib_temp_node = fib_heap.fib_temp_node->right;
    }
}

/*
 * This function will decrease the distance of the nodes that are connected to the deleted node.
 * Denote the node whose distance is decrease "A"
 * After decrease, if A's distance is smaller than that of A's parent, then A must be move to
 * be a new root instantly, and A's parent should be marked.
 * If A's parent has been marked before, then this procedure will be done cascadingly.
 * All the nodes that are roots are not marked.
 */
void fib_decrease(int index) {
```

```
    ptr_fib_node fib_decreased = &fib_heap.fib_node[index];
    ptr_fib_node temp_parent = fib_decreased->parent;
    if (temp_parent == NULL || temp_parent->distance <= fib_decreased->distance) return;
    while(1) {
        //If the node is a root, then we are done.
        if (temp_parent == NULL){
            fib_decreased->mark = false;
            break;
        }
        //Otherwise, this node will be moved to the root chain.
        //Before that, parent's degree should be check.
        if (temp_parent->degree == 1) {
            temp_parent->child = NULL;
        } else {
            //The moved node has some siblings.
            temp_parent->child = fib_decreased->right;
            fib_decreased->right->left = fib_decreased->left;
            fib_decreased->left->right = fib_decreased->right;
        }
        //Common changes.
        temp_parent->degree--;
        fib_decreased->mark = false;
        fib_decreased->right = fib_heap.fib_min_node;
        fib_decreased->left = fib_heap.fib_min_node->left;
        fib_heap.fib_min_node->left->right = fib_decreased;
        fib_heap.fib_min_node->left = fib_decreased;
        fib_decreased->parent = NULL;
        fib_heap.tree_num++;
        if (temp_parent->mark == false) {
            //If it's parent hasn't been marked before, then we are done.
            temp_parent->mark = true;
            break;
        } else {
            //If it's parent has been marked before, then we have to repeat the previous procedure.
            fib_decreased = temp_parent;
            temp_parent = fib_decreased->parent;
        }
    }
    //Find the new min_node.
    fib_heap.fib_temp_node = fib_heap.fib_min_node;
    for (int i = 0 ;i < fib_heap.tree_num; i++) {
        if (fib_heap.fib_min_node->distance > fib_heap.fib_temp_node->distance) {
            fib_heap.fib_min_node = fib_heap.fib_temp_node;
        }
        fib_heap.fib_temp_node = fib_heap.fib_temp_node->right;
    }
}


// This function will delete a tree whose root contains the minimum distance.
int fib_delete_min() {
    if (fib_heap.fib_min_node == NULL) return 0;
    int previous_index = fib_heap.fib_min_node->index;
    //Denote some nodes for convenience.
    ptr_fib_node temp_deleted = fib_heap.fib_min_node;
    ptr_fib_node temp_child = temp_deleted->child;
    ptr_fib_node temp_left = temp_deleted->left;
    if (temp_deleted->left == temp_deleted) {  //Fibonacci heap only has one tree
        fib_heap.fib_min_node = temp_child;
        for (int i = 0; i < temp_deleted->degree; i++) {
            temp_child->parent = NULL;
            temp_child->mark = false;
            temp_child = temp_child->right;
        }
    } else { //The deleted node is not the only node in root chain.
        //This is not really updating fib_min_node. True node will be updated in reconstruct function.
        fib_heap.fib_min_node = temp_left;
        for (int i = 0; i < temp_deleted->degree; i++) {
            //Move all the children to the root chain.
            temp_left->right = temp_child;
            temp_child->left = temp_left;
            temp_child->parent = NULL;
            temp_child->mark = false;
            temp_child = temp_child->right;
            temp_left = temp_left->right;
        }
        temp_left->right = temp_deleted->right;
        temp_deleted->right->left = temp_left;
    }
    //Update the number of nodes in root chain.
    fib_heap.tree_num = fib_heap.tree_num - 1 + temp_deleted->degree;
    fib_reconstruct();
    return previous_index;
}
```

```c
/*
 * This function is to modify the fibonacci heap after deletion and is VERY important.
 * A temp stack will be used to combine the nodes with the same degree to maintain fibonacci heap's properties
 */
void fib_reconstruct() {
    if (fib_heap.fib_min_node == NULL) return;
    int max_occupy_index = INT_MIN;  //Stores the maximal index of temp stack that are used for faster traversal.
    ptr_fib_node this,next;
    this = fib_heap.fib_min_node;
    //This loop will traverse the root chain and try to put them to the temp stack.
    //If collision happens, then these two will be combined. This will be done in iteration
    for (int i = 0; i < fib_heap.tree_num; i++) {
        next = this->right;
        if (fib_heap.fib_temp_stack[this->degree] == NULL) {
            //This slot hasn't been occupied, then will continue.
            fib_heap.fib_temp_stack[this->degree] = this;
        } else {  //Otherwise, combine them in iteratively
            while (fib_heap.fib_temp_stack[this->degree] != NULL) {
                this = fib_combine_two_nodes_in_temp_stack(this);
            }
            fib_heap.fib_temp_stack[this->degree] = this;
        }
        max_occupy_index = (this->degree > max_occupy_index) ? this->degree : max_occupy_index;
        this = next;
    }
    fib_heap.fib_min_node = NULL;
    fib_heap.tree_num = 0;
    //Combine all the node to form the new fibonacci heap.
    for (int i = 0; i <= max_occupy_index; i++) {
        if (fib_heap.fib_temp_stack[i] == NULL) continue;
        fib_heap.tree_num++;
        if (fib_heap.fib_min_node == NULL) {
            fib_heap.fib_min_node = fib_heap.fib_temp_stack[i];
            fib_heap.fib_min_node->left = fib_heap.fib_min_node;
            fib_heap.fib_min_node->right = fib_heap.fib_min_node;
        } else {
            fib_heap.fib_temp_stack[i]->right = fib_heap.fib_min_node;
            fib_heap.fib_temp_stack[i]->left = fib_heap.fib_min_node->left;
            fib_heap.fib_min_node->left->right = fib_heap.fib_temp_stack[i];
            fib_heap.fib_min_node->left = fib_heap.fib_temp_stack[i];
            fib_heap.fib_min_node = (fib_heap.fib_min_node->distance > fib_heap.fib_temp_stack[i]->distance) ?
                                    fib_heap.fib_temp_stack[i] : fib_heap.fib_min_node;
        }
        //Empty the temp stack
        fib_heap.fib_temp_stack[i] = NULL;
    }
}

