Advanced Data Structures and Algorithm Analysis

Project 2: Shortest Path Algorithm with Heaps

Date: 2024/10/19

Table of Contents

Chapter 1: Introduction

1.1 Problem description & background of the algorithms

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.

Given that the problem uses the USA road networks for evaluation, the graph should be defined as *undirected*.

DIJKSTRA'S ALGORITHM:

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

Dijkstra's algorithm finds the shortest path from a given source node to every other node. The algorithm uses a *min-priority queue* data structure for selecting the shortest paths known so far.

MIN-PRIORITY QUEUE:

A Min-Priority Queue is a special type of priority queue where the element with the smallest priority is dequeued first. In a Min-Priority Queue, the smaller the priority value, the higher the element's priority, and it is processed earlier.

In this problem, a smaller priority value corresponds to a shorter path distance, meaning that the vertex with the shortest path distance is prioritized for removal from the queue each time. The actual realization in our project is done by *binary heaps* and *Fibonacci heaps*.

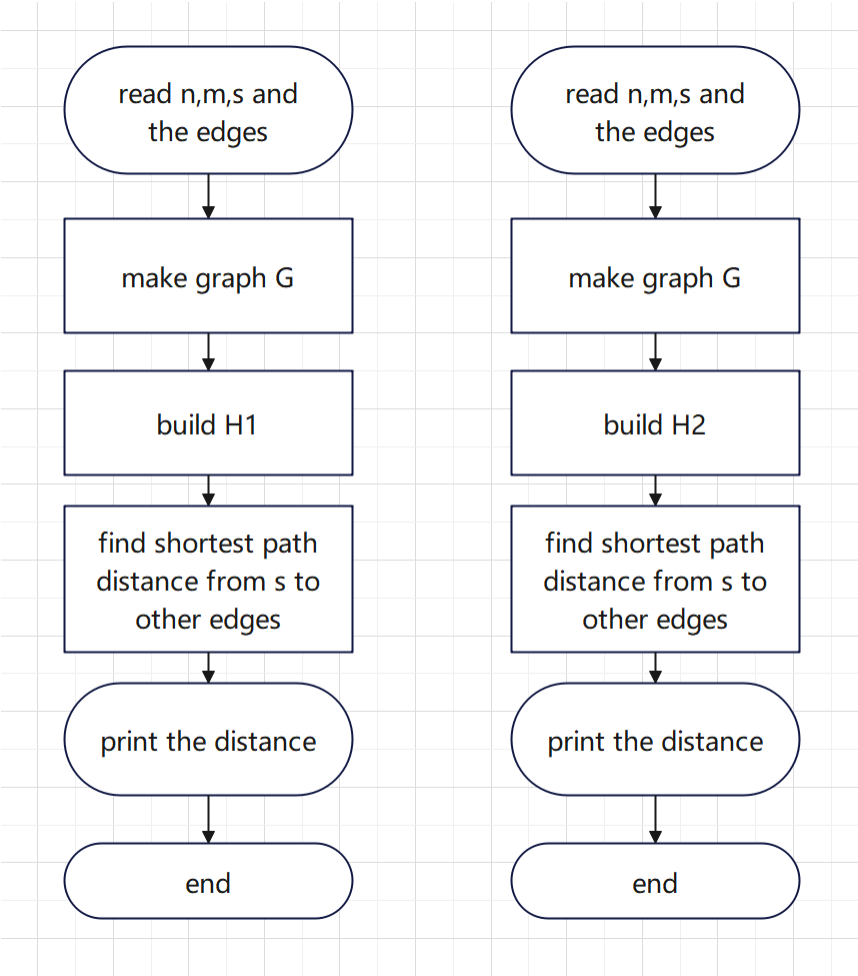
1.2 Our goals

1. Implement the algorithm with two different heap structure, binary heap & Fibonacci heap.
2. Use the USA road networks for evaluation. (http://www.diag.uniroma1.it/challenge9/download.shtml)
3.

