Shortest Path Algorithm with Heaps

Author Names
(temporary vacancy during anonymous peer review)

Date: 2020-03-31

Chapter 1: Introduction

In the project, it's required to implement the Dijkstra's algorithm to calculate the shortest paths using two different heap structures and one of which must be Fibonacci heap. We choose the priority queue and the Fibonacci heap.

Chapter 2: Data Structure / Algorithm Specification

In the chapter, we will briefly discuss the Dijkstra's algorithm, priority queue and the Fibonacci heap.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, traffic networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published in 1959. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Pseudocode for the original algorithm:

```
function Dijkstra(Graph, source):
    create vertex set Q
        for each vertex v in Graph:
            dist[v] ← INFINITY
            prev[v] ← UNDEFINED
            add v to Q
```

```
        dist[source] ← 0
        while Q is not empty:
            u ← vertex in Q with min dist[u]

            remove u from Q
            for each neighbor v of u: // only v that are still in Q
                alt ← dist[u] + length(u, v)
                if alt < dist[v]:
                    dist[v] ← alt
                    prev[v] ← u
    return dist[], prev[]
```

2.2 Priority Queue

A min-priority queue is an abstract data type that provides 3 basic operations : `add_with_priority()`, `decrease_priority()` and `extract_min()`. Using such a data structure can lead to faster computing times than using a basic queue.

Pseudocode for the algorithm:

```
function Dijkstra(Graph, source):
    dist[source] ← 0                    // Initialization
    create vertex priority queue Q
    for each vertex v in Graph:
        if v ≠ source
            dist[v] ← INFINITY      // Unknown distance from source to v
            prev[v] ← UNDEFINED         // Predecessor of v
        Q.add_with_priority(v, dist[v])
    while Q is not empty:               // The main loop
        u ← Q.extract_min()             // Remove and return best vertex
        for each neighbor v of u:       // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)
    return dist, prev
```

2.3 Fibonacci Heap

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a lazy manner, postponing the work for later operations. For example, merging heaps is done simply by concatenating the two lists of trees, and operation decrease key sometimes cuts a node from its parent and forms a new tree.

Using such a data structure can lead to faster computing times than using a basic queue.

Pseudocode for the algorithm:

```
DijkstraUsingFibonacciHeap(PGraph G, AdjVertex start, AdjVertex end)
{
    PFibonacciHeap FibHeap = BuildAFibonacciHeap();
    Dis *dis = (Dis *)malloc(sizeof(Dis) * G->sumVertexes); //The tempo
rary dis from the start to the node
    for (each i < G->sumVertexes)
    {
        dis[i] = INF;
        visit[i] = 0;
    }
    PFibonacciNode FibNode = CreateAFibonacciNode(0, start);
    InsertToFibonacciHeap(FibHeap, FibNode);
    while (FibHeap->minTree)
    {
        PFibonacciNode tmpNode = PopMinFibonacciHeap(FibHeap);
```

```
            AdjVertex tmpV = tmpNode->index;
            if (visit[tmpV])
                continue;
            visit[tmpV] = true;
            for (PAdjEdge i = G->HeadList[tmpV]; i; i = i->nextEdge)
            {
                AdjVertex to = i->toVertex;
                if (!visit[to] && dis[tmpV] + i->dis < dis[to])
                {
                    dis[to] = dis[tmpV] + i->dis;
                    FibNode = CreateAFibonacciNode(dis[to], to);
                    InsertToFibonacciHeap(FibHeap, FibNode); //
                }
            }
        }
        Dis ret = dis[end];
        return ret;
    }
```

Chapter 3: Testing Results

3.1 Test Environment

Our test environment is just as following:

OS: Windows 10 Pro for Workstations, Build 18363.720

Compiler: gcc 8.1.0, x86_64-posix-seh-rev0

Compile command: g++ -Wall -std=c++14 source.cpp -o source -O2

RAM: 64GiB

3.2 Test Results and Analysis

We test six cases. The 1st, 2nd and 3rd cases are and the 4th, 5th and 6th cases are provided by teachers.

