Project 2

Shortest Path Algorithm with Heaps

Date:2023-4-1

Chapter 1 Introduction

Problem description: In this project, we are required to compute the shortest paths of a weighted digraph with single source using Dijkstra algorithm. And at least two heap structures are needed to be used, while Fibonacci heap must be one of them. (We choose Binary heap to be the other heap structure.)

Purpose: Find the best data structure for Dijkstra algorithm. Study the use and efficiency of Fibonacci heap. Compare the Fibonacci heap with Binary heap.

Background: We have learned the Binomial heap data structure in privious lessons, and we also have learned Dijkstra algorithm in the course Fundation of Data Structure last term. The Fibonacci heap structure we used in the program is an optimization of binomial heap in time complexity. In this project we will also use a basic data structure, binary heap, to observe the advantage of Fibonacci heap in time complexity. And we use C++ to solve the problem.

Input: The test data set can be download from

http://www.diag.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.USA.gr.gz

In the data file, the line beginning with a character 'p' is followed by the type of the type of the graph 'sd', then the number of nodes and edges of the graph respectively. The next each line beginning with 'a' is followed by the head, tail and weight of each edge. The lines beginning with 'c' is some explanations of the file which will not be read by our program. The graph includes 23947347 vertices and 58333344 edges.

After the graph being read, the user are supposed to input the starting vertex and end vertex, which should be seperated by a space.

Output: We will calculate the shortest path form source to every vertex in the graph, and we write the result shortest distance between sources and each vertex in a text file "Result_of_all_node.txt".

Chapter 2 Algorithm Specification / Data Structure

Header: We ues following Header in our program and attach our explanation after each of them.

<iostream> Header that defines the standard input/output stream objects

<utility> Header containing utilities in unrelated domains including pairs(objects that can hold two values of different types)

<fstream> Header providing input/output file stream class

<vector> Header that defines the vector container class. Vectors are sequence containers representing arrays that can change in size.

<queue> Header that defines the queue and priority_queue container adaptor classes. Queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

<algorithm> Header that defines a collection of functions especially designed to be used on ranges of elements.

<cmath> Header that declares a set of functions to compute common mathematical operations and transformations

<string> //The standard string class provides support for such objects with an interface similar to that of a standard container of bytes, but adding features specifically designed to operate with strings of single−byte characters.

<time.h> Used for tesing programing time.

"Fib_Heap.h" Header that includes the Fibonacci heap code we wrote.

Dijkstra Algorithm: The Dijksrta shortest path can solve the shortest path probleam as following steps.

- Create a priority queue to store the shortest path from source to each vertwx.
- Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.
- While sptSet doesn' t include all vertices
  - Pick a vertex u that is not there in sptSet and has a minimum distance value. (Delete the minimum vertex from priority queue.)
  - Include u to sptSet.
  - Then update the distance value of all adjacent vertices of u.
    - To update the distance values, iterate through all adjacent vertices.
    - For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u−v, is less than the distance value of v, then update the distance value of v. (Adjust the position of the vertex v in th priority queue.)

We use a int array vis[] to represent the set of vertices included in SPT. If a value sptSet[v] is 1, then vertex v is included in SPT, otherwise not. Array d[] is used to store the shortest distance values of all vertices.

Note: There are some differences between our Dijkstra algorithm and the traditional Dijkstra algorithm showed above. In our Dijkstra algorithm, we omitted the creating heap operation but insert each vertex into the binary heap through the path we have visited. So the decresing key operation is replaced by insertion operation. Although it may cause the same vertex to be inserted into heap repeatedly when its shortest distance is updated, but its older node in heap will be ignored with a judgement vis[], which is used to mark if the vertex has been collected. Thus the method will not impact correctness of the final result. Moreover, it can save the space when deal with big data so we don' t have to save redundant vertices in the heap which is not in the path from start point to end point. It also save the time for creating heap.

