实验 1:Bait 游戏

王琛然（151220104、17721502736@163.com）

(南京大学 计算机科学与技术系, 南京 210093)

摘 要: 本次实验使用了 GVG-AI，是为了通用人工智能的研究开发的游戏框架，基于 VGDL（视觉游戏描述语言），能够构成多种游戏。在 Bait 游戏中，共有深度优先搜索算法、深度受限的深度优先搜索算法、A*算法和 MCTS 算法（蒙特卡洛树算法）4 中算法实现自主搜索路径。前三种需自主实现，下文会进行详细介绍；第四种算法在框架中已实现，下文会进行分析。

关键词: 深度优先搜索、深度受限的深度优先搜索、A*算法、MCTS 算法

中图法分类号: TP301 **文献标识码**: A

1 引言

本次游戏 Bait 游戏是一款推箱子游戏，基于 VGDL 实现，游戏规则为：精灵需要先拿到钥匙，然后走到目标；如果精灵吃了蘑菇，则额外加 1 分；精灵不能掉进洞里，否则游戏失败；精灵可以推箱子把洞填上并通过，每填上一个洞会有 1 分奖励；一次只能推一个箱子，蘑菇是不可移动的物体，箱子不可以推到蘑菇上，但可以覆盖钥匙；游戏的时间为 1000ticket，时间结束即失败。

实验中共有 4 个任务：

1. 针对第一个关卡实现深度优先搜索，在游戏一开始就使用深度优先搜索找到成功的路径通关，记录下路径，并在之后每一步按照路径执行动作；
2. 在任务 1 的基础上实现深度受限的深度优先搜索，修改为每一步进行一次深度搜索，但这时不需要一定搜索到通关 ，而是搜索到一定的深度，再设计一个启发式函数判断局面好坏；
3. 在任务 2 的基础上，将深度优先搜索换成 A*算法
4. 阅读并介绍 MCTS 算法

该文的第二部分会介绍 4 种算法，第三部分会展示实验结果

2 算法介绍

2.1 深度优先搜索

深度优先搜索是一种常用的图算法，目的是要达到搜索结构的叶结点，采用递归的方式，对图进行搜索，搜索过程中每个结点仅访问一次。在本次实验中，需要先进行深度优先搜索算法搜索出所有的路径并执行，核心思想在是将每一个可以走的路径存入数组，在当前状态搜索所有可以执行的动作，用 stCopy 来模拟动作的执行，判断是否与当前已走的状态相同或遇到不可移动的物体，若存在，则进行下一个动作的搜索；若不存在，则加入路径并更新状态，对新状态进行深度优先搜索。当目前状态的所有可执行动作均执行完毕并没有到达游戏结束状态，则回退到上一状态，删除动作数组中对应的数组。

Agent 类定义的变量：

states 数组是所有的局面状态集合，actions 数组是最终执行的动作集合，finish 来判断游戏是否结束，step 是执行的步数，用来选择返回的动作。

```
protected ArrayList<StateObservation> states = new ArrayList<>(); //all of the states in this game
protected ArrayList<Types.ACTIONS> actions = new ArrayList<>(); //the selected actions
boolean finish; // the state of game state - 0: not finish 1: finish
int step;
```

Agent 类中的 act 函数来执行动作：

在第一次进入 act 函数时进行深度优先搜索，搜索完成后所得结果会存在 actions 数组中，之后对于每一步的执行由下标 step 来选择并返回该动作。

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
    if(step == 0)
        dfs(stateObs);
    step++;
    //System.out.println(count);
    //System.out.println(actions.size());
    try {
        Thread.sleep( millis: 500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return actions.get(step - 1);
}
```

DFS 算法：

getAvailableActions()函数获得当前状态可以执行的动作集合，在当前可以执行的动作集合中循环，stCopy.advance 模拟动作的进行，用 equalPosition()函数来判断是否与之前的状态有相同来避免回路，并判断墙等不可移动物体，获得可以移动的动作。如果没有回路，则将当前动作加入最终执行动作集合，更新状态并进行新状态的深度优先搜索。

当从循环中跳出时，若游戏没有结束，则表明所选的动作不能够结束游戏，则需要回退到上一状态，并将最终动作集合中的该动作和状态集合中的该状态删除，退出当前递归。