Chapter 2: Algorithm Specification

2.1 Overall architecture design

The main program reads information such as n, m, s, and calls the *add_edge* function to make a graph G based on the information. Using two functions *bio_heap* and *fib_heap*, it constructs a binary heap $H1$,and a Fibonacci heap $H2$, then uses Dijkstra's algorithm to find the shortest path distances from the given node s to all the other nodes. Finally, it outputs the results in the required format.



2.2 About the Dijkstra's Algorithm

In the first chapter, we briefly introduced the Dijkstra's algorithm. Now we will provide a more detailed description.

Dijkstra's algorithm is a method used to find the shortest path in a weighted graph. Its main idea is to use a *greedy strategy* to progressively expand the shortest path from the

starting point to other vertices. The steps are as follows:

1. Initialization: Start from the initial vertex, set the distance to the start as 0, and set the distance to all other vertices as infinity.
2. Vertex selection: From the unvisited vertices, choose the one that has the shortest current distance to the starting point.
3. Path update: Check the neighbors of this vertex and calculate the distance from the start to these neighbors. If the path through the current vertex is shorter than previously recorded, update the neighbor's distance.
4. Mark as visited: Mark the current vertex as visited and stop processing it.
5. Repeat steps 2 to 4 until all vertices are visited or the shortest path to the target vertex is determined.

The pseudocode is as follows,

```
function Dijkstra(Graph, source):
    // Initialization
    distance[source] = 0
    for each vertex v in Graph:
        if v ≠ source:
            distance[v] = infinity

    priority_queue.push(source)

    // main loop
    while priority_queue is not empty:
        // select the nearest vertex u form the source
        u = priority_queue.pop()

        // path update
        for each neighbor v of u:
            alt = distance[u] + weight(u, v)
            if alt < distance[v]:
                distance[v] = alt
                priority_queue.push(v)

    return distance
```

2.3 About the Binary Heap

A binary heap is a heap data structure that takes the form of a binary tree. Binary heaps are a common way of implementing priority queues.

A binary heap is defined as a binary tree with two additional constraints:

1. Shape property: a binary heap is a complete binary tree; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
2. Heap property: the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) (in our case, less than or equal to) the keys in the node's children, according to some total order.

The *struct node* is as follows,

```
struct node{
    int id;Element_Type val;//dis[id] = val
    node(int Id = 0,Element_Type Val = 0){
        id = Id;
        val = Val;
    }
    bool operator<(node other){
        return val < other.val;
    }
    bool operator>(node other){
        return val > other.val;
    }
}H[maxn];
```

2.4 About the Fibonacci Heap

A Fibonacci heap is a collection of trees satisfying the minimum-heap property. 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.

However, at some point order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes are kept quite low: every node has degree at most $logn$ and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_i is the ith Fibonacci number. This is achieved by the rule: *at most one child can be cut off each non-root node*.

When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree. The number of trees is decreased in the operation *delete-min*, where trees are linked together.

The *struct node* is as follow,

```
struct node{
    int id;Element_Type val;
    int deg;//度数
    node *l,*r;//左兄弟和右兄弟，双向循环链表维护兄弟关系
    node *son,*fa;//第一个孩子，父亲
    bool mark;//是否被删除过孩子
    node(int Id = 0,Element_Type Val = 0){
        id = Id;val = Val;
        deg = 0;l = r = this ;son = fa = 0; mark = 0;
        //初始 左右兄弟都指向自己，从而保证循环
    }
    bool operator<(node other){
        return val < other.val;
    }
    bool operator>(node other){
        return val > other.val;
    }
}*Min,*pos[maxn];
```

