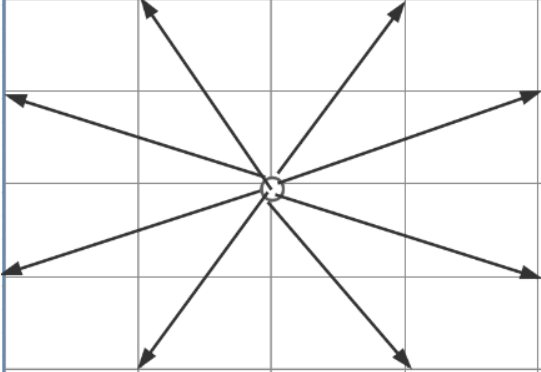
**实验6-跳马问题**

**问题分析**

* 跳马问题也称骑士遍历问题，在本题中运用到中国象棋中的马走日的规则。即跳马的方向总共有八个，如图所示。



预先定义八个方向的“日”形的横纵坐标偏移量

1. **int** x[8] = { 1,1,2,2,-1,-1,-2,-2 };
2. **int** y[8] = { 2,-2,1,-1,2,-2,1,-1 };
3. //右1上2 右1下2 右2上1 右二下1 左1 上2 左1下2 左2上1 左1下2

* 要求最少步数，采用广度优先搜索算法。因为广度优先搜索算法是从起点开始一层一层地向外扩展，因此最终求得地步数一定是最小的。同时因为一层结束后才会开始新的一层，所以可以保证所有情况都包含而不遗漏。
* 同时需要定义一个二维数组state[x][y]记录（x,y）是否已经被访问过。确保不会重复经过同一个点，假使重复经过则说明两次经过该点之间的路程是不必要的，说明该路径不是最短，同时也会使各方案之间路径重叠增加复杂度。
* 广度优先搜索算法部分：

采用stl库中的队列作为容器，已（x,y）的数对作为其中元素。首先将队头元素弹出，然后遍历该元素能跳到的八个方向的位置，对访问到的位置先进性越界判断即是否超出棋盘，其次更新该位置的status[x][y]为true并将其入队，最后更新该位置的最小步数值，即dist[tx][ty] = dist[x2][y2] + 1;

1. p.push({ sx,sy });
2. **while** (!p.empty())
3. {
4. **int** x2 = p.front().first;
5. **int** y2 = p.front().second;
6. p.pop();
7. **for** (**int** i = 0; i < 8; i++)
8. {
9. **int** tx = x2 + x[i];
10. **int** ty = y2 + y[i];
11. //如果越界或者被访问过则跳过
12. **if** (tx < 1 || ty < 1 || tx >8 || ty>8 || state[tx][ty])
13. {
14. **continue**;
15. }
16. state[tx][ty] = **true**;
17. p.push({ tx,ty });
18. dist[tx][ty] = dist[x2][y2] + 1;
19. }
20. }

**进阶任务**

尝试采用DFS实现跳马问题。

采用深度优先搜索算法时需要用到递归调用和回溯法，同时定义一个minstep来记录最小步长数。当到达目标位置时进行判断是否是最小步长并进行更新。但是经过调试后发现采用递归调用的深度优先搜索算法来解决该跳马问题，容易出现递归栈特别深，从而导致运行时间特别长等情况。

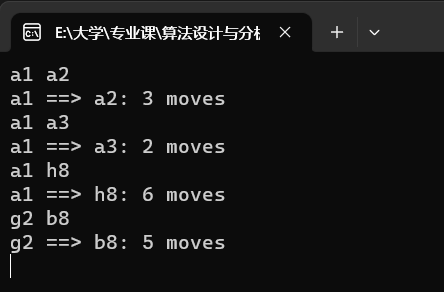
后采取剪枝操作，在每次进行扩展一个新的节点时，先进行比较：如果当前步数step已经大于最短步长minstep则直接剪枝，其后续的步长也就没有意义了。

1. **if** (step < minstep)
2. {
3. state[tx][ty] = **true**;
4. dfs(tx, ty, ex, ey, step + 1);
5. state[tx][ty] = **false**;
6. }

DFS具体代码：

1. **void** dfs(**int** sx, **int** sy, **int** ex, **int** ey,**int** step)
2. {
3. **if** (sx == ex && sy == ey)
4. {
5. **if** (step < minstep)
6. {
7. minstep = step;
8. //cout << minstep << endl;
9. **return**;
10. }
11. **return**;
12. }
13. **for** (**int** i = 0; i < 8; i++)
14. {
15. **int** tx = sx + x[i];
16. **int** ty = sy + y[i];
17. //如果越界或者被访问过则跳过
18. **if** (tx < 1 || ty < 1 || tx >8 || ty>8 || state[tx][ty])
19. **continue**;
20. **if** (step < minstep)
21. {
22. state[tx][ty] = **true**;
23. dfs(tx, ty, ex, ey, step + 1);
24. state[tx][ty] = **false**;
25. }
26. }
27. **return** ;
28. }

**运行结果截图**



**设计调试中的问题**

* 对于输入的处理

调试初始时，误将输入中的字母和数字对应行和列混淆，后将其调换即可。同时也应注意在字母映射到数字时的一些细节处理。

1. **int** sx, sy, ex, ey;
2. sx = st[1] - '0';
3. sy = st[0] - 'a' + 1;
4. ex = ed[1] - '0';
5. ey = ed[0] - 'a' + 1;

