AI lab1

PB21051012 刘祥辉

1. Astar

算法思路

算法伪代码如下：

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

`frontier` 是一个优先级队列，用于存储待探索的节点，节点的优先级按照启发式函数的值确定。`came_from` 字典用于记录每个节点的前一个节点，构成路径。`cost_so_far` 字典用于记录从起始节点到当前节点的代价。在每次迭代中，从 `frontier` 中取出优先级最高的节点作为当前节点，然后遍历当前节点的邻居节点，更新它们的代价和优先级，并将它们加入 `frontier` 中。搜索直到找到终点或者 `frontier` 为空。

实现细节

这段代码实现了A*搜索算法，用于寻找从起点到终点的最佳路径。下面是代码实现细节的分条说明：

1. **循环遍历直到开放列表为空：**

```
cppCopy codewhile(!open_list.empty())
{
    // A*搜索过程实现
}
```

2. **从开放列表中取出优先级最高的节点作为当前节点：**

```
cppCopy codeSearch_Cell * current;
current = open_list.top();
open_list.pop();
```

3. **检查当前节点是否为终点：**

```
cppCopy codeif (current->x == end_point.first && current->y ==
end_point.second){
    break;
}
```

4. **检查当前节点的上下左右四个邻居节点：**

```
cppCopy code// 检查上方邻居节点
if( current->x - 1 >= 0 && Map[current->x-1][current->y].type != 1){
    // 实现细节...
}

// 检查下方邻居节点
if( current->x + 1 < M  && Map[current->x+1][current->y].type != 1){
    // 实现细节...
}

// 检查左方邻居节点
if( current->y - 1 >= 0 && Map[current->x][current->y-1].type != 1){
    // 实现细节...
}

// 检查右方邻居节点
if( current->y + 1 < N && Map[current->x][current->y+1].type != 1){
    // 实现细节...
}
```

5. **对于每个邻居节点，计算新的代价并更新信息：**

```
cppCopy code// 计算新的代价
int new_cost = current->g + 1;

// 检查是否已经遍历过当前邻居节点，并且计算新的代价是否更小
bool visited = false;  // 是否已经遍历
bool dis_less = false;
Search_Cell *new_search;
for(auto &c : close_list)
{
    if(c->x == current->x - 1 && c->y == current->y)
    {
        new_search = c;
        visited = true;
        if(c->g > new_cost)
            dis_less = true;
        break;
    }
}

// 如果当前节点的食物数不为零，或者邻居节点是食物，则继续搜索
if(!(current->food <= 0 && cur_map->type!=2)){
    // 如果邻居节点未被遍历过，或者新路径的代价更小，则更新邻居节点的信息
    if(!visited || dis_less){
        if(!visited)  {
            new_search = new Search_Cell;
```

```
            new_search->x = current->x-1;
            new_search->y = current->y;
            new_search->food = current->food;
        }
        new_search->g = new_cost;
        new_search->h = Heuristic_Funtion(new_search->x,new_search-
>y,end_point);  // 计算启发式函数的值
        if(cur_map->type==2) new_search->food = T;  // 如果邻居节点是食物，更新食
物数
        else    new_search->food -= 1;
        if(!visited) close_list.push_back(new_search);  // 将邻居节点加入已遍历
节点列表
        open_list.push(new_search);  // 将邻居节点加入优先级队列
        auto pre_pair = make_pair(new_search->x,new_search->y);
        come_from[pre_pair] = make_pair(current->x,current->y);  // 记录邻居节
点的前一个节点
    }
}
```

6. **重复步骤4和步骤5，直到找到终点或者优先级队列为空。**

启发式函数

启发式函数采用到终点的曼哈顿距离。

可采纳性证明

曼哈顿距离是不考虑补给、不可通行处等因素时的路径长度，实际代价要考虑这些因素，因此启发式函数值一定小于实际代价，是可采纳的。

一致性讨论

在曼哈顿距离意义下, 每一步最多减少 1, 而路径长度也为 1, 因此是一致的

与一致代价搜索比较

在启发式函数总为 0 时，相当于是Dijkstra算法，处理十个输入的总步数为：10945

在采用曼哈顿距离后，总步数为：6149，故能有效降低搜索次数。

2. alpha-beta剪枝

评估函数

采用框架中给定的数据，根据当前棋盘进行棋子价值和棋力评估。

alpha-beta剪枝

代码如下：

```
int32_t min_max(ChessBoard& cb, int32_t searchDepth, int32_t alpha, int32_t beta,
PlaySide side){
    if (searchDepth == 0){
        return evaluateNode(cb);
    }
```

```
    int32_t minMaxValue;
    if (side == BLACK){
        int32_t minValue = std::numeric_limits<int32_t>::max();
        std::vector<Move> possibleMoves = gen_possible_moves(cb, BLACK);

        for (const Move& node : possibleMoves) {
            cb.move(node);
            minMaxValue = min_max(cb, searchDepth - 1, alpha, beta, RED);
            minValue = std::min(minValue, minMaxValue);
            cb.undo();

            beta = std::min(beta, minValue);
            if (alpha >= beta){
                break;
            }
        }

        return minValue;
    }
    else if (side == RED){
        int32_t maxValue = std::numeric_limits<int32_t>::min();
        std::vector<Move> possibleMoves = gen_possible_moves(cb, RED);

        for (const Move& node : possibleMoves) {
            cb.move(node);
            minMaxValue = min_max(cb, searchDepth - 1, alpha, beta, BLACK);
            maxValue = std::max(maxValue, minMaxValue);
            cb.undo();

            alpha = std::max(alpha, maxValue);
            if (alpha >= beta){
                break;
            }
        }

        return maxValue;
    }
    else return 0;
}
```

实现细节

1. 棋盘类中增添`void set(*int32_t* *r*, *int32_t* *c*, char *p*)`和`void undo()`函数用于设定棋子和取消棋子，可以不用申请新的棋盘类。
2. 考虑了绊马腿和挤象眼的情况。
3. ChessBoard类删去了不必要的数据。

实验结果

以下为搜索深度为5时候的结果

```
R(1,1) (1,4)
R(5,3) (0,3)
P(3,5) (3,6)
R(8,1) (0,1)
N(3,1) (2,3)
C(9,2) (1,2)
P(3,4) (2,4)
R(2,4) (2,6)
C(1,4) (3,4)
P(6,8) (5,8)
```

剪枝的影响

采用剪枝，深度为4的总耗时：`Total time for all AI moves: 10791 milliseconds.`

不采用剪枝时，深度为4，测试搜索阶段总耗时：

`Total time for all AI moves: 243520 milliseconds.`

增长了24倍左右，剪枝效果显著。

评估函数的设计

关于评估函数, 主要由以下三部分构成:

- 棋力评估
- 行棋可能性评估
- 棋子价值评估
  同时可以考虑多个棋子联合站位对于棋盘局势的影响，例如连环炮，连环马，车马炮等特殊组合