哈尔滨工业大学

**<<数据结构与算法>>**

**实验报告**

**(2024年秋季学期)**

|  |  |
| --- | --- |
| **姓名：** | **郭屹鸣** |
| **学号：** | **2023111958** |
| **学院：** | **计算机学院** |
| **教师：** | **翟德明** |

实验三 图性结构及其应用

## 一、实验内容

**实验题目 2：最小生成树算法**

**实验内容：**

最小生成树是数据结构与算法中图的一种重要应用，在图中对于具有 n 个顶

点的连通网可以建立许多不同结构的生成树，最小生成树就是在所有生成树中边

权值之和最小的生成树。在计算机领域和实际工程中具有广泛的应用，如局域网的搭建，道路网（畅通工程）、地下管网的设计等。本实验要求设计和实现 Prim

和 Kruskal等算法，求解最小生成树问题。

**实验要求：**

1． 选择并建立加权连通图的存储结构，实现求解加权连通图的 Prim 算法，并

输出连接各顶点的最小生成树。

2． 利用并查集，实现求解加权连通图的 Kruskal 算法，并输出连接各顶点的最

小生成树。

3． 以文件形式输入图的顶点和边，并以适当的方式展示相应的结果。要求顶点

不少于 10个，边不少于 13 个。

4． 通过实验的方法，比较 Prim和 Kruskal 算法的时间性能，并与理论

分析结果进行比较。你的实验结果是否与理论分析结果一致？你的实验结果

是否支持“Prim算法对边稠密的图更有优势，而 Kruskal 算法对边稀疏的图

更具优势”这个结论？

5． 利用堆结构（优先级队列）改进和优化 Prim 算法，实现改进和优

化的 Prim 算法，并与原算法进行实验比较。

6． (选做)设计并实现其他最小生成树算法。例如，管梅谷破圈算法、Sollin

（Boruvka）算法。

## 二、实验过程及结果

实验采用了模块化编程，采用了面向对象编程的思想理念，完成了树形结构的设计和算法的实现，完成了本次实验的全部任务。

**2.1 主要流程**：

**(1)图的初始化与数据导入**

读取文件 cleaned\_edges.txt（50个顶点，244条边的无向图），将节点和边的信息加载到邻接矩阵中进行存储。

**（2）初始化图的结构**

**（3）Kruskal 最小生成树算法实现**.

初始化节点的连接状态，使每个节点属于不同连通分量。收集所有边并按照权重排序。依次选择最小权重的边并判断是否会形成环。输出生成树的边及其总权重。如图1.

**（4）Prim 最小生成树算法实现**

从指定中心节点开始，依次选择最小权重的边，将相邻未访问的节点加入树中。

不断更新其他节点的权重，确保每次选取最优边。输出生成树的结构及总权重。如图2

**（5）Heapify 优化的 Prim 算法实现**

使用堆结构动态维护最小边权，减少搜索的复杂度。实现堆化（Heapify）和向上浮动的插入逻辑。通过堆结构实现更高效的 Prim 算法，输出生成树及其总权重。如图3,4

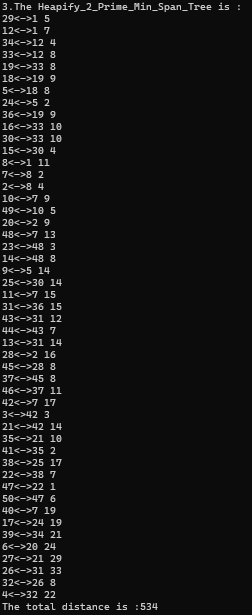
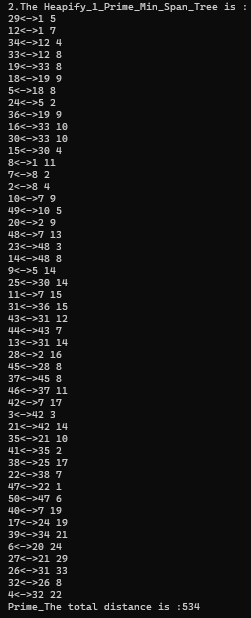
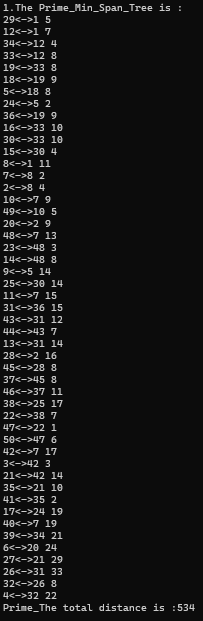
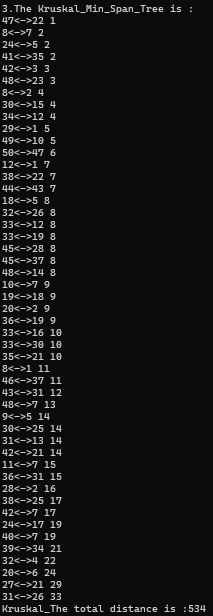


