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：202300130183 | 姓名： 宋浩宇 | | 班级：23级人工智能班 |
| 实验题目：2024数据结构--数据/智能 实验12 图 | | | |
| 实验学时：2 | | 实验日期：2024/12/4 | |
| 实验目的：  1.掌握图结构的定义与实现；  2.掌握图结构的使用。 | | | |
| 软件开发工具：   1. visual studio code 2022（使用C/C++、C/C++ Extension Pack、C/C++ Themes插件） 2. mingw64工具包 | | | |
| 1. 实验内容   完成2024数据结构--数据/智能 实验12 图 A 图论基础  1.实现图类，包含功能：删除边，插入边  2.完成实验的七个操作（七行输出）   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）   本题使用的数据结构是无向无权图。题目要求我们实现的七个操作分别是：  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  简单来说，题目需要我们实现一个DFS、一个BFS以及一个字典序优先的DFS、一个字典序优先的BFS，一个获取连通分量的函数，一个获取连通子图中最小编号的函数，一个搜索最短路径的函数，因为在实现前两行要求的输出的时候实际使用到了BFS或者DFS的，因此我们先来描述BFS和DFS的实现方式。  首先需要声明的是，因为题目的第一行、第二行、第七行也需要用到BFS和DFS其中一种，考虑到这两种搜索算法的时间复杂度在进行无目标搜索时是一样的且在无向无权图中Dijkstra算法和BFS是一致的，我们选择用递归的方式实现DFS，用循环的方式实现BFS，这样可以简化代码，且保证BFS高效运行。  Ⅰ.关于BFS：  因为我们是使用循环的方式来实现，故有如下内容。  在此处我们需要使用到队列数据结构。  我们创建一个队列，队列里存储的是我们将要处理的节点。  我们在创建一个bool数组，用于存储哪些节点是我们已经搜索到的。  我们在每一个循环里取出队列中的第一个节点，并对这个节点进行处理（即按照传入的函数指针进行处理），再对这个节点的字节点进行处理，把这个节点的子节点中还未搜索到的节点压入队列，并将这些节点标记为已经被搜索到。重复这个过程直到队列为空，则此时我们的无目标BFS结束。  而开始BFS需要进行的操作就是把起始节点压入队列并将其标记为已经搜索到了。  II.关于DFS：  因为我们是使用递归的方式来实现，故有如下内容。  我们在每一层DFS中都需要如下参数：该层DFS处理的节点索引、每一层DFS通用的用于标记节点是否被搜索到的bool数组、（可选）处理节点使用的函数指针  在每一层DFS中，我们先对该层进行处理的节点进行处理（即按照传入的函数指针进行处理），再对这个节点的子节点进行处理，先把这个节点的子节点中未被搜索到的节点标记为已被搜索到，再让这些子节点进入下一层DFS。如此下去，当递归结束的时候就是我们无目标DFS结束的时候。  III.关于字典序优先。因为题目给我们的数据都是int类型的数字，因此我们只需要在DFS和BFS时对于当前处理的子节点序列进行排序再按照排序后的结果进行递归或压入队列的操作即可。  显然，我们只需要一个简单的计数函数，将它的指针分别传入DFS和BFS就可以得到第三行和第五行的结果。  第四行和第六行的输出我们使用字典序DFS和字典序BFS再传入一个输出节点数据的函数指针即可。  而第一行要求的输出，即连通分量的个数，我们只需要开一个bool数组存储节点是否已被遍历到，然后遍历所有节点，如果这个节点没有被遍历过，就将连通分量的计数加1，再通过BFS将与这个节点所有连通的节点都设为已被遍历过即可，经过这个过程我们可以轻易的得到图的连通分量的个数。同样的思路，我们用同样的方式处理第二行的输出，只需要开一个数组存储子连通图的最小节点，并在上文中每次进行搜索过程时找出这个子连通图中的最小节点并将这个节点存储到上述数组里，把这个数组排序后再输出即可。  最后是最短路径的搜索，我们是用最简单的Dijkstra算法来进行搜索，因为Dijkstra的原理，实际的搜索过程在无向无权图中与BFS是一致的，因此我们使用BFS的方式来搜索最短路径的长度。总之这个题虽然要求较多，但实际上并不困难。   1. 测试结果（测试输入，测试输出）   测试输入：  10 20 4 5  0 6 4  0 10 3  0 4 8  0 4 10  1 4 10  0 2 1  0 5 8  0 5 2  0 10 7  0 9 6  0 9 1  0 7 1  0 8 10  0 7 5  0 8 3  0 6 7  1 6 4  1 8 3  0 7 8  0 9 2  输出为：  1  1  10  4 8 5 2 1 7 6 9 10 3  10  5 2 7 8 1 9 6 10 4 3  2   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   从测试结果来看，我们的算法成功解决了这个问题。存在的问题主要是，一个是我们递归时的内存管理存在问题，因为我们的DFS在实现的时候使用的是递归的方式，且这个递归中有大量的值传递，因此当节点和边的数量非常多的时候有可能会出现栈内存溢出的情况。另一个是我们的最短路径是使用的BFS，对于有目标搜索使用这种穷举式的搜索是效率较低的，使用启发式搜索则可以提高搜索效率。还有就是因为，为了做这个题目，我们的算法基本都是为了数值类型设计的（int、char、float等），如果使用其他类型会产生不可预见的bug，因此可拓展性较差。   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. /\*2024级数据结构--数据智能 实验12 图 A 图论基础.cpp\*/ 3. #include <iostream> 4. #include <vector> 5. #include <algorithm> 6. #include <queue> 8. **using** **namespace** std; 10. **template**<**typename** T> 11. **struct** GraphNode 12. { 13. T data; 14. //索引方案 15. **size\_t** index; 16. vector<**size\_t**> neighbors; 17. //指针方案 18. vector<GraphNode<T>\*> neighbors\_ptr; 19. **void** addNeighbor(**size\_t** neighbor\_index); 20. **void** delNeighbor(**size\_t** neighbor\_index); 21. **void** addNeighbor(GraphNode<T>\* neighbor\_ptr); 22. **void** delNeighbor(GraphNode<T>\* neighbor\_ptr); 23. GraphNode() {} 24. GraphNode(T data, **size\_t** index) : data(data), index(index) {} 25. }; 