人工智能基础第一次实验报告

PB21020651 武宇星

A*搜索

启发式函数

启发函数即选用曼哈顿距离即坐标值差的绝对值之和，显然对于地图上任意一点到一点所需的天数即它们的曼哈顿距离，若考虑途经障碍物绕行或者因补给不足需要绕行补给站都会增加到达目标的天数，显然曼哈顿距离作为启发式函数是 admissible 的；而对于任一点 n，任意后继为 n'，显然 h(n')=h(n)-1 或 h(n')=h(n)+1，而每一步代价 c=1，故总有 h(n)<=c+h(n')，曼哈顿距离作为启发式函数也是 consistent 的。

算法思路

在原有框架的基础上，地图信息没有在做改动，而对于搜索状态则添加了 int x,y 记录节点坐标，int supply 记录节点剩余的物资天数，Search_Cell *parent 记录它的前驱节点这样在搜索到终点时即可获得最优路径的全部节点。

而对于 A*搜索实现部分即最开始将起点加入边缘队列，由于使用优先队列存储边缘队列，g+h 最小的总是在队首，故每次将边缘队列队首元素拿出并加入访问过的节点队列，每个节点共有上下左右四种选择，对于每个节点检查坐标是否在地图边界内，是否为障碍物，走到这一步是否物资耗尽或者补给站补满物资，是否在访问过节点队列后加入边缘队列，如此重复直至探索节点为终点即可根据 parent 指针得到节点坐标和该节点的差推出这一步的前进方向，以此类推最优路径的每一步方向；或者边缘队列为空则说明无法达到终点。

优化效果

设置启发式函数为 0，进行代价一致性搜索，最后一个地图过大四十分钟都没有搜完，

```
Input 0 time cost: 0us
Input 1 time cost: 0us
Input 2 time cost: 0us
Input 3 time cost: 1443us
Input 4 time cost: 0us
Input 5 time cost: 2004us
Input 6 time cost: 0us
Input 7 time cost: 1004us
Input 8 time cost: 3000us
Input 9 time cost: 3060us
```

```
Input 0 time cost: 0us
Input 1 time cost: 0us
Input 2 time cost: 0us
Input 3 time cost: 0us
Input 4 time cost: 0us
Input 5 time cost: 0us
Input 6 time cost: 0us
Input 7 time cost: 0us
Input 8 time cost: 0us
Input 9 time cost: 0us
Input 10 time cost: 890514us
```

所以耗费一点时间但也远远优于代价一致性搜索由于每一步代价均为 1，此时代价一致性搜索即广度优先搜索，取决于地图大小和障碍物的多少来减少边缘队列的大小而 A*则更合理的拓展边缘队列，除了最后一个地图大没有障碍物有很多条最优路径，

Alpha-beta 剪枝算法

算法实现过程

生成合法动作

马：对于拌马脚部分，添加对于的坐标数组，在循环中判断马脚坐标是否有棋子，若有则直接进入下个循环，由于落子坐标已经进行过棋盘边界判断，故马脚不需要。

```
int fx[] = {1, 0, 0, -1, -1, 0, 0, 1};
int fy[] = {0, 1, 1, 0, 0, -1, -1, 0};
for(int i = 0; i < 8; i++) {
    Move cur_move;
    int nx = x + dx[i];
    int ny = y + dy[i];
    if (nx < 0 || nx >= 9 || ny < 0 || ny >= 10) continue;
    int ox = x + fx[i];
    int oy = y + fy[i];
    if (board[oy][ox] != '.') continue;
```

炮：对于炮，和车一样进行四个方向搜索，用 flag 表示是否碰到过棋子，如果 flag 为 false 即没碰到过，则只能落子于空白处，碰到棋子置 flag 为 true，接下来只能吃不同颜色棋子，并在再次碰到棋子时结束循环。

```
bool flag = false;
for(int i = x + 1; i < sizeY; i++) {
    Move cur_move;
    cur_move.init_x = x;
    cur_move.init_y = y;
    cur_move.next_x = i;
    cur_move.next_y = y;
    cur_move.score = 0;
    if (board[y][i] != '.') {
        if (!flag) {
            flag = true;
        }
        else {
            bool cur_color = (board[y][i] >= 'A' && board[y][i] <= 'Z');
            if (cur_color != color) {
                PaoMoves.push_back(cur_move);
            }
            break;
        }
    }
    else if (!flag){
        PaoMoves.push_back(cur_move);
    }
```

相：相的动作生成与马一样，只需改变落点位置和相脚位置即可

```
    int dx[] = {2, 2, -2, -2};
    int dy[] = {2, -2, 2, -2};
    int fx[] = {1, 1, -1, -1};
    int fy[] = {1, -1, 1, -1};
    for(int i = 0; i < 4; i++) {
        Move cur_move;
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (nx < 0 || nx >= 9 || ny < 0 || ny >= 10) continue;
        int ox = x + fx[i];
        int oy = y + fy[i];
        if (board[oy][ox] != '.') continue;
        cur_move.init_x = x;
        cur_move.init_y = y;
        cur_move.next_x = nx;
        cur_move.next_y = ny;
        cur_move.score = 0;
        if (board[ny][nx] != '.') {
            bool cur_color = (board[ny][nx] >= 'A' && board[ny][nx] <= 'Z');
            if (cur_color != color) {
                XiangMoves.push_back(cur_move);
            }
            continue;
        }
        XiangMoves.push_back(cur_move);
    }
```