```
public void dfs(StateObservation so){
    //ArrayList<Observation>[] movingPositions = so.getMovablePositions();
    //System.out.print("\n");
    //System.out.print(movingPositions[0].size());
    ArrayList<Types.ACTIONS> currentActions = so.getAvailableActions(); //all of the available actions in current state
    states.add(so); //add current state
    for(int i =0; i < currentActions.size(); i++) {
        boolean flag = true;
        if (finish)
            break;
        StateObservation stCopy = so.copy();
        stCopy.advance(currentActions.get(i));
        for (int j = 0; j < states.size(); j++) {
            if (stCopy.equalPosition(states.get(j))) { //avoid retreats
                flag = false;
                break;
            }
        }
        if (stCopy.getGameWinner() == Types.WINNER.PLAYER_LOSES)
            flag = false;
        else if (stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS)
            finish = true;

        if (flag) {
            actions.add(currentActions.get(i));
            dfs(stCopy);
        }
    }
    //the simulative action is not right, back to last state
    if(!finish) {
        if(!actions.isEmpty() && !states.isEmpty()) {
            actions.remove( index: actions.size() - 1);
            states.remove( index: states.size() - 1);
        }
    }
}
```

2.2 深度受限的深度优先搜索

深度受限的深度优先搜索是在深度优先搜索的基础上改进的一种算法，思想基本相似，不同之处是不同于深度优先搜索一直进行递归搜索，而是设定一个最大深度 limit_depth，当搜索深度到达 limit_depth 时，执行启发函数，通过启发函数返回的数值，判断当前局面是否最优，若最优，则将最终的动作集合更新

Agent 类变量：

设定最大深度为 5，actions 为所有的动作集合，states 是所有的状态局面集合，由于该算法不需要搜索到最终结果，而是根据启发函数选择最优，则 bestActions 是在未走到终点时，最好的动作集合。cost 是启发式函数的返回值

```
public final static int LIMIT_DEPTH = 5;
protected ArrayList<Types.ACTIONS> actions = new ArrayList<>();
protected ArrayList<StateObservation> states = new ArrayList<>();
protected ArrayList<Types.ACTIONS> bestActions = new ArrayList<>();
int depth;
int step;
boolean finish;
double cost;
```

Agent 类中的 act 函数：

cost 初始化为 10000，当游戏没有结束时，根据启发式函数的返回值选择最优动作并执行；当游戏结束时，则与深度优先搜索相同，由下标 step 在 actions 动作集合中选择合适的动作

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
    ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
    //System.out.print(movingPositions[0].size());
    cost = 10000;
    if(finish == false){ //don't search the result, just search to limit depth
        states.clear();
        actions.clear();
        bestActions .clear();
        ldfs(stateObs,  depth: 0);
        //System.out.println(actions.size());
        //System.out.println(bestActions .size());
        //System.out.println(bestActions .get(0));

        try {
            Thread.sleep( millis: 500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return bestActions.get(0);
    }
    else
        step++;
    try {
        Thread.sleep( millis: 500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return actions.get(step - 1);
}
```

启发函数：

首先获取精灵、钥匙和门的位置，根据精灵的状态进行不同计算，没有钥匙时，代价 judge 的值为到钥匙的距离与到门的距离之和；有钥匙的时候，judge 的值为到门的距离

```
public double heuristic(StateObservation stateObs){
    double judge;
    ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
    Vector2d goalpos = fixedPositions[1].get(0).position; //目标的坐标
    Vector2d keypos = movingPositions[0].get(0).position; //钥匙的坐标
    Vector2d currentpos = stateObs.getAvatarPosition();
    System.out.print(movingPositions[0].size());
    System.out.print("\n");
    if(movingPositions[0].size() == 1) { //without key, the judge is the distance from goal and key
        judge = currentpos.dist(keypos) + currentpos.dist(goalpos);
    }
    else //with key, the judge is the distance from goal
        judge = currentpos.dist(goalpos);
    return judge;
}
```