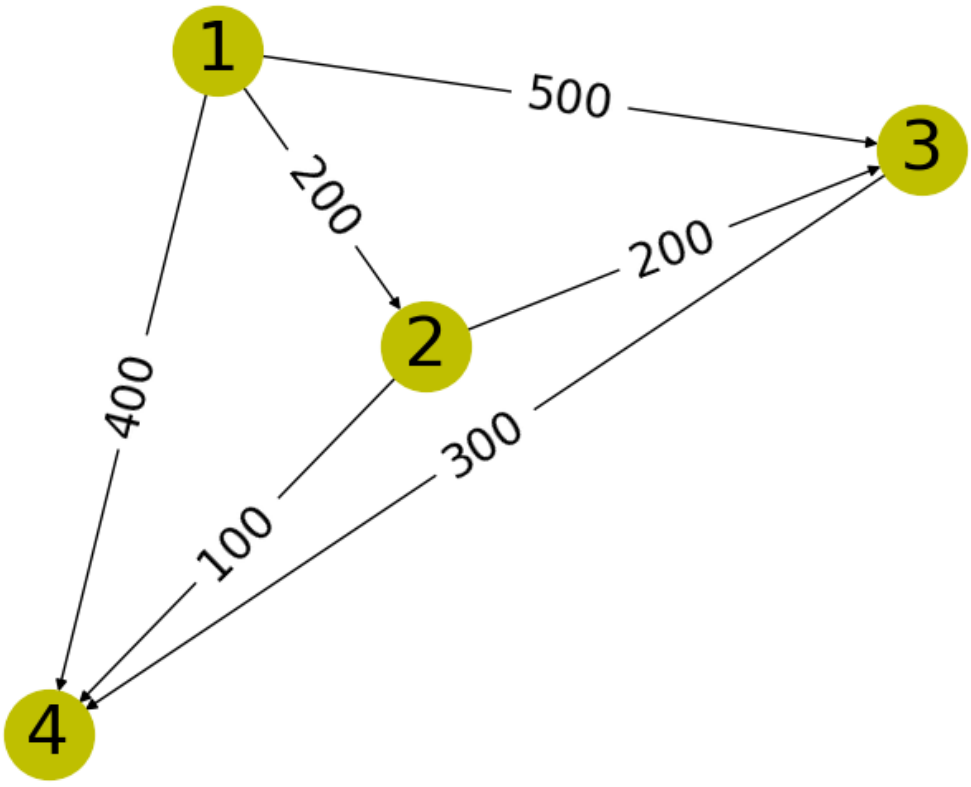
Format description of input data:

The first row of the input data consists of three integers V, E and Q, representing the number of vertices, the number of edges, and the number of queries, respectively.

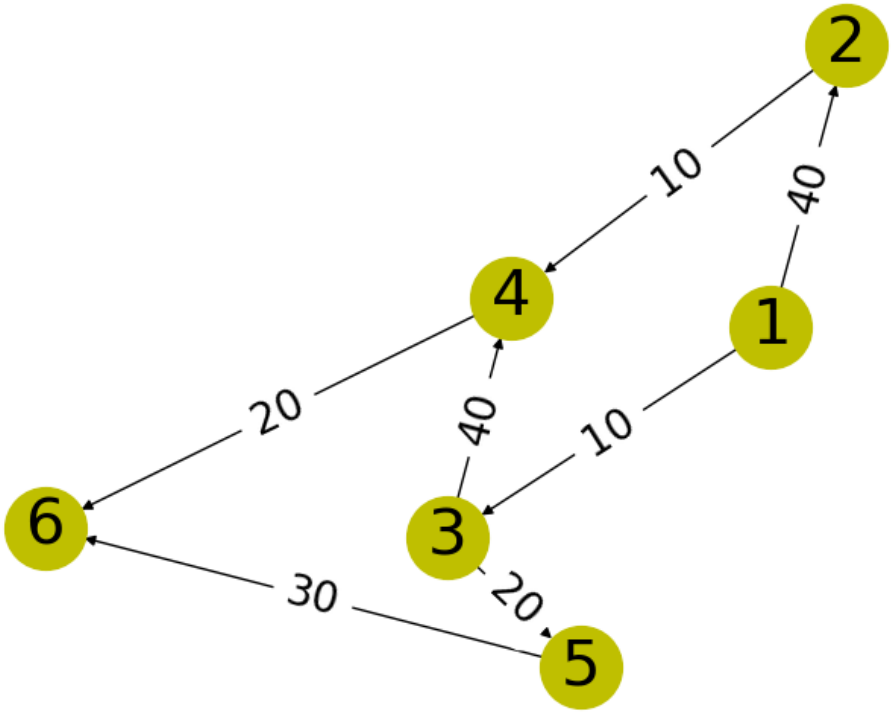
In the following E rows, three integers a_i, b_i, c_i, in each line, represents an arrow from a_i to b_i and the weights of this arrow is c_i.

