Advanced Data Structures and Algorithm Analysis

Laboratory Projects

Shortest Path Algorithm with Heaps

Author Names

Zhang Wenqi

Wu Qinran

Chen Yongchen

Date: 2020-03-29

Chapter 1: Introduction

Problem description:

The problem today we are solving is about Dijkstra's algorithm. The task is to implement Dijkstra's algorithm by Fibonacci heap and other min-priority queue structures. We're going to fulfill the algorithm by Fibonacci heap and the minimum heap, and analyze the complexity of algorithm then.

Background of the data structures and the algorithms:

1) Dijkstra's algorithm:

Dijkstra algorithm is a typical single source shortest path algorithm, which is used to calculate the shortest path from one node to all other nodes. The main feature is to expand layer by layer from the starting point to the end point.

The approximate process of this algorithm can be expressed as:

① Treat the initial points on the graph as one set S, and other points as another set

② According to the initial point, calculate the distance d [i] from other points to the initial point (if adjacent, d [i] is the edge weight; if not adjacent, d [i] is infinite)

③ Select the smallest d [i] (record as d [x]), and add the corresponding point (record as x) of this d [i] edge to the set S. (in fact, the d [x] value of the point added to the set is the shortest distance from the initial point.)

④ According to x, update d [y] value of Y adjacent to X: D [y] = min {d [y], d [x] + edge weight w [x] [y]}, this update operation is called relaxation operation.

⑤ Repeat steps 3 and 4 until the target point is added to the set, and the corresponding d [i] of the target point is the shortest path length.

Pseudo code:

```
void Dijkstra( Table T )
{
    Vertex   V, W;
    for ( ; ; ) {
        V = smallest unknown distance vertex;
```

```
            if ( V == NotAVertex )
                break;
            T[ V ].Known = true;
            for ( each W adjacent to V )
                if ( !T[ W ].Known )
                    if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {
                        Decrease( T[ W ].Dist to T[ V ].Dist + Cvw );
                    T[ W ].Path = V;
                } /* end-if update W */
        } /* end-for( ; ; ) */
    }
```

2) Fibonacci heap

Fibonacci heap is a set of disordered trees. Each tree is a minimum heap, which satisfies the property of minimum heap. Fibonacci heap has the advantage that it can complete operations such as heap building, insertion, extraction of minimum key words, union in O (1) time. This is a huge improvement in the efficiency of the binomial reactor.

Chapter 2:   Data Structure / Algorithm Specification

• Minimum heap:

1. The minimum heap is a sort of complete binary tree, in which the data value of any non terminal node is not greater than the value of its child nodes.

2. In the project, we use priority_queue in C++ standard library to implement minimum heap:

priority_queue<pair<long long, long long>,vector<pair<long long, long long> >,greater<pair<long long, long long> > > minheap;

Each node in the heap is of the structure pair<distance, node number>. The comparison results is that the element distance is in increasing order. When the first element is equal, the second element is in increasing order.

3. Some basic operations used:

1) push(): add a node into the heap. Time complexity of push is O(logn).

2) pop(): delete the minimum node from heap. Time complexity of pop is O(logn).

3) empty(): judge whether the heap is empty.

4) top(): returns the constant reference of the minimum node. Time complexity of top is O(1).

4. The time complexity of Dijkstra is $O(n^2)$. By using the heap to optimize the relaxation process, the time complexity can be reduced to

O(eloge).

5. Pseudo code of DijkByMinheap

```
void DijkByMinheap(Type startnode, Type endnode) {
    push(pair(0, startnode));
    while (!empty()) {
        current node = top();
        pop();
        if (current node == endnode)    output shortest distance,return;
        else {
            if (current node is visted)    continue;
            else {
                if (adjnode isn't visited && dist[adjnode]>dist[adjnode]+edge) {
                    dist[adjnode] = dist[current node] + edge;
                }
            }
        }
    }
    output no way;
    return;
}
```

• Fibonacci Heap：

1. Fibonacci heap is a set of disordered trees. Each tree is a minimum heap, which satisfies the property of minimum heap. Fibonacci heap has the advantage that it can complete operations such as heap building, insertion, extraction of minimum key words, union in O (1) time. This is a huge improvement in the efficiency of the binomial reactor。

2. In this program we use two ADT, Fibonacci heap and Fibonacci node

```
struct FibonacciNode{
    Type key; //key value
    Type index; // node index
    Type degree;
    int mark; // whether marked or not
    FibNode left; //left node
    FibNode right; //right node
    FibNode child; // children node
    FibNode parent; //parent node
};
struct FibonacciHeap{
    Type keyNum; // the number of nodes
    Type maxDegree;
    FibNode min; //the minimum node
    FibNode *cons; //children of the previous minimum node
};
create_fibheap();//initialize the fibheap structure
```

```
insertKey(FibHeap heap, Type idx, Type ikey);//insert node with key and index
insertNode(FibNode node, FibNode root);//insert node to the double linked list
removemin(FibHeap heap);//remove and return the minimum node
extractmin(FibHeap heap);//delete the minimum node, adjust the heap, and return the root list
fiblink(FibHeap heap, FibNode node, FibNode root); //link node to the double linked list
fibconsolidate(FibHeap heap); //consolidate heaps with the same degree
fibnodeSearch(FibNode root, Type skey); //search and return the node with key
fibnodeUpdate(FibHeap heap, Type oldkey, Type idx, Type newkey);
```

3. To calculate the distance using Dijkstra algorithm, the pseudo code is as follows:

```
function: DijkByFibheap(startnode, endnode)
    initialize dist -> infinite, visit -> 0 for all the nodes in the adjacent list
    insert all the nodes into fibheap

    min -> startnode
    while min <> endnode
    do
        for p in Adjlist[min] do
            if visit[p] = 1 then
                continue
            else if visit[p] = 0 then
                compare dist[p] between heap.min + distance
                update dist[p] -> min( , )
                visit[p] -> 1
            end if
        end for

        extract the minimum node and update the heap

        if heap.min = infinity then
```