深度受限的深度优先搜索算法：

基本步骤同 DFS 算法相同，不同之处：当精灵到达门的时候，即成功状态，除了将 finish 的值改为 true，还需将当前的 bestActions 更新为 actions 数组；若执行的动作是合法的但没有到达深度，则将该动作加入到动作集合，并对新状态进行更深一层的搜索。若该执行的动作是合法的且到达搜索限定深度时，调用启发函数，选取代价较小的动作集合并更新 bestActions；

```
public void ldfs(StateObservation so, int depth){
    ArrayList<Types.ACTIONS> currentActions = so.getAvailableActions();
    states.add(so);
    for(int i = 0; i < currentActions.size(); i++){
        boolean flag = true;
        StateObservation stCopy = so.copy();
        stCopy.advance(currentActions.get(i));
        for(int j = 0; j < states.size(); j++){
            if(stCopy.equalPosition(states.get(j))){ //avoid retreats
                flag = false;
                break;
            }
        }
        if(stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS){
            //System.out.print("finish");
            //System.out.print(finish);
            finish = true;
            actions.add(currentActions.get(i));
            bestActions .clear();
            for(int j = 0; j<actions.size(); j++)
                bestActions.add(actions.get(j));
            break;
        }
        if(flag == true && depth == LIMIT_DEPTH){ //achieve the limit depth, find the best in current states
            //System.out.print("limit depth\n");
            if(heuristic(stCopy) < cost){
                cost = heuristic(stCopy);
                bestActions .clear();
                for(int j = 0; j < actions.size(); j++)
                    bestActions.add(actions.get(j));
            }
            flag = false;
        }

        //System.out.print(flag);
        if(flag){
            actions.add(currentActions.get(i));
            ldfs(stCopy, depth: depth + 1);
        }
    }
    }
    if(!finish) {
        if(!actions.isEmpty() && !states.isEmpty()) {
            actions.remove( index: actions.size() - 1);
            states.remove( index: states.size() - 1);
        }
    }
```

2.3 A*算法

A*算法是一种求解最短路径效率较高的直接搜索算法，也是一个启发式算法，它利用对起始点周围的可行步骤的评估值，选择最优解作为下一步骤，并且把当前节点设为下一步骤的父节点,以此依据搜索到目标点后，根据父节点倒序就能找到最优路径。

代价计算时，除了计算现在状态到目标状态的代价 cost1 之外，还需要计算从初始状态到现在状态的代价 cost2，总代价 cost = cost1 + cost2 越小的路径表示当前位置越优，将被放入优先队列

新建一个 Node 类，构建搜索树，每一个结点都是一个 Node 对象，变量 self 是当前的状态，father 为父节点，chosedAction 为当前所选择的动作，alreadyCost 是初始状态到现在状态的代价，goalCost 是现在状态到目标状态的代价，allCost 是总代价，rootpos 是搜索树的根结点。

```
StateObservation self;
Node father;
ArrayList<Types.ACTIONS> actions;
Types.ACTIONS chosedAction;
double alreadyCost;
double goalCost;
double allCost;
Vector2d rootpos;
```

AStar 算法：

A*算法中，新建两个优先级队列来记录该位置的状态，用自定义函数 compare 按照总代价从小到大排序，openQueue 记录还未搜索到的节点，closeQueue 记录已经搜索过的节点。如果该位置正在被遍历，如果已经在 openQueue 队列中，就对该位置评估，选择最优解，若不在 openQueue 队列中，就添加到 openQueue 队列，选择结束后，加入 closeQueue 队列。

rootpos 是精灵的起始位置，首先将初始节点存入 openQueue 队列，当游戏没有停止的时候，从 openQueue 中选择代价最小的节点 n，对该节点的可行动作进行遍历，根据选择的动作新建节点 nextStep 并将当前节点作为新建节点的父节点存入，并调用启发函数获得代价。在 openQueue 队列和 closeQueue 队列中判断状态，若在 openQueue 队列中，则根据总代价来寻找最优解；若不在 openQueue 中且合法，则记录 nextStep 的父节点为 n，并将 nextStep 加入 openQueue