Chapter 3: Testing results

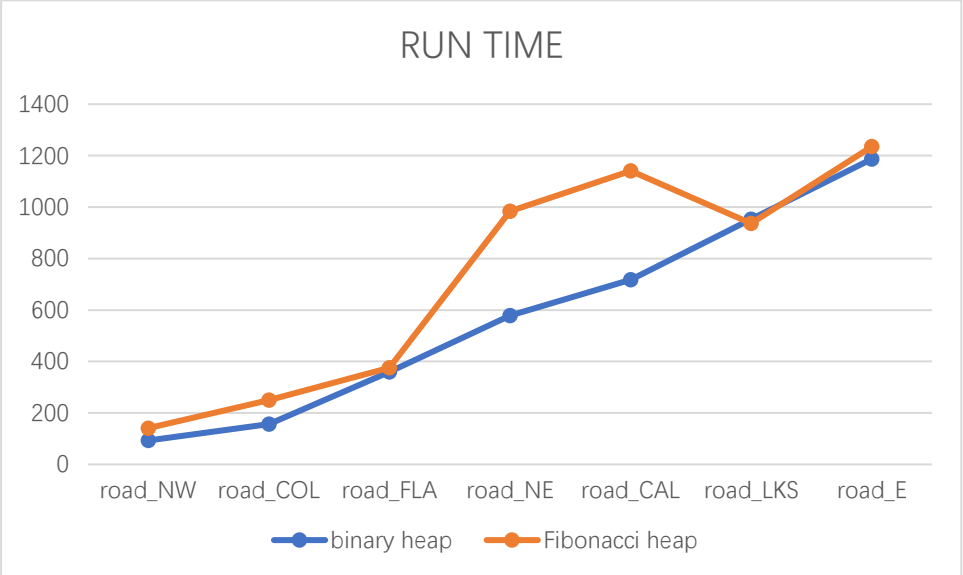
From website *"9th DIMACS Implementation Challenge - Shortest Paths"*, we download 7 datasets as our testing data, consisting of "USA-road-d.CAL", "USA-road-d.COL", "USA-road-d.E", "USA-road-d.FLA", "USA-road-d.LKS", "USA-road-d.NE", "USA-road-d.NY". Each with different number of vertexes and edges. The computer configuration we used for testing is *12th Gen Intel(R) Core(TM) i5-12500H, 2500 Mhz*.

We tested the correctness of both code implementations on Luogu. By generating the outputs *output_bio.txt* and *output_fib.txt*, we used the *fc* command in the command line to verify if the outputs are identical, ensuring the correctness of the code.

For each dataset, we used the Dijkstra algorithm based on a binary heap and the Dijkstra algorithm based on a Fibonacci heap, respectively, to test the runtime of the algorithm with both data structures and recorded the results in the table below,

test_case	vertexes	edges	binary heap	Fibonacci heap
road_CAL	1890815	4657742	718ms	1141ms
road_COL	435666	1057066	156ms	250ms
road_E	3598623	8778114	1187ms	1235ms
road_FLA	1070376	2712798	360ms	375ms
road_LKS	2758119	6885658	953ms	937ms
road_NE	1524453	3897636	578ms	984ms
road_NW	264346	733846	93ms	141ms

To illustrate it more clearly, we sorted the data by size and plotted the runtime as a chart.


RUN TIME
1400
1200
1000
800
600
400
200
0
road_NW
road_COL
road_FLA
road_NE
road_CAL
road_LKS
road_E
binary heap
Fibonacci heap

Chapter 4: Analysis and Comments

4.1 Time Complexity analysis

BINARY HEAP:

Insert The number of *percolate_up* needed is determined only by how many levels the new element must move up to maintain the heap property. Therefore, in the worst case, the insertion operation has a time complexity of $O(\log N)$.

Delete_min In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, so in the worst case the delete operation has a time complexity of $O(\log N)$.

Decrease_key Decrease key can be done by *finding the element*, *decreasing the key* and *perlocate_up*. Therefore, in the worst case, the time complexity is $O(find(x)) + O(perlocate\_up) = O(1) + O(\log N) = O(\log N)$
(By maintaining an array to record the position in heap,$find$ can be done in $O(1)$)

FIBONACCI HEAP:

Insert The insertion operation can be considered a special case of the merge operation, with a single node. The node is simply appended to the root list increasing the potential by one. The amortized cost is thus constant.

