lab1 report

PB21081601 张芷苒

Astar

TODO 部分

补全完成启发式函数定义部分、A*搜索部分等标注有 TODO 标记代码：
第一个TODO:

```
struct Map_Cell
{
    int type; // 地图的类型：0空地、1障碍、2补给、3起点、4终点
};
```

第二个TODO:

```
struct Search_Cell
{
    int x, y; // 位置坐标
    int g; // 从起点到当前节点的实际代价
    int h; // 从当前节点到终点的估计代价（启发式）
    Search_Cell *parent; // 父节点

    Search_Cell(int x, int y, int g, int h, Search_Cell* parent =
nullptr) : x(x), y(y), g(g), h(h), parent(parent) {}
};
```

第三个TODO:

```
// 启发式函数，使用曼哈顿距离
int Heuristic_Function(int x, int y, int target_x, int target_y) {
    return abs(x - target_x) + abs(y - target_y);
}
```

第四个TODO:

```
...
while(!open_list.empty() && !found)
    {
```

```
        current = open_list.top();
        open_list.pop();

        // 到达终点
        if (current->x == end_point.first && current->y ==
end_point.second) {
            found = true;
            break;
        }

        // 扩展当前节点
        const vector<pair<int, int>> directions{{0, 1}, {1, 0}, {0,
-1}, {-1, 0}}; // 四个可能的移动方向
        for (const auto& dir : directions) {
            int new_x = current->x + dir.first;
            int new_y = current->y + dir.second;

            if (new_x ≥ 0 && new_x < M && new_y ≥ 0 && new_y < N &&
Map[new_x][new_y].type ≠ 1) { // 确保在范围内且不是障碍
                int new_g = current->g + 1;
                int new_h = Heuristic_Function(new_x, new_y,
end_point.first, end_point.second);
                auto new_cell = new Search_Cell(new_x, new_y, new_g,
new_h, current);

                if (all_cells.find({new_x, new_y}) == all_cells.end()
|| new_g < all_cells[{new_x, new_y}]->g) { // 发现更短的路径
                    all_cells[{new_x, new_y}] = new_cell;
                    open_list.push(new_cell);
                }
            }
        }
    }
...
```

第五个TODO:

```
// 构建路径
    if (found) {
        vector<char> path;
        while (current ≠ nullptr && current->parent ≠ nullptr) {
            if (current->x == current->parent->x + 1)
path.push_back('D');
            else if (current->x == current->parent->x - 1)
path.push_back('U');
            else if (current->y == current->parent->y + 1)
```

```
            path.push_back('R');
            else if (current->y == current->parent->y - 1)
path.push_back('L');
            current = current->parent;
        }
        reverse(path.begin(), path.end());
        step_nums = path.size();
        way = string(path.begin(), path.end());
    } else {
        step_nums = -1;
        way = "";
    }
```

启发式函数描述与属性分析

启发式函数使用的是曼哈顿距离，计算如下：

```
int Heuristic_Function(int x, int y, int target_x, int target_y) {
    return abs(x - target_x) + abs(y - target_y);
}
```

这个函数计算了当前节点到目标节点的格子行列差的绝对值之和，适用于只能沿水平或垂直方向移动的情况。

Admissibility:
曼哈顿距离总是返回从当前点到终点所需的最少步数（每步只能沿格子边移动），从不高估实际的最小移动次数，所以它满足“永远不会高估任意节点到目标的最低成本”的性质，所以可以说这个启发式函数是 admissible的。

Consistency:
对于这个启发式函数，无论向哪个方向移动一格，估计的代价总是确切地减少了1（或增加了1），这正好等于从 n 到 p 的实际距离。这和一致性的要求：“如果对于任意节点 n 和其任一邻居节点 p，下列条件成立：h(n) ≤ d(n, p) + h(p)，其中 d(n, p) 是节点 n 和 p 之间的实际代价”相符合，所以也可以说这个函数有consistent 的性质。

算法的主要思路

A* 算法通过结合从起点到当前节点的实际路径长度（g 值）和从当前节点到目标节点的启发式估计距离（h 值）来工作。算法维护一个优先队列（开放列表），优先处理 g+h 值最小的节点，即认为最有可能达到终点的路径。