Binary heap: The following are the essential operations we use when implementing a bainary heap data structure:

heapify: rearranges the elements in the heap to maintain the heap property.

Drease_min: decrease the key of an item and ajust its position in the heap.

Delete_min: removes the minimum item in a heap.

Percolate_down: adjust the position of an item with key smaller than its child' s key.

Percolate_up: adjust the position of an item with key bigger than its parent' s key.

Fibonacci heap: As the name suggests, Fibonacci heaps use Fibonacci numbers in its structure. Like binomial heaps, Fibonacci heaps use doubly linked lists to allow for O(1) time for operations such as splicing off a part of a list, merging two lists, and finding the minimum (or maximum) value. Each node contains a pointer to its parent and any one of its children. The children are all linked together in a doubly linked list called the child list. Each child in the child list has pointers for its left sibling and its right sibling.

For each node, the linked list also maintains the number of children a node has, a pointer to the root containing the minimum key, and whether the node is marked. A node is marked to indicate that it has lost a single child and a node is unmarked if it loses no children.

Here is a pseudocode implementation of Fibonacci heaps[1][2].

```
Make-Fibonacci-Heap()
n[H] := 0
min[H] := NIL
return H

Fibonacci-Heap-Minimum(H)
return min[H]

Fibonacci-Heap-Link(H,y,x)
remove y from the root list of H
make y a child of x
degree[x] := degree[x] + 1
mark[y] := FALSE

CONSOLIDATE(H)
for i:=0 to D(n[H])
     Do A[i] := NIL
for each node w in the root list of H
    do x:= w
       d:= degree[x]
       while A[d] <> NIL
           do y:=A[d]
              if key[x]>key[y]
                then exchange x<->y
              Fibonacci-Heap-Link(H, y, x)
              A[d]:=NIL
```

```
            d:=d+1
       A[d]:=x
min[H]:=NIL
for i:=0 to D(n[H])
    do if A[i]<> NIL
          then add A[i] to the root list of H
               if min[H] = NIL or key[A[i]]<key[min[H]]
                  then min[H]:= A[i]

Fibonacci-Heap-Union(H1,H2)
H := Make-Fibonacci-Heap()
min[H] := min[H1]
Concatenate the root list of H2 with the root list of H
if (min[H1] = NIL) or (min[H2] <> NIL and min[H2] < min[H1])
   then min[H] := min[H2]
n[H] := n[H1] + n[H2]
free the objects H1 and H2
return H

Fibonacci-Heap-Insert(H,x)
degree[x] := 0
p[x] := NIL
child[x] := NIL
left[x] := x
right[x] := x
mark[x] := FALSE
concatenate the root list containing x with root list H
if min[H] = NIL or key[x]<key[min[H]]
        then min[H] := x
n[H]:= n[H]+1

Fibonacci-Heap-Extract-Min(H)
z:= min[H]
if x <> NIL
        then for each child x of z
             do add x to the root list of H
                p[x]:= NIL
             remove z from the root list of H
             if z = right[z]
                then min[H]:=NIL
                else min[H]:=right[z]
```

```
                    CONSOLIDATE(H)
            n[H] := n[H]-1
return z

Fibonacci-Heap-Decrease-Key(H,x,k)
if k > key[x]
   then error "new key is greater than current key"
key[x] := k
y := p[x]
if y <> NIL and key[x]<key[y]
   then CUT(H, x, y)
        CASCADING-CUT(H,y)
if key[x]<key[min[H]]
   then min[H] := x

CUT(H,x,y)
Remove x from the child list of y, decrementing degree[y]
Add x to the root list of H
p[x]:= NIL
mark[x]:= FALSE

CASCADING-CUT(H,y)
z:= p[y]
if z <> NIL
  then if mark[y] = FALSE
       then mark[y]:= TRUE
       else CUT(H, y, z)
            CASCADING-CUT(H, z)
```

To embed fibonacci heap into Dijkstra algorithm, we adjust some details of Fibonacci heap, but the core frame of the structure is the same.

