Chapter1: Introduction

1.1 Problem description

Shortest path problem is one of the most popular combinatorial optimization problems, relative algorithms are still under active research. And Dijkstra's algorithm is one of the widely used algorithms to solve this kind of problems.

In this project, we are asked to use different data structures(but limited to different kinds of min-priority queue) to implement Dijkstra's algorithm and test their performance so as to find the best data structure for the Dijkstra's algorithm.

We are supposed to use USA road networks for evaluation, and at least 1000 pairs of query are required in evaluating the run times of the algorithm with various of implementations.

In our project, we choose Fibonacci heap and an ordinary min-heap. We use C++ to implement this project.

1.2 Relative teminology

1.2.1 Shortest path problem

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

1.2.2 Dijkstra's algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

1.2.3 Fibonacci heap

A Fibonacci heap is a data structure for priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap.

Michael L. Fredman and Robert E. Tarjan developed Fibonacci heaps in 1984 and published them in a scientific journal in 1987. Fibonacci heaps are named after the Fibonacci numbers, which are used in their running time analysis.

Chapter2: Algorithm specification

2.1 Data structure

In the following class difinitions we ommited its functions and focus on the data structure. The total difinition can be seen in our source code.

2.1.1 min-heap

```
class node
{
private:
    /*suggest the id of the node*/
    int x;
    /*suggest the distance between the node with the source node*/
    double d;
};
```

```
/*Data type, this task requires the use of node*/
using DATATYPE = node;
class priority_queue
{
private:
    /*use vector as the storage type*/
    vector<DATATYPE> PriQueue;
    /*the current size of the queue*/
    int currentsize = 0;
}
```

2.1.2 Fibonacci heap

```
class FibNode {
    public:
    /*the node value*/
    node key;
    /*the degree of the node*/
    int degree;
    /*point to the left sibling*/
    FibNode *left;
    /*point to the right sibling*/
    FibNode *right;
    /*point to child node*/
    FibNode *child;
    /*point to the parent node, and this project doesn't use this property*/
    FibNode *parent;
    /*mark the child wheather it has been deleted, and this project doesn't use this property*/
    bool marked;
};
```

```
class FibHeap {
private:
    /*suggest the number of the nodes in the heap*/
    int keyNum;
    /*suggest the max degree of all nodes*/
    int maxDegree;
    /*point to the subtree with the minimun node as root*/
    FibNode *min;
    /*store the subtrees temporary*/
    FibNode **cons;
}
```

2.2.3 Graph structure

```
/*Used to record information in dijkstra procedure.*/
struct TableEntry {
    int Known;
    double Dist;
    int Path;
};
typedef struct TableEntry* Table;

/*used for store the infomation of one node*/
typedef struct VNode *PtrToVNode;
struct VNode {
    /*the id of the node*/
    Vertex Vert;
    /*the disance between the node i with it*/
```

```
    double dis;
    /*point to the next node*/
    PtrToVNode Next;
};

/*used for store the infomation of one graph*/
typedef struct GNode *MGraph;
struct GNode {
    /*the number of nodes of the graph*/
    int Nv;
    /*the number of edges of the graph*/
    int Ne;
    /*the array that store the infomation of all edges*/
    PtrToVNode *Array;
};
typedef struct GNode *PtrToGNode;
```

2.2 Procedure of the program

```
int main()
{
    Open a file which stores information of nodes;
    alloc space for a graph;
    use the file to implement the graph;

    /*test fibnarcci heap*/
    Open a file which stores our querys;
    Open a file which will store our results;
    Start time record;
    for(the number of querys)
    {
        Read in a and b;
        /*Find the shortest path between a and b*/
        Create a table with a the origin;
        Use Dijkstra algorithm to find the minmium distance of every node
to a;
        Print the distance between a and b into the result file;
    }
    Stop time record;
    Output the time spent;

    /*test min-heap query*/
    Open a file which will store our results;
    Start time record;
    for(the number of querys)
    {
        Read in a and b;
        /*Find the shortest path between a and b*/
        Create a table with a the origin;
        Use Dijkstra algorithm to find the minmium distance of every node
to a;
        Print the distance between a and b into the result file;
    }
    Stop time record;
    Output the time spent;

    Close files;
}
```

2.3 Pseudo-code of key algorithm

2.3.1 Dijkstra's algorithm

```
/**
 *@brief:Find the minimun distance between the source node with every other node
 *@param[in] Start: the id of the source node
 *@param[in] T: the table that used to record information in dijkstra procedure
 *@param[in] G: store the infomation of one graph
*/
void dijkstra(Vertex Start,Table T,MGraph G) {
    Heap q;
    /*create the heap*/
    for (int i = 0; i < G->Nv; i++) {
        if (the node have edge with source) {
            create a node x with i and T[i].Dist;
            Push it into the heap;
        }
    }
    while (the heap is not empty) {
        pop the min node of the heap as x;
        while(the min node have been visited)
        {
            pop the next min node;
        }
        mark the node as been visited;
        Use ptr to record node linked with the current node;
        while (ptr != NULL) {
            Vertex tempv = ptr->Vert;
            /*update the distance of all nodes link with this node*/
            if (T[tempv].Known == 0 && T[tempv].Dist > T[x.getx()].Dist + ptr->dis)
            {
                T[tempv].Dist = T[x.getx()].Dist + ptr->dis;
                create a node y with tempv and T[tempv].Dist;
                Push y into the heap;
                /*record the infomation of the path*/
                T[tempv].Path = x.getx();
            }
            ptr = ptr->Next;
        }
    }
}
```

2.3.2 Fibnacci Heap

All functions of Fibnacci heap is in our source code, in this part we declare two main operations on this heap.

