HW1：Bait游戏实验报告

匡亚明学院 张子谦 191240076 191240076@smail.nju.edu.cn

本人承诺该实验全程由本人独立自主完成，无抄袭或给予他人抄袭行为，代码已上传至本人github

摘要：Bait游戏是一个通过操纵精灵获得钥匙并返回出口从而获胜的小游戏。为获得钥匙，精灵必须通过策略性的推动箱子填洞开路。填洞的同时可以获得一定的加分。本实验在已有的代码框架下，针对第一个关卡设计实现了深度优先搜索与深度受限搜索算法，并在此基础上设计实现A*算法，观察三种算法在游戏中的实现情况。最后阅读蒙特卡洛树搜索的控制程序代码实现，进一步理解其算法本质。

1. 引言

如图1所示，将每一点的状态抽象成节点，Bait游戏便是一个经典的路径搜索模型，本实验拟以深度优先搜索，深度受限搜索，A*，蒙特卡洛树搜索四种树搜索算法为例，对四种算法如何搜素成功路径进行阐述。

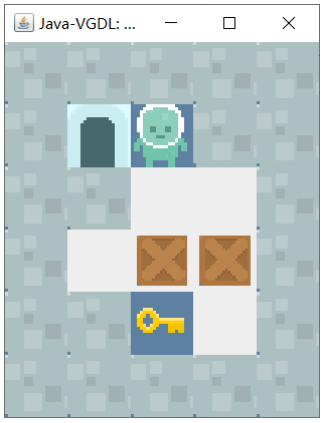

Fig1

2. 实验内容

Task1: 深度优先搜索

全局变量

变量名	变量含义
tag	是否搜索到成功路径
visited	历史路径
actions	从初状态到游戏胜利的每一步动作

依照实验要求，在游戏一开始就使用深度优先搜索找到成功的路径通关。

首先判断 $actions.size()$ 是否为0，若为0，则调用 DFS 函数进行深度优先搜索，将从初状态到游戏胜利的每一步都存储入$actions$，反之将对应的动作返回。

深度优先搜索DFS伪代码如下：

```
public boolean DFS(StateObservation stateObs,ElapsedCpuTimer elapsedTimer) {
        if(visited before) {
            return false;
```

```
        }
        if(win) {
            return true;
        }
        if(Lose) {
            return false;
        }
        visited.add(stateObs);
        stCopy = stateObs.copy();
        for(thisact in availableActions) {
            stCopy.advance(thisact);
            if(DFS(stCopy,elapsedTimer)) {
                actions.add(thisact);
                return true;
            }
            else {
                refresh stCopy;
                continue;
            }
        }
        return false;
    }
```

通过递归调用，判断采取 $thisact$ 时能否搜索成功，如成功，将其加入到 $actions$ 当中，反之，复原局面状态，从 $availableActions$ 中寻找另外的动作尝试，直至搜索到成功路径。显然成功路径不一定最优，该算法能通过1，2，3关的测试，不过在2，3关中有不少冗余的行为操作。

Task2 深度受限搜索

不同于Task1中，在一开始就搜索完，在受限的时间内，每一步进行一次深度搜索，搜索到一定的深度，通过启发式函数判断局面好坏。考虑到游戏本身性质，我将整个游戏流程划分为：找钥匙，回出口两部分；在未找到钥匙的情况下，找到钥匙为当前的第一准则。

主要全局变量

变量名	变量含义
visited	本次搜索中访问过的statement
visitedDepth	本次搜素中访问过的statement对应的搜索层数
actions	从初状态到搜索到最佳状态的每一步动作
depthLevel	搜索层数
bestScore	本次搜索局面评分
wholeVisited	游戏初状态到结束真实行动的访问路径
getkey	对于整个游戏过程是否已经取到钥匙
finalAct	本次搜索采取的act