The last Q rows of the input data contains two integers in each row, indicating the label of the starting point and the label of the terminal point of a group of queries.

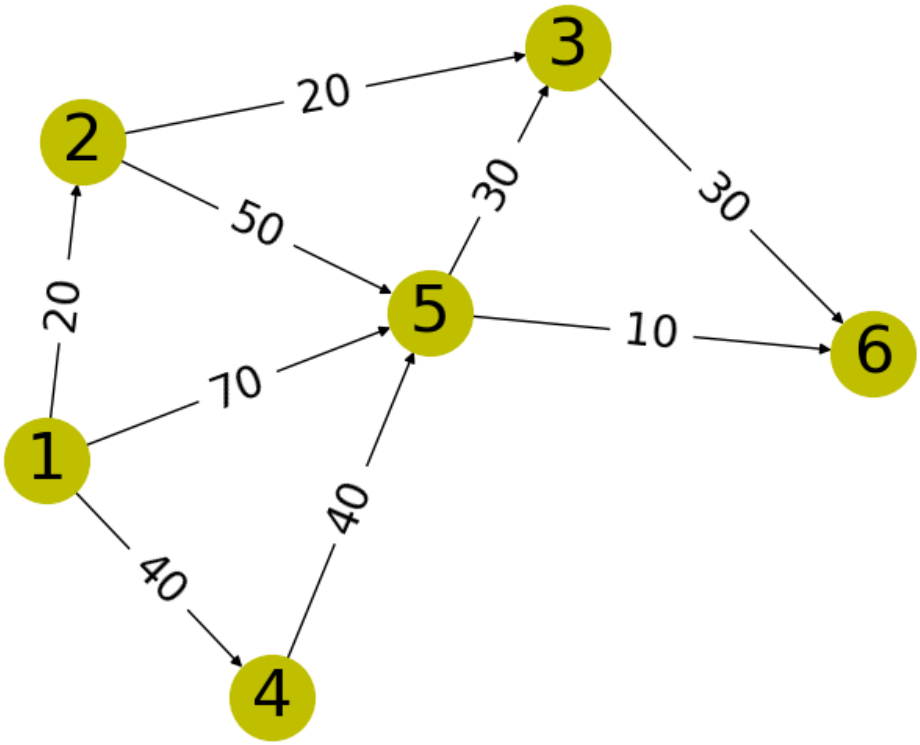
1st



2nd



3rd



The results all agree with our expectations.

Chapter 4: Analysis and Comments

4.1 Time Complexity Analysis

As we all know, for the original Dijkstra's algorithm on a graph with edges E and vertices V, the time complexity is $O(|E|T_{dk}+|V|T_{em})$ where T_{dk} and T_{em} are the complexities of the decrease-key and extract-minimum operations in data structure Q.

For priority queue, $T_{dk} = \log|V|$(time complexity for a binary search) and $T_{em} =|V|$. It's $O(|V|^2+|E|\log|V|)$.

For Fibonacci heap, it's $O(|E|+|V|\log|V)$.

4.2 Space Complexity Analysis

For priority queue implement case, if we implement Disjkstra's shortest path algorithm using a MinHeap and also if we use an array to store the values of the shortest distance for every node in the graph, the space complexity will be $O(V) + O(V) = O(2V) =\sim O(V)$.

For Fibonacci heap implement case, it's $O(V^2)$.

Appendix: Source Code

The source code folder consists of four files: FibonacciHeap.hpp, PriorityQueue.hpp, ShortestPath.hpp and the source.cpp.