Chapter 3 Test Results

1. Test for Correctness

Both two programs can solve the problem correctively with small data.

Test Example	Start & End	Binary Heap Result	Fibonacci Heap Result	Comment
c This is a simple graph. p sd 3 2 a 1 2 1 a 2 3 1	1 3	2	2	Correct
p sd 5 6 a 1 2 50 a 2 3 100 a 2 4 250 a 3 4 130 a 3 5 70 a 4 5 40	1 4	280	280	Corecct

2. **Test for Time complexity**

Data structure	Binary Heap	Fibonacci Heap
M, N	23947347, 58333344	
Pairs	1000	1000
Ticks	5971622.000000	15197298.000000
Total time(sec)	5971.622000	15197.298000
Iterations	1 / pair × 1000 pair	1 / pair × 1000 pair
Duration /pair(sec)	5.972	15.197

In the test, we choose 1000 pairs of vertices randomly to compute the shortest distance between them for both heaps.

Beyond our expectation, in the test for time complexity, binary heap performs better than Fibonacci heap.

Chapter4 Analysis and Comments

In the Dijkstra algorithm, firstly, we need to make a priority queue to store the shortest path from source to each vertex. Secondly, we divides all vertices into two part, one in the set SPT(as mentioned in Chapter 2) and the other in the set T(in the tree but not in the set SPT). Each time a point v is selected from the set T with the shortest distance from the set S, and the point v is added to the set S. Meanwhile, The shortest distance of the points in T is updated. Repeat this step until no point in T can be found adjacent to S.

But in our Dijkstra algorithm, we omitted the creating heap operation but insert each vertex into the binary heap with the path we have visited. So the decresing key is replaced by insertion. Thus the time complexity of Dijkstra algorithm can be calculated as the following formula,

$$(n-1) * (T_{DELETE\_MIN} + T_{INSERT} * k)$$

where the number of vertices is n and the number of edges is m, the average number of edges per vertex is k = m / n. The 2nd step will be executed n−1 times. There are two ways, binary heap and Fibonacci heap, to realize it in our project.

In binary heap, the time complexity of MAKE_HEAP is O(n), the time complexity of DELETE_MIN is O(log n), and the time complexity of PERCOLATE_UP is O(log n). Thus the time complexity of Dijkstra algorithm is O((n + m) * log n).

In Fibonacci heap, the time complexity of MAKE_HEAP is O(n), the time complexity of DELETE_MIN is O(log n), and the time complexity of INSERTION is O(1). Thus the time complexity of Dijkstra algorithm is O(n * log n + m).

Theoretically speaking, Fibonacci heap has better time complexity than binary heap. And The space complexity of Dijkstra algorithm by both ways is O(n).

However, in the actual test, we find that binary heap performs better than Fibonacci heap. We suspect that it is due to too many pointers being calling during the implementation of Fibonacci heap . This may cause a mess in memory allocation and prolong the process.

Appendix: Source Code

We use two program to solve the problem. One program use Binary heap and the other use Fibonacci. Both of them are attached to the "code" folder.