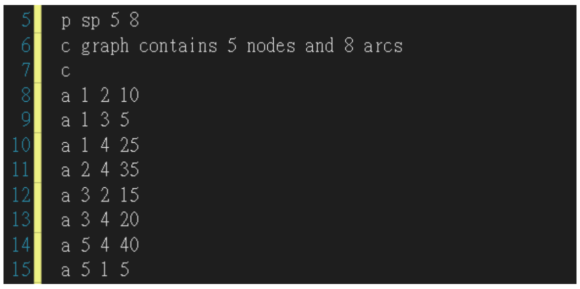
```
            fail to find, return
        else visit[p] -> 2
        end if
    end while
end function
```

Chapter 3: Testing Results

3.1 Accuracy of test program results

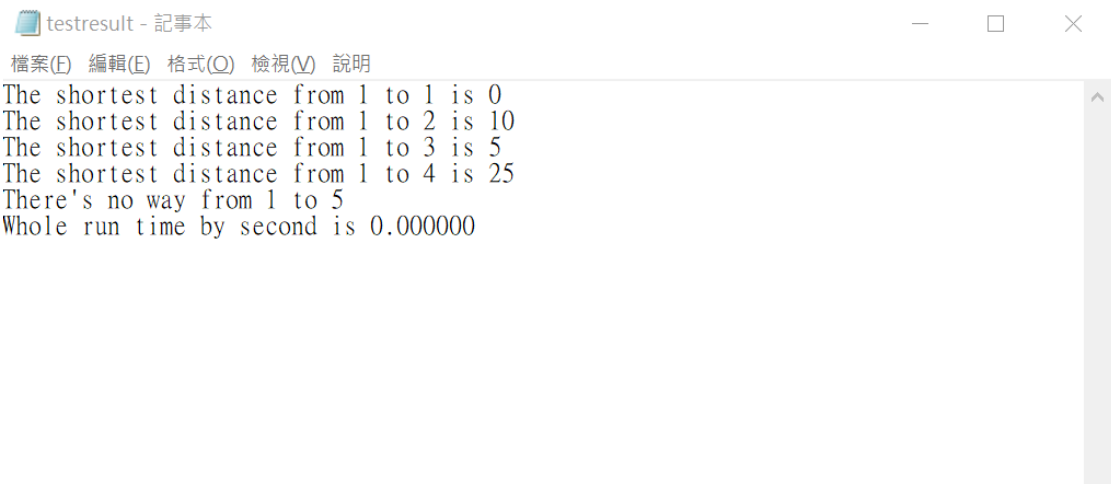
In order to test the correctness of the program, we randomly set a map to test.

Map:



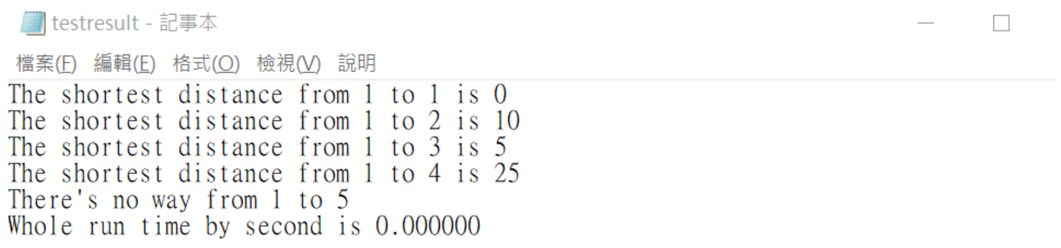
```
p sp 5 8
c graph contains 5 nodes and 8 arcs
c
a 1 2 10
a 1 3 5
a 1 4 25
a 2 4 35
a 3 2 15
a 3 4 20
a 5 4 40
a 5 1 5
```

Minheap			
Input	purpose of this case	output	the current status
1 to 1	Same point	0	*pass*
1 to 2	Choose the shortest path	10	*pass*
1 to 3	Only one path	5	*pass*
1 to 4	Two identical short paths	25	*pass*
1 to 5	No path	There's no way from 1 to 5	*pass*



```
The shortest distance from 1 to 1 is 0
The shortest distance from 1 to 2 is 10
The shortest distance from 1 to 3 is 5
The shortest distance from 1 to 4 is 25
There's no way from 1 to 5
Whole run time by second is 0.000000
```

Fibonacci heap			
Input	purpose of this case	output	the current status
1 to 1	Same point	0	*pass*
1 to 2	Choose the shortest path	10	*pass*
1 to 3	Only one path	5	*pass*
1 to 4	Two identical short paths	25	*pass*
1 to 5	No pass	There's no way from 1 to 5	*pass*



```
The shortest distance from 1 to 1 is 0
The shortest distance from 1 to 2 is 10
The shortest distance from 1 to 3 is 5
The shortest distance from 1 to 4 is 25
There's no way from 1 to 5
Whole run time by second is 0.000000
```

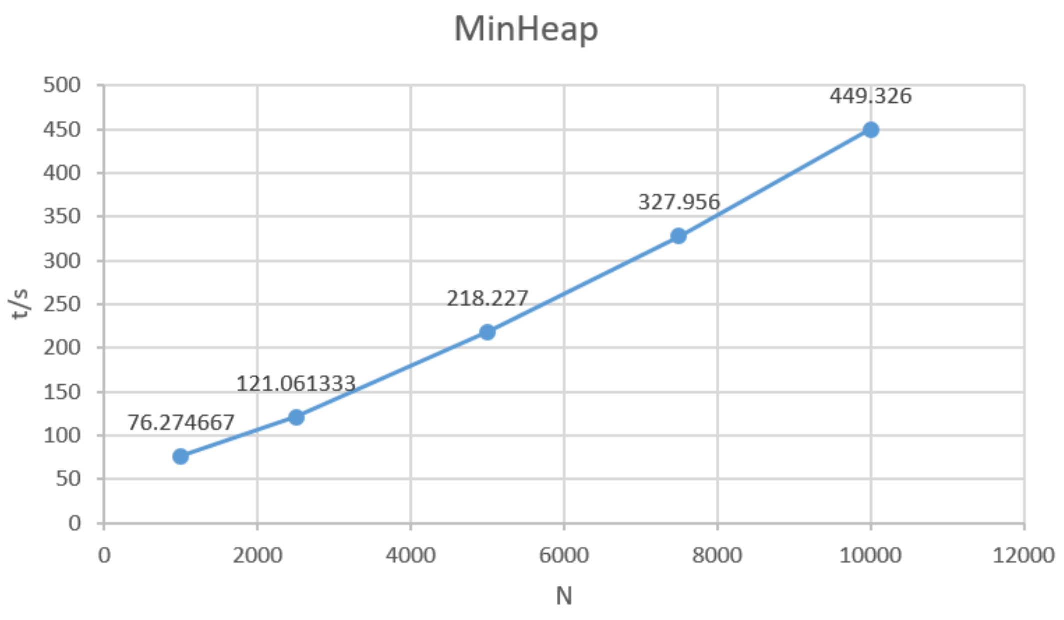
In terms of accuracy, the program has no problems