The FibonacciHeap.hpp:

```
#ifndef FibonacciHeap_Hpp
 // head files preprocessing
#define FibonacciHeap_Hpp
 // head files preprocessing
typedef int Dis;
typedef int AdjVertex;
/*
The struct for the nodes in the Fibonacci Heap
*/
typedef struct FibonacciNodeStruct *PFibonacciNode;
 // struct for the Fibonacci node
typedef struct FibonacciNodeStruct
 // struct for the Fibonacci node
{
    Dis key;
 // the temporary distance from the start node to the node.
    PFibonacciNode left, right, child;
 // nodes
    AdjVertex index;
 // index
    int degree;
 // node's degree
} FibonacciNode;
/*
The struct for the Fibonacci Heap
*/
typedef struct FibonacciHeapStruct *PFibonacciHeap;
 // struct for the Fibonacci heap
typedef struct FibonacciHeapStruct
 // struct for the Fibonacci heap
{
    int sumNodes;
 // the sum of the nodes
    PFibonacciNode minTree;
 // the pointer to the tree whose top node is min
} FibonacciHeap;
//The declarations of the functions
PFibonacciHeap BuildAFibonacciHeap();
 // function to build an empty fibonacci heap
void InsertToFibonacciHeap(PFibonacciHeap FibHeap, PFibonacciNode FibNo
de);// function to insert the FibNode into the FibHeap
```

```
PFibonacciNode CreateAFibonacciNode(Dis key, AdjVertex index);
// function to create a fibonacci node with key and index
PFibonacciNode PopMinFibonacciHeap(PFibonacciHeap FibHeap);
// function to return the min Node in the FibHeap and delete it
void AddNodeToList(PFibonacciNode rootNode, PFibonacciNode FibNode);
// function to insert FibNode to the right of rootNode in the doubly li
nked list
void RemoveNodeFromList(PFibonacciNode FibNode);
// function to remove the FibNode from the doubly list it exists in
void UpdateFibonacciHeap(PFibonacciHeap FibHeap);
// functin to update the heap after deleting the min node
//The definitions of the functions
PFibonacciHeap BuildAFibonacciHeap()
// function to build an empty fibonacci heap
{
    PFibonacciHeap FibHeap = (PFibonacciHeap)malloc(sizeof(FibonacciHea
p));
    FibHeap->minTree = NULL;
 // initial value for minTree
    FibHeap->sumNodes = 0;
 // initial value for sumNdes
    return FibHeap;
 // terminate the function and return
}
PFibonacciNode CreateAFibonacciNode(Dis key, AdjVertex index)
 // function to create a fibonacci node with key and index
{
    PFibonacciNode FibNode = (PFibonacciNode)malloc(sizeof(FibonacciNod
e));
    FibNode->key = key;
 // initial value for the key
    FibNode->child = NULL;
 // initial value for the child
    FibNode->left = FibNode->right = FibNode;
 // initial values
    FibNode->degree = 0;
 // initial value for the degree
    FibNode->index = index;
 // initial value for the index
    return FibNode;
 // terminate the function and return
}
```

```
void AddNodeToList(PFibonacciNode rootNode, PFibonacciNode FibNode)
 // function to insert FibNode to the right of rootNode in the doubly l
inked list
{
    FibNode->right = rootNode->right;
 // set value
    rootNode->right->left = FibNode;
 // set value
    rootNode->right = FibNode;
 // set value
    rootNode->right = FibNode;
 // set value
    FibNode->left = rootNode;
 // set value
}
void RemoveNodeFromList(PFibonacciNode FibNode)
 // function to remove the FibNode from the doubly list it exists in
{
    if (FibNode->right == FibNode)
 // if the FibNode's right is equal to FibNode
        return;
 // terminate the function and return
    FibNode->right->left = FibNode->left;
 // set value
    FibNode->left->right = FibNode->right;
 // set value
    FibNode->left = FibNode->right = FibNode;
 // set value
}
void InsertToFibonacciHeap(PFibonacciHeap FibHeap, PFibonacciNode FibNo
de)// function to insert the FibNode into the FibHeap
{
    FibHeap->sumNodes++;
 // increment sumNodes by 1
    if (!FibHeap->minTree)
 // empty heap
        FibHeap->minTree = FibNode;
 // set value
    else
    {
        AddNodeToList(FibHeap->minTree, FibNode);
 // lazy insertion
        if (FibNode->key < FibHeap->minTree->key)