minHeap.hpp

```
#ifndef _MIN_HEAP_HPP
#define _MIN_HEAP_HPP
#include <cmath>
#include <algorithm>
using namespace std;
template <class T, int capacity>
class minHeap
{
public:
    minHeap() = default; //constructor
    ~minHeap() = default; //destructor
    bool empty() //Judge if the heap is empty
    {
        return size == 0 ? true : false;
    }
    void push(T value) //push a new node to the heap
    {
        min_heap[++size] = value;
        PercolateUp(size); //Adjust its position in heap
    }
    T deletemin() //Delete the minimum vertex
    {
        T temp = min_heap[1];
        swap(min_heap[1], min_heap[size]); //Replace the  minimum vertex with the last vertex
        size--; //Size of heap minus 1
        PercolateDown(1); //Percolate down the vertex swapped to the first position
        return temp;
    }
```

```
    void clear() //Clear the heap
    {
        size = 0;
    }
private:
    T min_heap[capacity];
    int size = 0; //Size of the heap
    void PercolateUp(int p) //Percolate up
    {
        if (p == 1) //If it is in the first position, stop percolate down
        {
            return;
        }
        if (min_heap[p] < min_heap[p >> 1]) //If the key of the current vertex is smaller than its parent's
        {
            swap(min_heap[p], min_heap[p >> 1]); //swap the current vertex and its parent
            PercolateUp(p >> 1); //Continue to percolatedown
        }
        return;
    }
    void PercolateDown(int p) //Percolate down
    {
        int child = p << 1; //index of children = index of parent * 2
        if(child + 1 < size && min_heap[child + 1] < min_heap[child])
        {
            child++; //Find the child with smaller key
        }
        if(child < size && min_heap[child] < min_heap[p])
        {
            swap(min_heap[child], min_heap[p]); //swap the current vertex and its child with smaller key
            PercolateDown(child); //Continue to percolate
        }
    }
};
#endif
```