2.3.2.1 Merge

```
 /**
 *@brief:Merge trees of the same degree in the root list of the Fibonacci heap
 */
void FibHeap ::consolidate()
{
    int i, d, D;
    FibNode  *x, *y;
    Apply for the space for temporary storage;
    /*as two trees with largest degree merging, the degree will plus one*/
    Initial maxDegree+1 nodes to NULL;

    /*Merge nodes of the same degree in the root node list, to make the tree unique for each degree*/
    while (min != NULL){
```

```
        Get the tree with min node as x;
        Get the degree of the min tree as d;
        // if cons[d]!=NULL, then there are two trees with the same degree
        while (cons[d] != NULL){
            /*y point to the tree of same degree of x*/
            y = cons[d];
            Adjust the order so that x has a smaller value;
            Link two trees;
            /*update*/
            cons[d] = NULL;
            d++;
        }
        Store the tree;
        cons[d] = x;
    }
    min = NULL;

    Add the trees in the cons to the root list;
}
```

2.3.2.2 DeleteMin

```
/**
 *@brief: remove the min node
 */
void FibHeap ::removeMin()
{
    if (min is NULL)
        return;
    FibNode  *child = NULL;
    FibNode  *m = min;
    /*add the child of the min node and its brothers into the double linked list*/
    while (m->child != NULL){
        child = m->child;
        removeNode(child);
        if (child->right == child) {
            m->child = NULL;
        }
        else {
            m->child = child->right;
        }
        add child to the double linked list;
    }

    remove the node from the linked list;
    if (the node is the only node in the heap) {
         set min to NULL;
    }
    else{
        set the right node as the minimun node;
        adjust the heap;
    }
    decrease the number of node in the heap;
    /*release the space*/
    delete m;
}
```

2.3.3 Min-Heap

All functions of min heap is in our source code, in this part we declare two main operations on this heap.

2.3.3.1 Pop

```
/**
 *@brief:pop the node with the minimum value
 */
void priority_queue::pop() {
    if (the heap is empty)
        output error;
    else {
        delete the minimum node;
        Use temp to record the last node of the heap;
        /*downward filtering process*/
        for (i = 1; i * 2 <= currentsize; i = child) {
            child = i * 2;
            Find the valid child with bigger value;
            if (temp > PriQueue[child]) {
                PriQueue[i] = PriQueue[child];
            }
            /*The procedure is over- temp finds its place*/
            else {
                break;
            }
        }
        PriQueue[i] = temp;
    }
}
```

2.3.3.2 Push

```
/**
 *@brief:enqueue a node
 *@param[in] val: the node that need to be enqueued
 */
using DATATYPE = node;
void priority_queue::push(const DATATYPE& val) {
    if (the queue is empty) {
        Initialize the queue;
    }
    /*if the space is not enough, apply for more */
    if (the space is not enough) {
        Resize priqueue to two times its current size ;
    }
    Increase the size of the queue and record the current size as k;
    /*upward filtering process*/
    while (k > 1 && val < PriQueue[k / 2]) {
        PriQueue[k] = PriQueue[k / 2];
        k /= 2;
    }
    /*insert the new node*/
    PriQueue[k] = val;
}
```

Chapter 3: Testing results

Notice: our test datas are loaded from http://www.dis.uniroma1.it/challenge9/download.shtml http://www.dis.uniroma1.it/challenge9/download.shtml

Computer Information:

Hardware	Config
CPU	intel Core i7-8750H @ 2.20GHz 6-core
Memory	16GB
Harddisk	240G SSD + 1T

Test Method:
In the program, we provide a macro Size, by adjusting the value of Size we could test the result with diffrent size. Here are the psuedo code of the testing procedure.

```
#define Size 1
for (int i = 0; i < Size; i++) {
    int start_node;  //start place
    int dest_node;  //destination place
    Read_Test_Data_From_Files
    initTable(start_node, G, T);
    fib_dijkstra(start_node, T, G); //or dijkstra(start_node, T, G) for another heap algorithm
    PrintPath(dest_node, T, fpath);
}
```

Time Consume:

Size	Fibnarcci_heap (s)	Ordinary_min_heap (s)
1	0.895	2.725
2	1.799	5.379
5	4.466	13.508
10	8.777	26.850
20	17.429	53.390
50	43.231	133.413
100	86.206	267.653
200	172.830	533.268
500	429.746	1330.276
1000	857.572	2657.325
2000	1714.536	5319.843

Chapter4 : Analysis and comments

4.1 Analysis

Bounds of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of the number of edges, denoted $|E|$, and the number of vertices, denoted $|V|$.
In the procedure of this program, when we use ordinary binary heap, it takes $log|V|$ for average time to find the minimum number in the heap, because we need to traverse all the vertexs and edges, thus the average time of the dijkstra algorithm using ordinary heap is $O(|V|log|V| + |E|log|V|)$

But when we use fibnarcci heap to implement dijkstra algorithm, it takes $O(1)$ for find_min and $O(log|V|)$ for delete_min. Thus, the time bound of the algorithm is $O(|E| + |V|log|V|)$

4.2 Comments

Above all, we can draw a conclusion that using fibnarcci heap is faster than using ordinary min heap when implement the dijkstra algorithm in average cases. With the growing of the query size, using fibnarcci heap works more efficient.

References

https://en.wikipedia.org/wiki/Fibonacci_heap

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Declaration

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

Duty Assignments

Member	Duty
杨佳妮	Programmer
应承峻	Tester
贺婷婷	Documentation

Signitures

杨佳妮 贺婷婷 应承峻