lab1 report

牛庆源 PB21111733

lab1.1

- 启发式函数依据地图的特性，定义为距离终点的曼哈顿距离即可：

```
int Heuristic_Function(int nowx, int nowy, int endx, int endy)
{
    return abs(nowx - endx) + abs(nowy - endy);
}
```

admissible：起点到终点如果没有其他条件，曼哈顿距离为最优，在该地图上增加补给站和不可走的地块后，路径长只会比曼哈顿距离长，而不会短。
consistent：$h(n) \leq c(n, n') + h(n')$ 成立。

- 主要思路：
  - 定义当前搜索状态：

```
struct Search_Cell
{
    int h;
    int g;
    // TODO: 定义搜索状态
    int x, y;    // 当前坐标
    int T; // 身上的补给
    Search_Cell *parent; // 父节点
};
```

  - 可以上下左右移动，使用两个数组分别描述上下左右移动时x方向和y方向的偏移量：

```
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};
```

  - 对于每一个方向生成的新节点，先判断节点坐标是否合法（越界，走到山上都是非法的），再赋值，然后判断是否有其他非法（走到该节点时如果食物为0，是否是终点）

```
int new_x = now_cell->x + dx[i];
```

```
int new_y = now_cell->y + dy[i];
// 判断新节点是否越界
if(new_x < 0 || new_x >= M || new_y < 0 || new_y >= N)
{
    continue;
}
// 判断新节点是否是障碍物
if(Map[new_x][new_y].type == 1)
{
    continue;
}
// 给新节点赋值
Search_Cell *new_cell = new Search_Cell;
new_cell->x = new_x;
new_cell->y = new_y;
new_cell->g = now_cell->g + 1;
new_cell->h = Heuristic_Funtion(new_x, new_y, end_point.first,
end_point.second);
if(Map[new_x][new_y].type == 2)
{
    new_cell->T = T;
}
else
{
    new_cell->T = now_cell->T - 1;
}
new_cell->parent = now_cell;

// 当食物为0时，如果当前不是终点
if(new_cell->T == 0 && Map[new_x][new_y].type != 4)
{
    continue;
}
```

- 如果成功走到了新节点，则判断节点是否在`open_list`和`close_list`中，分别处理不同情况。对于`open_list`的每一个节点，由于`open_list`是优先队列，可以将出队列第一个元素出队然后入栈到一个向量中判断它是否是这个节点，如果是，则将`in_open_list`赋值为`true`。然后将该元素重新入到队尾，并从向量中`pop`出去。

```
std::vector<Search_Cell *> temp_open_vec;
for(int i = 0; i < open_list.size(); i++)
{
    temp_open_vec.push_back(open_list.top());
    open_list.pop();
    if(temp_open_vec.front()->x == new_x && temp_open_vec.front()->y
== new_y)
    {
        in_open_list = true;
    }
    open_list.push(temp_open_vec.front());
    temp_open_vec.pop_back();
}
```

对`close_list`操作简单一些，直接遍历检查即可。

- 如果不在关闭列表和开启列表中，则直接`push`到`open_list`中。

  如果在开启列表且新的g值小，则找到这个点并在`open_list`中更新它，遍历方法同上给出的代码，只需要将`if`语句内的动作改为赋值新的`g`和`parent`即可。

  如果在关闭列表中且新的g值小，则找到这个节点并将其从关闭列表中删除`erase`并入队`push`开启列表。

  另外，如果新的节点在关闭列表中且有更大的T值，则加入开启列表。

```
// 根据T值判断是否更新
else if(in_close_list && !in_open_list)
{   // 在close_list中，且T值更大，加入open_list
    for(auto cell : close_list)
    {
        if(cell->x == new_x && cell->y == new_y && new_cell->T >
cell->T)
        {
            open_list.push(new_cell);
            break;
        }
    }
}
```