Dijkstra_with_minHeap.cpp

```
#include <iostream>
#include <utility>
#include <fstream>
#include <cmath>
#include <string.h>
#include <stdio.h>
#include <time.h>
```

```
#include <math.h>
#include "minHeap.hpp" //Header that includes the Binart min-heap classes
using namespace std;
#define INF 0x7fffffff
#define MAX_NODE 23950000 //The maximum of vertices permitted
#define MAX_EDGE 58340000 //The maximum of edges permitted
clock_t start, stop;
double duration;
struct EDGE
{
    int to; //The tail of the edge
    int w; //The weight of the edge
    int next; //The next edge with the same head as that of the current edge
} edge[MAX_EDGE]; //Array of edges
int head[MAX_NODE]; //The first adjacent edge of each node
int cnt = 0; //The number of edges already added to the graph
bool vis[MAX_NODE]; //Flag if each node is collected
inline void addedge(int u, int v, int w) //Add the current edge to the graph
{
    edge[++cnt].to = v; //The number of edges+1 and set the end of  the edge to be v
    edge[cnt].next = head[u]; //The next the edge from u is linked to the current edge from u
    edge[cnt].w = w; //The weight of the current edge is set to be w
    head[u] = cnt; //The first edge from u is set to be current edge
}
int d[MAX_NODE]; //Store the shortest path from the source to each vertex
int n; //The number of nodes
int m; //The number of edges
int s; //The source of the path
int t; //The terminal of the path

class Node
{
public:
    int dis;
    int vertex;
    Node() = default; //ctor
    ~Node() = default; //detor
    Node(int a, int b)
    {
        dis = a; //The shortest distance
        vertex = b; //The identifier of the vertex
    }
    friend bool operator<(Node a, Node b)
    {
```

```
        return a.dis < b.dis;
    }
};
minHeap<Node, MAX_NODE> H; //Construct a Heap with struct Node as its key and max size is MAX_NODE
void dijkstra_with_minHeap(int start, int des) //Dijkstra algorithm to calculate the distance of the shortest
path from the source to each node
{
    d[start] = 0; //Set the shortest path from the source to the source to be 0
    H.push(Node(0, s)); //Push the source to binary heap
    while (!H.empty()) //While the binary heap is not empty
    {
        int now = H.deletemin().vertex; //Delete and return the minimum vertex from binary heap
        if (now == des) //If the current vertex is end vertex
        {
            cout << "shortest distance: " << d[des] << endl; //Print out the result
            return; //Exit the function
        }
        if (vis[now]) //If the vertex is collected
            continue; //Skip
        vis[now] = true; //Collect the minimum vertex
        for (int i = head[now]; i; i = edge[i].next) //Go through every edge from the current vertex
        {
            EDGE &e = edge[i];
            if (d[now] + e.w < d[e.to]) //If the shortest path through the current node to e.to is shorter than
the shortest path to e.to having stored
            {
                d[e.to] = d[now] + edge[i].w; //Update the shortest distance of the path to e.to
                H.push(Node(d[e.to], e.to)); //Push the edge to binary heap
            }
        }
    }
}
int main()
{
    cout << "Reading graph..." <<endl;
    fstream FILE_in;
    FILE_in.open("E:/ADS/USA-road-d.USA.gr", ios::in); //Open graph file to read in
    if (!FILE_in.is_open()) //If fail to open the graph file
    {
        cout << "Could not open the data file!" << endl; //Warning
        exit(EXIT_FAILURE); //An unsuccessful termination status is returned to the host environment
    }
    char flag;
    int u; //The head of an edge
```

```
    int v; //The tail of an edge
    int w; //The weight of an edge
    string temp;
    while (FILE_in >> flag)
    {
        if (flag == 'c') //If the line begins with 'c', it is a comment line
        {
            getline(FILE_in, temp); //It doesn't need to be read in to graph, so skip
        }
        else if (flag == 'p') //If the line begins with 'p'
        {
            FILE_in >> temp >> n >> m; //Read the identifier 'sp' into temp and read the number of nodes and edges
of the graph into n an m respectively
        }
        else if (flag == 'a') //If the line begins with 'a'
        {
            FILE_in >> u >> v >> w; //Read the head, tail and weight of the current edge
            addedge(u, v, w); //Add the curent edges to the graph
        }
    }
    FILE_in.close(); //Close the graph file
    cout << "Please input the start vertex and end vertex." << endl;
    cout << "For example: 1 19260817" << endl;
    cin >> s >> t; //Read in the start vertex and end vertex
    for (int i = 0; i <= n; i++)
    {
        d[i] = INF; //Initialize the shortest distance to be INFINITY for each vertex
    }
    memset(vis, 0, sizeof(vis)); //Flag all the vertices to be not collected
    H.clear(); //Clear the binary heap
    start = clock();
    dijkstra_with_minHeap(s, t); //Calculate the distance of the shortest path from source to each node with
Dijsktra algorithm
    stop = clock();
    duration = ((double)(stop - start)) / CLK_TCK;
    cout << "ticks: " << (double)(stop - start) << endl;
    cout << "duration: " << duration << "sec" << endl;//print the duration
//    fstream FILE_out;
//    FILE_out.open("Result_of_all_node_by_minHeap2.txt", ios::trunc | ios::out);
//    if (!FILE_out.is_open())
//    {
//        cout << "Could not open the result file!" << endl;
//        exit(EXIT_FAILURE);
//    }
```

```
//    for (int i = 1; i <= n; i++)
//    {
//        FILE_out << d[i] << " ";
//    }
//    FILE_out.close();
//    cout << "The result of all node has been submitted in the file \"Result_of_all_node_by_minHeap2.txt\"" <<
endl;
    return 0;
}
```