每次进行搜索前，将对应的变量 $clear$，然后判断是否取到钥匙。接下来进行深度受限搜索，返回对应的行动。

深度受限搜索 $recursiveDFS$ 伪代码如下：

```
public boolean recursiveDFS(StateObservation stateObs, ElapsedCpuTimer
elapsedTimer,int depth) {
        if(depth!=0&&visited before) {
            return false;
        }
        //将对应的statement和搜索层数存储
        visited.add(stateObs);
        visitedDepth.add(depth);
        if(Lose) {//如果游戏失败
            return false;
        }
        if(Win) {
            refresh bestScore and finalAct;
            return true;
        }
        if(!getkey&&在本步取得钥匙) {
            if(bestScore>depth) {
                refresh bestScore and finalAct;
                return true;
            }
        }
        if(depth==depthLevel) {
            score = heuristic(stateObs);
            if(score<bestScore) {//如果当前行动更优
                refresh bestScore and finalAct;
                return true;
            }
            else
                return false;
        }

        else {
            stCopy = stateObs.copy();
            get availableActions;
            for(thisact in availableActions) {
                if(depth==0) {
                    clear actions
                }
                stCopy = stateObs.copy();
                actions.add(thisact);
                stCopy.advance(thisact);
                if(!recursiveDFS(stCopy,elapsedTimer,depth+1)) {
                    remove newact from actions and refresh stCopy;
                    continue;
                }
                else {
                    continue;
                }
            }
        }
        return true;
    }
```

如上文所述，将找钥匙与回目标两部分割裂开来，所以对于 $getkey$ 是 $false$ 的情况，只要这一步找到钥匙，就不会向下拓展（但会考虑是否有搜索层数更少找到钥匙的情况）。

所以对于启发式函数的设计，采取：如未找到钥匙，则定义为精灵与钥匙的曼哈顿距离，如已找到，则定义为精灵与目标的曼哈顿距离。

该算法虽然可以通过第一关的测试，但当其应用于之后的关卡时，对于第二关必须调整 $depthLevel$ 参数才可能找到，但此时又会超出时间限制，总体实现效果并不理想。

Task3 A*算法

与深度受限搜索类似，无法一次完全搜完，需要在受限时间内给出步骤。这里参考了蒙特卡洛树搜索实现中的部分框架，设定只要在规定时间之内，就进行一步A*搜索，直至时间耗尽。返回act。对于下一次给出的新的statement，更新大部分全局变量，以该statement为起点再进行A*搜索

为了方便回溯，在 $A^*controller$ 块中定义了 $ScoreAndState$ 类用于描述某一结点的 $Statement$，$evaluation$, $fathernode$ 以及从父节点到这一步采取的 act

主要全局变量

变量名	变量含义
closeList	本次搜索走过的路径
openList	本次搜索expand但未走过的路径
realList	整个游戏过程走过的路径
nowState	本次搜索中每执行一步A*算法后实时更新的当前状态
getkey	整个游戏过程中是否已经取到钥匙

A*算法每次从openList中取出一个evaluation最佳的节点并visit，将其作为nowState并加入closeList，并对nowState的子节点进行拓展，分别计算它们的evaluation值，加入到openList中。如此反复直至终点。

act 以及 A^* 的部分伪代码如下：

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer
elapsedTimer){
        Init some variables;
        if(getKey(stateObs)) {
            getkey = true;
        }
        get stCopy;
        nowState = new ScoreAndState(stCopy,0,10000,null);
        openList.add(nowState);
        realList.add(stCopy);
        action = null;
        ...
        while(time enough){
            if(Win)
                break;
            if(!getkey&&now get the key)
                break;
            Astar(nowState);//执行一步A*算法
            refresh time;
            }
        action = FindAct();
        return action;
    }
```