Delete_min First,we include potential energy function $f(i) = t(i) + 2m(i)$,with $t(i)$ express the length of root list,and m(i) express the number of marked node.During *Delete_min*,we push all child of the min node into root_list,and we merge the node in root_list by degree.Consider a node have D degree at most,and we make k times merging.Therefore,the cost c should be $O(D) + k$. $f(i-1) = t(i-1) + 2m(i-1), t(i) = t(i-1) + D - k, m(i-1) = m(i), f(i) = f(i-1) + D - k, \Delta f = D - k$
Thus amortized cost $c' = c + \Delta f = O(D) + k + D - k = O(D)$

Decrease_key Consider the function clean is called k times. Thus the function cutoff is called $k + O(1)$ times.The cost is $O(k)$. $\Delta t = k + O(1), \Delta m = -k + O(1), \Delta f = -k + O(1)$ Thus,sum of amortized decrease_key cost should be $O(\sum \Delta f) = O(M)$,so the amortized cost = $O(1)$

Now we need to prove that $O(D) = O(\log N)$:
For a node x in Fibonacci heap,consider it's degree is k.And name it's child $s_1 .. s_k$ under the time they become x's child.(s_1 is the first) Notice s_i became the x's child only

when their had same degree,and in that time x had at least $s_1..s_{i-1}$;so degree of $s_i \geqslant i-1$ at that time;and s_i can lost only 1 child(otherwise it will be cutoff),so degree of $s\_i \geqslant i-2$ now.

Let's name F_k be the Fibonacci array ($F_0 = 0, F_1 = 1$)

Name $\phi = (1+\sqrt{5})/2$ = 1.618..,$\phi \times \phi = \phi + 1$.We have $F_{k+2} \geqslant \phi^k$

And we have $F_{k+1} = 1 + \sum_0^{k-1} F_i$ (Use inductive method)

Name $size(k)$ to be the max size of a node of k degree.

Finally let's prove $size(x) \geqslant F_{k+2} \geqslant \phi^k$ with inductive method:

$$size(k) = 1 + \sum size(s_i) \geqslant 1 + \sum size(i-2) \geqslant 1 + \sum F_i \geqslant F_{k+2} \geqslant \phi^k$$

Thus $size(D) \geqslant \phi^D, O(D) = O\big(\log\big(size(D)\big)\big) = O(\log N)$

4.2 comments

In this problem, we find that the running time for a Fibonacci heap realization is actually longer than a binary heap way.

Although Fibonacci heaps have better theoretical amortized time complexity compared to binary heaps, the main why it takes longer run time might be as follow,

1. Fibonacci heaps have a more complicated structure and require more intricate operations to maintaining a loose collection of trees, which introduce significant overhead in real implementations.
2. While Fibonacci heaps have better theoretical complexity, the actual constant factors involved in their operations are larger. Operations like merging and decreasing keys, although $O(1)$ in theory, is a kind of "large constant".
3. Binary heaps have a compact memory layout, since they are complete binary trees, making them more cache-efficient. Fibonacci heaps, on the other hand, distribute nodes across multiple trees, leading to more scattered memory access, which reduces cache performance.

Declaration

I hereby declare that all the work done in this project titled "Shortest Path Algorithm with Heaps " is of my independent effort.

Appendix

```
#include <cstdio>
#include <math.h>
#include<fstream>
#include<iostream>
#include<windows.h>

const int maxn = 6e6 + 5;
typedef long long Element_Type;

int n,m,s; // 点数，边数，起点

