

# 实验6-跳马问题

## 问题分析

- 问题要求求出马跳到某一点的最少步数，采用BFS求解即可。因为BFS是一层一层拓展的，所以最后的步数一定是最小的
- 定义`dist[x][y]`为马跳到 $(x,y)$ 的最小步数
- 定义`state[x][y]`记录 $(x,y)$ 是否已经被访问过，避免重复计算和错误更新答案
- 马是走“日”形的，所以预先定义八个方向的“日”形的横纵坐标偏移量

```
1  int dx[8]={1,1,2,2,-1,-1,-2,-2};
2  int dy[8]={2,-2,1,-1,2,-2,1,-1};
```

- 对于输入的处理-将字母映射到数字形式的横纵坐标即可

```
1  int sx,sy,ex,ey;
2  sx=st[1]-'0';//起点
3  sy=st[0]-'a'+1;
4  ex=ed[1]-'0';//终点
5  ey=ed[0]-'a'+1
```

- 本题中有多组测试数据，在BFS前要对用到的数组初始化

```
1  memset(dist, -1, sizeof dist);
2  memset(state, 0, sizeof state);
3  dist[x][y]=0;
4  state[x][y]=true;
```

- BFS算法部分

首先将队头元素弹出，然后遍历当前格子能跳的八个方向（注意判断坐标是否超出边界）

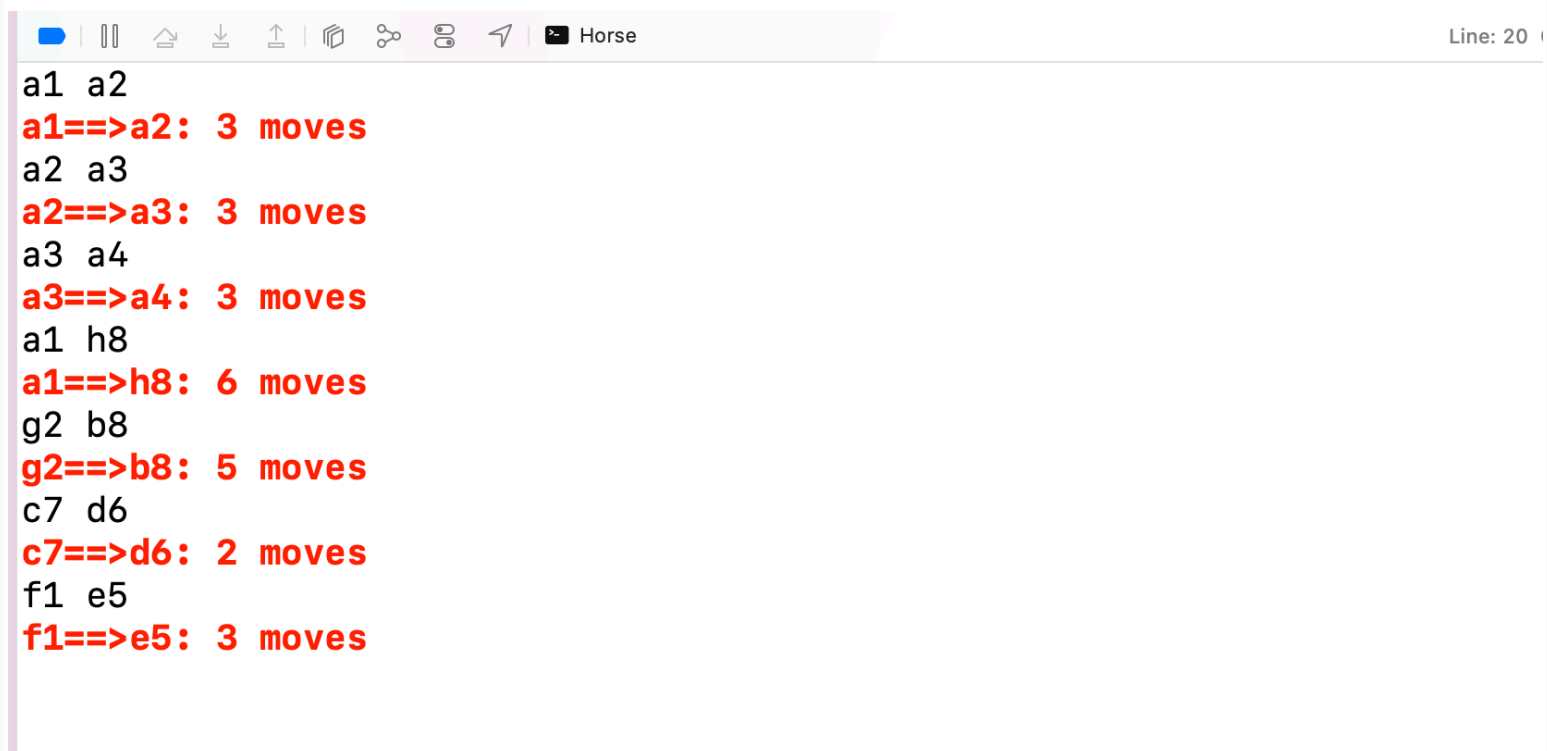
对访问到的格子的`state`值复制为`true`并让能访问到的格子坐标进入队列，最后更新该格子的最小步数值，即`dist[tx][ty]=dist[x2][y2]+1`

```

1  while (q.size()) {
2      auto [x2,y2]=q.front();
3      q.pop();
4      for (int i=0; i<8; i++) {
5          int tx=x2+dx[i],ty=y2+dy[i];
6          if (tx<1||tx>n||ty<1||ty>n||state[tx][ty]) {
7              continue;
8          }
9          state[tx][ty]=true;
10         q.push({tx,ty});
11         dist[tx][ty]=dist[x2][y2]+1;
12     }
13 }

