Project Report

Chapter 1: Problem Description

In the previous lessons, we have learned Dijkstra's algorithm. In this project, we will use this algorithm to calculate the single source shortest paths. More importantly, we need to use different types of priority queue (i.e. heaps) for Dijkstra's algorithm, and evaluate the impact of using various priority queue heaps on the efficiency of Dijkstra's algorithm through tests.

Chapter 2: Solutions

2.1 Dijkstra's Algorithm

First, let's briefly review the main content of Dijkstra's algorithm:
Starting from a given starting point, perform the following operations on the current point once a time:
Record the distance from the starting point to the current point included in the current node, which is the shortest distance from the starting point to the current point.
Generate a node for all points connected by this point, and record the distance from the starting point to these adjacent points in the node (the distance from the starting point to the current point plus the distance from the current point to the adjacent point). After storing these nodes, combine them with all the previously generated nodes, and find the node that belongs to the unreachable point and has the shortest distance as the next node.

2.2 Binomial Heap

In order to quickly find the smallest distance among all nodes, priority queue is used to complete this operation. In this project, we intend to use two implementations of priority queue, and one of these two is binomial heap.
Binomial heap is actually a group of k-power trees with 2 nodes. There are no two trees with the same number of nodes.
A tree with a number of nodes of the n-power of 2 is generated by merging two trees with a number of nodes of the (n-1)-power of 2. The generation process is simple: take the small top heap as an example, compare the root node sizes of the two, and convert the tree with a larger root node into a subtree of the smaller root node.

When adding a new node, it is equivalent to adding a tree with a size of 2 to the power of 0. If such a tree already exists in the binomial heap, merge it. Then check whether there is a tree of size 2. If there is one, merge it again, and so on.
Because the smallest node must be the root node of a tree, when searching for the smallest node, we only need to compare all the root nodes.
Due to the construction characteristics of the tree in the binomial heap, when deleting the root node of a tree whose size is 2 to the nth power, a tree whose size is 1, 2, 4 to the k-1 power of 2 will be generated.
After deleting the root node, these trees should be added to the binomial heap. This process is similar to binary addition.

2.3 Fibonacci Heap

Fibonacci heap is similar to binomial heap in thinking, but the former has more relaxed requirements for trees.
The number of sub nodes of a node is called degree. After adding a tree, check to see if there are trees of the same degree in the heap, and merge if there are any. After deleting a node, add all its subtrees to the Fibonacci heap. It has better amortized analysis performance than binomial reactor.

Chapter 3: Testing

We use 100 points and 4950 edges for test. The function clock() was used to record the running time.
The running time recorded are shown as follows:

Dijkstra's algorithm with binomial heap:

```
Total time: 120.000000ms
```

Dijkstra's algorithm with binomial heap:

```
Total time: 6.000000ms
```

It can be seen that when using the Fibonacci heap, the running implementation is significantly less than when using the binomial heap. From the perspective of amortized analysis, the time complexity of Fibonacci heap is also lower than that of binomial heap.