```

```
12. struct edge{
13.     int v,nxt;
14.     Element_Type w;
15. }e[maxn];
16. 
17. int fir[maxn],edge_cnt;
18. 
19. //链表方式存储边
20. void add_edge(int u,int v,Element_Type w){
21.     e[++edge_cnt].nxt = fir[u]; e[edge_cnt].v = v; e[edge_cnt].w = w;
     fir[u] = edge_cnt;
22. }
23. 
24. template<typename T>
25. inline void swap(T &a,T &b){
26.     T temp = a; a = b; b = temp;
27. }
28. 
29. struct BioHeap{
30.     //数组实现二叉堆（小根堆）
31.     struct node{
32.         int id;Element_Type val;//dis[id] = val
33.         node(int Id = 0,Element_Type Val = 0){
34.             id = Id;
35.             val = Val;
36.         }
37.         bool operator<(node other){
38.             return val < other.val;
39.         }
40.         bool operator>(node other){
41.             return val > other.val;
42.         }
43.     }H[maxn];
44.     int siz;//堆内元素个数
45. 
46.     int pos[maxn];//记录在堆内的位置,pos[H[k].id] = k
47. 
48.     //上浮
49.     void percolate_up(int id){
50.         for(;id > 1;id /= 2){
51.             if(H[id/2] > H[id]) swap(pos[H[id/2].id],pos[H[id].id]),s
    wap(H[id/2],H[id]);
52.             else break;
53.         }
```

```
54.     }
55.
56.     //下沉
57.     void percolate_down(int id){
58.         while(id*2 <= siz){
59.             int t = id*2;
60.             if(t<siz && H[t]>H[t+1]) t = t + 1;
61.             if(H[id] > H[t]){
62.                 swap(pos[H[id].id],pos[H[t].id]);
63.                 swap(H[id],H[t]); id = t;
64.             }
65.             else break;
66.         }
67.     }
68.
69.     void insert(int id,Element_Type val){
70.         pos[id] = ++siz;
71.         H[siz] = node(id,val); percolate_up(siz);
72.     }
73.     void decrease_key(int id,Element_Type val){
74.         int p = pos[id];
75.         if(H[p].val < val){
76.             puts("Error:meets a higher key when decrease key in BioHe
    ap");
77.             return ;
78.         }
79.         H[p].val = val; percolate_up(p);
80.     }
81.     //查询当前最近节点
82.     int get_min(){
83.         if(siz == 0){
84.             puts("Error:empty heap when get min in BioHeap");
85.             return 0;
86.         }
87.         return H[1].id;
88.     }
89.     void delete_min(){
90.         if(siz == 0){
91.             puts("Error:empty heap when delete_min in BioHeap");
92.             return ;
93.         }
94.         swap(pos[H[1].id],pos[H[siz].id]); swap(H[1],H[siz]);
95.         siz--;
96.         percolate_down(1);
```

```
97.     }
98.     int empty(){return siz == 0;}
99. }Bio;
100.
101.   //斐波那契堆
102.   struct FibHeap{
103.       struct node{
104.           int id;Element_Type val;
105.           int deg;//度数
106.           node *l,*r;//左兄弟和右兄弟，双向循环链表维护兄弟关系
107.           node *son,*fa;//第一个孩子，父亲
108.           bool mark;//是否被删除过孩子
109.           node(int Id = 0,Element_Type Val = 0){
110.               id = Id;val = Val;
111.               deg = 0;l = r = this ;son = fa = 0; mark = 0;
112.               //初始 左右兄弟都指向自己，从而保证循环
113.           }
114.           bool operator<(node other){
115.               return val < other.val;
116.           }
117.           bool operator>(node other){
118.               return val > other.val;
119.           }
120.       }*Min,*pos[maxn];
121.       //Min 表示整个 FibHeap 中的最小节点
122.       int siz;
123.
124.       //在双向链表中，将 t 接到 s 后面
125.       void insert_list(node *s,node *t){
126.           // if(s==0 || t==0) while(1);
127.           t->r = s->r; s->r->l = t;
128.           t->l = s; s->r = t;
129.       }
130.
131.       //在双向链表中，删掉 s
132.       void remove_list(node *s){
133.           // if(s == 0) while(1);
134.           s->l->r = s->r;
135.           s->r->l = s->l;
136.       }
137.
138.       //新建节点，并插入到 root_list 中
139.       void insert(int id,Element_Type val){
140.           node *cur = new node(id,val);
```

```
141.            if(siz == 0) Min = cur;
142.            else{
143.                insert_list(Min,cur);