- 如果都不是，则删除掉这个生成的新节点。

- 如果到达了终点，即`now_cell`的横纵坐标和`end_point`的`first`与`second`相同，则生成路径，生成路径使用向量来储存`output_way`，`output_way`通过`Search_cell`中的父指针实现循环赋值，将每一个走过的节点赋值到`output_way`中，然后输出`way`和`step_nums`。

```
// 注意终止条件放在函数开始
if(now_cell->x == end_point.first && now_cell->y == end_point.second)
{
    // 到达终点
    // 生成路径
    std::vector<Search_Cell *> output_way;
    Search_Cell *temp = now_cell;
    int steps = 0;
    while(temp != nullptr)
    {
        output_way.push_back(temp);
        temp = temp->parent;
    }
    while(output_way.size() != 0)
    {
        way += to_string(output_way.back()->x) + " " +
to_string(output_way.back()->y) + "\n";
        output_way.pop_back();
        steps ++;
    }
    step_nums = steps - 1;
    break;
}
```

- 相较于已知代价搜索，加入了启发式函数，可以更好的求解结果。

lab1.2

- 对棋盘坐标使用了正常的x-y系统，仅在算分数时倒置。
- 马的拌马脚：分别使用`obstacle_x`和`obstacle_y`来表示障碍物所在位置，分别与马的下一步所走位置一一对应。然后加入`if`语句判断棋盘`board`中对每一个`nx`，`ny`的拌马脚位置是否有棋子，若有则跳过这一步。（以下代码简化了其他内容）

```
int obstacle_x[] = {1, 0, 0, -1, -1, 0, 0, 1};
int obstacle_y[] = {0, 1, 1, 0, 0, -1, -1, 0};
for(int i = 0; i < 8; i++)
{
    if (board[x + obstacle_x[i]][y + obstacle_y[i]] ≠ '.') continue;//拌马脚
}
```

- 炮的动作：对于四个方向的每一个方向，用布尔类型表示是否碰到第一颗棋子。
  如果没有棋子且没有碰到过第一颗棋子，则`push_back`进去；
  如果有棋子且是第一颗棋子，则置`first_node`为`false`；
  如果没有棋子且碰过第一颗棋子，则`continue`，该位置不能走；
  如果有棋子且是第二颗棋子且是对方的棋子，则吃掉棋子（即`push_back`进去），该方向遍历结束，`break`掉；
  如果有棋子且碰到的是第二颗棋子且是自己的棋子，则直接`break`掉。
  （以下是一个方向上的遍历，其他方向同理）

```
bool first_node = true; // 碰到第一颗棋子
for(int i = x + 1; i < sizeX; 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;
    bool cur_color = (board[i][y] ≥ 'A' && board[i][y] ≤ 'Z');
    if(board[i][y] == '.' && first_node)
    {   // 如果没有棋子且没有碰到第一颗棋子
        PaoMoves.push_back(cur_move);
    }
    else if (board[i][y] ≠ '.' && first_node)
    {   // 如果有棋子且是碰到的第一颗棋子
        first_node = false;
    }
    else if (board[i][y] == '.' && !first_node)
    {   // 如过没有棋子，且碰到过第一颗棋子
        continue;
    }
    else if (board[i][y] ≠ '.' && !first_node && cur_color ≠ color)
    {   // 如果有棋子且是碰到的第二颗棋子，且是对方的棋子
        PaoMoves.push_back(cur_move);
        break;
```

```
    }
    else if (board[i][y] ≠ '.' && !first_node && cur_color == color)
    {   // 如果有棋子且碰到的是第二颗棋子，且是自己的棋子
        break;
    }
}
```

- 相的动作：和马类似，注意规定两方相的活动范围即可。
- 士的动作：和相类似，注意规定两方士的活动范围。
- 将的动作：先考虑对将，不同颜色的将遍历不同方向（每一个遍历一个方向），遍历时碰到棋子就 `break` 掉，碰到对方将则 `push_back` 进去。其余走法和士类似，注意活动范围。

```
for(int i = x - 1; i ≥ 0; 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[i][y] ≠ '.' && board[i][y] ≠ 'k')
    {
        break;
    }
    if(board[i][y] == 'k')
    {   // 如果是对将
        JiangMoves.push_back(cur_move);
        break;
    }
}
```

