

第一次实验

AStar

启发式函数

在这次的实验中，使用的启发式函数为曼哈顿距离

```
// TODO: 定义启发式函数
int Heuristic_Function(Search_Cell *p, pair<int,int> end_point)
{
    // 定义为曼哈顿距离
    return abs(end_point.first - p->x) + abs(end_point.second - p->y);
}
```

Admissible

易知，从起点出发到达终点的最短距离为横纵坐标距离之和

所以最短路径的步数一定 \geq 曼哈顿距离

所以满足Admissible

Consistent

由于一次移动，曼哈顿距离最多减小1（一次移动代价为1），即 $h(u) \leq h(v) + c(u \rightarrow v)$ ，所以满足Consistent

算法主要思路

A*搜索主体如下，维护两个表

`open_list`存储已经搜索到但未拓展的节点，使用优先队列存储，按f排序

`close_list`存储已经探索完全的节点

其中每个节点为一个状态，用x, y, t唯一标识（将相同x, y, 不同t的节点视作不同状态）

由于启发式满足consistent，所以可以使用图搜索，即进入`close_list`的节点不会再放入`open_list`
整个搜索过程类似Dijkstra

```
void Astar_search(const string input_file, int &step_nums, string &way)
{
    priority_queue<Search_Cell *, vector<Search_Cell *>, CompareF> open_list;
    vector<Search_Cell *> close_list;
    open_list.push(search_cell);
    int flag=0;
    while(!open_list.empty())
    {
        // TODO: A*搜索过程实现
        Search_Cell* curCell = open_list.top();
        if((curCell->x == end_point.first)&&(curCell->y == end_point.second)){
            flag = 1;
            break;
        }
        else{
            open_list.pop();
            close_list.push_back(curCell);
            vector<Search_Cell*> reach= cellCanReach(curCell,Map,M,N,T,close_list,end_point);
            for(int i=0;i<reach.size();i++){
                Search_Cell* tmp;
                Search_Cell* tarCell= reach[i];
                if(!(tmp=isInPriorityQueue(open_list,reach[i]))){
                    open_list.push(tarCell);
                }
                else{
                    if(tmp->g > reach[i]->g){
                        tmp->g = reach[i]->g;
                        tmp->f = curCell;
                        delete tarCell;
                    }
                }
            }
        }
    }
}
```

You, 3周前 · Astar

cellCanReach函数用来判断当前节点可以到达的所有节点

```

vector<Search_Cell*> cellCanReach(Search_Cell *curCell, Map_Cell** Map,int M,int N, int T,vector<Search_Cell *> closeCell, pair<int,int> end_point)
{
    int x = curCell->x;
    int y = curCell->y;
    vector<Search_Cell *> reach;
    vector<pair<int,int>> directs;
    directs.push_back(make_pair(1, 0));
    directs.push_back(make_pair(0, 1));
    directs.push_back(make_pair(-1, 0));
    directs.push_back(make_pair(0, -1));
    for(int i=0;i<directs.size();i++){
        int t = curCell->t;
        pair<int,int> direct=directs[i];
        if((x+direct.first>=0)&&(x+direct.first<M)&&(y+direct.second>=0)&&(y+direct.second<N)&&(Map[x+direct.first][y+direct.second].type==1)){
            int type=Map[x+direct.first][y+direct.second].type;
            if(curCell->t <= 1 && type == 0){
                continue;
            }
            if(type == 2) t = T+1;
            if(!isInVector(closeCell,x+direct.first,y+direct.second,t-1)){
                Search_Cell* addCell = new Search_Cell;
                addCell->x = x+direct.first;
                addCell->y = y+direct.second;
                addCell->t = t-1;
                addCell->g = curCell->g+1;
                addCell->h = Heuristic_Funtion(addCell,end_point);
                addCell->f = curCell;
                reach.push_back(addCell);
            }
        }
    }
    return reach;
}
}

```

isInPriorityQueue函数用来判断待添加节点是否在open_list中，如果在，则更新open_list的对应节点，如果不在，则将其添加进open_list

```

Search_Cell* isInPriorityQueue(priority_queue<Search_Cell *, vector<Search_Cell *>, CompareF> que, Search_Cell * cell)
{
    while(!que.empty()){
        Search_Cell *tmpCell = que.top();
        if( isEqualCell(tmpCell,cell)){
            return tmpCell;
        }
        que.pop();
    }
    return NULL;
}

```

节点结构如下：

```

struct Search_Cell
{
    int h;
    int g;
    // TODO: 定义搜索状态
    // 需要在所在位置的横纵坐标x, y, 当前食物剩余量t, 父节点f
    int x;
    int y;
    int t;
    Search_Cell *f;

};


```

修改CompareF

```

struct CompareF {
    bool operator()(const Search_Cell *a, const Search_Cell *b) const {
        return (a->g + a->h)  $\geq$  (b->g + b->h); // 较小的 g + h 值优先级更高
    }
};


