**实验5-Dijkstra**

**问题分析**

* Dijksta算法是一个基于贪心、广度优先搜索求一个图中一个点到其他所有点的最短路径的算法，时间复杂度为
* 朴素dijkstra原理
  + 初始化：

创建一个空的集合vis，用于存储已经找到最短路径的节点。创建一个数组dist，用于存储从起始节点到每个节点的当前最短距离。初始时，将起始节点的距离设置为0，其他节点的距离设置为无穷大。

创建一个优先级队列（本次实验通过一层循环比较达到优先队列的效果），用于选择下一个要探索的节点。

* + 迭代：

从优先级队列中选择距离最短的节点，将其标记为已经找到最短路径。

对于当前节点的每个相邻节点（即与当前节点有边相连的节点），计算从起始节点经过当前节点到达相邻节点的距离。如果通过当前节点到达相邻节点的距离小于相邻节点当前的最短距离，更新相邻节点的最短距离。

将更新后的距离和相邻节点添加到优先级队列中，以备下一轮迭代。

* + 重复步骤3，直到优先级队列为空或者所有节点都被标记为已找到最短路径。

1. **int** tmps = s;
2. dist[s] = 0;
3. vis[s] = 1;
4. **for** (**int** i = 1; i <= n; i++)
5. {
6. **for** (**int** j = 1; j <= n; j++)
7. {
8. **if** (m[tmps][j] == -1) **continue**;
9. **if** (dist[j] > dist[tmps] + m[tmps][j])
10. {
11. dist[j] = dist[tmps] + m[tmps][j];
12. pri[j].push\_back(tmps);//将路径保存
13. }
14. }
15. **int** mx = inf, mxi = 0;
16. **for** (**int** j = 1; j <= n; j++)//类似优先队列
17. {
18. **if** (vis[j]) **continue**;
19. **if** (dist[j] < mx)
20. {
21. mx = dist[j];
22. mxi = j;
23. }
24. }
25. vis[mxi] = 1;
26. tmps = mxi;//更新最短路径
27. }

* 如果从顶点s到顶点t有不止一条最短路径，那么输出路段数最少者。
* 如果具有最短路径且路段数也是最少的路径至少有2条，那么输出按顶点编号的字典序最小者。

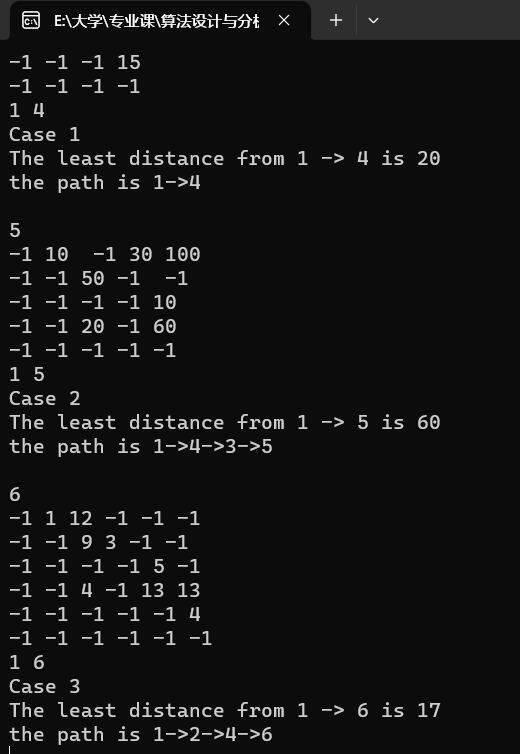
为实现两要求，需要定义一个二维vector数组用来保存所有可行的路径。

1. vector<**int**> pri[51];
2. vector<vector<**int**>> route;
3. **if** (dist[j] > dist[tmps] + m[tmps][j])
4. {
5. dist[j] = dist[tmps] + m[tmps][j];
6. pri[j].clear();//当出现更短路径，则清空更新
7. pri[j].push\_back(tmps);
8. }
9. **else**
10. {
11. **if** (dist[j] == dist[tmps] + m[tmps][j])
12. {
13. pri[j].push\_back(tmps);//若长度相同，则都保存
14. }
15. }

定义一个cmp函数使vector内部按照顶点编号的字典序和路径长短进行排序。

1. **bool** my\_comp(vector<**int**> v1, vector<**int**> v2)
2. {
3. **if** (v1.size() != v2.size())
4. **return** v1.size() < v2.size();
5. **else**
6. {
7. **int** s = v1.size();
8. **for** (**int** i = 0; i < s; i++)
9. **if** (v1[i] != v2[i])
10. **return** v1[i] < v2[i];
11. **return** 0;
12. }
13. }