27. **template**<**typename** T> 28. **void** GraphNode<T>::addNeighbor(**size\_t** neighbor\_index) 29. { 30. **for** (auto neighbor : neighbors) 31. { 32. **if** (neighbor == neighbor\_index) 33. { 34. **return**; 35. } 36. } 37. neighbors.push\_back(neighbor\_index); 38. } 40. **template**<**typename** T> 41. **void** GraphNode<T>::delNeighbor(**size\_t** neighbor\_index) 42. { 43. **for** (auto it = neighbors.begin(); it != neighbors.end(); it++) 44. { 45. **if** (\*it == neighbor\_index) 46. { 47. neighbors.erase(it); 48. **return**; 49. } 50. } 51. } 53. **template**<**typename** T> 54. **void** GraphNode<T>::addNeighbor(GraphNode<T>\* neighbor\_ptr) 55. { 56. neighbors\_ptr.push\_back(neighbor\_ptr); 57. } 59. **template**<**typename** T> 60. **void** GraphNode<T>::delNeighbor(GraphNode<T>\* neighbor\_ptr) 61. { 62. neighbors\_ptr.erase(find(neighbors\_ptr.begin(), neighbors\_ptr.end(), neighbor\_ptr)); 63. } 65. **template**<**typename** T> 66. **void** printGraphNodeData(GraphNode<T>& node) 67. { 68. cout << node.data << " "; 69. } 71. **template**<**typename** T> 72. **void** printGraphNodeEdges(GraphNode<T>& node) 73. { 74. cout << "Neighbors: "; 75. **for** (**size\_t** i = 0; i < node.neighbors.size(); i++) 76. { 77. cout << node.neighbors[i] + 1 << " "; 78. } 79. cout << endl; 80. } 82. **template**<**typename** T> 83. **void** printGraphNode(GraphNode<T>& node) 84. { 85. cout << "Node: " << node.data << endl; 86. printGraphNodeEdges(node); 87. }   91. **template**<**typename** T> 92. **class** GraphWithoutDirection 93. { 94. **private**: 95. vector<GraphNode<T>> nodes; 96. T tem\_subgraph\_data; 97. **void** subgraphMinPoints(GraphNode<T>& node); 98. **size\_t** DFSCount; 99. **void** DFSCounter(); 100. **size\_t** BFSCount; 101. **void** BFSCounter(); 102. **public**: 104. vector<GraphNode<T>>& getNodes() { **return** nodes; } 106. GraphWithoutDirection(**size\_t** n); 107. **void** addEdge(**size\_t** u, **size\_t** v); 108. **void** delEdge(**size\_t** u, **size\_t** v); 109. **void** ConnectedComponents(); 110. **size\_t** DFSLength(**size\_t** start\_index); 111. **size\_t** BFSLength(**size\_t** start\_index); 112. **void** DepthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited, **void** (\*operation)(GraphNode<T>& node) = nullptr);//调用类外函数 113. **void** DepthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited, **void** (GraphWithoutDirection<T>::\* operation)(GraphNode<T>& node));//调用类内函数 114. **void** DepthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited, **void** (GraphWithoutDirection<T>::\* operation)());//调用类内函数 116. **void** DictionaryPrefaceDFS(**size\_t** start\_index, vector<**bool**>& visited, **void** (\*operation)(GraphNode<T>& node) = nullptr);//字典序优先深搜+调用类外函数 117. **void** DictionaryPrefaceDFS(**size\_t** start\_index, vector<**bool**>& visited, **void** (GraphWithoutDirection<T>::\* operation)(GraphNode<T>& node));//字典序优先深搜+调用类内函数 118. **void** DictionaryPrefaceDFS(**size\_t** start\_index, vector<**bool**>& visited, **void** (GraphWithoutDirection<T>::\* operation)());//字典序优先深搜+调用类内函数 120. **void** DPDFSPrint(**size\_t** start\_index); 122. **void** BreadthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited,  **void** (\*operation)(GraphNode<T>& node) = nullptr);//调用类外函数 123. **void** BreadthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited,  **void** (GraphWithoutDirection<T>::\* operation)(GraphNode<T>& node));//调用类内函数 124. **void** BreadthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited,  **void** (GraphWithoutDirection<T>::\* operation)());//调用类内函数 126. **void** DictionaryPrefaceBFS(**size\_t** start\_index, vector<**bool**>& visited,  **void** (\*operation)(GraphNode<T>& node) = nullptr);//字典序优先广搜+调用类外函数 127. **void** DictionaryPrefaceBFS(**size\_t** start\_index, vector<**bool**>& visited,  **void** (GraphWithoutDirection<T>::\* operation)(GraphNode<T>& node));//字典序优先广搜+调用类内函数 128. **void** DictionaryPrefaceBFS(**size\_t** start\_index, vector<**bool**>& visited,  **void** (GraphWithoutDirection<T>::\* operation)());//字典序优先广搜+调用类内函数 130. **void** DPBFSPrint(**size\_t** start\_index); 132. **void** AllConnectivitySubgraphs(); 134. **int** Dijkstra(**size\_t** start\_index, **size\_t** end\_index); 135. }; 137. **template**<**typename** T> 138. GraphWithoutDirection<T>::GraphWithoutDirection(**size\_t** n) 139. { 140. nodes.resize(n); 141. **for** (**size\_t** i = 0; i < n; i++) 142. { 143. nodes[i].index = i; 144. nodes[i].data = i + 1; 145. // nodes[i] = GraphNode<T>(i + 1, i); 146. } 147. } 149. **template**<**typename** T> 150. **void** GraphWithoutDirection<T>::addEdge(**size\_t** u, **size\_t** v) 151. { 152. nodes[u - 1].addNeighbor(v - 1); 153. nodes[v - 1].addNeighbor(u - 1); 154. } 156. **template**<**typename** T> 157. **void** GraphWithoutDirection<T>::delEdge(**size\_t** u, **size\_t** v) 158. { 159. nodes[u - 1].delNeighbor(v - 1); 160. nodes[v - 1].delNeighbor(u - 1); 161. } 163. **template**<**typename** T> 164. **void** GraphWithoutDirection<T>::ConnectedComponents() 165. { 166. **size\_t** count = 0; 167. vector<**bool**> visited(nodes.size(), **false**); 168. **for** (**size\_t** i = 0; i < nodes.size(); i++) 169. { 170. **if** (!visited[i]) 171. { 172. count++; 173. visited[i] = **true**; 174. DepthFirstSearch 175. (   i, 176. visited 177. // , 178. // [&](GraphNode<T>& node) 179. // { 180. //     visited[node.index] = true; 181. // } 182. ); 183. } 184. } 185. // cout << "Connected Components: " << count << endl; 186. cout << count << endl; 187. } 189. **template**<**typename** T> 190. **void** GraphWithoutDirection<T>::subgraphMinPoints(GraphNode<T>& node) 191. { 192. **if** (node.data < tem\_subgraph\_data) 193. { 194. tem\_subgraph\_data = node.data; 195. } 196. } 198. **template**<**typename** T> 199. **void** GraphWithoutDirection<T>::AllConnectivitySubgraphs() 200. { 201. vector<T> subgraph\_data; 202. vector<**bool**> visited(nodes.size(), **false**); 203. // auto func = 204. //     [&min\_key](GraphNode<T>& node) mutable -> void 205. //         { 206. //             if (node.data < min\_key) 207. //             { 208. //                 min\_key = node.data; 209. //             } 210. //     }; 211. // void (\*func\_ptr)(GraphNode<T>&node) = [](GraphNode<T>& node); 212. // func\_ptr = func; 213. **for** (**size\_t** i = 0; i < nodes.size(); i++) 214. { 215. **if** (!visited[i]) 216. { 217. tem\_subgraph\_data = nodes[i].data; 218. visited[i] = **true**; 