```

将

> 修改为 \geq ，使得在相同f的情况下，后进入open_list的节点位列队列前端，达到近似贪婪的效果

与一致代价搜索对比

使用A*搜索的总用时为0.5s

将启发式改为0，退化为一致代价搜索，360s没有返回结果

```

[Done] exited with code=0 in 0.479 seconds

[Running] cd "/Users/crush/Documents/exper/人工智能/lab1/Astar/src/" && g++ -std=c++11 Astar/src/"Astar

[Done] exited with code=null in 363.311 seconds

```

A*搜索使用了曼哈顿距离的启发式信息，使得每一次的节点选择更倾向于起点到终点的直线上的节点，而更少拓展不在直线上的节点。

一致代价搜索更倾向于拓展代价小的节点，所以会拓展距离起点近的节点，拓展大量无用节点。特别是input10，由于地图过于大，导致常规时间无法求得答案。

AlphaBeta

算法实现

alphaBeta具体实现如下：

- 当到达目标深度时，直接返回EvaluationScore
- 当未到达目标深度时，不断获取子节点并对子节点调用alphaBeta，并比较beta和maxEval的关系（对于maximizer），来判断是否剪枝，同时不断更新alpha和beta的值。

```
int alphaBeta(GameTreeNode *node, int alpha, int beta, int depth, bool isMaximizer) {
    if (depth == 0) {
        return node->getEvaluationScore();
    }
    You, 3周前 · add ai lab1
    //TODO alpha-beta剪枝过程

    if (isMaximizer) {
        int maxEval = std::numeric_limits<int>::min();
        for (int i = 0; i < node->getMoveSize(); i++) {
            int eval = alphaBeta((node->getChild()), alpha, beta, depth - 1, false);
            maxEval = std::max(maxEval, eval);
            node->setEvaluationScore(maxEval);
            alpha = std::max(alpha, eval);
            if (beta <= maxEval) {
                break;
            }
        }
        return maxEval;
    } else {
        int minEval = std::numeric_limits<int>::max();
        for (int i = 0; i < node->getMoveSize(); i++) {
            int eval = alphaBeta((node->getChild()), alpha, beta, depth - 1, true);
            minEval = std::min(minEval, eval);
            node->setEvaluationScore(minEval);
            beta = std::min(beta, eval);
            if (minEval <= alpha) {
                break;
            }
        }
        return minEval;
    }
}

return 0;
}
```

Node结构如下：

```

class GameTreeNode {
private:
    bool color; // 当前玩家类型, true为红色方、false为黑色方
    ChessBoard board; // 当前棋盘状态
    int step=0;
    std::vector<std::vector<char>> cur_board;
public:
    // 构造函数
    int evaluationScore; // 棋盘评估分数
    std::vector<GameTreeNode*> children; // 子节点列表
    std::vector<Move> moves;
}

```

其中board负责：存储棋局情况，计算所有可能的移动，评估棋局分数

Board结构如下：

```

class ChessBoard {
private:
    int sizeX, sizeY; // 棋盘大小, 固定
    std::vector<ChessPiece> pieces; // 棋盘上所有棋子
    std::vector<std::vector<char>> board; // 当前棋盘、二维数组表示
    std::set<char> red_next_eat; // 红方下一步可以吃的棋子
    std::set<char> black_next_eat; // 黑方下一步可以吃的棋子
    std::vector<Move> red_moves; // 红方棋子的合法动作
    std::vector<Move> black_moves; // 黑方棋子的合法动作
}

```

Moves的计算：

对于每个棋局，计算moves，只需要计算每个棋子的Moves，再汇总即可
在象棋中，每个棋子的移动方式不同，所以要分别对待

例如，马的可能移动如下：

```

void generateMaMoves(int x, int y, bool color) {
    // 便利所有可能动作, 筛选
    std::vector<Move> MaMoves;
    int dx[] = {2, 1, -1, -2, -2, -1, 1, 2};
    int dy[] = {1, 2, 2, 1, -1, -2, -2, -1};
    int px[] = {1, 0, 0, -1, -1, 0, 0, 1};
    int py[] = {0, 1, 1, 0, 0, -1, -1, 0}; // 马脚位置
    // 简化, 不考虑拌马脚
    // TODO 可以实现拌马脚过程
    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 || existChess(x + px[i], y + py[i])) continue; // 马脚位置有棋子或者超出棋盘
        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] != '.') {
            // 注意棋盘坐标系, 这里nx、ny相反是正确的
            bool cur_color = (board[ny][nx] >='A' && board[ny][nx] <='Z');
            if (cur_color != color) {
                if(color)
                    red_next_eat.insert(board[ny][nx]);
                else
                    black_next_eat.insert(board[ny][nx]);
            }
            MaMoves.push_back(cur_move);
        }
        continue;
    }
    MaMoves.push_back(cur_move);
}