 // update the min node
```

```
            FibHeap->minTree = FibNode;
 // set value
    }
}
PFibonacciNode PopMinFibonacciHeap(PFibonacciHeap FibHeap)
 // function to return the min Node in the FibHeap and delete it
{
    PFibonacciNode minNode = FibHeap->minTree;
 // minimum node
    PFibonacciNode tmpNode = minNode->child;
 // tmp node
    PFibonacciNode t;
    while (tmpNode)
 // add the children of the minNode into the Root List
    {
        if (tmpNode->right == tmpNode)
 // if the tmpNode's right equals to the tmpNode
            t = NULL;
 // set t to NULL
        else
            t = tmpNode->right;
 // set value
        RemoveNodeFromList(tmpNode);
 // remove the child from its origin doubly list
        minNode->child = t;
        AddNodeToList(minNode, tmpNode);
 // add the child into the root list
        tmpNode = t;
    }
    if (minNode->right == minNode)
 // judge
        t = NULL;
 // set t to NULL
    else
        t = minNode->right;
 // set value
    RemoveNodeFromList(minNode);
 // remove the minNode
    FibHeap->minTree = t;
    UpdateFibonacciHeap(FibHeap);
 // update the FibHeap Structure
    FibHeap->sumNodes--;
 // decrement sumNodes by 1
    return minNode;
```

```
}
void UpdateFibonacciHeap(PFibonacciHeap FibHeap)
 // functin to update the heap after deleting the min node
{
    if (!FibHeap->minTree)
        return;
    int hashTableSize = ceil(log2(FibHeap->sumNodes)) + 1;
 // use a hashTable to obtain the trees with the same degree
    PFibonacciNode *hashTable = new PFibonacciNode[hashTableSize];
    for (int i = 0; i < hashTableSize; i++)
        hashTable[i] = NULL;
 // re-aggregate the trees on the Root List into the hashtable. We will
 link those trees with the same degree
    while (FibHeap->minTree)
    {
        PFibonacciNode minNode = FibHeap->minTree, t;
 // define new nodes
        int degree = minNode->degree;
        if (minNode->right == minNode)
 // judge
            t = NULL;
 // set t to NULL
        else
            t = minNode->right;
 // set value
        RemoveNodeFromList(minNode);
 // remove node minNode from list
        while (hashTable[degree])
 // aggregate trees until there doesn't exist any trees having the same
 degree
        {
            PFibonacciNode eNode = hashTable[degree];

 // we always set the tree in the hash table as the child of the tree f
rom the Root List

 // so we should make it sure that the eNode->key is larger than the mi
nNode->key
            if (eNode->key < minNode->key)
 // if the eNode's key is greater than the minimum node's key
            {
                PFibonacciNode t = eNode;
 // new a node eode
```

```
                eNode = minNode;
// set the new node to the minimum node
                minNode = t;
            }
            if (!minNode->child)
// if the minimum node's child is NULL
                minNode->child = eNode;
// set the minimum nodes' child to eNode
            else
                AddNodeToList(minNode->child, eNode);
// add the node to the list
            minNode->degree++;
// increment degree by 1
            hashTable[degree++] = NULL;
// the degree has been enlarged after aggregating two trees
        }
        hashTable[degree] = minNode;
// set to minimum node
        FibHeap->minTree = t;
// set to t
    }
    for (int i = 0; i < hashTableSize; i++)
// move the trees in the hash table to the root list
    {
        if (hashTable[i])
// judge
        {
            if (FibHeap->minTree)
// if the FibHeap's minTree is not false or null.
            {
                AddNodeToList(FibHeap->minTree, hashTable[i]);
// add the node to list
                if (hashTable[i]->key < FibHeap->minTree->key)
// judge
                    FibHeap->minTree = hashTable[i];
// set value
            }
            else
                FibHeap->minTree = hashTable[i];
// set value
        }
    }
  delete [] hashTable;
// delete the hash table
```

```
}
#endif
 // head files preprocessing
```