```

## 运行结果截图



```

a1 a2
a1==>a2: 3 moves
a2 a3
a2==>a3: 3 moves
a3 a4
a3==>a4: 3 moves
a1 h8
a1==>h8: 6 moves
g2 b8
g2==>b8: 5 moves
c7 d6
c7==>d6: 2 moves
f1 e5
f1==>e5: 3 moves

```

## 设计调试中的问题

访问队列队头元素后，忘记将队头弹出队列，导致程序死循环，加上弹出队头操作即可

```

1  q.pop();

```

# 用DFS与BFS求解此题的效率和可能的问题

我们都知道dfs是一条路搜到黑，所以不可以保证最开始搜到的就是最近点。也就是说，我们在遍历的过程中，结果是在不断更新的。如果用dfs求解我们把马所有可能走的路径全都搜索了一遍，有很多搜索都是累赘的。在数据量大的情况下这可能会导致程序运行超时，而bfs每步搜索离初始点的步数都是相等的，在我们找到最小步数前所走的方案数会在一个较小的数量级范围内，使得我们能很快地找到最小步数。

## 实验体会

本次实验是bfs的一个实际应用场景，题目比较有意思，跟象棋中的马走日结合在了一起。暴力枚举策略就是将所有可能的情况都枚举一遍以获得最优解，但是枚举全部元素的效率如同愚翁移山，无法应付数据范围稍大的情形。而深度优先搜索和广度优先搜索，从起点开始，逐渐扩大寻找范围，直到找到需要的答案为止。严格来说，搜索算法也算是一种暴力枚举策略，但是其算法特性决定了效率比直接的枚举所有答案要高，因为搜索可以跳过一些无效状态，降低问题规模。这次实验让我认识到了搜索这一策略让某些复杂问题的求解变得更容易去应对。

## 程序源码

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  const int N=1e3+10;
4  int n=8;
5  int dx[8]={1,1,2,2,-1,-1,-2,-2},dy[8]={2,-2,1,-1,2,-2,1,-1};
6  int dist[N][N];
7  bool state[N][N];
8  int solve(int x,int y,int ex,int ey){
9      queue<pair<int, int>> q;
10     q.push({x,y});
11     memset(dist, -1, sizeof dist);
12     memset(state, 0, sizeof state);
13     dist[x][y]=0;
14     state[x][y]=true;
15     while (q.size()) {
16         auto [x2,y2]=q.front();
17         q.pop();
18         for (int i=0; i<8; i++) {
19             int tx=x2+dx[i],ty=y2+dy[i];
20             if (tx<1||tx>n||ty<1||ty>n||state[tx][ty]) {
21                 continue;
```

```
22         }
23         state[tx][ty]=true;
24         q.push({tx,ty});
25         dist[tx][ty]=dist[x2][y2]+1;
26     }
27 }
28 return dist[ex][ey];
29 }
30 int main()
31 {
32     ios::sync_with_stdio(false);
33     cin.tie(nullptr);
34     string st,ed;
35     while (cin>>st) {
36         cin>>ed;
37         int sx,sy,ex,ey;
38         sx=st[1]-'0';
39         sy=st[0]-'a'+1;
40         ex=ed[1]-'0';
41         ey=ed[0]-'a'+1;
42         int dist=solve(sx,sy,ex,ey);
43         cout<<st<<"==>"<<ed<<": "<<dist<<" moves\n";
44     }
45 }
```