219. DepthFirstSearch 220. ( 221. i, 222. visited, 223. &GraphWithoutDirection<T>::subgraphMinPoints 224. ); 225. subgraph\_data.push\_back(tem\_subgraph\_data); 226. } 227. } 228. sort(subgraph\_data.begin(), subgraph\_data.end()); 229. // cout << "All Connectivity Subgraphs: "; 230. **for** (auto data : subgraph\_data) 231. { 232. cout << data << " "; 233. } 234. cout << endl; 235. } 236. **template**<**typename** T> 237. **void** GraphWithoutDirection<T>::DepthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited, **void** (\*operation)(GraphNode<T>& node)) 238. { 239. **for** (auto neighbor\_index : nodes[start\_index].neighbors) 240. { 241. **if** (!visited[neighbor\_index]) 242. { 243. visited[neighbor\_index] = **true**; 244. **if** (operation != nullptr) 245. { 246. operation(nodes[neighbor\_index]); 247. DepthFirstSearch(neighbor\_index, visited, operation); 248. } 249. **else** 250. { 251. DepthFirstSearch(neighbor\_index, visited); 252. } 253. } 254. } 255. } 257. **template**<**typename** T> 258. **void** GraphWithoutDirection<T>::DepthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited, **void** (GraphWithoutDirection<T>::\*operation)(GraphNode<T>& node)) 259. { 260. **for** (auto neighbor\_index : nodes[start\_index].neighbors) 261. { 262. **if** (!visited[neighbor\_index]) 263. { 264. visited[neighbor\_index] = **true**; 265. **if** (operation != nullptr) 266. { 267. (**this**->\*operation)(nodes[neighbor\_index]); 268. DepthFirstSearch(neighbor\_index, visited, operation); 269. } 270. **else** 271. { 272. DepthFirstSearch(neighbor\_index, visited); 273. } 274. } 275. } 276. } 278. **template**<**typename** T> 279. **void** GraphWithoutDirection<T>::DepthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited, **void** (GraphWithoutDirection<T>::\*operation)()) 280. { 281. **if** (operation != nullptr) 282. { 283. (**this**->\*operation)(); 284. visited[start\_index] = **true**; 285. } 286. **for** (auto neighbor\_index : nodes[start\_index].neighbors) 287. { 288. **if** (!visited[neighbor\_index]) 289. { 290. **if** (operation != nullptr) 291. { 292. DepthFirstSearch(neighbor\_index, visited, operation); 293. } 294. **else** 295. { 296. DepthFirstSearch(neighbor\_index, visited); 297. } 298. } 299. } 300. } 302. **template**<**typename** T> 303. **void** GraphWithoutDirection<T>::DictionaryPrefaceDFS(**size\_t** start\_index, vector<**bool**>& visited, **void** (\*operation)(GraphNode<T>& node)) 304. { 305. sort(nodes[start\_index].neighbors.begin(), nodes[start\_index].neighbors.end()); 306. **for** (auto neighbor\_index : nodes[start\_index].neighbors) 307. { 308. **if** (!visited[neighbor\_index]) 309. { 310. visited[neighbor\_index] = **true**; 311. **if** (operation != nullptr) 312. { 313. operation(nodes[neighbor\_index]); 314. DictionaryPrefaceDFS(neighbor\_index, visited, operation); 315. } 316. **else** 317. { 318. DictionaryPrefaceDFS(neighbor\_index, visited); 319. } 320. } 321. } 322. } 324. **template**<**typename** T> 325. **void** GraphWithoutDirection<T>::DictionaryPrefaceDFS(**size\_t** start\_index, vector<**bool**>& visited, **void** (GraphWithoutDirection<T>::\*operation)()) 326. { 327. sort(nodes[start\_index].neighbors.begin(), nodes[start\_index].neighbors.end()); 