图1 图2 图3 图4

**（6）算法性能测量**

重复运行每种算法多次（10 \* MVINT 次），测试其性能和运行时间。使用 std::chrono 进行时间测量，并输出运行时间结果。

**（7）实验结果展示与比较**

输出各算法生成的最小生成树及其总权重。

在Heavy\_load的情况下，比较不同算法的运行时间，堆可优化Prime算法选取临近顶点的过程，将线性搜索n降低到logn。但是根据实验，静态与动态建立堆的时间复杂度相同，但是动态建堆优化了整体的堆操作过程，速度大大增加。静态堆运行时间甚至多于Naive\_Prime算法。其中在trail.cpp中展示了light\_payload过程中第一次堆优化，由于存在大量重建而**操作缓慢**如图4。在Heavy\_load.cpp的Heavy\_1中性能有所提高，改重建堆为上浮调整，但是仍然运行缓慢，Heavy\_load.cpp的Heavy\_2使用动态建堆，速度比Naive\_Prime提高了近**20**倍。如图5.

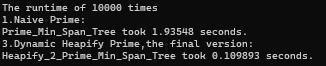
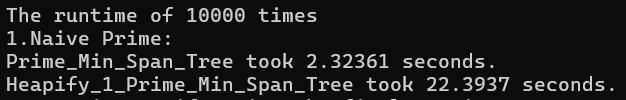


图4 图5

**（8）补充**：

支持“Prim算法对边稠密的图更有优势，而 Kruskal 算法对边稀疏的图

更具优势” 由于Prime主要是加入点，而Kruskal则是对边进行操作。关于稀疏图和稠密图构建如GRAPH.txt和Cleared\_Edge.txt.如图6的稠密Cleared\_Edge.txt运行结果。

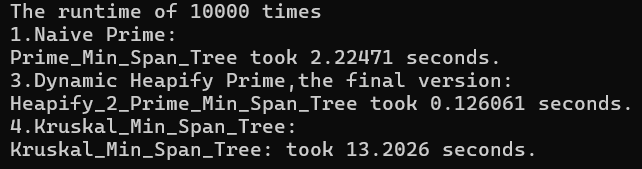


图6

1. **可以继续探讨改进**：

堆优化的Prime算法在最好情况下应该是(m+n)logn,而我使用邻接矩阵存储Graph，带来了额外的一次for循环，我的算法复杂度是n(m+n)logn,同时矩阵存储了很多冗余的信息，其本身的结构在导致输出可视化不友好。

以后有时间可以改用邻接表进行存储。

## 三、实验心得

* **遇到的问题和解决方法**

1. 这次实验的逻辑思维含量比较大，正常两个算法的实现比较简单。但是在堆优化prime的过程中遇到了困难。其中在trail.cpp中展示了light\_payload过程中第一次堆优化，由于存在大量重建而操作缓慢。在Heavy\_load.cpp的Heavy\_1中性能有所提高，改重建堆为上浮调整，但是仍然运行缓慢（静态和图存储方法的bug），Heavy\_load.cpp的Heavy\_2使用动态建堆，速度比Naive\_Prime提高了近20倍。
2. 同时，由于最开始没有正确选择邻接表作为图存储形式，给后续的debug'带来了巨大无比的困难(),为了输出可视化让图平添了许多冗余的信息。但是好在熬过来了（），debug之后跑出来还是很开心的。

* **体会和感受**

通过本次实验，我对图论中的**最小生成树算法**有了更深入的理解，并且学会了如何通过不同的算法实现同一问题.

在性能测试环节中，通过多次运行算法并记录时间，我对算法的时间复杂度有了更直观的感受。尤其是对比原始 Prim 和 Heapify 优化后的 Prim 算法的运行时间后，我理解了算法复杂度在处理大规模数据时的关键作用。