The PriorityQueue.hpp:

```
#ifndef PriorityQueue_Hpp
 // head files preprocessing
#define PriorityQueue_Hpp
 // head files preprocessing
typedef int Dis;
typedef int AdjVertex;
typedef struct HeapNodeStruct
 // the heap node struct
{
    Dis key;
 // the dis from the start node to this node
    AdjVertex index;
 // the index of vertexes
} HeapNode, *PHeapNode;
typedef struct HeapStruct
 // the heap struct
{
    int size;
 // current size of the heap
    int space;
 // current space of the heap
    PHeapNode nodeList;
} Heap;
typedef struct HeapStruct *PriorityQueue;
 // define the priority queue data sruct
//The declarations of the functions
PriorityQueue BuildAHeap(int sumVertexes);
 // build an empty heap with sumVertexes nodes
void InsertToPriorityQueue(HeapNode x, PriorityQueue H);
 // insert an node x into the heap H. You should create the node yourse
lf
HeapNode PopMinPriorityQueue(PriorityQueue H);
 // return the min node in the heap H and delete it.
bool IsEmptyPriorityQueue(PriorityQueue H);
 // check the heap whether empty or not
//The definitions of the functions
PriorityQueue BuildAHeap(int sumVertexes)
 // function to build a heap
{
```

```
    PriorityQueue H = (PriorityQueue)malloc(sizeof(HeapStruct));
 // malloc the memory
    H->size = 0;
 // initial value
    H->nodeList=(PHeapNode)malloc(sizeof(HeapNode) * (sumVertexes + 1))
;// initial value
    H->nodeList[0].key = -1;
 // initial value
    return H;
 // return result
}
void InsertToPriorityQueue(HeapNode x, PriorityQueue H)
 // function to insert into priority queue
{
    int i;
    for (i = ++H->size; H->nodeList[i / 2].key > x.key; i /= 2)
 // add the node to the tail of the list and percolate up it
        H->nodeList[i] = H->nodeList[i / 2];
    H->nodeList[i] = x;
 // set value
}
HeapNode PopMinPriorityQueue(PriorityQueue H)
 // fuction to pop minimum priority queue
{
    int i, child;
    HeapNode minNode = H->nodeList[1],
 // define the minimum node
             LastNode = H->nodeList[H->size--];
 // delete the node at the tail of the list, and replace it with the to
p node
    for (i = 1; i * 2 <= H->size; i = child)
 // percolate down the LastNode
    {
        child = i * 2;
        if (child != H->size && H->nodeList[child + 1].key < H->nodeLis
t[child].key)// judge
            child++;
        if (LastNode.key > H->nodeList[child].key)
 // if the last node's key is greater than the key of the current child
's node
            H->nodeList[i] = H->nodeList[child];
        else
            break;
 // find the minumum node
```

```
    }
    H->nodeList[i] = LastNode;
 // set the value
    return minNode;
 // retur the result
}
bool IsEmptyPriorityQueue(PriorityQueue H)
 // function to judge if the priority queue is empty
{
    return H->size == 0 ? true : false;
 // if the priority queue is empty, return true. return false otherwise
.
}
#endif
```