3.2 Time complexity of program

Next, we will compare the runtimes of two different heaps (Fibonacci heap, Minheap). We will set different values of N (randomly extract N query pairs from the data) and analyze the test results by calculating the running time of the algorithm

The input data sets can be downloaded from http://www.dis.uniroma1.it/challenge9/download.shtml which provides the benchmarks for the 9th DIMACS Implementation Challenge.

Minheap				
N=1000	t1/s	t2/s	t3/s	tavg/s
	81.736000s	71.100000s	75.988000s	76.274667s
N=2500	t1/s	t2/s	t3/s	tavg/s
	118.787000	122.391000	122.006000	121.061333s
N=5000	t1/s	t2/s	t3/s	tavg/s
	226.003000s	220.640000s	208.038000s	218.227000s
N=7500	t1/s	t2/s	t3/s	tavg/s
	329.097000s	323.169000s	331.602000s	327.956000s
N=10000	t1/s	t2/s	t3/s	tavg/s
	443.608000s	427.868000s	476.502000s	449.326000s

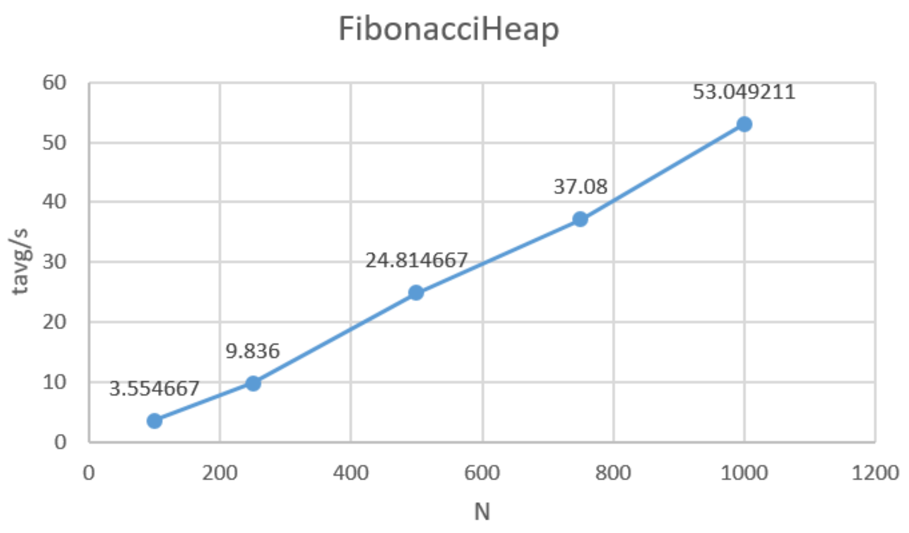


FibonacciHeap				
N=100	t1/s	t2/s	t3/s	tavg/s
	3.745000s	3.490000s	3.429000s	3.554667s
N=250	t1/s	t2/s	t3/s	tavg/s
	9.156000s	11.648000s	8.704000s	9.836000s
N=500	t1/s	t2/s	t3/s	tavg/s

	17.774000s	26.588000s	30.082000s	24.814667s
N=750	t1/s	t2/s	t3/s	tavg/s
	44.713000s	36.709000s	29.818000s	37.080000s
N=1000	t1/s	t2/s	t3/s	tavg/s
	51.106000s	62.035632s	46.006000s	53.049211s

**The input data is self-filled

**Program function is not perfect, there are certain problems



Chapter 4: Analysis and Comments

Analysis :

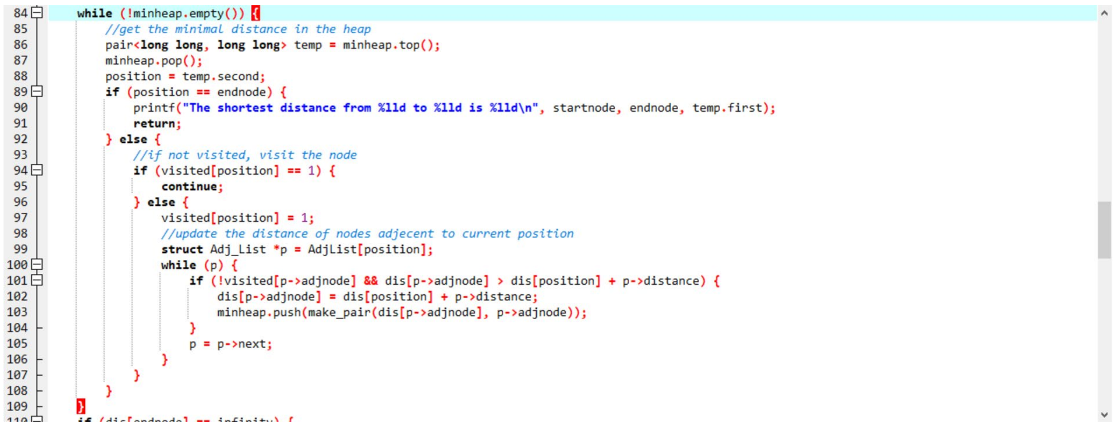
The Dijkstra algorithm is an algorithm to find the shortest path. But its time complexity is too high. The time complexity of this algorithm is O (n^2). So there is a heap for optimization, here we use the minimum heap and FibonacciHeap.