Appendix: Source Code

ShortestPathWithBinomialHeap.cpp

```
#include <cstdio>
#include <time.h>
#define NODE_MAX 40

using namespace std;
```

```
struct edgeNode;
typedef struct edgeNode edge;
```

```
struct pointNode {  //用于记录点的信息：点的编号、与其连接的所有边
    int index;
    edge *edgeList;
    int edgeCnt;
};
typedef struct pointNode *point;
```

```
struct edgeNode {   //用于记录边的信息：连接的相邻点、长度
    point destination;
    int length;
};
```

```
inline void addEdge(point A, point B, int l) {  //在点的信息中添加相应边的记录
    A->edgeList[A->edgeCnt].destination = B;
    A->edgeList[A->edgeCnt].length = l;
    A->edgeCnt ++;
    B->edgeList[B->edgeCnt].destination = A;
    B->edgeList[B->edgeCnt].length = l;
    B->edgeCnt ++;
}
```

```
struct heapNode {   //堆中的结点的结构定义
    point currentPoint;     //所在点
    int distanceTraveled;   //从起点到达此结点的距离
    int depth;              //记录子树的深度
    heapNode *firstChild;
    heapNode *nextSibling;  //使用了firstChild，nextSibling 的方式记录子结点
    bool operator<(heapNode &another) const {
        return this->distanceTraveled < another.distanceTraveled;
    }
};
typedef struct heapNode *heapPtr;
```

```
void addChild(heapPtr parent, heapPtr child) {  //将某一结点并入另一结点的子结点
    child->nextSibling = parent->firstChild;
    parent->firstChild = child;
    parent->depth ++;
}
```

```
heapPtr heapJoin(heapPtr heapA, heapPtr heapB) {    //合并堆
```

```
    if(heapA < heapB) {
        addChild(heapA, heapB);
        return heapA;
    } else {
        addChild(heapB, heapA);
        return heapB;
    }
}
```

```
void forestJoin(heapPtr forestA[], heapPtr forestB[]) { //二项堆合并，其过程类似二进制加法
    heapPtr c = NULL;
    for(int i = 0; i < NODE_MAX; i ++) {
        if(forestA[i] == NULL) {
            if(forestB[i] == NULL) {
                forestA[i] = c;
                c = NULL;
            } else {
                if(c == NULL) {
                    forestA[i] = forestB[i];
                } else {
                    c = heapJoin(forestB[i], c);
                }
            }
        } else {
            if(forestB[i] == NULL) {
                if(c != NULL) {
                    c = heapJoin(forestA[i], c);
                    forestA[i] = NULL;
                }
            } else {
                if(c == NULL) {
                    c = heapJoin(forestA[i], forestB[i]);
                    forestA[i] = NULL;
                } else {
                    c = heapJoin(forestB[i], c);
                }
            }
        }
    }
}
```

```
class BinomialHeap {    //二项堆定义
    private:
```

```
        heapPtr *forest;    //forest 数组用于记录所有树

    public:
        BinomialHeap() {
            forest = new(heapPtr[NODE_MAX]);
            for(int i = 0; i < NODE_MAX; i ++) {
                forest[i] = NULL;
            }
        }
        heapPtr top() { //查看最小值，通过比较根结点得到
            int ret = -1;
            for(int i = 0; i < NODE_MAX; i ++) {
                if(forest[i] != NULL && (ret == -1 || forest[ret] > forest[i])) {
                    ret = i;
                }
            }
            return forest[ret];
        }
        void pop() {    //取出最小值结点
            int split = -1;
            for(int i = 0; i < NODE_MAX; i ++) {
                if(forest[i] != NULL && (split == -1 || forest[split] > forest[i]))
{
                    split = i;
                }
            }
            heapPtr splitHeap = forest[split];
            forest[split] = NULL;   //删除该树
            heapPtr *splitForest = new(heapPtr[NODE_MAX]);
            for(int i = 0; i < NODE_MAX; i ++) {    //记录所有子树成为新的二项堆
                splitForest[i] = NULL;
            }
            for(heapPtr i = splitHeap->firstChild; i != NULL; i = i->nextSibling)
{
                splitForest[i->depth] = i;
            }
            forestJoin(forest, splitForest);    //二项堆合并
        }
        void push(heapPtr insertHeap) { //插入新结点，相当于与只有一个结点的二项堆合
并
            heapPtr *insertForest = new(heapPtr[NODE_MAX]);
            for(int i = 0; i < NODE_MAX; i ++) {
                insertForest[i] = NULL;
            }
```

```
            insertForest[0] = insertHeap;
            forestJoin(forest, insertForest);
        }
        bool empty() {
            for(int i = 0; i < NODE_MAX; i ++) {
                if(forest[i] != NULL) {
                    return false;
                }
            }
            return true;
        }
};
BinomialHeap priorityQueue = BinomialHeap();
```

```
bool confirmed[50000] = {false};
int dis[50000] = {0};
void dijkstra();
```

```
int main() {
    int n, m;
    scanf("%d %d", &n, &m); //0 到n-1 共n 个点，以 0 为起点
    point pointList[50000];
    for(int i = 0; i < n; i ++) {   //读入点
        pointList[i] = new(pointNode);
        pointList[i]->index = i;
        pointList[i]->edgeList = new(edge[50000]);
        pointList[i]->edgeCnt = 0;
    }
```

```
    for(int i = 0; i < m; i ++) {   //读入边
        int pointA, pointB, length;
        scanf("%d %d %d", &pointA, &pointB, &length);
        addEdge(pointList[pointA], pointList[pointB], length);
    }
```

```
    heapPtr initialHeap = new(heapNode);    //初始结点
    initialHeap->currentPoint = pointList[0];
    initialHeap->distanceTraveled = 0;
    initialHeap->depth = 0;
    initialHeap->firstChild = initialHeap->nextSibling = NULL;
    priorityQueue.push(initialHeap);
```

```
    double begin = clock();
    dijkstra();
```

```
    double end = clock();

    for(int i = 0; i < n; i ++) {
        printf("%d: %d\n", i, dis[i]);
    }
    printf("Total time: %lfms", end - begin);
    return 0;
}

void dijkstra() {   //Dijkstra 算法
    while(!priorityQueue.empty()) {
        heapPtr topHeap = priorityQueue.top();
        priorityQueue.pop();
        while(confirmed[topHeap->currentPoint->index] && !priorityQueue.empty())
{
            topHeap = priorityQueue.top();
            priorityQueue.pop();
        }
        if(confirmed[topHeap->currentPoint->index] && priorityQueue.empty()) {
            break;
        }

        point currentPoint = topHeap->currentPoint;
        int distanceTraveled = topHeap->distanceTraveled;
        confirmed[currentPoint->index] = true;
        dis[currentPoint->index] = distanceTraveled;

        for(int i = 0; i < currentPoint->edgeCnt; i ++) {
            edge nextEdge = currentPoint->edgeList[i];

            if(!confirmed[nextEdge.destination->index]) {
                heapPtr newHeap = new(heapNode);
                newHeap->currentPoint = nextEdge.destination;
                newHeap->distanceTraveled = distanceTraveled + nextEdge.length;
                newHeap->depth = 0;
                newHeap->firstChild = newHeap->nextSibling = NULL;
                priorityQueue.push(newHeap);
            }
        }
    }
}
```