The ShortestPath.hpp

```
#ifndef ShortestPath_Hpp
 // head files preprocessing
#define ShortestPath_Hpp
 // head files preprocessing
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include "FibonacciHeap.hpp"
#include "PriorityQueue.hpp"
#define INF 2147483647
 // global constant
typedef int Dis;
 // global variable int
/*
Adjacency list is used to describe the graph
*/
typedef int AdjVertex;
 // the index of vertexes. Notice that the index should begin as 0
/*
The struct storing the edges in the adjacency list.
*/
typedef struct AdjEdgeStruct *PAdjEdge;
 // padjedge
typedef struct AdjEdgeStruct
 // adjedge
{
```

```
    Dis dis;
 // the length of the edge
    AdjVertex fromVertex, toVertex;
 // the vertexes the edge connects
    PAdjEdge nextEdge;
 // the following edges whose FROM VERTEX is the same
} AdjEdge;
 // the struct and the pointer to an edge
typedef PAdjEdge *AdjList;
 // the adjacency list
typedef struct GraphStruct
 // the graph
{
    int sumVertexes;
 // the sum of all the vertexes in the graph;
    int sumEdges;
 // the sum of all the edges in the graph;
    AdjList HeadList;
 // the head list
} * PGraph;
 // end
//The declarations of the functions
PGraph BuildAGraph(int sumVertexes, int sumEdges);
 // build an empty adjacency list with sumVertexes vertexes
void AddEdge(PGraph G, AdjVertex fromV, AdjVertex toV, Dis dis);
 // add an edge from fromV to toV
Dis DijkstraUsingFibonacciHeap(PGraph G, AdjVertex start, AdjVertex end
);// Dijkstra algorithm using fibonacci heap
Dis DijkstraUsingPriorityQueue(PGraph G, AdjVertex start, AdjVertex end
);// Dijkstra algorithm using priority queue
//The definitions of the functions
PGraph BuildAGraph(int sumVertexes, int sumEdges)
 // build an empty adjacency list with sumVertexes vertexes
{
    PGraph Graph = (PGraph)malloc(sizeof(struct GraphStruct));
 // new graph
    Graph->sumVertexes = sumVertexes;
 // set value
    Graph->sumEdges = sumEdges;
 // set value
    Graph->HeadList = (AdjList)malloc(sizeof(PAdjEdge) * sumVertexes);
 // set value
    for (int i = 0; i < sumVertexes; i++)
 // recurse
```

```
    {
        Graph->HeadList[i] = NULL;
 // set value
    }
    return Graph;
 // terminate the function and return
}

void AddEdge(PGraph G, AdjVertex fromV, AdjVertex toV, Dis dis)
 // build an empty adjacency list with sumVertexes vertexes
{
    PAdjEdge edge = (PAdjEdge)malloc(sizeof(AdjEdge));
    edge->fromVertex = fromV;
 // set value
    edge->toVertex = toV;
 // set value
    edge->dis = dis;
 // set value
    edge->nextEdge = G->HeadList[fromV];
 // set value
    G->HeadList[fromV] = edge;
 // set value
}
Dis DijkstraUsingFibonacciHeap(PGraph G, AdjVertex start, AdjVertex end
)// Dijkstra algorithm using fibonacci heap
{
    PFibonacciHeap FibHeap = BuildAFibonacciHeap();
    Dis *dis = (Dis *)malloc(sizeof(Dis) * G->sumVertexes);
 // the temporary dis from the start to the node
    bool *visit = (bool *)malloc(sizeof(bool) * G->sumVertexes);
 // malloc new variable
    for (int i = 0; i < G->sumVertexes; i++)
 // recurse
    {
        dis[i] = INF;
 // set value
        visit[i] = 0;
 // set value
    }
    dis[start] = 0;
 // set value
    PFibonacciNode FibNode = CreateAFibonacciNode(0, start);
    InsertToFibonacciHeap(FibHeap, FibNode);
 // insert the startNode into the fibonacci heap
```

```
    while (FibHeap->minTree)
    {
        PFibonacciNode tmpNode = PopMinFibonacciHeap(FibHeap);
// malloc new variable
        AdjVertex tmpV = tmpNode->index;
// get the node which has the min dis from startNode
        if (visit[tmpV])
// if visit[tmpV] is false
            continue;
// skip it if the node has been visited
        visit[tmpV] = true;
// mark the node
        for (PAdjEdge i = G->HeadList[tmpV]; i; i = i->nextEdge)
// use the tmpV to relax all the advancency nodes
        {
            AdjVertex to = i->toVertex;
// set value
            if (!visit[to] && dis[tmpV] + i->dis < dis[to])
// judge
            {
                dis[to] = dis[tmpV] + i->dis;
// set value
                FibNode = CreateAFibonacciNode(dis[to], to);
// create new node
                InsertToFibonacciHeap(FibHeap, FibNode);
// insert the node that is been relaxed into the heap
            }
        }
    }
    Dis ret = dis[end];
// set value
    free(FibNode);
// free space
    free(dis);
// free space
    free(visit);
// free space
    free(FibHeap);
// free space
    return ret;
// terminate the function and return
}
Dis DijkstraUsingPriorityQueue(PGraph G, AdjVertex start, AdjVertex end
)// Dijkstra algorithm using priority queue
```

```
{
    Dis *dis = (Dis *)malloc(sizeof(Dis) * G->sumVertexes);
// malloc new variable
    bool *visit = (bool *)malloc(sizeof(bool) * G->sumVertexes);
// malloc new variable
    for (int i = 0; i < G->sumVertexes; i++)
// recurse
    {
        dis[i] = INF;
// set value
        visit[i] = 0;
// set value
    }
    dis[start] = 0;
// set value
    PriorityQueue heap = BuildAHeap(G->sumVertexes + G->sumEdges);
    HeapNode newNode;
    newNode.index = start;
// set value
    newNode.key = 0;
// set value
    InsertToPriorityQueue(newNode, heap);
// insert the startNode into the fibonacci heap
    while (!IsEmptyPriorityQueue(heap))
    {
        HeapNode tmpNode = PopMinPriorityQueue(heap);
        AdjVertex tmpV = tmpNode.index;
        if (visit[tmpV])
// judge if the visit[tmpV] is empty
            continue;
// skip it if the node has been visited
        visit[tmpV] = true;
// mark the node
        for (PAdjEdge i = G->HeadList[tmpV]; i; i = i->nextEdge)
// use the tmpV to relax all the advancency nodes
        {
            AdjVertex to = i->toVertex;
// define new variable
            if (!visit[to] && dis[tmpV] + i->dis < dis[to])
// judge
            {
                dis[to] = dis[tmpV] + i->dis;
// set value
```

```
                newNode.index = to;
 // set value
                newNode.key = dis[to];
 // set value
                InsertToPriorityQueue(newNode, heap);
 // insert the node that is been relaxed into the heap
            }
        }
    }
    Dis ret = dis[end];
 // set value
    free(dis);
 // free space
    free(visit);
 // free space
    free(heap);
 // free space
    return ret;
 // terminate the function and return
}
#endif
 // head files preprocessing
```