在实验过程中遇到了诸如文件路径读取错误、算法边界条件处理不当等问题。这些问题让我认识到，细节上的严谨和不断调试是确保程序正确性的重要环节。

最后，非常感谢为我答疑解惑的助教童老师。

1. **源代码**
2. #pragma once
3. #include <iostream>
4. #include <fstream>
5. #include <sstream>
6. #include <stack>
7. #include <vector>
8. #include <deque>
9. #include <string>
10. #include <chrono>
11. #include <functional>
12. *//------------------------Trial--------static Heapify\_Prime------------------//*
13. #include "ulity.h"
14. constexpr int MVINT = 10000;
15. class Mat\_Graph\_Operator{
16. public:
17. typedef struct{
18. char vex\_name;
19. int connected;
20. int visited;
21. }data;
22. typedef struct{
23. std::vector<data>vexs;
24. std::vector<std::vector<int>>arcs;
25. }Mat\_Graph;
26. typedef struct{
27. int real\_index;
28. int adjvex\_index;
29. int lowcost;
30. }Cost;
31. typedef struct{
32. int front\_index;
33. int rear\_index;
34. int lowcost;
35. }edge;
36. Mat\_Graph MG;
37. std::vector<Mat\_Graph\_Operator::Cost>cost;
38. std::vector<edge>Edge;
39. int vexnum,arcnum;
40. Mat\_Graph\_Operator(int \_vexnum,int \_arcnum):vexnum(\_vexnum),arcnum(\_arcnum){
41. Inital\_Mat\_Graph();
42. Read\_File\_Generate\_Mat();
43. }
44. virtual ~Mat\_Graph\_Operator(){}
46. void Prime\_Min\_Span\_Tree(bool apparent){
47. if(apparent){
48. std::cout<<"1.The Prime\_Min\_Span\_Tree is :"<<std::endl;
49. }
50. cost.resize(vexnum);
51. int center = 0;
52. for(int i = 0; i<vexnum; i++){
53. cost[i].adjvex\_index = center;
54. cost[i].real\_index = i;
55. cost[i].lowcost = center!=i ? MG.arcs[center][i]:0;
56. }
57. for(int k = 0; k<vexnum-1; k++){
58. int next\_index = Select\_Min(cost);
59. if(apparent){
60. std::cout<<MG.vexs[next\_index].vex\_name<<MG.vexs[cost[next\_index].adjvex\_index].vex\_name<<" "
61. <<MG.arcs[next\_index][cost[next\_index].adjvex\_index]<<std::endl;
62. }
63. cost[next\_index].lowcost = 0;
64. cost[next\_index].adjvex\_index = next\_index;
65. for(int i = 0; i<vexnum; i++){
66. if(cost[i].lowcost!=0 && cost[i].lowcost > MG.arcs[i][next\_index]){
67. cost[i].lowcost = MG.arcs[i][next\_index];
68. cost[i].adjvex\_index = next\_index;
69. }
70. }
71. }
72. if(apparent) std::cout<<std::endl;
73. }
74. void Kruskal\_Min\_Span\_Tree(){
75. std::cout<<"3.The Kruskal\_Min\_Span\_Tree is :"<<std::endl;
76. int sum\_distance = 0;
77. for(int i = 0; i<vexnum ;i++){
78. MG.vexs[i].connected = i;
79. }
80. for(int i = 0; i<vexnum; i++){
81. for(int j = 0; j<=i; j++){
82. if(MG.arcs[i][j] != MVINT&&i!=j ){
83. Edge.push\_back({i,j,MG.arcs[i][j]});
84. }
85. }
86. }
87. Sort\_Min\_Edge();
88. *// for(int i = 0;i<arcnum;i++){*
89. *//     std::cout<<Edge[i].lowcost<<" ";*
90. *// }*
91. *// std::cout<<std::endl;*
93. for(int i = 0; i<arcnum; i++){
95. int v1 = Edge[i].front\_index;
96. int v2 = Edge[i].rear\_index;
97. int vs1 = MG.vexs[v1].connected;
98. int vs2 = MG.vexs[v2].connected;
100. if(vs1!=vs2){
101. sum\_distance += Edge[i].lowcost;
102. std::cout<<MG.vexs[v1].vex\_name<<MG.vexs[v2].vex\_name<<" "<<Edge[i].lowcost<<std::endl;
103. MG.vexs[v2].connected = vs1;
104. for(int j = 0; j<vexnum; j++){
105. if(MG.vexs[j].connected == vs2) MG.vexs[j].connected = vs1;
106. }
107. }
108. }
109. std::cout<<"The total distance is :"<<sum\_distance<<std::endl;
110. }
111. void Measure\_Heapify\_Prime\_Min\_Span\_Tree() {
112. auto start = std::chrono::high\_resolution\_clock::now();
113. for(int i = 0; i< 10\*MVINT;i++) Heapify\_Prime\_Min\_Span\_Tree(false);
114. auto end = std::chrono::high\_resolution\_clock::now();
115. std::chrono::duration<double> duration = end - start;
116. std::cout << "Heapify\_Prime\_Min\_Span\_Tree took " << duration.count() << " seconds.\n";
117. }
118. void Measure\_Prime\_Min\_Span\_Tree() {
119. auto start = std::chrono::high\_resolution\_clock::now();