328. **for** (auto neighbor\_index : nodes[start\_index].neighbors) 329. { 330. **if** (!visited[neighbor\_index]) 331. { 332. visited[neighbor\_index] = **true**; 333. **if** (operation != nullptr) 334. { 335. (**this**->\*operation)(); 336. DictionaryPrefaceDFS(neighbor\_index, visited, operation); 337. } 338. **else** 339. { 340. DictionaryPrefaceDFS(neighbor\_index, visited); 341. } 342. } 343. } 344. } 346. **template**<**typename** T> 347. **void** GraphWithoutDirection<T>::DictionaryPrefaceDFS(**size\_t** start\_index, vector<**bool**>& visited, **void** (GraphWithoutDirection<T>::\*operation)(GraphNode<T>& node)) 348. { 349. sort(nodes[start\_index].neighbors.begin(), nodes[start\_index].neighbors.end()); 350. **for** (auto neighbor\_index : nodes[start\_index].neighbors) 351. { 352. **if** (!visited[neighbor\_index]) 353. { 354. visited[neighbor\_index] = **true**; 355. **if** (operation != nullptr) 356. { 357. (**this**->\*operation)(nodes[neighbor\_index]); 358. DictionaryPrefaceDFS(neighbor\_index, visited, operation); 359. } 360. **else** 361. { 362. DictionaryPrefaceDFS(neighbor\_index, visited); 363. } 364. } 365. } 366. } 368. **template**<**typename** T> 369. **void** GraphWithoutDirection<T>::BreadthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited,  **void** (\*operation)(GraphNode<T>& node)) 370. { 371. queue<**size\_t**> list; 372. list.push(start\_index); 373. **while** (!list.empty()) 374. { 375. **size\_t** index = list.front(); 376. list.pop(); 377. **if** (operation != nullptr) 378. { 379. operation(nodes[index]); 380. } 381. // sort(nodes[index].neighbors.begin(), nodes[index].neighbors.end()); 382. **for** (auto neighbor\_index : nodes[index].neighbors) 383. { 384. **if** (!visited[neighbor\_index]) 385. { 386. visited[neighbor\_index] = **true**; 387. list.push(neighbor\_index); 388. } 389. } 390. } 391. } 393. **template**<**typename** T> 394. **void** GraphWithoutDirection<T>::BreadthFirstSearch(**size\_t** start\_index,vector<**bool**>& visited,  **void** (GraphWithoutDirection<T>::\*operation)()) 395. { 396. queue<**size\_t**> list; 397. list.push(start\_index); 398. **while** (!list.empty()) 399. { 400. **size\_t** index = list.front(); 401. list.pop(); 402. **if** (operation != nullptr) 403. { 404. (**this**->\*operation)(); 405. } 406. // sort(nodes[index].neighbors.begin(), nodes[index].neighbors.end()); 407. **for** (auto neighbor\_index : nodes[index].neighbors) 408. { 409. **if** (!visited[neighbor\_index]) 410. { 411. visited[neighbor\_index] = **true**; 412. list.push(neighbor\_index); 413. } 414. } 415. } 416. } 418. **template**<**typename** T> 419. **void** GraphWithoutDirection<T>::BreadthFirstSearch(**size\_t** start\_index, vector<**bool**>& visited,  **void** (GraphWithoutDirection<T>::\* operation)(GraphNode<T>& node)) 420. { 422. queue<**size\_t**> list; 423. list.push(start\_index); 424. **while** (!list.empty()) 425. { 426. **size\_t** index = list.front(); 427. list.pop(); 428. **if** (operation != nullptr) 429. { 430. (**this**->\*operation)(nodes[index]); 431. } 432. // sort(nodes[index].neighbors.begin(), nodes[index].neighbors.end()); 433. **for** (auto neighbor\_index : nodes[index].neighbors) 