山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202000130198 | 姓名： 隋春雨 | | 班级： 20.4 |
| 实验题目：图 | | | |
| 实验学时：2 | | 实验日期： 2021-12-16 | |
| 实验目的：  1、掌握图的基本概念，图的描述方法；图上的操作方法实现。  2、掌握图结构的应用。 | | | |
| 软件开发环境：  CLION2020 | | | |
| 1. 实验内容   题目描述：  创建无向图类，存储结构使用邻接链表，提供操作：插入一条边，删除一条边，BFS，DFS。  输入输出格式：  输入：  第一行四个整数n，m，s，t。n (10≤n≤100000) 代表图中点的个数，m (10≤m≤200000) 代表接下来共有m个操作，s代表起始点，t代表终点。  接下来m行，每行代表一次插入或删除边的操作，操作格式为：  0 u v 在点u和v之间增加一条边；  1 u v 删除点u和v之间的边。  输出：  第一行输出图中有多少个连通分量；  第二行输出所有连通子图中最小点的编号（升序），编号间用空格分隔；  第三行输出从s点开始的dfs序列长度；  第四行输出从s点开始的字典序最小的dfs序列；  第五行输出从t点开始的bfs序列的长度；  第六行输出从t点开始字典序最小的bfs序列；  第七行输出从s点到t点的最短路径，若是不存在路径则输出-1。   1. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法） 2. 数据结构：图和临接链表 3. 临接链表：我们维护一个有序链表。通过这个链表，访问其结点，使用有序链表能够提高时间性能。 4. 图：使用图这个数据结构，进行dfs、bfs、查询联通分量等操作。图内的数据成员有chain数组和结点数量，提供的操作如下：     图的实例：     1. 算法： 2. 计算连通分量：我们遍历每一个结点，并维护一个jud数组，对每一个未被访问的结点进行bfs访问。每一次在for循环里压入一个新的结点的时候，cnt自增，表示有一个新的连通分量。同时，对这个结点所在的联通图进行搜寻，把与其临接的每个结点都标记为不可访问即可。代码如下：       保存每个联通分量的最小点：在for循环中进行保存即可。   1. Dfs操作：我们使用一个共有接口函数分别调用两个操作。代码如下：      1. 输出dfs序列：对输入的变量的临接的每一个未相邻的结点进行访问并进行dfs，同时使用一个数组进行保存。代码如下：      1. 计算两点之间距离：由于这个是无权重图，所以我们可以使用bfs直接进行计算。我们使用一个队列来进行完成该函数，每次从队列中取出来队头，判断一下是否为我们想要到达的顶点，如果是，直接退出。否则，遍历临接的每一个没有被访问过的点，压入队列。若最后退出的时候也没有访问到，返回-1   代码如下：       1. 测试结果（测试输入，测试输出） 2. a题   输入：    输出：    提交OJ的结果：     1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径） 2. 对于复杂的函数，我们可以将其分解成两个protected函数，然后通过public函数调用。这样能够减少耦合度。例子如下：      1. 要看好数据范围，比如下标是从0开始还是1开始很重要，如果我们数组开小了可能会导致我们RE。 2. 要保持好数据的一致性，比如本次实验中初始化的时候，外面给的是n+1，但是里面在计算size的时候，也自增了 ，这就导致这个数据结构的size不对。最后会导致数组下标访问越界。 3. 在提交oj的时候要把自己的测试删除，比如多输出了一个换行可能就会导致全部的判错。 4. 对于BOOL数组的初始化，绝对不能想当然，每一次的定义都要对其进行初始化。在本次的实验中，因为bool数组没有初始化导致debug了很久。        1. 对于一个只有两个私有成员的struct，我们可以直接使用pair来存储。简化了我们的代码量。同时对于返回多个值的函数，我们只能使用引用来返回。这也是为什么兴起了Go语言的一个原因。 2. 对于height等操作的计算，一定要特判是否合法。因为对于root结点来说，它的父节点是自身，不能直接增加。      1. 我们在写循环条件判断的时候，对于短路情况的判断一定要慎重，我们对于指针的使用一定要先判断是否为空，在进行取值操作，如下：      1. 对于维护私有变量的时候，要特殊情况特殊判：比如说删除结点的时候，要考虑这个是不是头结点，如果是，那么更新私有变量。如下：     (10)要注意私有成员的更新，public函数知道自己所处的状态都是靠着私有成员才知道的，如果没有及时更新，那数据就成了垃圾数据，没有任何意义。我在写实验的时候，也经历过没更新导致的Bug，最终debug查出来，就是下面这个：     1. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释） 2. #include <iostream> 3. #include <queue> 4. using namespace std; 5. template<class T> 6. class Graph 7. { 8. chain<T>\* ptr;*//指针数组* 9. int sizeofNode;*//结点数量* 10. public: 11. Graph(int n) 12. { 13. ptr = new chain<T>[n + 1];*//从1开始* 14. sizeofNode = n; 15. } 16. virtual ~Graph(); 17. void insert(int i, int j);*//插入* 18. void erase(int i, int j);*//删除* 19. pair<int, vector<int>> Connected\_Component()const;*//联通分量，返回个数和联通集* 20. pair<int, vector<int>>dfs(int s);*//返回dfs序列长度和字典序最小的dfs序列* 21. pair<int, vector<int>>bfs(int s)const;*//返回bfs序列和字典序最小的dfs序列* 22. int distance(int i, int j)const;*//返回i和j之间的长度* 23. protected: 24. void dfs\_sequence(int s, vector<int>& v, bool\* jud)const; 25. int dfs\_length(int s, bool\* jud) const; 26. }; 27. template<class T> 28. void Graph<T>::insert(int i, int j) 29. { 30. ptr[i].insert(j); 31. ptr[j].insert(i); 32. } 33. template<class T> 34. void Graph<T>::erase(int i, int j) 35. { 36. ptr[i].erase(j); 