120. for(int i = 0; i< 10\*MVINT;i++) Prime\_Min\_Span\_Tree(false);
121. auto end = std::chrono::high\_resolution\_clock::now();
122. std::chrono::duration<double> duration = end - start;
123. std::cout << "Prime\_Min\_Span\_Tree took " << duration.count() << " seconds.\n";
124. }
125. virtual void Mat\_Graph\_Control(){
126. *//Display\_Mat\_Graph();*
127. Prime\_Min\_Span\_Tree(true);
128. Heapify\_Prime\_Min\_Span\_Tree(true);
129. Kruskal\_Min\_Span\_Tree();
130. Measure\_Prime\_Min\_Span\_Tree();
131. Measure\_Heapify\_Prime\_Min\_Span\_Tree();
132. }
133. void Heapify\_Prime\_Min\_Span\_Tree(bool apparent){
134. if(apparent){
135. std::cout<<"2.The Heapify\_Prime\_Min\_Span\_Tree is :"<<std::endl;
136. }
137. std::deque<Mat\_Graph\_Operator::Cost>virable\_cost;
138. std::vector<Mat\_Graph\_Operator::Cost>const\_cost;
139. int center = 0;
140. for(int i = 0; i<vexnum; i++){
141. int lowcost;
142. if(i == 0) lowcost = 0;
143. else lowcost = MG.arcs[center][i];
144. virable\_cost.push\_back({i,0,lowcost});
145. const\_cost.push\_back({i,0,lowcost});
146. }
147. virable\_cost.pop\_front();
148. Bulid\_Virable\_Cost\_Heap(virable\_cost,virable\_cost.size());
149. *//Print(virable\_cost);*
150. for(int k = 0; k<vexnum-1; k++){
151. *//Print(virable\_cost);*
152. int next\_index = Get\_Min\_Index(virable\_cost);
153. *//std::cout<<next\_index<<" ";*
154. if(apparent){
155. std::cout<<MG.vexs[next\_index].vex\_name<<MG.vexs[const\_cost[next\_index].adjvex\_index].vex\_name<<" "
156. <<MG.arcs[next\_index][const\_cost[next\_index].adjvex\_index]<<std::endl;
157. }
158. *//const\_cost[next\_index].adjvex\_index = next\_index;*
160. for(int i = 0; i<virable\_cost.size();i++){
161. if(virable\_cost[i].lowcost > MG.arcs[virable\_cost[i].real\_index][next\_index]){
162. const\_cost[virable\_cost[i].real\_index].adjvex\_index = next\_index;
163. virable\_cost[i].lowcost = MG.arcs[virable\_cost[i].real\_index][next\_index];
164. virable\_cost[i].adjvex\_index = next\_index;
165. Heapify\_Float\_Up(virable\_cost,i);
166. }
167. }
168. }
169. if(apparent) std::cout<<std::endl;
170. }
171. protected:
172. inline void Heapify\_Float\_Up(std::deque<Cost> &virable\_cost,int current){
173. while(current > 0){
174. int parent = (current-1)/2;
175. if(virable\_cost[parent].lowcost>virable\_cost[current].lowcost){
176. Swap(virable\_cost[parent],virable\_cost[current]);
177. }
178. else{
179. break;
180. }
181. }
182. }
183. inline void Swap(Cost &cost1,Cost&cost2){
184. Cost temp;
185. temp = cost1;
186. cost1 = cost2;
187. cost2 = temp;
188. }
189. inline void Heapify(std::deque<Cost> &virable\_cost,int size,int root){
190. int small = root;
191. int lchild = root\*2+1;
192. int rchild = root\*2+2;
193. if(lchild < size && virable\_cost[small].lowcost>virable\_cost[lchild].lowcost){
194. small = lchild;
195. }
196. if(rchild < size && virable\_cost[small].lowcost>virable\_cost[rchild].lowcost){
197. small = rchild;
198. }
199. if(small !=root){
200. Swap(virable\_cost[small],virable\_cost[root]);
201. Heapify(virable\_cost,size,small);
202. }
203. }
204. inline void Bulid\_Virable\_Cost\_Heap(std::deque<Cost> &virable\_cost,int size){
205. for(int i = size/2-1;i>=0;i--){
206. Heapify(virable\_cost,size,i);
207. }
208. }
209. inline int Get\_Min\_Index(std::deque<Cost> &virable\_cost){
210. int size = virable\_cost.size();
211. if(size<=0) return -1;
212. int real\_index = virable\_cost[0].real\_index;
213. virable\_cost.pop\_front();
214. return real\_index;
215. }
216. inline void Sort\_Min\_Edge(){
217. edge temp;
218. for(int i = 0;i <arcnum-1; i++){
219. for(int j = 0; j<arcnum-1-i; j++){
220. if(Edge[j].lowcost>Edge[j+1].lowcost){
221. temp = Edge[j];
222. Edge[j] = Edge[j+1];
223. Edge[j+1] = temp;
224. }
225. }
226. }
227. }
