编程 1 报告

——李昭阳 2021013445

一、 算法思路

在规则设计部分，限制了棋子运动范围为棋盘内部非己方棋子已占有的点，同时根据“蹩马腿”规则制定了跳跃的判断规则。代码如下。

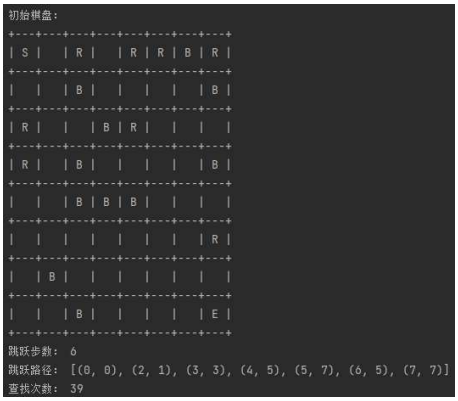
```python
def is_valid_move(self, x, y, prev_x, prev_y):
    '''
    判断当前移动是否有效
    :param x: 移动后点的x坐标
    :param y: 移动后点的y坐标
    :param prev_x: 移动前点的x坐标
    :param prev_y: 移动前点的y坐标
    :return: True或False
    '''
    # 移动点超出棋盘范围
    if x < 0 or x >= self.size or y < 0 or y >= self.size:
        return False
    # 已有己方棋子的地方不能下棋（B方）
    if self.board[x][y] == 'B':
        return False
    # 不允许不动
    if (x, y) == (prev_x, prev_y):
        return False

    # 当马脚越过角线的两侧长边相邻位置，检查是否有己方或敌方棋子
    diff_x = abs(x - prev_x)
    diff_y = abs(y - prev_y)
    if diff_x == 2 and diff_y == 1:
        if x - prev_x == 2:
            if self.board[prev_x + 1][prev_y] != ' ':
                return False
        else:
            if self.board[prev_x - 1][prev_y] != ' ':
                return False
    elif diff_x == 1 and diff_y == 2:
        if y - prev_y == 2:
            if self.board[prev_x][prev_y + 1] != ' ':
                return False
        else:
            if self.board[prev_x][prev_y - 1] != ' ':
                return False

    return True
```

在实现部分，采用了广度优先搜索的策略，通过维护一个队列，访问棋盘上所有合法节点。同时，为了减小计算代价，将已访问的节点进行标记，使得每个合法节点仅被操作一次。代码如下。

```python
def solve(self):
    '''
    使用广度优先搜索算法解决棋盘问题
    :return: None
    '''
    if not self.start or not self.end:
        print("起始点或终止点未设置")
        return

    self.search_time = 0
    visited = set()
    q = queue.Queue()
    q.put((self.start, []))

    while not q.empty():
        (x, y), path = q.get()
        if self.is_goal_reached(x, y):
            print("跳跃步数：", len(path))
            print("跳跃路径：", path + [(x, y)])
            print("查找次数：", self.search_time)
            return

        if (x, y) in visited:
            continue
        visited.add((x, y))
        self.search_time += 1

        for next_x, next_y in self.get_possible_moves(x, y):
            if (next_x, next_y) in visited:
                continue
            q.put(((next_x, next_y), path + [(x, y)]))

    print("未找到路径，目标不可达")
```

二、 算法效果

在 8x8 的棋盘中，广度优先算法表现出较高的性能，在 39 次查找后找到了解，同时由于广度优先搜索具有最优性，所以可知该解为最优解。具体搜索效果如下图。



在 100x100 的棋盘中，广度优先算法查找了 8353 次才找到最优解，近乎查询了棋盘中的每一个合法点，效率过低。我认为可以采用 A*算法，以向右下角移动的距离为启发函数，鼓励算法向右下角探索终点，使得搜索速度更快。