```

```

        for (int i = 0; i < MaMoves.size(); i++) {
            if(color) {
                MaMoves[i].score = MaPosition[MaMoves[i].next_x][9 - MaMoves[i].next_y] - MaPosition[x][9 - y];
                red_moves.push_back(MaMoves[i]);
            }
            else {
                MaMoves[i].score = MaPosition[MaMoves[i].next_x][MaMoves[i].next_y] - MaPosition[x][y];
                black_moves.push_back(MaMoves[i]);
            }
        }
    }
}

```

效率影响

当depth为3时，使用alphaBeta，运行时间为**3s**，不使用alphaBeta，运行时间为**20s**

当depth为4时，使用alphaBeta，运行时间为18s

所以剪枝显著减少了运行时间，提升了效率

评估函数

评估函数分为三个部分：

1. 棋力评估：每个棋子在不同位置所获得的分数

```

// 棋力评估，这里的棋盘方向和输入棋盘方向不同，在使用时需要仔细
// 生成合法动作代码部分已经使用，经过测试是正确的，大家可以参考
std::vector<std::vector<int>> JiangPosition = {
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 12 rows
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, -8, -9, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {5, -8, -9, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, -8, -9, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
};

std::vector<std::vector<int>> ShiPosition = {
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // 12 rows
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
};

```

2. 行棋可能性评估：下一步的可能动作所获得的分数

```
// 行期可能性评估，这里更多是对下一步动作的评估
std::map<std::string, int> next_move_values = {
    {"K", 9999},
    {"N", 100},
    {"R", 500},
    {"C", 100},
    {"P", -20},
    {"A", 0},
    {"B", 0},
    {"k", 9999},
    {"n", 100},
    {"r", 500},
    {"c", 100},
    {"p", -20},
    {"a", 0},
    {"b", 0}
};
```

3. 棋子价值评估：各棋子的固定价值

```
// 棋子价值评估
std::map<std::string, int> piece_values = {
    {"K", 20000},
    {"A", 10},
    {"B", 30},
    {"N", 300},
    {"R", 500},
    {"C", 300},
    {"P", 90},
    {"k", 20000},
    {"a", 10},
    {"b", 30},
    {"n", 300},
    {"r", 500},
    {"c", 300},
    {"p", 90},
};
```

在统计Moves的同时，我们也可以统计移动可以吃的棋子和移动前后棋力差
前者可以在我们计算评估分数时，帮助计算行棋可能性分数
后者可能作为Moves排序的依据，让棋力提升大的子节点先被拓展，协助我们剪枝

```

void generateMaMoves(int x, int y, bool color) {
    // 便利所有可能动作，筛选
    std::vector<Move> MaMoves;
    int dx[] = {2, 1, -1, -2, -2, -1, 1, 2};
    int dy[] = {1, 2, 2, 1, -1, -2, -2, -1};
    int px[] = {1, 0, 0, -1, -1, 0, 0, 1}; // 马脚位置
    int py[] = {0, 1, 1, 0, 0, -1, -1, 0}; // 马脚位置
    // 简化，不考虑拌马脚
    // TODO 可以实现拌马脚过程
    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 || existChess(x + px[i], y + py[i])) continue; // 马脚位置有棋子或者超出棋盘
        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] != '.') {
            // 注意棋盘坐标系，这里nx、ny相反是正确的
            bool cur_color = (board[ny][nx] >= 'A' && board[ny][nx] <= 'Z');
            if (cur_color != color) {
                if(color)
                    red_next_eat.insert(board[ny][nx]);
                else
                    black_next_eat.insert(board[ny][nx]);
            }
            MaMoves.push_back(cur_move);
        }
        continue;
    }
    MaMoves.push_back(cur_move);
}

```

```

for (int i = 0; i < MaMoves.size(); i++) {
    if(color) {
        MaMoves[i].score = MaPosition[MaMoves[i].next_x][9 - MaMoves[i].next_y] - MaPosition[x][9 - y];
        red_moves.push_back(MaMoves[i]);
    }
    else {
        MaMoves[i].score = MaPosition[MaMoves[i].next_x][MaMoves[i].next_y] - MaPosition[x][y];
        black_moves.push_back(MaMoves[i]);
    }
}

```

如上图，使用两个set, red_next_eat, black_next_eat分别统计红黑两方下一步可以吃的棋子种类。

并使用MaMoves[i].score存储移动前后棋力差，作为拓展顺序的依据

最终在evaluateNode函数获得评估分数

效果

对于input1，给出的output为R (1,1) (4,1)，评估分为9930

该Move可以将黑方将一步将死，为正解



对于input2, output为R (3,5) (3,0), 评估分数为9561, 也是正解



说明在三种评估方式的共同作用下，对棋局的判断很准确