144.                if(cur->val < Min->val) Min = cur;
145.            }
146.            siz ++ ;
147.            pos[id] = cur;
148.        }
149.
150.        //将 t 接为 s 的儿子
151.        void link(node *s,node *t){
152.            if(s->son) insert_list(s->son,t);
153.            else s->son = t,t->l=t->r=t;
154.            t->fa = s;
155.            s->son = t;
156.            s->deg ++ ;
157.        }
158.
159.        node* tmp[maxn];
160.
161.        void delete_min(){
162.            //处理特殊情况
163.            if(siz == 0){
164.                puts("Error:empty heap when delete_min in FibHeap");
165.                return ;
166.            }
167.            if(siz == 1){
168.                pos[Min->id] =0;
169.                delete Min;
170.                Min = 0;siz = 0;
171.                return ;
172.            }
173.            int D = ceil(log(siz)/log(1.6)); //度数上界
174.            //将 Min 的所有儿子放入 root_list 中
175.            node *cur = Min->son;
176.            if(cur){
177.                do{
178.                    node *nxt = cur->r;
179.                    cur->fa = 0 ;
180.                    insert_list(Min,cur);
181.                    cur = nxt;
182.                }while(cur != Min->son);
183.            }
184.
```

```
185.          //对 root_list 中除了 Min 之外的节点按度数合并
186.          cur = Min->r;
187.          for(int i = 0 ; i <= D ; i ++ ) tmp[i] = 0;
188.          while(cur != Min){
189.              node *nxt = cur->r; remove_list(cur);
190.              cur->l = cur->r = cur;
191.              int d = cur->deg;
192.              while(tmp[d]){
193.                  //保证 cur 更小
194.                  if(tmp[d]->val < cur->val) swap(tmp[d],cur);
195.                  link(cur,tmp[d]);
196.                  tmp[d] = 0;
197.                  d ++ ; //合并后度数 + 1
198.              }
199.              tmp[d] = cur; cur = nxt;
200.          }
201.          pos[Min->id] = 0; delete Min ; Min = 0; //删除 Min
202.          //将合并后的结果重新连成 root_list
203.          for(int i = 0; i <= D; i ++ ){
204.              if(tmp[i] == 0) continue;
205.              if(Min == 0) Min = tmp[i];
206.              else{
207.                  insert_list(Min,tmp[i]);
208.                  if(Min->val > tmp[i]->val) Min = tmp[i];
209.              }
210.              tmp[i] = 0;
211.          }
212.          siz -- ;
213.      }
214.
215.      //将 cur 结点从其父亲上脱离，并放到 root_list 中，mark 置 0
216.      void cutoff(node *cur){
217.          if(cur->fa){
218.              //维护父亲的 deg , son
219.              cur->fa->deg -- ;
220.              if(cur->fa->son == cur){
221.                  if(cur->r != cur) cur->fa->son = cur->r;
222.                  else cur->fa->son = 0;
223.              }
224.              remove_list(cur),insert_list(Min,cur) ; //这里如果
    在 cur->fa == 0 的情况下执行这行代码就会出问题
225.              cur->fa = 0; cur->mark = 0 ;
226.          }
227.      }
```

```
228.
229.        //递归剪切，将 mark == 0 的 mark 置为 1 并返回，否则将其 cutoff 并
    继续递归
230.        void clean(node *cur){
231.            if(cur == 0) return ;
232.            if(cur->mark == 0) cur->mark = 1;
233.            else{
234.                node *fa = cur->fa ;
235.                cutoff(cur); clean(fa);
236.            }
237.        }
238.
239.        void decrease_key(int id,Element_Type val){
240.            node *cur = pos[id];
241.            if(cur == 0){
242.                puts("Error:ZERO");
243.                return ;
244.            }
245.            if(cur->id != id){
246.                puts("Error:wrong position!");
247.                return ;
248.            }
249.            if(cur->val < val){
250.                puts("Error:meets a higher key when decrease key in Fi
    bHeap");
251.                return ;
252.            }
253.            cur->val = val;
254.            if(Min == cur) return ;
255.            node *fa = cur->fa;
256.            if(fa && cur->val < fa->val){
257.                cutoff(cur) ; clean(fa);
258.            }