主要步骤：

- 初始化时，起点加入开放列表。

- 每次从开放列表中取出一个节点（当前代价最小的节点），对其进行扩展，即考察其所有可能的下一步移动。
- 对于每一个有效的移动（不越界且不是障碍物），计算新节点的 g+h 值，并更新路径信息。
- 如果到达终点，回溯路径并记录；否则，继续从开放列表中取节点扩展。
- 使用地图和节点信息储存结构来跟踪已访问和待访问的节点。

与一致代价搜索比较

一致代价搜索只考虑从起点到当前节点的实际路径长度（g 值），不使用启发式信息（h 值）。UCS 虽然可以保证找到从起点到任一节点的最小成本路径，但可能会探索更多的节点，因为它没有利用启发式信息指导搜索方向。

A* 方法的优化效果:
A* 搜索通过启发式信息有效地引导搜索方向，减少了搜索空间和时间。当启发式函数比较精确时，A* 的效率会很高。
相较于 UCS，A* 在大多数情况下可以更快地达到目标，因为它能跳过那些不太可能到达目标的路径（不像UCS会探索更多无用的节点）。在大地图或复杂环境中，比如这个Astar问题里，这个性质特别有效，因为节省的计算资源显著。

In conclusion，A* 搜索通过使用启发式函数，提供了一个既快速又能找到最优解的强大搜索方法。

Alpha-Beta

TODO 部分

alpha-beta剪枝过程：

```
...
if (isMaximizer) {
    int maxEval = INT_MIN;
    for (GameTreeNode* child : node.getChildren()) { // 遍历子节点
        int eval = alphaBeta(*child, alpha, beta, depth - 1, false); //
递归搜索
        maxEval = std::max(maxEval, eval); // 更新最大值
        alpha = std::max(alpha, eval);
        if (beta ≤ alpha) {
            break;
        }
    }
    return maxEval;
    } else {
        int minEval = INT_MAX;
```

```
        for (GameTreeNode* child : node.getChildren()) {
            int eval = alphaBeta(*child, alpha, beta, depth - 1, true);
            beta = std::min(beta, eval);
            if (beta ≤ alpha) {
                break;
            }
        }
        return minEval;
    }
...
```

alphaBeta(root,):

```
int depth = 3; // 搜索深度
int bestScore = alphaBeta(root, std::numeric_limits<int>::min(),
std::numeric_limits<int>::max(), depth, true);
std::cout << "Best score: " << bestScore << std::endl;
```

生成各个棋子的合法动作（只举了几个例子，具体可见完整代码）：

```
//马，要考虑拌马脚
// 检查"拌马脚"
int mx = x + dx[i] / 2;
int my = y + dy[i] / 2;
if (board[my][mx] ≠ '.') continue;  // 拌马脚，跳过
```

```
//炮，要考虑炮翻山吃子的情况
...
for (int i = 0; i < 4; ++i) {
        int nx = x + dx[i], ny = y + dy[i];
        // 跳过炮的位置
        while (nx ≥ 0 && nx < 9 && ny ≥ 0 && ny < 10 && board[ny][nx]
== '.') {
                Move cur_move;
                cur_move.init_x = x;
                cur_move.init_y = y;
                cur_move.next_x = nx;
                cur_move.next_y = ny;
                cur_move.score = 0;
                PaoMoves.push_back(cur_move);
                nx += dx[i];
                ny += dy[i];
        }
        // 跳过炮的位置
```

```
        if (nx ≥ 0 && nx < 9 && ny ≥ 0 && ny < 10) {
                nx += dx[i];
                ny += dy[i];
        }
        // 跳过炮的位置
        while (nx ≥ 0 && nx < 9 && ny ≥ 0 && ny < 10) {
                if (board[ny][nx] ≠ '.') {
                        bool cur_color = (board[ny][nx] ≥ 'A' &&
board[ny][nx] ≤ 'Z');
                        if (cur_color ≠ color) {
                                Move cur_move;
                                cur_move.init_x = x;
                                cur_move.init_y = y;
                                cur_move.next_x = nx;
                                cur_move.next_y = ny;
                                cur_move.score = 0;
                                PaoMoves.push_back(cur_move);
                        }
                        break;
                }
                nx += dx[i];
                ny += dy[i];
        }
}
...
```

```
//兵，需要分条件考虑，小兵在过楚河汉界之前只能前进，之后可以左右前
// 如果兵已经过河，可以左右前
if ((color && y < 5) || (!color && y ≥ 5)) {
        for (int i = 0; i < 3; ++i) {
                int nx = x + dx[i], ny = y + dy[i];
                if (nx ≥ 0 && nx < 9 && ny ≥ 0 && ny < 10) {
                        Move cur_move;
                        cur_move.init_x = x;
                        cur_move.init_y = y;
                        cur_move.next_x = nx;
                        cur_move.next_y = ny;
                        cur_move.score = 0;
                        BingMoves.push_back(cur_move);
                }
        }
} else { // 否则只能前进
        int nx = x, ny = y + dy[2];
        if (ny ≥ 0 && ny < 10) {
                Move cur_move;
                cur_move.init_x = x;
```

```
                cur_move.init_y = y;
                cur_move.next_x = nx;
                cur_move.next_y = ny;
                cur_move.score = 0;
                BingMoves.push_back(cur_move);
        }
}
```