The source.cpp

```
#include <cstdio>
#include <cstring>
#include <cmath>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "ShortestPath.hpp"
using namespace std;
typedef struct QueryPairStruct
 // the query pair struct
{
    int start, end;
 // struct members
} QueryPair;
int main()
 // the main function
{
    char filename[30];
    FILE* PFileIn =  fopen("data.in", "r");
 // open the input data file
```

```
    clock_t begin, finish;
// timer variables
    int n, m, q;
    fscanf(PFileIn,"%d %d %d",&n,&m,&q);
// read the input data
    PGraph G = BuildAGraph(n, m);
// build graph
    QueryPair *QueryPairs = (QueryPair *)malloc(sizeof(QueryPair) * q);
    for (int i = 0; i < m; i++)
// recurse
    {
        int f, t, d;
        fscanf(PFileIn,"%d %d %d",&f,&t,&d);
// read the input data
        AddEdge(G, f - 1, t - 1, d);
// add edge to the graph
    }
    for (int i = 0; i < q; i++)
// recurse
    {
        int s, e;
        fscanf(PFileIn,"%d %d",&s,&e);
// read the input data
        QueryPairs[i].start = s - 1;
// record the query pairs
        QueryPairs[i].end = e - 1;
// record the query pairs
    }
    fclose(PFileIn);
// close the fle

    sprintf(filename, "PriorityQueue.out");
// help word
    FILE *PFileOut=fopen(filename, "w");
// define the output file string
    fprintf(PFileOut,"The result using Priority Queue is: \n");
// help word
    begin = clock();
// begin the timer
    for (int i = 0; i < q; i++)
    {
        Dis ans = DijkstraUsingPriorityQueue(G, QueryPairs[i].start, Qu
eryPairs[i].end);
        fprintf(PFileOut,"%d\n", ans);
```

```
    }
    finish = clock();
 // terminate the timer
    fprintf(PFileOut,"The time using Priority Queue is: %d", (finish -
begin) / CLOCKS_PER_SEC);
    fclose(PFileOut);
 // close the written file string
    cout << "The function using Priority Queue completed!" << endl;
 // help word

    sprintf(filename, "FibonacciHeap.out");

    PFileOut=fopen(filename, "w");
 // open the written file string
    fprintf(PFileOut,"The result using Fibonacci Heap is: \n");
    begin = clock();
 // begin the timer
    for (int i = 0; i < q; i++)
    {
        Dis ans = DijkstraUsingFibonacciHeap(G, QueryPairs[i].start, Qu
eryPairs[i].end);
        fprintf(PFileOut,"%d\n", ans);
 // print the output
    }
    finish = clock();
 // terminate the timer
    fprintf(PFileOut,"The time using Fibonacci Heap is: %d", (finish -
begin) / CLOCKS_PER_SEC);
    fclose(PFileOut);
    printf("The function using Priority Queue completed!");
 // print the output
    return 0;
}
```

References

[1] Author, “Title of the article”, *Name of the journal*, page no., (year)

[2] Author, “Title of the book”, *Name of the publisher*, year

[3] Author, “Title of the article”, Web site links

Author List

Temporary vacancy during anonymous peer review.

Declaration

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

Signatures

Temporary vacancy during anonymous peer review.