37. ptr[j].erase(i); 38. } 39. template<class T> 40. pair<int, vector<int>> Graph<T>::Connected\_Component() const 41. { 42. int cnt = 0; 43. queue<int>q; 44. vector<int>sequence; 45. bool\* jud = new bool[1 + sizeofNode]();*//初始化* 46. for (int i = 1; i <= sizeofNode; i++) 47. { 48. if (!jud[i])*//没有被访问过* 49. { 50. sequence.push\_back(i); 51. cnt++; 52. q.push(i); 53. jud[i] = true; 54. while (!q.empty()) 55. { 56. int front = q.front(); 57. q.pop(); 58. *//将临接的所有顶点压入* 59. for (chainNode<T>\* p = ptr[front].getFirstNode(); p; p = p->next) 60. { 61. if (!jud[p->element]) 62. { 63. q.push(p->element); 64. jud[p->element] = true; 65. } 66. } 67. } 68. } 69. } 70. return pair<int, vector<int>>(cnt, sequence); 71. } 72. template<class T> 73. int Graph<T>::dfs\_length(int s, bool\* jud) const 74. { 75. int length = 1; 76. for (chainNode<T>\* p = ptr[s].getFirstNode(); p; p = p->next) 77. { 78. if (!jud[p->element]) 79. { 80. jud[p->element] = true; 81. length += dfs\_length(p->element, jud);*//递归调用* 82. } 83. } 84. return length; 85. } 86. template<class T> 87. void Graph<T>::dfs\_sequence(int s, vector<int>& v, bool\* jud) const 88. { 89. for (chainNode<T>\* p = ptr[s].getFirstNode(); p; p = p->next) 90. { 91. if (!jud[p->element]) 92. { 93. jud[p->element] = true; 94. v.push\_back(p->element); 95. dfs\_sequence(p->element, v, jud);*//递归调用* 96. } 97. } 98. } 99. template<class T> 100. pair<int, vector<int>> Graph<T>::dfs(int s) 101. { 102. bool\* jud = new bool[sizeofNode + 1]();*//初始化* 103. jud[s] = true; 104. int length = dfs\_length(s, jud); 105. for (int i = 0; i <= sizeofNode; i++) 106. { 107. jud[i] = false;*//标记为false* 108. } 109. vector<int>ans; 110. ans.push\_back(s); 111. jud[s] = true; 112. dfs\_sequence(s, ans, jud); 113. return pair<int, vector<int>>(length, ans); 114. } 115. template<class T> 116. pair<int, vector<int>> Graph<T>::bfs(int s) const 117. { 118. int length = 1; 119. bool\* jud = new bool[sizeofNode + 1](); 120. vector<int>sequence;*//保存序列* 121. sequence.push\_back(s); 122. queue<int>q; 123. q.push(s); 124. jud[s] = true; 125. while (!q.empty()) 126. { 127. int front = q.front(); 128. q.pop(); 129. *//压入临接顶点* 130. for (chainNode<T>\* p = ptr[front].getFirstNode(); p; p = p->next) 131. { 132. if (!jud[p->element]) 133. { 134. jud[p->element] = true; 135. length += 1;*//更新* 136. sequence.push\_back(p->element); 137. q.push(p->element); 138. } 139. } 140. } 141. return { length, sequence };*//返回* 142. } 143. template<class T> 144. int Graph<T>::distance(int i, int j) const*//返回i和j之间的距离* 145. { 146. int length = 0; 147. bool\* jud = new bool[sizeofNode + 1](); 148. queue<int>q; 149. q.push(i);*//压入* 150. jud[i] = true; 151. while (!q.empty()) 152. { 153. int front = q.front(); 154. q.pop(); 155. if (front == j) 156. { 157. return length; 158. } 159. else 160. { 161. length++; 162. for (chainNode<T>\* p = ptr[front].getFirstNode(); p; p = p->next) 163. { 164. if (!jud[p->element]) 165. { 166. q.push(p->element); 167. jud[p->element] = true; 168. } 169. } 170. } 171. } 172. return -1; 173. } 174. template<class T> 175. Graph<T>::~Graph() { 176. } 177. template <class T> 178. struct chainNode 179. {*//数据成员* 180. T element; 181. chainNode<T>\* next; 182. *//方法（三种）* 183. chainNode() {} 184. chainNode(const T& element) { this->element = element; } 185. chainNode(const T& element, chainNode<T>\* next) { this->element = element; this->next = next; } 186. }; 187. template <class T> 188. class chain 189. { 190. protected: 191. chainNode<T>\* firstNode;*//指向链表中第一个节点的指针* 192. int listSize;*//线性表的元素个数* 193. public: 194. *//构造函数、析构函数* 195. chain(int initialcapacity = 10); 196. ~chain(); 197. void erase(int& theElement); 198. void insert(const T& theElement); 199. chainNode<T>\* getFirstNode() { return firstNode; } 200. }; 201. template <class T> 202. chain<T>::chain(int initialcapacity) 203. {*//构造函数* 204. firstNode = nullptr; 205. listSize = 0; 206. } 207. template <class T> 208. chain<T>::~chain() 209. {*//链表的析构函数，重复删除链表中的首节点直到全部删除* 210. while (firstNode != nullptr) 211. {*//删除首节点* 212. chainNode<T>\* nextNode = firstNode->next; 213. delete firstNode; 214. firstNode = nextNode; 215. } 216. } 217. template <class T> 218. void chain<T>::erase(int& theElement) 219. {*//删除元素，保证有序* 220. chainNode<T>\* p = firstNode, \* tp = nullptr; 221. while (p != nullptr && p->element < theElement) 222. { *//查找删除位置* 223. tp = p; 224. p = p->next; 225. } 226. if (p != nullptr && p->element == theElement) 227. { 228. if (tp == nullptr) 229. firstNode = p->next; 230. else 231. tp->next = p->next; 232. listSize--; 233. delete p; 234. } 235. } 236. template <class T> 237. void chain<T>::insert(const T& theElement) 238. {*//插入元素，保证有序* 239. chainNode<T>\* p = firstNode, \* tp = nullptr;*//儿子* 240. while (p != nullptr && p->element < theElement) 241. { *//查找插入位置* 242. tp = p; 243. p = p->next; 244. } 245. if (p != nullptr && p->element == theElement) 246. { *//元素已经存在，则直接返回* 247. return; 248. } 249. *//插入* 250. chainNode<T>\* newNode = new chainNode<T>(theElement, p); 251. if (tp == nullptr) 252. firstNode = newNode; 253. else 254. tp->next = newNode; 255. listSize++; 256. } 257. int main() 258. { 259. int n, m, s, t; 260. cin >> n >> m >> s >> t; 261. Graph<int>g(n); 262. for (int i = 0; i < m; i++) 263. { 264. int flag, from, to; 265. cin >> flag >> from >> to; 266. if (flag) 267. { 268. g.erase(from, to);*//删除* 269. } 270. else 271. { 272. g.insert(from, to);*//插入* 273. } 274. } 275. pair<int, vector<int>>component = g.Connected\_Component();*//联通分量* 276. cout << component.first << endl;*//联通分量个数* 277. for (int i = 0; i < component.second.size(); i++) 278. { 279. cout << component.second[i] << " ";*//所有连通子图中最小点的编号* 280. } 281. cout << endl; 282. *//dfs* 283. pair<int, vector<int>>dfs = g.dfs(s); 284. cout << dfs.first << endl;*//dfs序列长度* 285. for (int i = 0; i < dfs.second.size(); i++) 286. { 287. cout << dfs.second[i] << " ";*//字典序最小的dfs序列* 288. } 289. cout << endl; 290. *//bfs* 291. pair<int, vector<int>>bfs = g.bfs(t);*//bfs序列长度* 292. cout << bfs.first << endl; 293. for (int i = 0; i < bfs.second.size(); i++) 294. { 295. cout << bfs.second[i] << " ";*//字典序最小的bfs序列* 296. } 297. cout << endl; 298. cout << g.distance(s, t); 299. return 0; 300. } | | | |
|  | | | |