在 openQueue 队列中判断当前局面状态，若精灵到达目标节点，根据父节点倒序遍历直到根节点，则将节点中存储的动作存入 actions 数组（此时数组中存储的动作是倒序的），在 act 函数中倒序执行。

```
public void AStar(StateObservation so){
    ArrayList<Observation>[] movingPositions = so.getMovablePositions();
    ArrayList<Observation>[] fixedPositions = so.getImmovablePositions();
    rootpos = so.getAvatarPosition();
    Node root = new Node(so, rootpos);
    openQueue.add(root);
    while(!finish){
        Node n = openQueue.poll();
        System.out.print(n.self.getAvatarPosition());
        System.out.print("\n");closeQueue.add(n);
        ArrayList<Types.ACTIONS> currentActions = so.getAvailableActions();
        for(int i = 0; i < currentActions.size(); i++){
            StateObservation stCopy = n.self.copy();
            stCopy.advance(currentActions.get(i));
            Node nextStep = new Node(stCopy, rootpos);
            nextStep.chosedAction = currentActions.get(i);


            boolean reachable = true;
            boolean open = false;
            boolean close = false;
            for(Node s: openQueue){
                if(stCopy.equalPosition(s.self))
                    open = true;     //the node is in the openQueue
            }
            for(Node s: closeQueue){
                if(stCopy.equalPosition(s.self))
                    close = true;    //the node is in the closeQueue
            }
```

```
        if(stCopy.getGameWinner() == Types.WINNER.PLAYER_LOSES)
            reachable = false;
        if(stCopy.equalPosition(n.self))
            reachable = false;   //the node is unreachable
        if(reachable && !close && !open){     //not in the openQueue
            nextStep.father = n;
            openQueue.add(nextStep);
        }
        else if(reachable && !close && open){    //in the openQueue
            for(Node s: openQueue){
                if(stCopy.equalPosition(s.self)){
                    if(s.allCost > nextStep.allCost){
                        s.father = n;
                        n.alreadyCost = nextStep.alreadyCost;
                        n.goalCost = nextStep.goalCost;
                        n.allCost = nextStep.allCost;
                    }
                }
            }
        }

        for(Node s: openQueue){
            if(s.self.getGameWinner() == Types.WINNER.PLAYER_WINS) {
                while (s != null) {
                    actions.add(s.chosedAction);
                    s = s.father;
                }
                finish = true;
            }
        }
```

启发式函数：

```
public double heuristic(StateObservation stateObs){
    if(stateObs.getGameWinner() == Types.WINNER.PLAYER_WINS)
        return 0;
    else if(stateObs.isGameOver())
        return -1;
    ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();

    Vector2d currentpos = stateObs.getAvatarPosition(); //avatar position
    Vector2d goalpos = fixedPositions[fixedPositions.length - 1].get(0).position;   //door position

    //holes positions
    ArrayList<Observation> holes = null;
    if (fixedPositions.length > 2) holes = fixedPositions[fixedPositions.length - 2];

    //boxes positions
    ArrayList<Observation> boxes = null;
    if (movingPositions != null) boxes = movingPositions[movingPositions.length - 1];

    alreadyCost = currentpos.dist(rootpos);

    double keyCost =0, boxCost = 0;
    if(holes != null && boxes != null && holes.size() > 0 && boxes.size() > 0) {
        boxCost = currentpos.dist(boxes.get(boxes.size() - 1).position);
        if (stateObs.getAvatarType() != 4) { //do not get key
            Vector2d keypos = movingPositions[0].get(0).position;
            keyCost = 5*currentpos.dist(keypos);
        }
    }
    goalCost = keyCost + boxCost;
    allCost = alreadyCost + goalCost;
    return allCost;
}
```

分别把箱子和洞放入不同的数组中，在没有拿到钥匙且有洞有箱子的时候，到目标节点的代价是钥匙的代价+箱子的代价（将钥匙的优先级设为 5）

2.4 蒙特卡洛树算法

蒙特卡洛树搜索构建树分为四个步骤，选择（ selection），扩展（ extension），模拟（ simulation）和反向传播（backPropagation，本实验中的代码是 backup）。

选择是从根节点开始，即需要决策的局面 A 开始，局面 A 是第一个被检查的节点，则局面 A 的子节点即为要评价的节点。利用 treePolicy()来判断 A 局面的下一步动作：若局面 A 的可行子节点均被评价完，则用

ucb 公式得到一个拥有最大 ucb 值的可行动作（即调用 uct()函数），并且对这个可行动作产生的新局面再次进行检查直到叶子结点，实现扩展。

再利用 rollOut()函数，随机的产生动作并模拟，用 value()函数给每个节点估值，获得模拟后的胜负值：如果达到胜利局面，赋给一个无穷大的值，如果失败，赋给一个无穷小的值，并更新终止节点的胜负值

最后利用 backup()函数，将终止节点的父节点以及其所有的祖先节点依次更新胜利率，一直反向传播到根节点 R。一个节点的胜利率为这个节点所有的子节点的平均胜利率，从而路径上所有的节点的胜利率