/*
 * This function handles collision in the temp stack
 */
ptr_fib_node fib_combine_two_nodes_in_temp_stack(ptr_fib_node new_node) {
    ptr_fib_node old_node;  //The node that is already in the stack
    old_node = fib_heap.fib_temp_stack[new_node->degree];
    ptr_fib_node big_node = (new_node->distance > old_node->distance) ? new_node : old_node;
    ptr_fib_node small_node = (new_node->distance <= old_node->distance) ? new_node : old_node;
    fib_heap.fib_temp_stack[new_node->degree] = NULL;
    //Always combine the greater one to the smaller one.
    if (small_node->child == NULL) {
        small_node->child = big_node;
        big_node->left = big_node;
        big_node->right = big_node;
    } else {
        big_node->right = small_node->child;
        big_node->left = small_node->child->left;
        small_node->child->left->right = big_node;
        small_node->child->left = big_node;
    }
    small_node->degree++;
    big_node->parent = small_node;
    fib_heap.fib_temp_node = small_node;
    return fib_heap.fib_temp_node;
}

/*
 * This function will initiate the fibonacci heap.
 */
void fib_construct(int source) {
    ptr_fib_node temp = &fib_heap.fib_node[1];
    temp->right = temp;
    temp->left = temp;
    temp->parent = NULL;
    temp->child = NULL;
    temp->degree = 0;
```

```
    temp->distance = INT_MAX;
    temp->mark = false;
    temp->index = 1;
    for (int i = 2; i <= adj_list.nodes; i++) {
        fib_heap.fib_node[i].right = &fib_heap.fib_node[i - 1];
        fib_heap.fib_node[i - 1].left = &fib_heap.fib_node[i];
        fib_heap.fib_node[i].left = &fib_heap.fib_node[1];
        fib_heap.fib_node[1].right = &fib_heap.fib_node[i];
        fib_heap.fib_node[i].parent = NULL;
        fib_heap.fib_node[i].child = NULL;
        fib_heap.fib_node[i].mark = false;
        fib_heap.fib_node[i].distance = INT_MAX;
        fib_heap.fib_node[i].degree = 0;
        fib_heap.fib_node[i].index = i;
    }
    fib_heap.fib_node[source].distance = 0;
    fib_heap.fib_min_node = &fib_heap.fib_node[source];
    fib_heap.tree_num = adj_list.nodes;
}

/*
 * This function will read data.
 */
void read_data() {
    char in[200]; //Used to absorb nonsense
    clock_t start, total;
    start =clock();
    int from, to, weight;
    freopen("USA-road-d.NY.gr","r",stdin);  //Read data from a file
//    freopen("test.txt","r",stdin);  //Read data from a file
    for (int i = 0; i < 4; i++) gets(in);  //filter. The first four lines are not needed
    scanf("%s",in);
    scanf("%s",in);
    scanf("%d%d",&adj_list.nodes,&adj_list.edges);
    for (int i = 0; i < 2; i++) gets(in); //filter. The following two lines are not needed
    //Initiate part of data
    for (int i = 0; i < MAX_NODE; i++) {
        adj_list.adj_list_node[i].to = 0;
        adj_list.adj_list_node[i].next = NULL;
        fib_heap.fib_temp_stack[i/2] = NULL;
    }
    for (int i = 0; i < MAX_TEMP_STACK; i++) {
        bin_heap.bin_stack[i] = NULL;
        bin_heap.bin_temp_stack[i] = NULL;
    }
    ptr_adj_list_node temp_node;
    //Read all the possible infomation.
    while (scanf("%s%d%d%d",in,&from,&to,&weight)!=EOF) {
        temp_node = &adj_list.adj_list_node[from];
        while (temp_node->next != NULL) temp_node = temp_node->next;
        if (temp_node->to != 0) {
            temp_node->next = (ptr_adj_list_node)malloc(sizeof(struct adj_list_node));
            temp_node = temp_node->next;
        }
        temp_node->to = to;
        temp_node->weight = weight;
        temp_node->next = NULL;
    }
    freopen("CON","r",stdin);
    total = clock() - start;
    printf("finish read. Running time : %.3f s\n",(double)total/(double)CLOCKS_PER_SEC);
}
```

Appendix III Declaration and Signatures

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.*

Signatures

1. *XXX*
2. *XXX*
3. *XXX*