MinHeap-Optimized Dijkstra:

The time complexity of MinHeap-Optimized Dijkstra is O(eloge).

Dijkstra's algorithm uses the time complexity of O(e) to find the edge with the least weight. Now we use the priority queue of STL to optimize this process, reducing the time complexity to O (loge). so that the time complexity of MinHeap-Optimized Dijkstra is reduced to O (eloge).

The space complexity of MinHeap-Optimized Dijkstra is o(e). The space complexity is decided by the total number of edges.

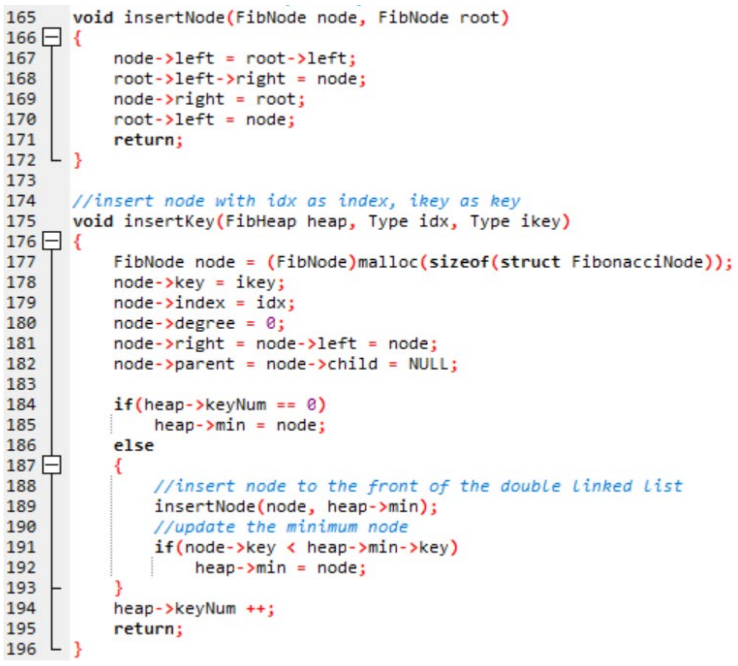


```
    while (!minheap.empty()) {
        //get the minimal distance in the heap
        pair<long long, long long> temp = minheap.top();
        minheap.pop();
        position = temp.second;
        if (position == endnode) {
            printf("The shortest distance from %lld to %lld is %lld\n", startnode, endnode, temp.first);
            return;
        } else {
            //if not visited, visit the node
            if (visited[position] == 1) {
                continue;
            } else {
                visited[position] = 1;
                //update the distance of nodes adjecent to current position
                struct Adj_List *p = AdjList[position];
                while (p) {
                    if (!visited[p->adjnode] && dis[p->adjnode] > dis[position] + p->distance) {
                        dis[p->adjnode] = dis[position] + p->distance;
                        minheap.push(make_pair(dis[p->adjnode], p->adjnode));
                    }
                    p = p->next;
                }
            }
        }
    }
```

(Part of void DijkByMinheap())

FibonacciHeap -Optimized Dijkstra:

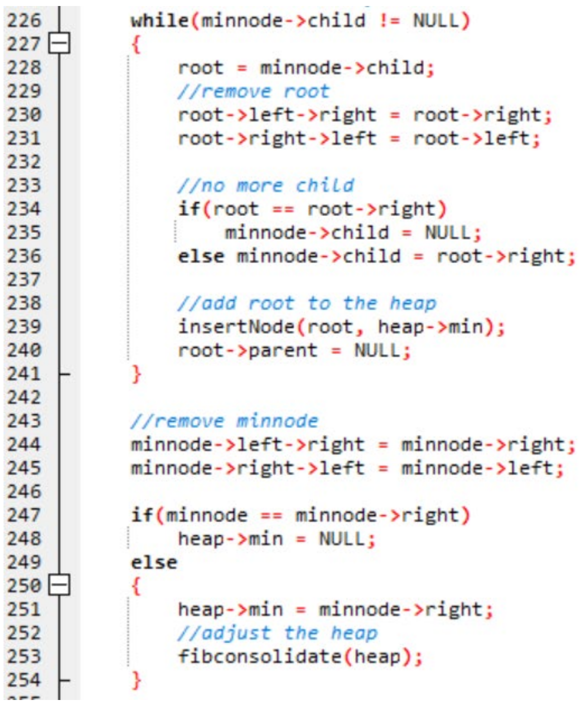
It is very obvious the time complexity of insertKey() and insertNode() are O(1). For the Insert operation, we can know that the number of trees in the tree increases by 1, while the number of labeled nodes does not change. So the amortization time is O (1).



```
void insertNode(FibNode node, FibNode root)
{
    node->left = root->left;
    root->left->right = node;
    node->right = root;
    root->left = node;
    return;
}

//insert node with idx as index, ikey as key
void insertKey(FibHeap heap, Type idx, Type ikey)
{
    FibNode node = (FibNode)malloc(sizeof(struct FibonacciNode));
    node->key = ikey;
    node->index = idx;
    node->degree = 0;
    node->right = node->left = node;
    node->parent = node->child = NULL;

    if(heap->keyNum == 0)
        heap->min = node;
    else
    {
        //insert node to the front of the double linked list
        insertNode(node, heap->min);
        //update the minimum node
        if(node->key < heap->min->key)
            heap->min = node;
    }
    heap->keyNum ++;
    return;
}
```

(insertKey() & insertNode())