434. { 435. **if** (!visited[neighbor\_index]) 436. { 437. visited[neighbor\_index] = **true**; 438. list.push(neighbor\_index); 439. } 440. } 441. } 442. } 444. **template**<**typename** T> 445. **void** GraphWithoutDirection<T>::DictionaryPrefaceBFS(**size\_t** start\_index, vector<**bool**>& visited,  **void** (\*operation)(GraphNode<T>& node)) 446. { 447. queue<**size\_t**> list; 448. list.push(start\_index); 449. **while** (!list.empty()) 450. { 451. **size\_t** index = list.front(); 452. list.pop(); 453. **if** (operation != nullptr) 454. { 455. operation(nodes[index]); 456. } 457. sort(nodes[index].neighbors.begin(), nodes[index].neighbors.end()); 458. **for** (auto neighbor\_index : nodes[index].neighbors) 459. { 460. **if** (!visited[neighbor\_index]) 461. { 462. visited[neighbor\_index] = **true**; 463. list.push(neighbor\_index); 464. } 465. } 466. } 467. } 469. **template**<**typename** T> 470. **void** GraphWithoutDirection<T>::DictionaryPrefaceBFS(**size\_t** start\_index, vector<**bool**>& visited,  **void** (GraphWithoutDirection<T>::\* operation)()) 471. { 472. queue<**size\_t**> list; 473. list.push(start\_index); 474. **while** (!list.empty()) 475. { 476. **size\_t** index = list.front(); 477. list.pop(); 478. **if** (operation != nullptr) 479. { 480. (**this**->\*operation)(); 481. } 482. sort(nodes[index].neighbors.begin(), nodes[index].neighbors.end()); 483. **for** (auto neighbor\_index : nodes[index].neighbors) 484. { 485. **if** (!visited[neighbor\_index]) 486. { 487. visited[neighbor\_index] = **true**; 488. list.push(neighbor\_index); 489. } 490. } 491. } 492. }  495. **template**<**typename** T> 496. **void** GraphWithoutDirection<T>::DictionaryPrefaceBFS(**size\_t** start\_index, vector<**bool**>& visited,  **void** (GraphWithoutDirection<T>::\* operation)(GraphNode<T>& node)) 497. { 498. queue<**size\_t**> list; 499. list.push(start\_index); 500. **while** (!list.empty()) 501. { 502. **size\_t** index = list.front(); 503. list.pop(); 504. **if** (operation != nullptr) 505. { 506. (**this**->\*operation)(nodes[index]); 507. } 508. sort(nodes[index].neighbors.begin(), nodes[index].neighbors.end()); 509. **for** (auto neighbor\_index : nodes[index].neighbors) 510. { 511. **if** (!visited[neighbor\_index]) 512. { 513. visited[neighbor\_index] = **true**; 514. list.push(neighbor\_index); 515. } 516. } 517. } 518. } 520. **template**<**typename** T> 521. **void** GraphWithoutDirection<T>::DPDFSPrint(**size\_t** start\_index) 522. { 523. // cout << "Dictionary Pre-Order DFS:" << endl; 524. vector<**bool**> visited(nodes.size(), **false**); 525. cout << start\_index + 1 << " "; 526. visited[start\_index] = **true**; 527. DictionaryPrefaceDFS(start\_index, visited, printGraphNodeData); 528. cout << endl; 529. } 531. **template**<**typename** T> 532. **void** GraphWithoutDirection<T>::DPBFSPrint(**size\_t** start\_index) 533. { 534. // cout << "Dictionary Pre-Order BFS:" << endl; 535. vector<**bool**> visited(nodes.size(), **false**); 536. // cout << start\_index + 1 << " "; 537. // printGraphNode(nodes[start\_index]); 538. visited[start\_index] = **true**; 539. DictionaryPrefaceBFS(start\_index, visited, printGraphNodeData); 540. cout << endl; 541. } 