ShortestPathWithFibonaccilHeap.cpp

```cpp
#include <cstdio>
#include <time.h>
#define NODE_MAX 10000

using namespace std;

struct edgeNode;
typedef struct edgeNode edge;

struct pointNode {  //用于记录点的信息：点的编号、与其连接的所有边
    int index;
    edge *edgeList;
    int edgeCnt;
};
typedef struct pointNode *point;

struct edgeNode {   //用于记录边的信息：连接的相邻点、长度
    point destination;
    int length;
};

inline void addEdge(point A, point B, int l) {  //在点的信息中添加相应边的记录
    A->edgeList[A->edgeCnt].destination = B;
    A->edgeList[A->edgeCnt].length = l;
    A->edgeCnt ++;
    B->edgeList[B->edgeCnt].destination = A;
    B->edgeList[B->edgeCnt].length = l;
    B->edgeCnt ++;
}

struct heapNode {   //堆中的结点的结构定义
    point currentPoint;     //所在点
    int distanceTraveled;   //从起点到达此结点的距离
    int degree;             //记录了子结点的数量
    heapNode *firstChild;
    heapNode *preSibling;   //记录了前一个子结点，形成双向链表
    heapNode *nextSibling;  //使用了firstChild，nextSibling 的方式记录子结点
    bool operator<(heapNode* &another) const {
        return this->distanceTraveled < another->distanceTraveled;
    }
};
typedef struct heapNode *heapPtr;
```

```
void addChild(heapPtr parent, heapPtr child) {  //将某一结点并入另一结点的子结点
    child->nextSibling = parent->firstChild;
    parent->firstChild = child;
    parent->degree ++;
}
```

```
heapPtr heapJoin(heapPtr heapA, heapPtr heapB) {    //合并堆
    if(heapA < heapB) {
        addChild(heapA, heapB);
        return heapA;
    } else {
        addChild(heapB, heapA);
        return heapB;
    }
}
```

```
class FibonacciHeap {    //斐波那契堆定义
    private:
        heapPtr fibonacciRoot;  //使用一个虚拟的根结点作为所有树的父结点

    public:
        FibonacciHeap() {
            fibonacciRoot = new(heapNode);
            fibonacciRoot->firstChild = NULL;
        }
        void nodeDelete(heapPtr hp) { //从虚拟的根节点的子树中取出一棵树
            if(hp->preSibling != NULL) {
                hp->preSibling->nextSibling = hp->nextSibling;
            } else {
                fibonacciRoot->firstChild = hp->nextSibling;
            }
            if(hp->nextSibling != NULL) {
                hp->nextSibling->preSibling = hp->preSibling;
            }
        }
        void nodeInsert(heapPtr hp) {   //加入一棵树
            hp->preSibling = NULL;
            hp->nextSibling = fibonacciRoot->firstChild;
            if(hp->nextSibling != NULL) {
                hp->nextSibling->preSibling = hp;
            }
            fibonacciRoot->firstChild = hp;
        }
        heapPtr top() { //查看最小值，通过比较根结点得到
```

```
            heapPtr ret = NULL;
            for(heapPtr i = fibonacciRoot->firstChild; i != NULL; i = i->nextSibling)
{
                if(ret == NULL || i->distanceTraveled < ret->distanceTraveled) {
                    ret = i;
                }
            }
            return ret;
        }
        void pop() {    //取出最小值结点
            heapPtr split = NULL;
            for(heapPtr i = fibonacciRoot->firstChild; i != NULL; i = i->nextSibling)
{
                if(split == NULL || i->distanceTraveled < split->distanceTraveled)
{
                    split = i;
                }
            }
            nodeDelete(split);  //删除该树
            heapPtr next = NULL;
            for(heapPtr i = split->firstChild; i != NULL; i = next) {
                next = i->nextSibling;
                nodeInsert(i);  //并入所有子树
            }
            scan();
        }
        void push(heapPtr insertHeap) {