259.            if(Min->val > cur->val) Min = cur;
260.        }
261.        int get_min(){
262.            if(Min == 0){
263.                printf("siz:%d\n",siz);
264.                puts("Error:empty heap when get min in FibHeap");
265.                return 0 ;
266.            }
267.            return Min->id;
268.        }
269.        int empty(){return siz == 0;}
```

```
270.
271.        //调试用代码
272.        void prints(node *cur){
273.            if(cur == 0) return ;
274.            node *p=cur;
275.            do{
276.                printf("id:%d fa:%d\n",p->id,p->fa?p->fa->id:0);
277.                prints(p->son);
278.                p = p->r;
279.            }while(p!=cur);
280.            puts("");
281.        }
282.        void print(){
283.            puts("-----------------------------------------------");
284.            printf("siz:%d\n",siz);
285.            prints(Min);
286.            puts("-----------------------------------------------");
287.            fflush(stdout);
288.        }
289.    }Fib;
290.
291.    const Element_Type inf = 1e18;
292.
293.    //Dijktra 二叉堆实现
294.    Element_Type dis_bio[maxn];bool vis_bio[maxn];
295.    void Dij_bioheap(){
296.        for(int i = 1; i <= n; i ++ ) dis_bio[i] = inf;
297.        dis_bio[s] = 0;Bio.insert(s,0);vis_bio[s] = 1;
298.        while(!Bio.empty()){
299.            int u = Bio.get_min() ; Bio.delete_min() ;
300.            // printf("u:%d\n",u);
301.            for(int i = fir[u]; i ; i = e[i].nxt){
302.                int v = e[i].v;
303.                if(dis_bio[v] > dis_bio[u] + e[i].w){
304.                    dis_bio[v] = dis_bio[u] + e[i].w;
305.                    if(vis_bio[v] == 0){
306.                        vis_bio[v] = 1;
307.                        Bio.insert(v,dis_bio[v]);
308.                    }
309.                    else Bio.decrease_key(v,dis_bio[v]);
310.                }
311.            }
312.        }
```

```
313.      for(int i = 1; i <= n; i ++ ) printf("%lld%c",dis_bio[i],i==n?
    '\n':' ');
314.  }
315.
316.  //斐波那契堆实现
317.  Element_Type dis_fib[maxn];bool vis_fib[maxn];
318.  void Dij_fibheap(){
319.      for(int i = 1; i <= n; i ++ ) dis_fib[i] = inf;
320.      dis_fib[s] = 0;Fib.insert(s,0);vis_fib[s] = 1;
321.      while(!Fib.empty()){
322.          int u = Fib.get_min() ; Fib.delete_min() ;
323.          for(int i = fir[u]; i ; i = e[i].nxt){
324.              int v = e[i].v;
325.              if(dis_fib[v] > dis_fib[u] + e[i].w){
326.                  dis_fib[v] = dis_fib[u] + e[i].w;
327.                  if(vis_fib[v] == 0){
328.                      vis_fib[v] = 1;
329.                      Fib.insert(v,dis_fib[v]);
330.                  }
331.                  else Fib.decrease_key(v,dis_fib[v]);
332.              }
333.          }
334.      }
335.      for(int i = 1; i <= n; i ++ ) printf("%lld%c",dis_fib[i],i==n?
    '\n':' ');
336.  }
337.
338.  int main(){
339.      freopen("D:\\Project-2\\Dij\\road\\road_NY.txt","r",stdin);
340.      freopen("output_NY_try.txt","w",stdout);
341.      // std::ifstream infile("road_CAL.txt");
342.      //  if(!infile){
343.      //      std::cerr<<"无法打开文件! "<<std::endl;
344.      //      return 1;
345.      //  }
346.
347.      scanf("%d%d%d",&n,&m,&s);   //n 个点，m 条边，s 是起点
348.      //infile>>n>>m>>s;
349.      // printf("%d\n",n); return 0 ;
350.      DWORD t1,t2,t3,t4;
351.      for(int i = 1; i <= m; i ++ ){
352.          int u,v; Element_Type w;
353.          scanf("%d%d%lld",&u,&v,&w);
354.          // infile>>u>>v>>w;
```

```
355.            add_edge(u,v,w);
356.            // add_edge(v,u,w); //如果是无向图
357.        }
358.        t1=GetTickCount();
359.        Dij_bioheap();
360.        t2=GetTickCount();
361.        t3 = GetTickCount();
362.        Dij_fibheap();
363.        t4 = GetTickCount();
364.        printf("Bioheap time:%dms\n",t2-t1);
365.        printf("Fibheap time:%dms\n",t4-t3);
366.        return 0 ;
367.    }
```