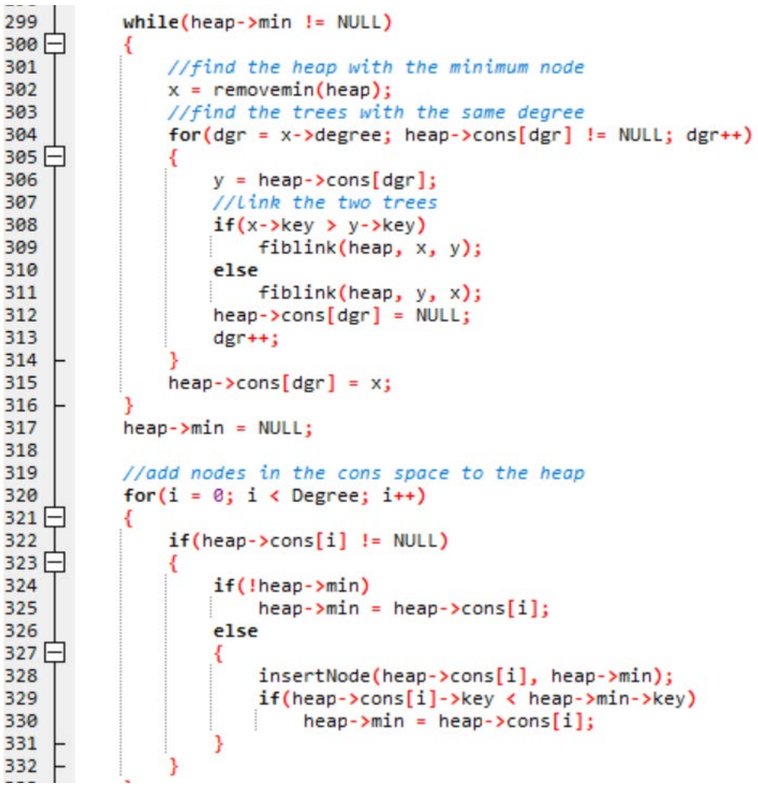
The operation of extracting the minimum node is complicated operation in the Fibonacci heap. First we need to find the minimum root node and delete it, and then all its children are added the root to the heap.



```
    while(minnode->child != NULL)
    {
        root = minnode->child;
        //remove root
        root->left->right = root->right;
        root->right->left = root->left;

        //no more child
        if(root == root->right)
            minnode->child = NULL;
        else minnode->child = root->right;

        //add root to the heap
        insertNode(root, heap->min);
        root->parent = NULL;
    }

    //remove minnode
    minnode->left->right = minnode->right;
    minnode->right->left = minnode->left;

    if(minnode == minnode->right)
        heap->min = NULL;
    else
    {
        heap->min = minnode->right;
        //adjust the heap
        fibconsolidate(heap);
    }
```

(Part of extractmin())

We need to find and maintain the minimum root node of the heap, so we merge the other root nodes with the same degree to the two trees and maintain the state of the array.



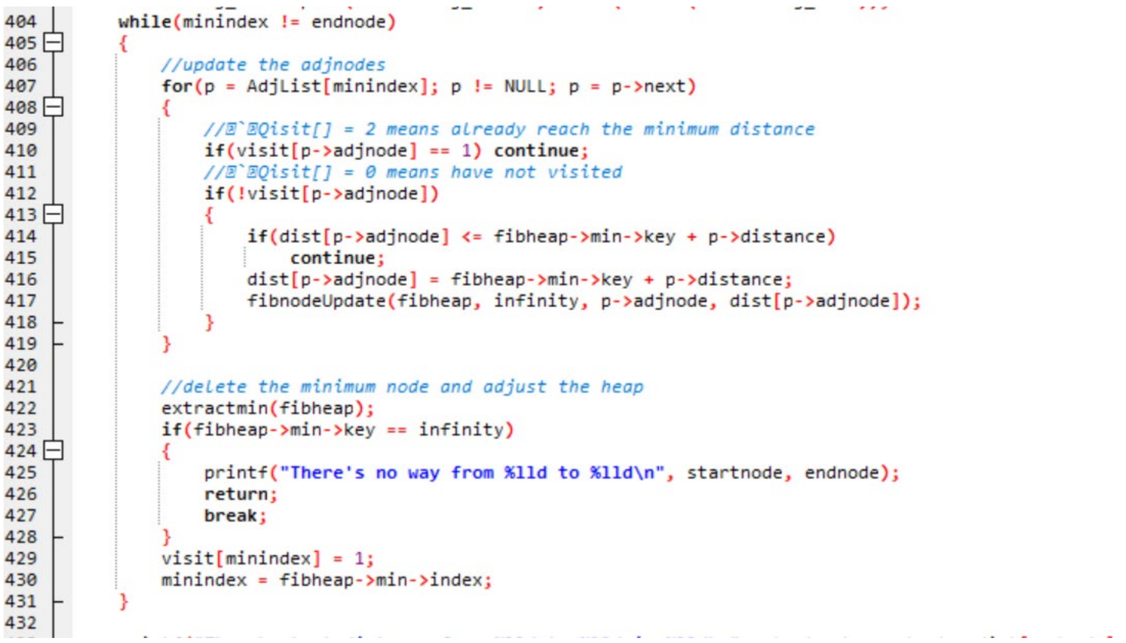
```
    while(heap->min != NULL)
    {
        //find the heap with the minimum node
        x = removemin(heap);
        //find the trees with the same degree
        for(dgr = x->degree; heap->cons[dgr] != NULL; dgr++)
        {
            y = heap->cons[dgr];
            //link the two trees
            if(x->key > y->key)
                fiblink(heap, x, y);
            else
                fiblink(heap, y, x);
            heap->cons[dgr] = NULL;
            dgr++;
        }
        heap->cons[dgr] = x;
    }
    heap->min = NULL;

    //add nodes in the cons space to the heap
    for(i = 0; i < Degree; i++)
    {
        if(heap->cons[i] != NULL)
        {
            if(!heap->min)
                heap->min = heap->cons[i];
            else
            {
                insertNode(heap->cons[i], heap->min);
                if(heap->cons[i]->key < heap->min->key)
                    heap->min = heap->cons[i];
            }
        }
```

(Part of fib_consolidate())

So the time complexity of fib_consolidate() is O(logv).