            nodeInsert(insertHeap);
            scan();
        }
        void scan() {   //每次树的组成发生变化后，查看是否有度相同的树，有则将其合并
            heapPtr heapList[NODE_MAX];
            for(int i = 0; i < NODE_MAX; i ++) {
                heapList[i] = NULL;
            }
            while(fibonacciRoot->firstChild != NULL) {
                heapPtr temp = fibonacciRoot->firstChild;
                nodeDelete(temp);
                while(heapList[temp->degree] != NULL) {
                    int d = temp->degree;
                    temp = heapJoin(temp, heapList[temp->degree]);
                    heapList[d] = NULL;
                }
                heapList[temp->degree] = temp;
```

```
            }
            for(int i = 0; i < NODE_MAX; i ++) {
                if(heapList[i] != NULL) {
                    nodeInsert(heapList[i]);
                }
            }
        }
        bool empty() {
            return fibonacciRoot->firstChild == NULL;
        }
};
FibonacciHeap priorityQueue = FibonacciHeap();
```

```
bool confirmed[50000] = {false};
int dis[50000] = {0};
void dijkstra();
```

```
int main() {
    int n, m;
    scanf("%d %d", &n, &m); //0 到n-1 共n 个点，以 0 为起点
    point pointList[50000];
    for(int i = 0; i < n; i ++) {   //读入点
        pointList[i] = new(pointNode);
        pointList[i]->index = i;
        pointList[i]->edgeList = new(edge[50000]);
        pointList[i]->edgeCnt = 0;
    }
```

```
    for(int i = 0; i < m; i ++) {   //读入边
        int pointA, pointB, length;
        scanf("%d %d %d", &pointA, &pointB, &length);
        addEdge(pointList[pointA], pointList[pointB], length);
    }
```

```
    heapPtr initialHeap = new(heapNode);    //初始结点
    initialHeap->currentPoint = pointList[0];
    initialHeap->distanceTraveled = 0;
    initialHeap->degree = 0;
    initialHeap->firstChild = initialHeap->preSibling = initialHeap->nextSibling
= NULL;
    priorityQueue.push(initialHeap);
```

```
    double begin = clock();
    dijkstra();
```

```
    double end = clock();

    for(int i = 0; i < n; i ++) {
        printf("%d: %d\n", i, dis[i]);
    }
    printf("Total time: %lfms", end - begin);
    return 0;
}

void dijkstra() {   //Dijkstra 算法
    while(!priorityQueue.empty()) {
        heapPtr topHeap = priorityQueue.top();
        priorityQueue.pop();
        while(confirmed[topHeap->currentPoint->index] && !priorityQueue.empty())
{
            topHeap = priorityQueue.top();
            priorityQueue.pop();
        }
        if(confirmed[topHeap->currentPoint->index] && priorityQueue.empty()) {
            break;
        }

        point currentPoint = topHeap->currentPoint;
        int distanceTraveled = topHeap->distanceTraveled;
        confirmed[currentPoint->index] = true;
        dis[currentPoint->index] = distanceTraveled;

        for(int i = 0; i < currentPoint->edgeCnt; i ++) {
            edge nextEdge = currentPoint->edgeList[i];

            if(!confirmed[nextEdge.destination->index]) {
                heapPtr newHeap = new(heapNode);
                newHeap->currentPoint = nextEdge.destination;
                newHeap->distanceTraveled = distanceTraveled + nextEdge.length;
                newHeap->degree = 0;
                newHeap->firstChild = NULL;
                newHeap->preSibling = NULL;
                newHeap->nextSibling = NULL;
                priorityQueue.push(newHeap);
            }
        }
    }
}
```