228. inline int Select\_Min(std::vector<Cost>&cost)const{
229. int index;
230. for(int i = 0; i<vexnum; i++){
231. if(cost[i].lowcost!=0){
232. index = i;
233. break;
234. }
235. }
236. for(int j = 0; j<vexnum; j++){
237. if(cost[j].lowcost!=0 && cost[j].lowcost < cost[index].lowcost){
238. index = j;
239. }
240. }
241. return index;
242. }
244. inline int Locate\_Vex(char vex){
245. for(int i = 0; i<vexnum;i++){
246. if(MG.vexs[i].vex\_name == vex) return i;
247. }
248. return -1;
249. }
250. inline void Inital\_Mat\_Graph(){
251. MG.arcs.resize(vexnum, std::vector<int>(vexnum,MVINT));
252. }
253. inline void Display\_Mat\_Graph(){
254. std::cout<<"1.Display Mat\_Graph"<<std::endl;
255. for(int i = 0; i<vexnum;i++){
256. std::cout<<MG.vexs[i].vex\_name<<" ";
257. }
258. std::cout<<std::endl;
259. for(int i = 0; i<vexnum; i++){
260. for(int j = 0; j<vexnum; j++){
261. std::cout<<MG.arcs[i][j]<<" ";
262. }
263. std::cout<<std::endl;
264. }
265. }
266. inline void Read\_File\_Generate\_Mat(){
267. std::ifstream file("E:\\HomeWork\\lab3\\GRAPH.txt");
268. if(!file.is\_open()){
269. std::cout<<"Fail to open the GRAPH.txt";
270. return ;
271. }
272. std::string line;
273. if(std::getline(file,line)){
274. std::stringstream ss(line);
275. char vertex;
276. while(ss>>vertex){
277. MG.vexs.push\_back({vertex,false,false});
278. }
279. }
280. std::string edge;
281. int weight;
282. while(file>>edge>>weight){
283. char v1 = edge[0];
284. char v2 = edge[1];
285. int pos1 = Locate\_Vex(v1);
286. int pos2 = Locate\_Vex(v2);
287. if(pos1!=-1 && pos2!=-1){
288. MG.arcs[pos1][pos2] = weight ;
289. MG.arcs[pos2][pos1] = weight ;
290. }
291. }
292. file.close();
293. }
294. };
295. int main(void)
296. {
297. Mat\_Graph\_Operator MG(10,19);
298. MG.Mat\_Graph\_Control();
299. return 0;
300. }
301. *//----------------------------------Heavy\_Payload----------------------------------------------------------//*
302. #include "ulity.h"
303. constexpr int MVINT = 10000;
304. class Mat\_Graph\_Operator{
305. public:
306. typedef struct{
307. int vex\_name;
308. int connected;
309. int visited;
310. }data;
311. typedef struct{
312. std::vector<data>vexs;
313. std::vector<std::vector<int>>arcs;
314. }Mat\_Graph;
315. typedef struct{
316. int real\_index;
317. int adjvex\_index;
318. int lowcost;
319. }Cost;
320. typedef struct{
321. int front\_index;
322. int rear\_index;
323. int lowcost;
324. }edge;
325. Mat\_Graph MG;
326. std::vector<Mat\_Graph\_Operator::Cost>cost;
327. std::vector<edge>Edge;
328. int vexnum,arcnum;
329. Mat\_Graph\_Operator(int \_vexnum,int \_arcnum):vexnum(\_vexnum),arcnum(\_arcnum){
330. Inital\_Mat\_Graph();
331. Read\_File\_Generate\_Mat();
332. }
333. virtual ~Mat\_Graph\_Operator(){}
334. void Kruskal\_Min\_Span\_Tree(){
335. std::cout<<"3.The Kruskal\_Min\_Span\_Tree is :"<<std::endl;
336. int sum\_distance = 0;
337. for(int i = 0; i<vexnum ;i++){
338. MG.vexs[i].connected = i;
339. }
340. for(int i = 0; i<vexnum; i++){
341. for(int j = 0; j<=i; j++){
342. if(MG.arcs[i][j] != MVINT){
343. Edge.push\_back({i,j,MG.arcs[i][j]});
344. }
345. }
346. }
347. Sort\_Min\_Edge();
348. *// for(int i = 0;i<arcnum;i++){*
349. *//     std::cout<<Edge[i].lowcost<<" ";*
350. *// }*
351. *// std::cout<<std::endl;*
352. int valid = 0;
353. for(int i = 0; i<arcnum; i++){
355. int v1 = Edge[i].front\_index;
356. int v2 = Edge[i].rear\_index;
357. int vs1 = MG.vexs[v1].connected;
358. int vs2 = MG.vexs[v2].connected;
360. if(vs1!=vs2){
361. sum\_distance += Edge[i].lowcost;
362. std::cout<<MG.vexs[v1].vex\_name<<"<->"<<MG.vexs[v2].vex\_name<<" "<<Edge[i].lowcost<<std::endl;
363. MG.vexs[v2].connected = vs1;
364. valid ++;
365. if(valid == vexnum-1){
366. *//break;*
367. }
368. for(int j = 0; j<vexnum; j++){