士：士的动作生成即检查斜着四步是否在棋盘以及己方九宫格范围内，并判断是否不同色合法吃子

```
int dx[] = {1, 1, -1, -1};
int dy[] = {1, -1, 1, -1};
for(int i = 0; i < 4; i++) {
    Move cur_move;
    int nx = x + dx[i];
    int ny = y + dy[i];
    if (nx < 3 || nx >= 6 || ny < 0 || ny >= 3 && ny <= 6 || ny >= 10) continue;
    cur_move.init_x = x;
    cur_move.init_y = y;
    cur_move.next_x = nx;
    cur_move.next_y = ny;
    cur_move.score = 0;
    if (board[ny][nx] != '.') {
        bool cur_color = (board[ny][nx] >= 'A' && board[ny][nx] <= 'Z');
        if (cur_color != color) {
            ShiMoves.push_back(cur_move);
        }
        continue;
    }
    ShiMoves.push_back(cur_move);
}
```

将：将除了和士相似检查上下左右四步是否在棋盘以及己方九宫格范围内，并判断是否不同色合法吃子，还需要考虑将帅互吃的情况，需遍历 y 轴查看能否直接吃到敌方将帅

```
for(int j = y + 1; j < sizeX; j++) {
    Move cur_move;
    cur_move.init_x = x;
    cur_move.init_y = y;
    cur_move.next_x = x;
    cur_move.next_y = j;
    cur_move.score = 0;
    if (board[j][x] != '.') {
        if (board[j][x] == 'k' || board[j][x] == 'K') {
            JiangMoves.push_back(cur_move);
        }
        break;
    }
}
```

兵：兵需要根据颜色判断是往上走还是往下走，还需根据坐标判断是否过河进行左右判定

```
bool flag = color ? (y < 5) : (y > 4);
int forward_dy = (color ? -1 : 1);
```

终止判断

遍历棋盘统计是否有将帅即可，缺少将或帅即游戏结束

```
bool judgeTermination() {
    //TODO
    int red_king = 0;
    int black_king = 0;
    for (int i = 0; i < pieces.size(); i++) {
        if (pieces[i].name == 'K') {
            red_king++;
        }
        else if (pieces[i].name == 'k') {
            black_king++;
        }
    }
    if (red_king == 0 || black_king == 0) {
        // std::cout << "Game Over!" << std::endl;
        return true;
    }
    return false;
}
```

棋盘分数评估

遍历棋盘中棋子的价值以及位置分数累加得到双方分数，然后用红方减去黑方

```
int evaluateNode() {
    //TODO
    int red_score = 0;
    int black_score = 0;
    // int red_king = 0;…
    for (int i = 0; i < pieces.size(); i++) {
        // if (pieces[i].name == 'K') {…
        std::string s;
        s = pieces[i].name;
        int cur_score = piece_values[s];
        switch (pieces[i].name)…
        if (pieces[i].color) {
            red_score += cur_score;
        }
        else {
            black_score += cur_score;
        }
    }
    // if (red_king == 0) {…
    return red_score - black_score;
}
```

构建新棋盘

根据动作更新棋盘，然后利用构造函数生成子节点即可

alpha-beta 剪枝过程

isMaximizer 为 true 即最大化节点，红方下寻求分数最大化，遍历动作利用递归计算黑方下的最小化即 isMaximizer 为 false，当β≤α时，对手不会有更好/更坏选择，故跳出循环；而递归则需考虑深度，深度为 0 则直接返回棋盘评分，不再进一步搜索，同样若棋局已结束，也不能进行操作，直接返回棋盘评分。Alpha-beta 剪枝减少了评估节点的数量，搜索选用的最大深度越深，相比不减枝优化的效果就越明显。

棋局搜索

红方先手遍历所有合法动作，更新棋盘让黑方下搜索最小分数，取分数最大化，为进一步提高搜索效率，若存在将死结局，直接取该动作，不再继续搜索其他动作。一开始尝试使用 int 最大最小值代表，但发现运行过程中会内存不够用，而将帅价值 10000，故以+/-9000

判断是否将死。

```
GameTreeNode* childptr = root.updateBoard(cur_board.getBoard(), move, player);
GameTreeNode child = *childptr;
delete childptr;
int score = alphaBeta(child, alpha, beta, depth, !player);
std::cout << "score: " << score << std::endl;
if (player) {
    if(score > 9000) {
        best_score = score;
        best_move = move;
        break;
    }
    if (score > best_score) {
        best_score = score;
        best_move = move;
    }
} else {
    if(score < -9000) {
        best_score = score;
        best_move = move;
        break;
    }
    if (score < best_score) {
        best_score = score;
        best_move = move;
    }
}
```

评估函数效果

除了第八题和第九题，红方均以深度 3 搜索到了将死对面的走法，其中第八题因为有炮勾引，而红方需要三步才能将死，搜索深度为 6 才能搜到，故无法正确判断，第九题则是需要更多步。对于第十题，除了可能因为黑方也没有走最优解致使被将死，但在评估函数中加入位置评估也诱导算法走向更优的局面，最后将死对面，评估函数效果还是比较好的。