**DFS和BFS的选用**

深度优先搜索（DFS）通俗来说就是一直往下走，走不通回头，换条路再走，直到无路可走。在实验最初打算采用深度优先搜索算法的策略去解题，但是采用dfs求解时，首先如果不采用剪枝算法会造成运行时间过长。其次，即使使用剪枝也无法保证最开始搜到的就是最近的点，换句话说在一开始就有很大的概率走一步会选中不能到达终点或者能到达但步数不是最小的点，所以接下来的操作其实都是不必要进行的，是累赘的，但是我们无法预知该种情况。在大规模的数据下，这些情况使得复杂度急剧升高。

广度优先搜索（BFS）每一次搜索离初始点的步数都是相等的，可以理解为所有方案“同时”执行，每次每个方案都执行一步，这在保证不遗漏的情况下，使得找到最小步数的方案会在一个较小的数量级范围内。

**实验体会**

此次实验虽然代码量不大，但是运用bfs去解题的思路却十分巧妙。

在实验最初打算采用深度优先搜索算法的策略去解题，但是如上文所说采用dfs反而会使得复杂度升高。最终通过思考以及上网查阅选择广度优先搜索，广度优先搜索在本题中从起点开始一层层地向外扩张计数，直至达到终点。其中一层一层地搜索既保证了不遗漏同时也确保了最终所得的答案一定是最小值。实际上搜索算法的本质也是一种暴力枚举的策略，但是相较于直接枚举，广度优先算法可以通过一些条件的设置跳过一些无效状态从而降低问题的规模，也因此效率会比枚举要来得高。综上，此次实验让我深刻体会了广度优先搜索算法的奥妙之处，同时通过从dfs到bfs的转换让我意识到一个恰当的算法在一些复杂问题中能有很大的改善，也让我理解了bfs和dfs这两者之间的优劣以及面对不同类型问题所达到的效率和可能遇到的问题。

**程序源码**

1. #include<iostream>
2. #include<bits/stdc++.h>
3. **using** **namespace** std;
4. **const** **int** inf = 0x3f3f3f3f;
5. **int** x[8] = { 1,1,2,2,-1,-1,-2,-2 };
6. **int** y[8] = { 2,-2,1,-1,2,-2,1,-1 };
7. //右1上2 右1下2 右2上1 右二下1 左1 上2 左1下2 左2上1 左1下2
8. **int** dist[9][9];
9. **bool** state[9][9];
10. **int** bfs(**int** sx,**int** sy,**int** ex,**int** ey )
11. {
12. queue<pair<**int**, **int**>> p;
13. dist[sx][sy] = 0;
14. state[sx][sy] = 1;
15. p.push({ sx,sy });
16. **while** (!p.empty())
17. {
18. **int** x2 = p.front().first;
19. **int** y2 = p.front().second;
20. p.pop();
21. **for** (**int** i = 0; i < 8; i++)
22. {
23. **int** tx = x2 + x[i];
24. **int** ty = y2 + y[i];
25. //如果越界或者被访问过则跳过
26. **if** (tx < 1 || ty < 1 || tx >8 || ty>8 || state[tx][ty])
27. {
28. **continue**;
29. }
30. state[tx][ty] = **true**;
31. p.push({ tx,ty });
32. dist[tx][ty] = dist[x2][y2] + 1;
33. }
34. }
35. **return** dist[ex][ey];
36. }
37. **int** minstep = inf;
38. **void** dfs(**int** sx, **int** sy, **int** ex, **int** ey,**int** step)
39. {
40. **if** (sx == ex && sy == ey)
41. {
42. **if** (step < minstep)
43. {
44. minstep = step;
45. //cout << minstep << endl;
46. **return**;
47. }
48. **return**;
49. }
50. **for** (**int** i = 0; i < 8; i++)
51. {
52. **int** tx = sx + x[i];
53. **int** ty = sy + y[i];
54. //如果越界或者被访问过则跳过
55. **if** (tx < 1 || ty < 1 || tx >8 || ty>8 || state[tx][ty])
56. **continue**;
57. **if** (step < minstep)
58. {
59. state[tx][ty] = **true**;
60. dfs(tx, ty, ex, ey, step + 1);
61. state[tx][ty] = **false**;
62. }
63. }
64. **return** ;
65. }
66. **int** main()
67. {
68. string st, ed;
69. **while** (cin >> st)
70. {
71. minstep = inf;
72. memset(dist, -1, **sizeof**(dist));
73. memset(state, 0, **sizeof**(state));
74. cin >> ed;
75. **int** sx, sy, ex, ey;
76. sx = st[1] - '0';
77. sy = st[0] - 'a' + 1;
78. ex = ed[1] - '0';
79. ey = ed[0] - 'a' + 1;
80. state[sx][sy] = 1;//设置原点走过
81. //int b\_ans = bfs(sx, sy, ex, ey);
82. dfs(sx, sy, ex, ey, 0);
83. **int** d\_ans = minstep;
84. cout << st << " ==> " << ed << ": " << d\_ans << " moves" << endl;
85. }
86. }