We only need O(e) time to put all the edges in the fibheap

Then loop v times to find the edge with the smallest value, so the time required is O (vlogv).



```c
    while(minindex != endnode)
    {
        //update the adjnodes
        for(p = AdjList[minindex]; p != NULL; p = p->next)
        {
            //▯`▯Qisit[] = 2 means already reach the minimum distance
            if(visit[p->adjnode] == 1) continue;
            //▯`▯Qisit[] = 0 means have not visited
            if(!visit[p->adjnode])
            {
                if(dist[p->adjnode] <= fibheap->min->key + p->distance)
                    continue;
                dist[p->adjnode] = fibheap->min->key + p->distance;
                fibnodeUpdate(fibheap, infinity, p->adjnode, dist[p->adjnode]);
            }
        }

        //delete the minimum node and adjust the heap
        extractmin(fibheap);
        if(fibheap->min->key == infinity)
        {
            printf("There's no way from %lld to %lld\n", startnode, endnode);
            return;
            break;
        }
        visit[minindex] = 1;
        minindex = fibheap->min->index;
    }
```

The time complexity of FibonacciHeap -Optimized Dijkstra is o(e +vlogv).

The space complexity of FibonacciHeap -Optimized Dijkstra is o(e). The space complexity is decided by the total number of edges.

Comments :

There are about 4 ways to implement Dijkstra Algorithm from known data structures and algorithms. As shown in the table.

Dijkstra Algorithm	time complexity	Space complexity
Array	O(e^2)	O(e)
priority queue of STL	O(eloge)	O(e)
binary heap	O(elogv)	O(e)
FibonacciHeap	O(e+vlogv)	O(e)

This time we chose the implementation of the priority queue of STL and FibonacciHeap. From the table, the time complexity of the binary heap is less than the priority queue of STL. The implementation effect of the binary heap is better but not obvious and the program is more complicated than the program using the priority queue of STL.

Fibonacci heaps except for DeleteMineleteMin and Delete take O (logN) amortization time. All other basic heap operations are a data conclusion of O (1) amortization time.

The use of Fibonacci heaps is more efficient than using the MinHeap after processing each selection, which requires updating the distance between all vertices and the source point. This also means that the time complexity of using the Fibonacci heap will be lower than the time complexity of using the MinHeap.

The time complexity of FibonacciHeap -Optimized Dijkstra is o(ke +vlogv). k is a constant. However, because k is relatively large, the data must have a large scale to show the advantage of complexity.

In choosing which implementation version to use, we need to consider the actual situation, such as the size of the data, the density of the graph, and the time that can be used to write the program.