Fib_Heap.hpp

```
#ifndef _FIB_HEAP
#define _FIB_HEAP
#include <cmath>
using namespace std;
template <class T> //data field we need
class FibNode //the structure of nodes in FibHeap
{
public:
    FibNode() // initialize a new and empty FibHeap
        : p(nullptr), child(nullptr), left(this), right(this), degree(0), mark(false){};
    FibNode(T value) // initialize a new and empty FibHeap with the data we store
        : p(nullptr), child(nullptr), left(this), right(this), degree(0), mark(false), data(value){};
    ~FibNode() // free the FibHeap
    {
        if(child != nullptr)//delete the non-empty child node
        {
            delete child;
        }
        FibNode *temp = nullptr;
        FibNode *that = this;
        while(that != nullptr)//delete this node
        {
            temp = that;
            that = right;//while pointer move to its right sibling
            delete temp;
        }
    }
    T data; //data field we store
    int degree; //the number of children the node have
    bool mark; //if the node has losen its child after it became someone's child last time
    FibNode *p; //the parent of the node
    FibNode *child; //one of the children of the node
    FibNode *left; //the left siblings of the node
```

```
    FibNode *right; //the right siblings of the node
};
template <class T>
class FibHeap  //the structure of FibHeap
{
public:
    typedef FibNode<T> *FibNodePtr;
    FibNodePtr minNode; //point to the minimum root of the whole heap
    FibNodePtr head; //the head of root linked list
    int size; //the number of root nodes in the root linked list
    FibHeap() // initialize a new and empty FibHeap
    {
        minNode = nullptr;
        head = nullptr;
        size = 0;
    }
    ~FibHeap() // free the FibHeap
    {
        FibNodePtr temp = nullptr;
        while(head->right != head) //remove all root nodes from head to the end
        {
            temp = head->right;
            RemoveNode(head);
            delete head;
            head = temp;
        }
        delete head;
        delete minNode; //delete the minimum root of the heap
    }
    void Insert(T value) //insert a node to Fibheap whose datas are "value"
    {
        FibNodePtr temp = new FibNode<T>(value); //allocate memory for newNode
        if (minNode == nullptr)//if minNode is empty, fill it with the new node
        {
            minNode = new FibNode<T>();
            head = new FibNode<T>();
            minNode = temp;
            LeftInsert(head, minNode);//insert minNode to the left of head
        }
        else
        {
            LeftInsert(minNode, temp);//insert temp to the left of minNode
            if (value < minNode->data) //if new value is smaller, adjust the location of minNode
            {
```

```
            minNode = temp;//to make sure minNode is the minimum data
        }
    }
    size++;//update the the size of heap
}
T DeleteMin()// to delete the minNode and return the value of it
{
    T result = minNode->data; //fetch the minimum value
    FibNodePtr tempMin = minNode;
    if (tempMin != nullptr)//ensure the minNode does exist
    {
        FibNodePtr temp = minNode->child;//record the child of minNode
        FibNodePtr next;
        for (int i = 0; i < minNode->degree; i++) //for each child of minNode
        {
            next = temp->right;//record the location of next child
            LeftInsert(head, temp);///insert temp to the left of head in root list
            temp = next;//continue to do it
        }
        RemoveNode(tempMin); //remove the minNode from root list
        if (tempMin == tempMin->right)//if only minNode exists in root list
        {
            minNode = nullptr;//set it as nullptr because it is empty
        }
        else //if still other nodes in root list
        {
            minNode = head->right; //let the node on right of head be minNode temporarily
            Consolidate();//consolidate the root nodes which have the same degree, and find the location of
minNode
        }
        size--;//update the size of heap
    }
    return result;//return data of minNode
}
bool isEmpty()//recognize the heap is empty or not
{