543. **template**<**typename** T> 544. **size\_t** GraphWithoutDirection<T>::BFSLength(**size\_t** start\_index) 545. { 546. BFSCount = 0; 547. vector<**bool**> visited(nodes.size(), **false**); 548. visited[start\_index] = **true**; 549. BreadthFirstSearch(start\_index, visited, &GraphWithoutDirection<T>::BFSCounter); 550. **return** BFSCount; 551. } 553. **template**<**typename** T> 554. **void** GraphWithoutDirection<T>::BFSCounter() 555. { 556. BFSCount++; 557. } 559. **template**<**typename** T> 560. **size\_t** GraphWithoutDirection<T>::DFSLength(**size\_t** start\_index) 561. { 562. DFSCount = 0; 563. vector<**bool**> visited(nodes.size(), **false**); 564. DepthFirstSearch(start\_index, visited, &GraphWithoutDirection<T>::DFSCounter); 565. **return** DFSCount; 566. } 568. **template**<**typename** T> 569. **void** GraphWithoutDirection<T>::DFSCounter() 570. { 571. DFSCount++; 572. } 574. **template**<**typename** T> 575. **int** GraphWithoutDirection<T>::Dijkstra(**size\_t** start\_index, **size\_t** end\_index) 576. { 577. //由于本题目的图是无权图，实际上使用Dijkstra算法与使用BFS算法的效果是一样的 578. //但由于BFS算法的实现更简单，所以我们使用BFS算法来实现Dijkstra算法 579. queue<**size\_t**> list; 580. vector<**int**> dist(nodes.size(), 2147483647); 581. vector<**bool**> visited(nodes.size(), **false**); 582. dist[start\_index] = 0; 583. list.push(start\_index); 584. **while** (!list.empty()) 585. { 586. **size\_t** index = list.front(); 587. list.pop(); 588. **if** (!visited[index]) 589. { 590. visited[index] = **true**; 591. **for** (auto neighbor\_index : nodes[index].neighbors) 592. { 593. list.push(neighbor\_index); 594. dist[neighbor\_index] = min(dist[neighbor\_index], dist[index] + 1); 595. } 596. } 598. } 599. **if** (dist[end\_index] == 2147483647) 600. { 601. **return** -1; 602. } 603. **return** dist[end\_index]; 604. }  607. **class** Solution 608. { 609. **public**: 610. **void** solve(); 611. }; 613. **void** Solution::solve() 614. { 615. ios::sync\_with\_stdio(**false**); 616. **size\_t** n, m; 617. cin >> n >> m; 618. **size\_t** start, end; 619. cin >> start >> end; 620. GraphWithoutDirection<**int**> graph(n); 621. **for** (**size\_t** i = 0; i < m; i++) 622. { 623. **size\_t** op; 624. cin >> op; 625. **if** (op == 0) 626. { 627. **size\_t** u, v; 628. cin >> u >> v; 629. graph.addEdge(u, v); 630. } 631. **else** **if** (op == 1) 632. { 633. **size\_t** u, v; 634. cin >> u >> v; 635. graph.delEdge(u, v); 636. } 637. } 638. graph.ConnectedComponents(); 639. graph.AllConnectivitySubgraphs(); 640. // cout << "DFS Length: " << graph.DFSLength(start - 1) << endl; 641. // cout << "BFS Length: " << graph.BFSLength(end - 1) << endl; 642. // graph.DPDFSPrint(start - 1); 643. // graph.DPBFSPrint(end - 1); 644. // cout << "Dijkstra: " << graph.Dijkstra(start - 1, end - 1) << endl; 645. cout << graph.DFSLength(start - 1) << endl; 646. graph.DPDFSPrint(start - 1); 647. cout << graph.BFSLength(end - 1) << endl; 648. graph.DPBFSPrint(end - 1); 649. cout << graph.Dijkstra(start - 1, end - 1) << endl; 650. // for (auto node : graph.getNodes()) 651. // { 652. //     printGraphNode(node); 653. // } 654. **return**; 655. } 657. **int** main() 658. { 659. Solution solution; 660. solution.solve(); 661. **return** 0; 662. } | | | |