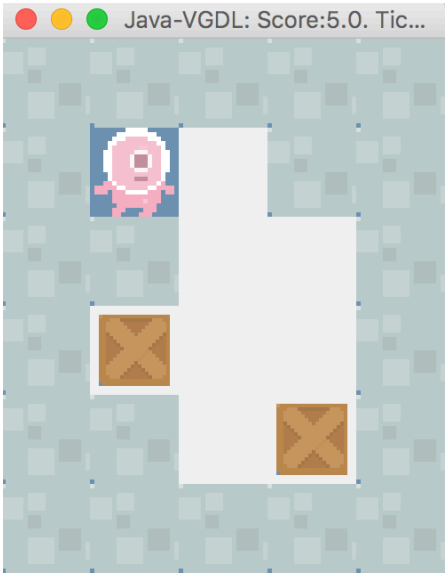
在 Agent.java 中实现 act 函数，用于单步执行可行步骤。 SingleMCTSPlayer 提供搜索的框架，并且每次随机传入一个节点进行后续工作。 SingleTreeNode 给出了树中节点的结构、评估函数、估值比较和选择判断。

其中，uct 公式为：$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$

v_i 是节点估计的值，n_i 是节点被访问的次数，而 N 则是其父节点已经被访问的总次数。C 是可调整参数。

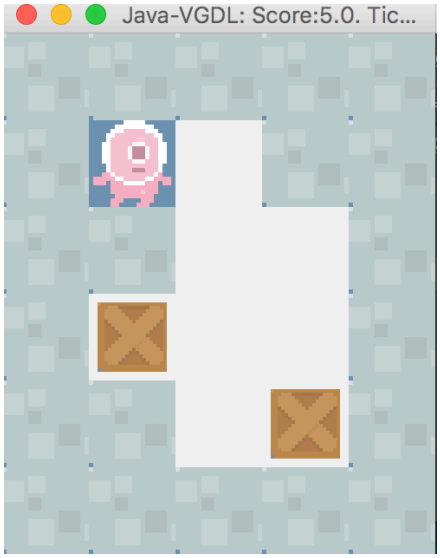
3 实验结果

任务一：深度优先搜索




```
 ** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl0.txt **
Controller initialization time: 0 ms.
Result (1->win; 0->lose):1, Score:5.0, timesteps:10
Controller tear down time: 0 ms.
```

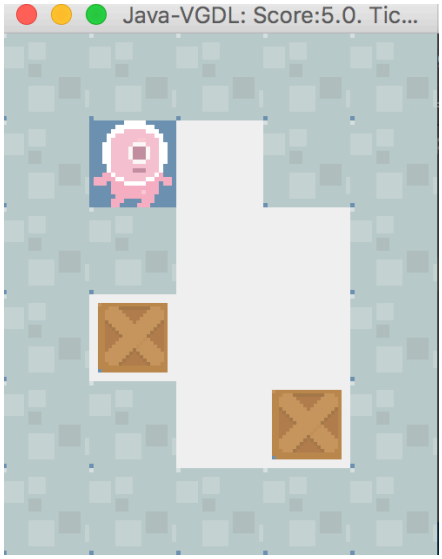
任务二：深度受限的深度优先搜索




```
 ** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl0.txt **
Controller initialization time: 0 ms.
Result (1->win; 0->lose):1, Score:5.0, timesteps:9
Controller tear down time: 0 ms.
```

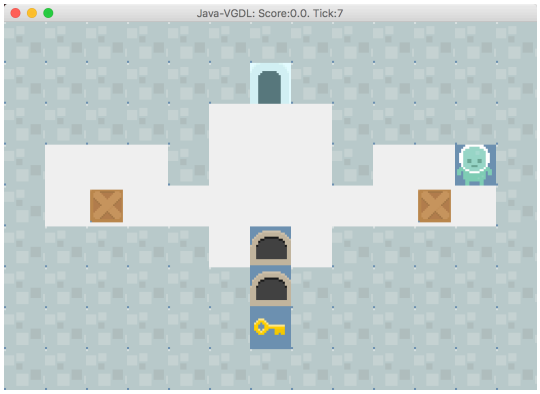
任务三：A*算法

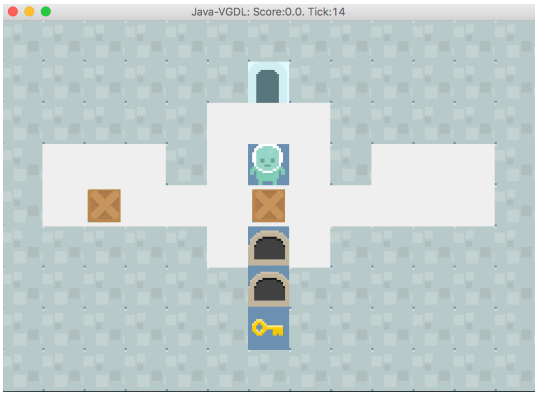
level0:

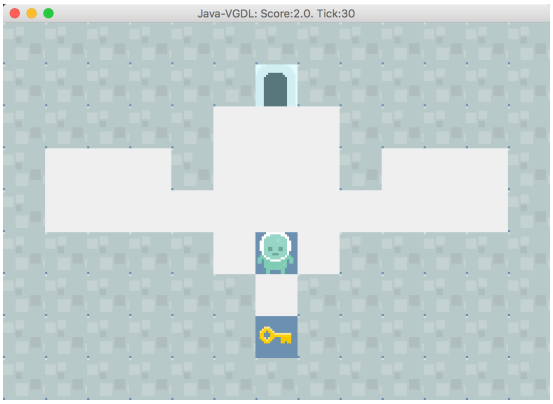


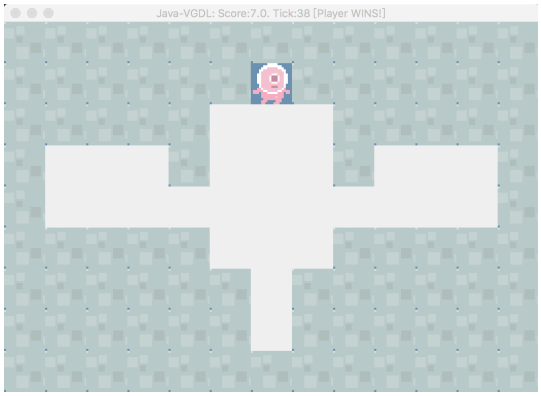

```
 ** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl0.txt **
Controller initialization time: 1 ms.
Result (1->win; 0->lose):1, Score:5.0, timesteps:9
Controller tear down time: 0 ms.
```

level1:



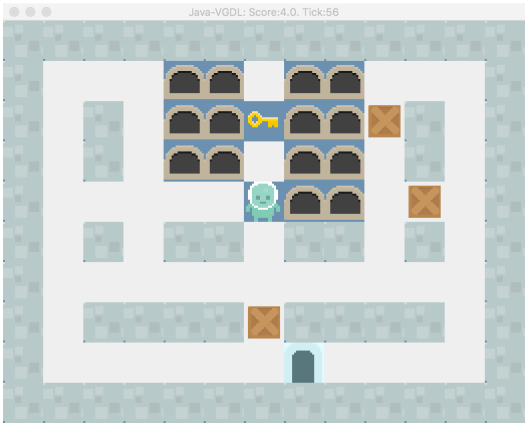


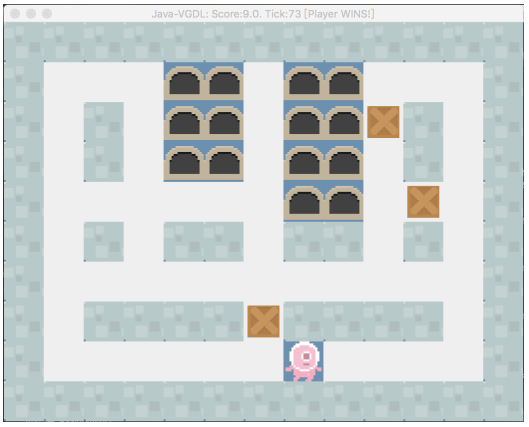





```
 ** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl1.txt **
Controller initialization time: 0 ms.
Result (1->win; 0->lose):1, Score:7.0, timesteps:42
Controller tear down time: 0 ms.
```

level2:






```
 ** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl2.txt **
Controller initialization time: 1 ms.
Result (1->win; 0->lose):1, Score:9.0, timesteps:73
Controller tear down time: 0 ms.
```

致谢　在此,向对人工智能课程老师和助教表示感谢.

References:

[1]　《人工智能》

[2]　人工智能课程 ppt

[3]　蒙特卡洛树算法(MCTS) https://blog.csdn.net/Jaster_wisdom/article/details/50845090.

[4]　蒙特卡洛树搜索 MCTS https://blog.csdn.net/zkl99999/article/details/50677996.

[5]　人工智能中的常用搜索策略　https://www.cnblogs.com/bgmind/p/AI.html.

[6]　人工智能搜索算法 http://www.thebigdata.cn/JiShuBoKe/14335.html