    return (size == 0);//size = 0 means it is empty
}
void clear()//clear the heap
{
    size = 0; //size to be zero
    minNode = nullptr;
    head = nullptr; //pointer to be nullptr
}
```

```
private://private functions play in inner space
    void LeftInsert(FibNodePtr old, FibNodePtr cur) //set cur to the left of old
    {
        cur->left = old->left;
        cur->right = old;
        old->left->right = cur;
        old->left = cur;
    }
    void RemoveNode(FibNodePtr node) //to remove node from the current list
    {
        node->left->right = node->right;
        node->right->left = node->left;//just let its siblings pointer to each other
    }
    void Link(FibNodePtr y, FibNodePtr x)//to make y child of x
    {
        RemoveNode(y);//remove y from its current child of root list
        y->p = x; //set y.parent as x
```

```
        if (x->child == nullptr)//if x has no child
        {
            x->child = y;//just let x.child pointer to y
            y->left = y->right = y;
        }
        else //if children exist in x'child list
        {
            LeftInsert(x->child, y);//add y into the child list
        }
```

```
        x->degree++;//update the degree of x (the number of children x has)
        y->mark = false; //y to be child again, not in the root list
    }
    void Consolidate()//consolidate the root nodes which have the same degree, and find the location of minNode
    {
        int D = (int)log2(size) + 1; //the upper bound of the number of roots
        FibNodePtr *Array = new FibNodePtr[D];//allocate memory for a node array Array
        for (int i = 0; i < D; i++)
        {
            Array[i] = nullptr;//initialize the array Array
        }
        FibNodePtr temp = head->right; //locate one of the root
        FibNodePtr next; //locate the right node of current node
```

```
        do
        {
            if (temp == head)//while iterate all nodes and come back the beginning of root
            {
                break;//exit the loop
            }
            int deg;
            FibNodePtr x, y;
            x = temp;
            deg = x->degree; //record the degree of the current node
            next = temp->right;//store the next location
            while (Array[deg] != nullptr)//if there has existed a node with same degree in the array
            {
                y = Array[deg];//fetch the existed node as y
                if ((y->data) < (x->data))//compare x and y, if y is smaller
                {
                    swap(x, y);//swap x and y, to ensure x is smaller than y, and x will become the parent of
y
                }
```

```
                Link(y, x);//set y as the child of x, which means consolidate them
                Array[deg] = nullptr;//clear Array[degree]
                deg++;//after consolidation, the degree of new node will increase, so enter the loop again
            }
            Array[deg] = x;//node x with degree "deg" to store in Array[deg]
            x->degree = deg;
            temp = next;//come to next root node
        } while (temp != head);//until all root node has been traveled
        minNode = nullptr;//clear minNode for the following relinking
        head->left = head->right = head;//clear root list and remain only the empty head
        for (int i = 0; i < D; i++)//iterate the node Array to rebuild the root list
        {
            if (Array[i] != nullptr)//if exist node with degree of i, include it in the list
            {
                if (minNode == nullptr)//if root list is empty
                {
                    minNode = Array[i];
                    LeftInsert(head, minNode);//add it as minNode
                }
                else//if exist nodes in root list
                {
                    LeftInsert(head, Array[i]);//first include it in list
```

```
                    if (Array[i]->data < minNode->data)//then update the minNode if new node is smaller than the
former one
                    {
                        minNode = Array[i];
                    }
                }
            }
        }
        delete[] Array;//delete the auxiliary array
    }
};
#endif
```