Appendix: Source Code (if required)

```
//Shortest Path Algorithm with Heaps
//Implement Dijkstra's algorithm with Fibonacci heap and min heap

//Input file is like:
//      p sp 264346 733846
//      c graph contains 264346 nodes and 733846 arcs
//      a 1 2 803
//      a 2 1 803
//      a 3 4 158
//      a 4 3 158
//      a 5 6 774
//      a 6 5 774
//      ......
//

#include<cstdio>
#include<cstdlib>
```

```
#include<cmath>
#include<algorithm>
#include<vector>
#include<queue>
#include<cstring>
#include<string>
#include<cctype>
#include<iostream>
#include<time.h>
#include<windows.h>
using namespace std;


#define maxnode    1000000    //the maximal number of nodes in a map
const int maxline = 100;    //the maximal character number in a line
const long long infinity = 10000000000;

//use adjecency list to save the distance map
struct Adj_List {
        long long adjnode;
        long long distance;
        struct Adj_List *next;
};
struct Adj_List *AdjList[maxnode+5];
long long totalnodes, totalarcs;

//Clock_t is the variable type returned by the clock () function
clock_t start,stop;
//Record the running time of the function under test in seconds
double duration;


typedef long long Type;

typedef struct FibonacciNode *FibNode;
struct FibonacciNode{
        Type key; //key value
        Type index; // node index
        Type degree;
        int mark; // whether marked or not
        FibNode left; //left node
        FibNode right; //right node
        FibNode child; // children node
        FibNode parent; //parent node
};

typedef struct FibonacciHeap *FibHeap;
struct FibonacciHeap{
        Type keyNum; // the number of nodes
        Type maxDegree;
        FibNode min; //the minimum node
        FibNode *cons; //children of the previous minimum node
};

void AddRoad(long long node1, long long node2, long long distance);
void DijkByMinheap(long long startnode, long long endnode);
void DijkByFibheap(long long startnode, long long endnode);

FibHeap create_fibheap();

void insertKey(FibHeap heap, Type idx, Type ikey);
void insertNode(FibNode node, FibNode root);

FibNode getmin(FibHeap heap);//remove and return the minimum node
void deletemin(FibHeap heap);//delete the minimum node, adjust the heap, and return the root list

void fiblink(FibHeap heap, FibNode node, FibNode root); //link node to the double linked list
void fibconsolidate(FibHeap heap); //consolidate heaps with the same degree

int main() {
        char flag;
        char str[maxline];
```

```
long long i=0,j=0;
long long node1, node2, distance;
long long startnode, endnode;
long long query_pairs;    //    total number of query pairs
int heapflag;

printf("Please input the flag of heap(Fibonacci heap is 1, min heap is 2): ");
scanf("%ld", &heapflag);
printf("Please input total number of query pairs : ");
scanf("%lld", &query_pairs);
//read from file
//replace the filename to read different test data file
freopen("USA-road-d.NY.gr", "r", stdin);
while (1) {    //get the flag in front of each line
        scanf("%c", &flag);
        if (flag == 'c') {
                //if 'c', skip the line
                gets(str);
        } else if (flag == 'p') {
                //if 'p', get total number of nodes and arcs
                getchar();getchar();getchar();
                scanf("%lld%lld", &totalnodes, &totalarcs);
                for (i=1; i<=totalnodes ;i++) {
                         AdjList[i] = NULL;
                 }
                gets(str);
        } else if (flag == 'a') {
                //if 'a', get 'node1 node2 distance'
                scanf("%lld%lld%lld", &node1,&node2,&distance);
         AddRoad(node1,node2,distance);
         j++;
                if (j == totalarcs) {    //end input
                        fclose(stdin);
                        break;
                }
        }
        }
        //print the test result into file "testtresult.txt"
        freopen("testresult.txt", "w", stdout);
        srand((unsigned)time(NULL));
        start = clock();
        for (i=0; i<query_pairs; i++) {
        //get startnode and endnode randomly
                startnode = rand()%totalnodes + 1;
                endnode = rand()%totalnodes + 1;
                if (heapflag == 1) {
                 DijkByFibheap(startnode, endnode);
                 } else {
                        DijkByMinheap(startnode, endnode);
                 }
        }
        stop = clock();
        duration = ((double)(stop - start))/CLK_TCK;
        /*Output total run time in S*/
        printf("Whole run time by second is %lf\n", duration);
        fclose(stdout);
        return 0;
}

void AddRoad(long long node1, long long node2, long long distance) {
        //add road into the adjecency list
        struct Adj_List *s1 = (struct Adj_List *)malloc(sizeof(struct Adj_List));
        s1->adjnode = node2;
        s1->distance = distance;
        s1->next = NULL;
        if (!AdjList[node1]) {    //the first adjecent node
                AdjList[node1] = s1;
        } else {
                 s1->next = AdjList[node1]->next;
                 AdjList[node1]->next = s1;
        }
}
```

```
//create Fibonacci heap
FibHeap create_fibheap()
{
    FibHeap heap = (FibHeap)malloc(sizeof(struct FibonacciHeap));
    heap->keyNum = 0;
    heap->maxDegree = 0;
    heap->min = NULL;
    heap->cons = NULL;
    return heap;
}

//insert node to the front of the double linked list root
void insertNode(FibNode node, FibNode root)
{
    node->left = root->left;
    root->left->right = node;
    node->right = root;
    root->left = node;
    return;
}

//insert node with idx as index, ikey as key
void insertKey(FibHeap heap, Type idx, Type ikey)
{
    FibNode node = (FibNode)malloc(sizeof(struct FibonacciNode));
    node->key = ikey;
    node->index = idx;
    node->degree = 0;
    node->right = node->left = node;
    node->parent = node->child = NULL;

    if(heap->keyNum == 0)
        heap->min = node;
    else
    {
        //insert node to the front of the double linked list
        insertNode(node, heap->min);
        //update the minimum node
        if(node->key < heap->min->key)
            heap->min = node;
    }
    heap->keyNum ++;
    return;
}

FibNode getmin(FibHeap heap)
{
    if(!heap || !heap->min)
        return NULL;
    else
        return heap->min;
}

//delete the minimum node and adjust its root nodes
void deletemin(FibHeap heap)
{
    if(!heap || !heap->min)
        return;

    FibNode root = NULL;
    FibNode minnode = heap->min;

    //add the children of the minimum node to the root
    while(minnode->child != NULL)
    {
        root = minnode->child;
        root->mark = 0;
        //remove root
        root->left->right = root->right;
        root->right->left = root->left;
```

```
            //no more child
            if(root == root->right)
                minnode->child = NULL;
            else minnode->child = root->right;

            //add root to the heap
            insertNode(root, heap->min);
            root->parent = NULL;
        }

        //remove minnode
        minnode = heap->min;
        heap->min->left->right = heap->min->right;
        heap->min->right->left = heap->min->left;

        if(minnode == minnode->right)
            heap->min = NULL;
        else
        {
            heap->min = minnode->right;
            //adjust the heap
            fibconsolidate(heap);
        }

        heap->keyNum--;
        free(minnode);
    }

    //add node to be the child of root
    void fiblink(FibHeap heap, FibNode node, FibNode root)
    {
        //remove node
        node->left->right = node->right;
        node->right->left = node->left;
        if(!root->child)
            root->child = node;
        else
            insertNode(node, root->child);

        node->parent = root;
        root->degree++;
        node->mark = 0;
    }

    //consolidate all the subtrees with the same degree
    void fibconsolidate(FibHeap heap)
    {
        Type tmp = heap->maxDegree;
        //the maxDegree of Fibnacci heap is log2N round up to an integar
        heap->maxDegree = (Type)(heap->keyNum) / log(2.0) + 1;

        if(tmp >= heap->maxDegree)
            return;

        //realloc space for cons
        Type Degree = heap->maxDegree + 1;