369. if(MG.vexs[j].connected == vs2) MG.vexs[j].connected = vs1;
370. }
371. }
372. }
373. std::cout<<"Kruskal\_The total distance is :"<<sum\_distance<<std::endl;
374. }
376. void Prime\_Min\_Span\_Tree(bool apparent){
377. if(apparent){
378. std::cout<<"1.The Prime\_Min\_Span\_Tree is :"<<std::endl;
379. }
380. int sum\_distance = 0;
381. cost.resize(vexnum);
382. int center = 0;
383. for(int i = 0; i<vexnum; i++){
384. cost[i].adjvex\_index = center;
385. cost[i].real\_index = i;
386. cost[i].lowcost = center!=i ? MG.arcs[center][i]:0;
387. }
388. for(int k = 0; k<vexnum-1; k++){
389. int next\_index = Select\_Min(cost);
390. sum\_distance += MG.arcs[next\_index][cost[next\_index].adjvex\_index];
391. if(apparent){
392. std::cout<<MG.vexs[next\_index].vex\_name<<"<->"<<MG.vexs[cost[next\_index].adjvex\_index].vex\_name<<" "
393. <<MG.arcs[next\_index][cost[next\_index].adjvex\_index]<<std::endl;
394. }
395. cost[next\_index].lowcost = 0;
396. cost[next\_index].adjvex\_index = next\_index;
397. for(int i = 0; i<vexnum; i++){
398. if(cost[i].lowcost!=0 && cost[i].lowcost > MG.arcs[i][next\_index]){
399. cost[i].lowcost = MG.arcs[i][next\_index];
400. cost[i].adjvex\_index = next\_index;
401. }
402. }
403. }
404. if(apparent){
405. std::cout<<"Prime\_The total distance is :"<<sum\_distance<<std::endl;
406. std::cout<<std::endl;
407. }
408. }
409. void Measure\_Heapify\_1\_Prime\_Min\_Span\_Tree() {
410. auto start = std::chrono::high\_resolution\_clock::now();
411. for(int i = 0; i< 10\*MVINT;i++) Heapify\_1\_Prime\_Min\_Span\_Tree(false);
412. auto end = std::chrono::high\_resolution\_clock::now();
413. std::chrono::duration<double> duration = end - start;
414. std::cout << "Heapify\_1\_Prime\_Min\_Span\_Tree took " << duration.count() << " seconds.\n";
415. }
416. void Measure\_Heapify\_2\_Prime\_Min\_Span\_Tree() {
417. auto start = std::chrono::high\_resolution\_clock::now();
418. for(int i = 0; i< 10\*MVINT;i++) Heapify\_2\_Prime\_Min\_Span\_Tree(false);
419. auto end = std::chrono::high\_resolution\_clock::now();
420. std::chrono::duration<double> duration = end - start;
421. std::cout << "Heapify\_2\_Prime\_Min\_Span\_Tree took " << duration.count() << " seconds.\n";
422. }
423. void Measure\_Prime\_Min\_Span\_Tree() {
424. auto start = std::chrono::high\_resolution\_clock::now();
425. for(int i = 0; i< 10\*MVINT;i++) Prime\_Min\_Span\_Tree(false);
426. auto end = std::chrono::high\_resolution\_clock::now();
427. std::chrono::duration<double> duration = end - start;
428. std::cout << "Prime\_Min\_Span\_Tree took " << duration.count() << " seconds.\n";
429. }
430. virtual void Mat\_Graph\_Control(){
431. *//Display\_Mat\_Graph();*
432. Prime\_Min\_Span\_Tree(true);
433. Heapify\_1\_Prime\_Min\_Span\_Tree(true);
434. Heapify\_2\_Prime\_Min\_Span\_Tree(true);
435. Kruskal\_Min\_Span\_Tree();
436. std::cout<<"The runtime of 10000 times"<<std::endl;
437. std::cout<<"1.Naive Prime:"<<std::endl;
438. Measure\_Prime\_Min\_Span\_Tree();
439. *//std::cout<<"2.Static Heapify Prime,the constant num is the large:"<<std::endl;*
440. *//Measure\_Heapify\_1\_Prime\_Min\_Span\_Tree();*
441. std::cout<<"3.Dynamic Heapify Prime,the final version:"<<std::endl;
442. Measure\_Heapify\_2\_Prime\_Min\_Span\_Tree();
443. }
444. void Heapify\_1\_Prime\_Min\_Span\_Tree(bool apparent){
445. if(apparent){
446. std::cout<<"2.The Heapify\_1\_Prime\_Min\_Span\_Tree is :"<<std::endl;
447. }
448. *//std::cout<<MG.vexs[0].vex\_name<<" ";*
449. std::vector<Cost>cost;
450. int sum\_distance = 0;
451. int center = 0;
452. cost.resize(vexnum);
453. for(int i = 0; i<vexnum; i++){
454. cost[i].adjvex\_index = center;
455. cost[i].real\_index = i;
456. cost[i].lowcost = center!=i ? MG.arcs[center][i]:0;
457. }
458. for(int k = 0; k<vexnum-1; k++){