Dijkstra_with_FibHeap.cpp

```
#include <iostream>
#include <utility>
#include <fstream>
#include <cmath>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include "Fib_Heap.hpp"
using namespace std;
#define INF 0x7fffffff
#define MAX_NODE 23950000
#define MAX_EDGE 58340000
clock_t start, stop; //tools for timing test
double duration; //tools for timing test
struct EDGE
{
    int to, w, next;//"to" is the tail of the edge, while "w" is the weight of the edge
} edge[MAX_EDGE];
int head[MAX_NODE], cnt = 0; //"cnt" is the number of edges added to the graph
bool vis[MAX_NODE]; //flag if each node is collected
inline void addedge(int u, int v, int w) //Add the current edge to the graph
{
    edge[++cnt].to = v; //the number of edges+1 and set the end of  the edge to be v
    edge[cnt].next = head[u]; //the next the edge from u is linked to the current edge from u
    edge[cnt].w = w; //the weight of the current edge is set to be w
    head[u] = cnt; //the first edge from u is set to be current edge
}
int d[MAX_NODE];
```

```
int t; //the destination
int n; //the number of nodes
int m; //the number of edges
int s; //the source
class Node //node structrue of graph
{
public:
    int dis; //distance
    int vertex; //vertex num
    Node() = default; //make node
    ~Node() = default; //clear node
    Node(int a, int b) //make a new node with "a" as its dis and "b" as its vertex
    {
        dis = a;
        vertex = b;
    }
    friend bool operator<(Node a, Node b)//compare dis of a and b
    {
        return a.dis < b.dis; //if a < b, return true
    }
};
FibHeap<Node> H; //build a FibHeap H with Node
void dijkstra_with_Fib_Heap(int start, int des) //Dijkstra algorithm to calculate the distance of the shortest
path from the source to each node
{
    d[start] = 0;
    H.Insert(Node(0, s));
    while (!H.isEmpty()) //While the Fibonacci heap is not null
    {
        int now = H.DeleteMin().vertex; //Delete and return the minimum node from Fibonacci heap
        if (now == des) //if already reach the destination
        {
            cout << "shortest distance: " << d[des] << endl;//print the minimum path it calculates
            return;//end the dijksta function
        }
        if (vis[now])//If the node is collected
            continue; //Skip
        vis[now] = true; //Collect the minimum node
        for (int i = head[now]; i; i = edge[i].next) //Go through every edge from the current edge
        {
            EDGE &e = edge[i];
            if (d[now] + e.w < d[e.to])//If the shortest path through the current node to e.to is shorter than
the shortest path to e.to having stored
            {
```

```
                d[e.to] = d[now] + edge[i].w;//Update the shortest distance to e.to
                H.Insert(Node(d[e.to], e.to));//Push the edge to min-heap
            }
        }
    }
}
int main()
{
    cout << "Reading graph..." <<endl;
    fstream FILE_in;
    FILE_in.open(/*"Sample_Data.gr" "easy_test.txt"*/"USA-road-d.USA.gr", ios::in);//Open graph file to read in
    if (!FILE_in.is_open())//If fail to open the graph file
    {
        cout << "Could not open the data file!" << endl; //Warning
        exit(EXIT_FAILURE); //exit program
    }
    char flag;
    int u; //The head of an edge
    int v; //The tail of an edge
    int w; //The weight of an edge
    string temp;
    while (FILE_in >> flag)
    {
        if (flag == 'c') //If the line begins with 'c', it is a comment line
        {
            getline(FILE_in, temp); //It doesn't need to be read in to graph, so skip
        }
        else if (flag == 'p') //If the line begins with 'p'
        {
            FILE_in >> temp >> n >> m; //Read the identifier 'sp' into temp and read the number of nodes and edges
of the graph into n an m respectively
        }
        else if (flag == 'a') //If the line begins with 'a'
        {
            FILE_in >> u >> v >> w; //Read the head, tail and weight of the current edge
            addedge(u, v, w); //Add the curent edges to the graph
        }
    }
    FILE_in.close(); //Close the graph file
    cout << "Please input the start vertex and end vertex." << endl;
    cout << "For example: 1 19260817" << endl;
    cin >> s >> t;
    for (int i = 0; i <= n; i++)
    {
```

```
        d[i] = INF;
    }
    memset(vis, 0, sizeof(vis));
    start = clock(); //record the beginning time of dijkstra function
    dijkstra_with_Fib_Heap(s, t); //Calculate the distance of the shortest path from source to each node with
Dijsktra algorithm
    stop = clock(); //record the endding time of dijkstra function
    duration = ((double)(stop - start)) / CLK_TCK; //calculate the duration for running dijkstra function
    cout << "ticks: " << (double)(stop - start) << endl;
    cout << "duration: " << duration << "sec" << endl;//print the duration
    /*
    fstream FILE_out;
    FILE_out.open("Result_of_all_node_by_minHeap2.txt", ios::out);
    if (!FILE_out.is_open())
    {
        cout << "Could not open the result file!" << endl;
        exit(EXIT_FAILURE);
    }
    for (int i = 1; i <= n; i++)
    {
        FILE_out << d[i] << " ";
    }
    FILE_out.close();
    */
    return 0;
}
```

References

[1] Stergiopoulos , S. Algorithm for Fibonacci Heap Operations. Retrieved June 7, 2016, from http://www.cse.yorku.ca/~aaw/Sotirios/BinomialHeapAlgorithm.html

[2] Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2001). Introduction to Algorithms (2nd edition) (pp. chapter 20). The MIT Press.

Author List

Declaration

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

Signatures