        heap->cons = (FibNode *)realloc(heap->cons,
            sizeof(FibNode) * Degree);

        Type i;
        for(i = 0; i < Degree; i++)
            heap->cons[i] = NULL;

        FibNode x = NULL, y = NULL;
        Type dgr;

        while(heap->min != NULL)
        {
            //find the heap with the minimum nodem and delete the min node
            x = getmin(heap);
            heap->min->left->right = heap->min->right;
            heap->min->right->left = heap->min->left;
            if(x->right == x)
                heap->min = NULL;
            else
                heap->min = x->right;
            x->left = x;
            x->right = x;

            //find the trees with the same degree
            for(dgr = x->degree; heap->cons[dgr] != NULL; dgr++)
            {
                y = heap->cons[dgr];
                //link the two trees
                if(x->key > y->key)
                    fiblink(heap, x, y);
                else
                    fiblink(heap, y, x);
                heap->cons[dgr] = NULL;
                dgr++;
            }
            heap->cons[dgr] = x;
        }

        //add nodes in the cons space to the heap
        for(i = 0; i < Degree; i++)
        {
            if(heap->cons[i] != NULL)
            {
                if(!heap->min)
                    heap->min = heap->cons[i];
                else
                {
                    insertNode(heap->cons[i], heap->min);
                    if(heap->cons[i]->key < heap->min->key)
                        heap->min = heap->cons[i];
                }
            }
        }
    }
    /*
    //find node with skey as key
    FibNode fibnodeSearch(FibNode root, Type skey)
    {
        if(root == NULL)
            return NULL;

        FibNode tmp = root;
        FibNode result = NULL;

        do
        {
            //equal then found
            if (tmp->key == skey)
            {
                result = tmp;
                break;
            }
            //search in its children
            else
            {
                result = fibnodeSearch(tmp->child, skey);
                if(result != NULL)
                    break;
            }
            tmp = tmp->right;
        }while(tmp != root); //end when double linked list matches

        return result;
    }

    //update the node with oldkey as former key, idx as index, newkey as new key
```

```
void fibnodeUpdate(FibHeap heap, Type oldkey, Type idx, Type newkey)
{
    if(!heap || !heap->min)
        return;
    //find the node with oldkey
    FibNode node = fibnodeSearch(heap->min, oldkey);
    //remove node
    node->left->right = node->right;
    node->right->left = node->left;
    //insert new key
    insertKey(heap, idx, newkey);
}
*/
//Implement Dijkstra's algorithm with Fibonacci heap.
Type dist[maxnode];
Type visit[maxnode];
void DijkByFibheap(long long startnode, long long endnode) {

    Type i;
    FibHeap fibheap = create_fibheap();
    FibNode minnode;

    for(i = 1; i <= totalnodes; i++)
    {
        dist[i] = infinity;
        visit[i] = 0;
    }
    dist[startnode] = 0;
    insertKey(fibheap, startnode, dist[startnode]);
    struct Adj_List *p = (struct Adj_List *)malloc(sizeof(struct Adj_List));

    while(fibheap->min != NULL)
    {
        minnode = getmin(fibheap);

        if(minnode->key <= dist[minnode->index])
        {
            Type minindex = minnode->index;
            visit[minindex] = 1;
            if(visit[endnode])
            {
                printf("The shortest distance from %lld to %lld is %lld\n", startnode, endnode, 
dist[endnode]);
                return ;
            }

            for(p = AdjList[minindex]; p != NULL; p = p->next)
            {
                //visit[] = 2 means already reach the minimum distance
                if(visit[p->adjnode]) continue;
                //visit[] = 0 means have not visited
                if(!visit[p->adjnode])
                {
                    if(dist[p->adjnode] <= dist[minindex] + p->distance)
                        continue;
                    dist[p->adjnode] = dist[minindex] + p->distance;
                    insertKey(fibheap, p->adjnode, dist[p->adjnode]);
                }
            }
        }
        deletemin(fibheap);
    }

    printf("There's no way from %lld to %lld\n", startnode, endnode);
    return;

/*
    Type minindex = startnode;
    struct Adj_List *p = (struct Adj_List *)malloc(sizeof(struct Adj_List));
    while(minindex != endnode)
    {
        //update the adjnodes
        for(p = AdjList[minindex]; p != NULL; p = p->next)
        {
            //visit[] = 2 means already reach the minimum distance
            if(visit[p->adjnode] == 1) continue;
            //visit[] = 0 means have not visited
            if(!visit[p->adjnode])
            {
                if(dist[p->adjnode] <= fibheap->min->key + p->distance)
                    continue;
                dist[p->adjnode] = fibheap->min->key + p->distance;
                fibnodeUpdate(fibheap, infinity, p->adjnode, dist[p->adjnode]);
                visit[p->adjnode] = 1;
            }
        }

        //delete the minimum node and adjust the heap
        extractmin(fibheap);
        if(fibheap->min->key == infinity)
        {
            printf("There's no way from %lld to %lld\n", startnode, endnode);
        return;
        }
        visit[minindex] = 1;
    }

    printf("The shortest distance from %lld to %lld is %lld\n", startnode, endnode, dist[endnode]);
    return ;
*/
}

//Implement Dijkstra's algorithm with min heap.
long long dis[maxnode];    //record the minimal distance
long long visited[maxnode];    //record the node be visited or not
void DijkByMinheap(long long startnode, long long endnode) {
    long long i;
    long long position;    //record the currrent position during searching
    //initialize all the distance to be infinite
    //initialize all the nodes to be unvisited
    for (i=1; i<=totalnodes; i++) {
        dis[i] = infinity;
        visited[i] = 0;
    }
    //set the distance of startnode itself to be 0
    dis[startnode] = 0;
    //set up the minheap of pair<distance, node>
    priority_queue<pair<long long,long long>,vector<pair<long long,long long> >,greater<pair<long
long,long long> > > minheap;
    minheap.push(make_pair(0, startnode));
    while (!minheap.empty()) {
        //get the minimal distance in the heap
        pair<long long, long long> temp = minheap.top();
        minheap.pop();
        position = temp.second;
        if (position == endnode) {
            printf("The shortest distance from %lld to %lld is %lld\n", startnode, endnode,
temp.first);
            return;
        } else {
            //if not visited, visit the node
            if (visited[position] == 1) {
                continue;
            } else {
                visited[position] = 1;
                //update the distance of nodes adjecent to current position
                struct Adj_List *p = AdjList[position];
                while (p) {
                    if (!visited[p->adjnode] && dis[p->adjnode] > dis[position] + p->distance) {
                        dis[p->adjnode] = dis[position] + p->distance;
                        minheap.push(make_pair(dis[p->adjnode], p->adjnode));
                    }
                    p = p->next;
                }
```

```
                    }
               }
          }
          if (dis[endnode] == infinity) {
          printf("There's no way from %lld to %lld\n", startnode, endnode);
          return;
          }
          return;
}
```

References

[1] Mark Allen Weiss, "Data Structures and Algorithm Analysis in C", *机械工业出版社*,p.345~351, (2004.1)

Author List

Zhang Wenqi: Implement Dijkstra's algorithm with Fibonacci heap.

Write Chapter 2 in the report (Fibonacci heap part).

Wu Qinran: Implement Dijkstra's algorithm with Minimum heap.

Write a test of performance program.

Write Chapter 2 in the report (Minimum heap part).

Write Chapter 1 in the report.

Chen Yongchen:

Provide the necessary inputs for testing and give the run time table.

Plot the run times vs. input sizes for illustration.

Write analysis and comments.

Write Chapter 3&4 in the report.

Complete the PPT.

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

张雯琪

陈咏琛

吴沁玥