459. int next\_index = Get\_Min\_Index\_1(cost,cost.size());
460. sum\_distance += MG.arcs[next\_index][cost[next\_index].adjvex\_index];
461. if(apparent){
462. std::cout<<MG.vexs[next\_index].vex\_name<<"<->"<<MG.vexs[cost[next\_index].adjvex\_index].vex\_name<<" "
463. <<MG.arcs[next\_index][cost[next\_index].adjvex\_index]<<std::endl;
464. }
465. cost[next\_index].lowcost = 0;
466. cost[next\_index].adjvex\_index = next\_index;
467. for(int i = 0; i<vexnum; i++){
468. if(cost[i].lowcost!=0 && cost[i].lowcost > MG.arcs[i][next\_index]){
469. cost[i].lowcost = MG.arcs[i][next\_index];
470. cost[i].adjvex\_index = next\_index;
471. }
472. }
473. }
474. if(apparent){
475. std::cout<<"Prime\_The total distance is :"<<sum\_distance<<std::endl;
476. std::cout<<std::endl;
477. }
478. }
479. inline void Print(std::deque<Cost> &virable\_cost){
480. for(int i = 0;i<virable\_cost.size();i++){
481. std::cout<<virable\_cost[i].lowcost<<" "<<virable\_cost[i].real\_index+1<<std::endl;
482. }
483. std::cout<<"---------------------------------"<<std::endl;
484. }
485. void Insert(std::deque<Cost> &virable\_cost,Cost value){
486. virable\_cost.push\_back(value);
487. Heapify\_Float\_Up\_2(virable\_cost,virable\_cost.size()-1);
488. }
489. void Heapify\_2\_Prime\_Min\_Span\_Tree(bool apparent){
490. if(apparent){
491. std::cout<<"3.The Heapify\_2\_Prime\_Min\_Span\_Tree is :"<<std::endl;
492. }
493. std::deque<Mat\_Graph\_Operator::Cost>virable\_cost;
494. std::vector<Mat\_Graph\_Operator::Cost>const\_cost(vexnum);
495. int center = 0,sum\_distance = 0,existed\_vex\_num = 0;
496. for(int i = 0; i<vexnum; i++){
497. const\_cost[i].adjvex\_index = center;
498. const\_cost[i].real\_index = i;
499. const\_cost[i].lowcost = MVINT;
500. }
501. Insert(virable\_cost,const\_cost[0]);
502. *//MG.vexs[0].visited = true;*
503. while(!virable\_cost.empty()){
504. if(existed\_vex\_num == vexnum) break;
505. int min\_index = Get\_Min\_Index\_2(virable\_cost);
506. if(MG.vexs[min\_index].visited) continue;
507. existed\_vex\_num++;
508. MG.vexs[min\_index].visited = true;
509. if(apparent && min\_index != const\_cost[min\_index].adjvex\_index){
510. sum\_distance += MG.arcs[min\_index][const\_cost[min\_index].adjvex\_index];
511. std::cout<<MG.vexs[min\_index].vex\_name<<"<->"<<MG.vexs[const\_cost[min\_index].adjvex\_index].vex\_name<<" "
512. <<MG.arcs[min\_index][const\_cost[min\_index].adjvex\_index]<<std::endl;
513. }
514. for(int i = 0; i <vexnum;i++){
515. if(!MG.vexs[i].visited && const\_cost[i].lowcost > MG.arcs[min\_index][i]){
516. const\_cost[i].adjvex\_index = min\_index;
517. const\_cost[i].lowcost = MG.arcs[min\_index][i];
518. Insert(virable\_cost,const\_cost[i]);
519. }
520. }
521. }
522. if(apparent) {
523. std::cout<<"The total distance is :"<<sum\_distance<<std::endl;
524. std::cout<<std::endl;
525. }
526. }
527. protected:
528. *//----------------------------------下面是Heapify\_1的辅助函数-------------------------------------//*
529. inline void Heapify(std::vector<Cost> &cost,int size,int root){
530. int small = root;
531. int lchild = root\*2+1;
532. int rchild = root\*2+2;
533. if(lchild < size && cost[small].lowcost>cost[lchild].lowcost){
534. small = lchild;
535. }
536. if(rchild < size && cost[small].lowcost>cost[rchild].lowcost){
537. small = rchild;
538. }
539. if(small !=root){
540. Swap(cost[small],cost[root]);
541. Heapify(cost,size,small);
542. }
543. }
544. inline void Bulid\_Cost\_Heap(std::vector<Cost>&cost,int size){
545. for(int i = size/2-1;i>=0;i--){
546. Heapify(cost,size,i);
547. }
548. }
549. inline  int Get\_Min\_Index\_1(std::vector<Cost> cost,int size){
550. Bulid\_Cost\_Heap(cost,size);
551. if(size<=0) return -1;
552. if(size == 1){
553. size--;
554. return cost[0].real\_index;
555. }
556. while(cost[0].lowcost == 0){