终止：

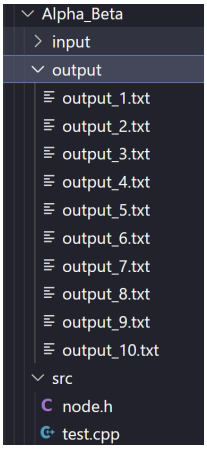
```
//终止判断
bool judgeTermination() {
        // 游戏结束条件：将被吃掉
        for (int i = 0; i < 9; ++i) {
                for (int j = 0; j < 10; ++j) {
                        if (board[j][i] == 'K' || board[j][i] == 'k') {
                                return false;
                        }
                }
        }
        return true;
}
```

棋盘评估：

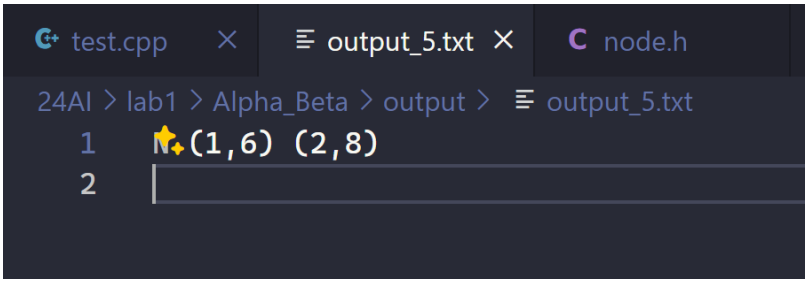
```
//棋盘分数评估，根据当前棋盘进行棋子价值和棋力评估，max玩家减去min玩家分数
int evaluateNode() {
        int score = 0;
        for (int i = 0; i < 9; ++i) {
                for (int j = 0; j < 10; ++j) {
                        if (board[j][i] ≠ '.') {
                                bool cur_color = (board[j][i] ≥ 'A' &&
board[j][i] ≤ 'Z');
                                int piece_value =
getPieceValue(board[j][i]);
                                score += cur_color ? piece_value : -
piece_value;
                        }
                }
        }
        return score;
}

// 计算棋子价值
int getPieceValue(char piece) {
```

```
        switch (piece) {
                case 'K': case 'k': return 10000;
                case 'A': case 'a': return 20;
                case 'B': case 'b': return 20;
                case 'N': case 'n': return 90;
                case 'R': case 'r': return 600;
                case 'C': case 'c': return 300;
                case 'P': case 'p': return 30;
                default: return 0;
        }
}
```

实验结果



其中举一个输出文件的例子:



alpha-beta剪枝对搜索效率的影响

不同深度时，处理所有输入，可以发现随着深度增加，耗时显著增加。

不采用剪枝时，取深度为3，测试搜索阶段总耗时比采用剪枝时更高的深度要慢许多，因此可以发现剪枝有着显著效果。

评估函数

评估函数设计核心在于通过棋子的价值来决定棋局的评估分数。

通过对不同棋子分配不同的数值，直接反映了每种棋子在游戏中的重要性和功能性。例如，将的价值最高，这反映了其在游戏中的核心地位；车和炮的价值较高，表明其在攻防

中的重要性。
在我的实现中，评估函数仅考虑棋子类型的固定价值，而未考虑棋子的具体位置。这种方法简化了评估。
通过将当前玩家的棋子价值加和，对手的棋子价值减和，直接计算出棋盘状态的总评分，其中正值表示对当前玩家有利，负值则相反。这种设计使评估结果直接支持极大极小搜索策略。