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 and background of the algorithms**

**1.2 Our goals**

**Chapter 2：**Algorithm Specification

**2.1 Overall architecture design**

**2.2 About the Binary Search Tree (BST)**

**2.3 About the In\_order Traversal**

**2.4 About the Pre\_order Traversal**

**2.6 About the “Two-pointer” Algorithm**

**Chapter 3：**Testing results

**Chapter 4:** Analysis and Comments

**4.1 Data analysis**

**4.2 Complexity analysis**

**4.3 comments**

**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.

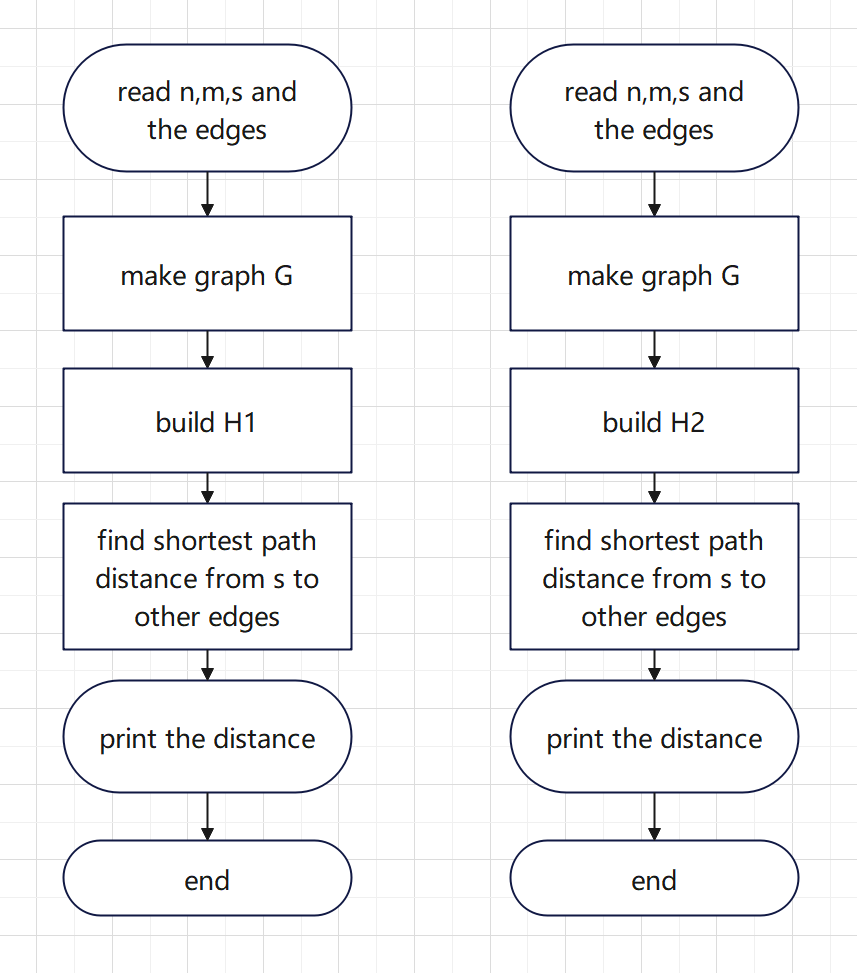
(<http://www.diag.uniroma1.it/challenge9/download.shtml>)

3.

**Chapter 2：**Algorithm Specification

**2.1 Overall architecture design**

The main program reads information such as , , , and calls the *add\_edge* function to make a graph based on the information. Using two functions *bio\_heap* and *fib\_heap*, it constructs a binary heap ,and a Fibonacci heap, then uses Dijkstra’s algorithm to find the shortest path distances from the given node 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,

1. function Dijkstra(Graph, source):
2. // Initialization
3. distance[source] = 0
4. **for** each vertex v in Graph:
5. **if** v ≠ source:
6. distance[v] = infinity
8. priority\_queue.push(source)
10. // main loop
11. **while** priority\_queue is not empty:
12. // select the nearest vertex u form the source
13. u = priority\_queue.pop()
15. // path update
16. **for** each neighbor v of u:
17. alt = distance[u] + weight(u, v)
18. **if** alt < distance[v]:
19. distance[v] = alt
20. priority\_queue.push(v)
22. **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,

1. **struct** node{
2. **int** id;Element\_Type val;//dis[id] = val
3. node(**int** Id = 0,Element\_Type Val = 0){
4. id = Id;
5. val = Val;
6. }
7. **bool** operator<(node other){
8. **return** val < other.val;
9. }
10. **bool** operator>(node other){
11. **return** val > other.val;
12. }
13. }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 and the size of a subtree rooted in a node of degree is at least , where is the th 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.

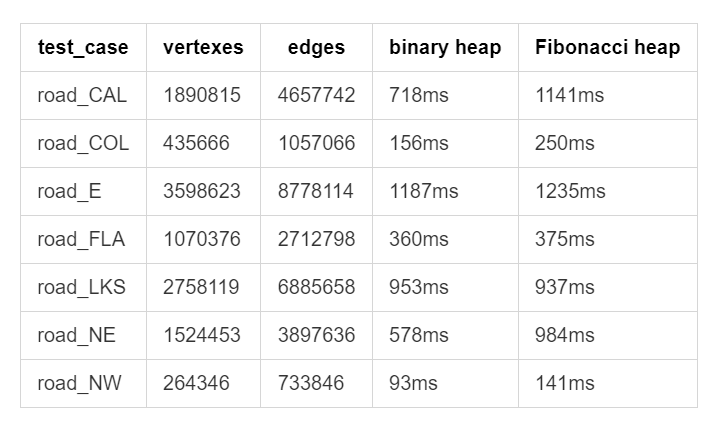
The *struct node* is as follow,

1. **struct** node{
2. **int** id;Element\_Type val;
3. **int** deg;//度数
4. node \*l,\*r;//左兄弟和右兄弟，双向循环链表维护兄弟关系
5. node \*son,\*fa;//第一个孩子，父亲
6. **bool** mark;//是否被删除过孩子
7. node(**int** Id = 0,Element\_Type Val = 0){
8. id = Id;val = Val;
9. deg = 0;l = r = **this** ;son = fa = 0; mark = 0;
10. //初始 左右兄弟都指向自己，从而保证循环
11. }
12. **bool** operator<(node other){
13. **return** val < other.val;
14. }
15. **bool** operator>(node other){
16. **return** val > other.val;
17. }
18. }\*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,

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

**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.

*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 .

*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

**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.

**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 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

1. #include <cstdio>
2. #include <math.h>
3. #include<fstream>
4. #include<iostream>
5. #include<windows.h>
7. **const** **int** maxn = 6e6 + 5;
8. **typedef** **long** **long** Element\_Type;
10. **int** n,m,s; // 点数，边数，起点
12. **struct** edge{
13. **int** v,nxt;
14. Element\_Type w;
15. }e[maxn];
17. **int** fir[maxn],edge\_cnt;
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. }
24. **template**<**typename** T>
25. **inline** **void** swap(T &a,T &b){
26. T temp = a; a = b; b = temp;
27. }
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
46. **int** pos[maxn];//记录在堆内的位置,pos[H[k].id] = k
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]),swap(H[id/2],H[id]);
52. **else** **break**;
53. }
54. }
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. }
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 BioHeap");
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;
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;
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. }
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. }
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. }
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. }
159. node\* tmp[maxn];
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. }
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. }
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. }
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. }
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 FibHeap");
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;}
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;
291. **const** Element\_Type inf = 1e18;
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. }
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. }
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. //  }
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. }