557. cost[0] = cost[size-1];
558. cost.pop\_back();
559. size--;
560. Heapify(cost,size,0);
561. }
562. int index = cost[0].real\_index;
563. cost[0] = cost[size-1];
564. cost.pop\_back();
565. size--;
566. Heapify(cost,size,0);
567. return index;
568. }
569. inline void Swap(Cost &cost1,Cost&cost2){
570. Cost temp;
571. temp = cost1;
572. cost1 = cost2;
573. cost2 = temp;
574. }
575. *//------------------------------------上面是Heapify 1的辅助将函数，下面是Heapify\_2的辅助函数------------------//*
576. inline void Heapify\_Float\_Up\_2(std::deque<Cost> &virable\_cost,int current){
578. while(current > 0){
579. int parent = (current-1)/2;
580. if(virable\_cost[parent].lowcost>virable\_cost[current].lowcost){
581. Swap(virable\_cost[parent],virable\_cost[current]);
582. current = parent;
583. }
584. else{
585. break;
586. }
587. }
588. }
589. inline void Heapify(std::deque<Cost> &virable\_cost,int size,int root){
590. int small = root;
591. int lchild = root\*2+1;
592. int rchild = root\*2+2;
593. if(lchild < size && virable\_cost[small].lowcost>virable\_cost[lchild].lowcost){
594. small = lchild;
595. }
596. if(rchild < size && virable\_cost[small].lowcost>virable\_cost[rchild].lowcost){
597. small = rchild;
598. }
599. if(small !=root){
600. Swap(virable\_cost[small],virable\_cost[root]);
601. Heapify(virable\_cost,size,small);
602. }
603. }
604. inline void Bulid\_Virable\_Cost\_Heap(std::deque<Cost> &virable\_cost,int size){
605. for(int i = size/2-1;i>=0;i--){
606. Heapify(virable\_cost,size,i);
607. }
608. }
609. inline int Get\_Min\_Index\_2(std::deque<Cost> &virable\_cost){
610. int size = virable\_cost.size();
611. if(size<=0) return -1;
612. int real\_index = virable\_cost[0].real\_index;
613. Swap(virable\_cost[0],virable\_cost[virable\_cost.size()-1]);
614. virable\_cost.pop\_back();
615. Heapify(virable\_cost, virable\_cost.size(), 0);
616. return real\_index;
617. }
618. *//----------------------------------------other---------------------------------//*
619. *// inline int Locate\_Vex(int vex\_name){*
620. *//   return vex\_name-1;*
621. *// }*
622. inline int Select\_Min(std::vector<Cost>&cost)const{
623. int index;
624. for(int i = 0; i<vexnum; i++){
625. if(cost[i].lowcost!=0){
626. index = i;
627. break;
628. }
629. }
630. for(int j = 0; j<vexnum; j++){
631. if(cost[j].lowcost!=0 && cost[j].lowcost < cost[index].lowcost){
632. index = j;
633. }
634. }
635. return index;
636. }
637. inline void Sort\_Min\_Edge(){
638. edge temp;
639. for(int i = 0;i <arcnum-1; i++){
640. for(int j = 0; j<arcnum-1-i; j++){
641. if(Edge[j].lowcost>Edge[j+1].lowcost){
642. temp = Edge[j];
643. Edge[j] = Edge[j+1];
644. Edge[j+1] = temp;
645. }
646. }
647. }
648. }
649. inline void Inital\_Mat\_Graph(){
650. MG.arcs.resize(vexnum, std::vector<int>(vexnum,MVINT));
651. }
652. inline void Display\_Mat\_Graph(){
653. std::cout<<"1.Display Mat\_Graph"<<std::endl;
654. for(int i = 0; i<vexnum;i++){
655. std::cout<<MG.vexs[i].vex\_name<<" ";
656. }
657. std::cout<<std::endl;
658. for(int i = 0; i<vexnum; i++){
659. for(int j = 0; j<vexnum; j++){
660. std::cout<<MG.arcs[i][j]<<" ";
661. }
662. std::cout<<std::endl;
663. }
664. }
665. inline void Read\_File\_Generate\_Mat(){
666. std::ifstream file("E:\\HomeWork\\lab3\\cleaned\_edges.txt");
667. if(!file.is\_open()){
668. std::cout<<"Fail to open the undirected\_graph.txt";
669. return ;
670. }
671. for(int i = 0;i<vexnum;i++){
672. MG.vexs.push\_back({i+1,false,false});
673. }
675. int edge1,edge2,weight;
676. while(file>>edge1>>edge2>>weight){
677. MG.arcs[edge1-1][edge2-1] = weight ;
678. MG.arcs[edge2-1][edge1-1] = weight ;
679. }
680. file.close();
681. }
682. };
683. int main(void)
684. {
685. Mat\_Graph\_Operator MG(50,244);
686. MG.Mat\_Graph\_Control();
687. return 0;
688. }