- 兵的动作：分别对不同颜色方考虑是否过河，对过河前后分别进行处理，过河前只有一个方向，并且一定不会越界，过河后三个方向，可能越界所以考虑越界。以下是对红方的处理，黑方类似：

```
if(x ≥ 5)
{
    Move cur_move;
    int nx = x - 1;
    int ny = y;
    // 在过河之前只能前进，不会越界，不考虑越界情况
    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[nx][ny] ≠ '.') {
        bool cur_color = (board[nx][ny] ≥ 'A' && board[nx][ny] ≤ 'Z');
        if(cur_color ≠ color) {
            BingMoves.push_back(cur_move);
        }
    }
    else BingMoves.push_back(cur_move);
}
else if(x < 5 && x ≥ 0)
{
    int dx[] = {0, -1, 0};
    int dy[] = {1, 0, -1};
    for(int i = 0; i < 3; i++)
    {
        Move cur_move;
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (ny < 0 || ny ≥ 9 || nx < 0) continue; // 不会后退所以不考虑x过大
的越界
        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[nx][ny] ≠ '.') {
            bool cur_color = (board[nx][ny] ≥ 'A' && board[nx][ny] ≤
'Z');
            if(cur_color ≠ color) {
                BingMoves.push_back(cur_move);
            }
            continue;
        }
        else BingMoves.push_back(cur_move);
    }
}
```

- 终止判断：看将的个数，遍历整个棋盘，看将的个数，如果是两个则`return false`，如果不是两个则`return true`。
- 评估函数：用`minscore`和`maxscore`，遍历整个棋盘，对每一种棋子，为红方赋值`maxscore`，是为红方该棋子的`value`和该棋子所在`Position`的价值之和；

为黑方赋值`minscore`，同理。

对于每一个红方棋子的合法动作，`maxscore`加上吃掉其吃掉棋子的价值。

对于每一个黑方棋子的合法动作，`minscore`加上其吃掉棋子的价值。

返回`maxscore-minscore`。

- 根据当前棋盘和动作构建新棋盘（子节点）：棋盘下一个位置置为刚刚的位置，棋盘当前位置置为`.`。
- Alpha-Beta剪枝：先获取棋盘内容，然后用棋盘内容迭代剪枝。套用剪枝模版，注意剪枝条件即可。

```
int bestscore = 0;
int bestmove = 0;
if(isMaximizer)
{
    bestscore = std::numeric_limits<int>::min();
    for(int i = 0; i < moves.size(); i++)
    {
        bool cur_color = cur_board[moves[i].init_x][moves[i].init_y] ⩾
'A'
            && cur_board[moves[i].init_x][moves[i].init_y] ⩽ 'Z';
        if(cur_color ≠ isMaximizer) continue;
        GameTreeNode *child = node.updateBoard(cur_board, moves[i],
isMaximizer);
        if(child→getBoardClass().judgeTermination())
        {
            bestscore = std::numeric_limits<int>::max();
            bestmove = i;
            break;
        }
        int score = alphaBeta(*child, alpha, beta, depth - 1,
!isMaximizer);
        alpha = std::max(alpha, score);
        bestscore = std::max(bestscore, score);
        if(bestscore == score) bestmove = i;
        if(alpha ⩾ beta) break;
    }
}
else
{
    bestscore = std::numeric_limits<int>::max();
    for(int i = 0; i < moves.size(); i++)
    {
```

```
        bool cur_color = cur_board[moves[i].init_x][moves[i].init_y] ≥
'A'
            && cur_board[moves[i].init_x][moves[i].init_y] ≤ 'Z';
        if(cur_color ≠ isMaximizer) continue;
        GameTreeNode *child = node.updateBoard(cur_board, moves[i],
isMaximizer);
        if(child→getBoardClass().judgeTermination())
        {
            bestscore = std::numeric_limits<int>::min();
            break;
        }
        int score = alphaBeta(*child, alpha, beta, depth - 1,
!isMaximizer);
        beta = std::min(beta, score);
        bestscore = std::min(bestscore, score);
        if(alpha ≥ beta) break;
    }
}
```

在MaxDepth层输出即可。

main函数调用为：(红方先手)

```
alphaBeta(root, std::numeric_limits<int>::min(),
std::numeric_limits<int>::max(), MaxDepth, true);
```

- 剪枝可以有效避免不必要的路径选择，加快工作效率。