```
public void Astar(ScoreAndState fatherState) {
        //每次Astar在openList找到score最好的状态，然后拓展，更新
        set bestScore MAX;
        get thisState from openList according to evaluation;
        nowState = thisState;
        if(Win)
            return;
        if(!getkey&&now get the key)
            return;
        openList.remove(thisState_id);
        closeList.add(thisState);
        //探寻新路径更新并加入openList
        get stCopy and availableActions;
        for(thisact in availableActions) {
            refresh stCopy;
            stCopy.advance(thisact);
            if(Loses||visited before) {//如果输了或者已经访问过
                continue;
            }
            if(Win) {
                Add thisStatement to openList with HIGH EVALUATION;
            }
            else if(expand before) {
                //如果此时的stateObservation在openList出现过，就要判断是否更新
                if(now evaluation is better) {
                    refresh the evaluation and fathernode of statement in
openList;
                }
            }
            else {
                add this statement to openList;
            }
        }
    }
```

A*算法本身的框架并不困难，但是问题在于启发式函数的设计。即 **evaluation function** $f(n) = h(n) + g(n)$ 中 $h(n)$ 部分，其中 $g(n)$ 表示从初始状态到状态 n 的路径耗散。这里仍然采用了取得钥匙前为钥匙与精灵的曼哈顿距离，取得钥匙后，目标与精灵的曼哈顿距离作为 $h(n)$ 的值。虽然次情况下能保证其可采纳性，但是在遇到第二关第三关的洞堵住取钥匙的路的情况，就难免出现 $h(n)$ 与 $h^*(n)$出入较大的情况。因为A^*算法的本质仍然是一种遍历，因此这时会访问很多 $useless$ 的节点，浪费了大量的时间，在100ms的限制下很难做出正确的响应。但在修改时间限制为较大值后测试发现能寻找到最优成功路径，所以算法本身正确性可以保证，但在启发式函数设计的优劣性上显示出明显的不足，在尝试对启发式函数进行一系列修改后，会出现异常（对游戏框架细节的认识存在不足），因此选择了暂时保留原启发式函数。

Task4 MCTS

MCTS算法的设计思路可以概括为以下四个过程：

1. Selection
2. Expansion
3. Simulation
4. Backpropagation

由 **SingleTreeNode.java** 中的 **mctSearch** 方法实现

首先，通过TreePolicy函数进行Selection与Expansion。采用UCB公式作为选择策略，对未完全展开的节点进一步拓展，并返回UCB最优的节点作为此次模拟的节点。其中向上，向下，向左，向右四个动作就类似于Bandits中的arm。

接下来，通过rollout函数进行Simulation，对选择的节点向下随机搜索直至finish，然后判断此时的value值更新bound。

最后，通过backUp函数进行Backpropagation，从子节点开始，沿刚刚向下的路径回头并沿途更新父节点的统计信息。

在时间耗尽前，重复上述步骤，最后返回访问次数最多的动作作为结果（即返回Bandits中roll的次数最多的arm）。

补充：为了不让摇臂的排列顺序决定摇臂的选择，引入了误差量 ϵ ，与课程讲解内容相一致。

3. 结束语

从四种搜索策略的实验结果来看，作为最简单的深度优先搜索算法，其在不复杂的地图下能搜索到成功路径，但面对地形复杂的地图（如第四关，第五关）由于搜索量过大，时间复杂度过高，且往往不是最优解，所以不总是令人满意。而对于深度受限搜索与A*算法，由于边行动边搜索的特点，在时间限制严格的情况下，对于局面打分的启发式函数设计要求往往更严格，简单的曼哈顿距离在需要推箱子堵洞的情况下会带来许多冗余的步骤，耗费大量的时间，如果可以设计判断当钥匙四周存在必须填补的洞时，优先推动箱子堵洞，并让精灵禁止将箱子推至角落（避免"自杀"行为），或许能够取得更好的效果。而MCTS从算法本身来讲，会有更好的实现效果，不过在尝试测试结果并不如人意，原因有待进一步探究。

References：课件 Lecture 3,5

https://blog.csdn.net/qq_16137569/article/details/83543641

https://www.jianshu.com/p/a8950fa19b72

注：介绍程序时的格式参考了https://blog.csdn.net/justice0/article/details/79688661并已征得作者（学长）本人同意。