**运行结果截图**



**设计调试中的问题**

* 在打印路径时遇到有多条路径的情况下，会将两个可行答案穿插错误出现。

出现错误的原因是因为在打印在递归调用打印函数时，用到了循环遍历，每次退出递归还需恢复现场。即r.pop\_back()

1. **void** find\_route(**int** t, vector<**int**> r)
2. {
3. **if** (t == s)
4. {
5. reverse(r.begin(), r.end());
6. route.push\_back(r);
7. }
8. **else**
9. {
10. **for** (auto it : pri[t])
11. {
12. r.push\_back(it);
13. find\_route(it, r);
14. r.pop\_back();
15. }
16. }
17. }

**实验体会**

此次实验让我再次温习了dijksra算法。同时在进阶任务中保存多条路径的步骤也让我学会了迭代器和递归调用的搭配使用。在一开始，进阶任务带给我不小的挑战，但通过不断的尝试，不断对算法主题修改，最终达成目标，也让我对dijkstra算法有了更深的认识。

在实验的最后，我发现其实原本最初的朴素的dijkstra代码其实可以一定程度上满足进阶任务的需求。取字典序最大路径：该题目中默认节点的编号为行号（即纯数字），而每次在遍历最近点时也是从小到大开始递增查找，因此先找到的最优解一定是字典序最大的。（如果节点编号不为纯数字则失效）通过以上的发现，让我意识到有时一些复杂的功能或规则其实就隐藏在原本的功能之中，需要我们多观察多分析就能达到化繁为简的妙处。

**程序源码**

1. #include <bits/stdc++.h>
3. **using** **namespace** std;
4. **int** n,s,t;
5. **int** m[51][51];
6. **int** inf = 1e9;
7. **int** dist[51], vis[51];
8. vector<**int**> pri[51];
9. vector<vector<**int**>> route;
11. **void** find\_route(**int** t, vector<**int**> r)
12. {
13. **if** (t == s)
14. {
15. reverse(r.begin(), r.end());
16. route.push\_back(r);
17. }
18. **else**
19. {
20. **for** (auto it : pri[t])
21. {
22. r.push\_back(it);
23. find\_route(it, r);
24. r.pop\_back();
25. }
26. }
27. }
29. **bool** my\_comp(vector<**int**> v1, vector<**int**> v2)
30. {
31. **if** (v1.size() != v2.size())
32. {
33. **return** v1.size() < v2.size();
34. }
35. **else**
36. {
37. **int** s = v1.size();
38. **for** (**int** i = 0; i < s; i++)
39. {
40. **if** (v1[i] != v2[i])
41. {
42. **return** v1[i] < v2[i];
43. }
44. }
45. **return** 0;
46. }
47. }

50. **int** main()
51. {
52. **int** count = 1;
53. **while** (cin >> n)
54. {
55. **for** (**int** i = 1; i <= n; i++)
56. {
57. **for** (**int** j = 1; j <= n; j++)
58. {
59. cin >> m[i][j];
60. }
61. }
62. cin >> s >> t;
63. **for** (**int** i = 1; i <= n; i++)
64. {
65. dist[i] = inf;
66. vis[i] = 0;
67. pri[i].clear();
68. }
69. **int** tmps = s;
70. dist[s] = 0;
71. vis[s] = 1;
72. **for** (**int** i = 1; i <= n; i++)
73. {
74. **for** (**int** j = 1; j <= n; j++)
75. {
76. **if** (m[tmps][j] == -1) **continue**;
77. **if** (dist[j] > dist[tmps] + m[tmps][j])
78. {
79. dist[j] = dist[tmps] + m[tmps][j];
80. pri[j].clear();
81. pri[j].push\_back(tmps);
82. }
83. **else**
84. {
85. **if** (dist[j] == dist[tmps] + m[tmps][j])
86. {
87. pri[j].push\_back(tmps);
88. }
89. }
90. }
91. **int** mx = inf, mxi = 0;
92. **for** (**int** j = 1; j <= n; j++)
93. {
94. **if** (vis[j]) **continue**;
95. **if** (dist[j] < mx)
96. {
97. mx = dist[j];
98. mxi = j;
99. }
100. }
101. vis[mxi] = 1;
102. tmps = mxi;
103. }
104. vector<**int**> r;
105. r.push\_back(t);
106. find\_route(t, r);
108. sort(route.begin(), route.end(), my\_comp);
109. cout << "Case " << count << endl;
110. cout << "The least distance from " << s << " -> " << t << " is " << dist[t] << endl;
111. cout << "the path is ";
112. **for** (auto it : route[0])//0代表排序后第一个
113. {
114. **if** (it == s)
115. cout << it;
116. **else**
117. cout << "->" << it;
118. }
119. cout << endl;
120. }
121. **return** 0;
122. }