作业一实验报告

银琦（141220132、141220132@smail.nju.edu.cn）

(南京大学　计算机科学与技术系，　南京　　210093)

摘　要:　本次作业 Bait 游戏是基于 VGDL（视觉游戏描述语言）的游戏，通过深度优先搜索算法、深度有限算法、A*算法、蒙特卡洛树搜索算法实现自主搜索成功路径完成游戏。前三种算法须自己实现，最后一种算法须阅读并理解。

关键词:　深度优先搜索，深度有限搜索，A*搜索，蒙特卡洛树搜索

1　实验内容

Bait 游戏是一个基于 VGDL 描述的游戏，游戏规则如下：精灵要先拿到钥匙，然后走到目标；如果精灵吃了蘑菇，那么额外加 1 分；精灵不能掉进洞里，否则失败；精灵可以推盒子吧洞填上，洞填上后就可以通过，并且每填一个洞有 1 分的奖励；只能向前推一个盒子，不能推两个盒子，也不能把盒子推到墙、蘑菇上（盒子可以推到目标上再推开）；箱子可以覆盖钥匙，一旦覆盖，必须推开箱子取得钥匙才能成功；另外，游戏的时间为 1000ticket，时间结束即失败。

本次实验分为四个任务：

任务一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。

任务二在任务一基础上，实现深度有限的深度有限搜索，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。

任务三在任务二的基础上，将深度优先搜索改为 A*算法，并且尝试通过第二关和第三关。

任务四需要阅读提供的 sample 蒙特卡洛树算法并且理解。

2　算法介绍

2.1　深度优先搜索算法

深度优先搜索是常用的搜索方法，核心思想是将每一步可走步骤加入存储路径的数组，并且判断它是否和当前局面已走的步骤相同，如果相同，说明已经走过，跳出算法的迭代，否则将新的步骤加入数组，并且更新当前局面，进行下一层的搜索。结束条件为当前局面搜索到成功步骤，那么标志着此次递归结束，可行步骤全部存入路径数组。具体实现如下：

在 Agent 类中实现 Act 函数，用于操纵精灵执行单步，Act 函数中调用 DFS 函数，用于搜索成功路径。

```
public ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
    // TODO Auto-generated method stub
    if (count == 0) {
        dfs(stateObs);
    }
    count++;
//  System.out.println(count);
    return act.get(count-1);
    return null;
}
```

在 DFS 函数中使用 STCopy 模拟器模拟下一步可走步骤对当前局面产生的影响。

如果四个可走步骤都尝试过（上下左右）还没有到达成功状态，那么就删去当前路径中的最后一步，回溯到上一步继续查找可走步骤并且更新状态。这里我设置了一个 stepcount 记录当前局面可行步骤（上下左右）中不可行的步数，以便回溯时进行步数比对。

```
if (state.get(state.size()-1).stepCount == newAction.size()) {
    act.remove(act.size()-1);
    state.remove(state.size()-1);
    state.get(state.size()-1).stepCount++;
}
```

需要注意的是，在搜索到成功路径的时候，需要置一个成功 flag，在此步骤执行完再跳出循环， 否则会导致最终路径少走最后一步。

```
if (stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS) {
    //System.out.println(flag);
    finish = true;
}
```

2.2 深度有限搜索算法

深度有限搜索是在深度优先搜索的基础上进行改进的一种算法。程序在执行到一定深度后，将执行启发式函数，通过启发式函数返回的值，判断此时局面是否是最优的，如果是， 就将此步骤加入可走步骤中。其他思想与深度优先搜索算法相同，实现方法也类似，增加了一个启发式函数以及对搜索层数的判断。

启发式函数的实现方法是，获取精灵当前位置、目标位置、钥匙位置的坐标，如果没有吃到钥匙，那么代价（cost）就定为精灵到目标的距离+精灵到钥匙的距离，如果吃到钥匙，那么代价就为精灵到目标的距离。

```
public double distance(StateObservation stateObs) {
    double distance = 0;
    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 currentPosition = stateObs.getAvatarPosition();
    System.out.println(movingPositions.length);
    System.out.println(currentPosition);
    if (movingPositions.length == 2) {//如果还没吃到钥匙
        Vector2d keypos = movingPositions[0].get(0).position;
        System.out.println(keypos);
        distance = currentPosition.dist(goalpos) + currentPosition.dist(keypos);
    //  System.out.println(distance);
    }
    else {//如果吃到钥匙
        Vector2d keypos1 = movingPositions[0].get(0).position;
        System.out.println(keypos1);
        distance = currentPosition.dist(goalpos);
        System.out.println(distance);
    }
    return distance;
}
```

代价（cost）初始化为一个较大的值（如我设为 10000），启发式函数返回的值越小，就用新的值代替之前的代价。需要注意的是，在某一层搜索结束后，需要把当前存储可行步骤的数组清空，重新赋值更好局面

下可行步骤的情况，否则 act 函数会执行重复和错误的步骤。由于层数对搜索局面有些影响，比如层数小于等于 5 都无法搜索到成功的局面，因此层数必须设置在 6 层及 6 层以上。

```
if (depth == 6 && flag == true) {
    System.out.println("ok");
    if (distance(stCopy) < cost) {
        cost = distance(stCopy);
        //System.out.println("222222");
        updateAction.clear();
        for (int k = 0;k < act.size();k++)
            updateAction.add(act.get(k));
    }
    flag = false;
}
```

2.3 A*搜索算法

A*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。启发函数使用三个值来计算代价（cost），gcost 代表从起始位置到当前位置的距离，fcost 代表当前位置到钥匙位置乘以系数 100，hcost 代表 gcost 和 fcost 之和。hcost 越小，说明当前位置越优，将被放入优先队列。

```
public double heuristics(StateObservation stateObs) {
    if (stateObs.getGameWinner() == Types.WINNER.PLAYER_WINS) return 0;
    if (stateObs.isGameOver()) return 1000000;
    ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
    Vector2d currentPosition = stateObs.getAvatarPosition();
    gcost =  currentPosition.dist(rootPosition);
    if (stateObs.getAvatarType() != 4) {
        Vector2d keypos = movingPositions[0].get(0).position;
        fcost = 100*currentPosition.dist(keypos);
    }
    else {
        Vector2d goalpos = fixedPositions[fixedPositions.length-1].get(0).position;
        fcost = currentPosition.dist(goalpos);
    }
    hcost = gcost + fcost;
    return hcost;
}
```

A*启用两个优先级队列来记录该位置的状态，如果该位置正在被遍历，就放入 open 队列，如果该位置不可达或者已经在 closed 队列里，那么就什么都不做，如果在 open 队列中，那么就查看该位置的评估值，选择最优解。选择结束后，将该位置放入 closed 队列中。

```
do {
    Astar priorityAstar = Open.poll();
    Closed.add(priorityAstar);
    ArrayList<Types.ACTIONS> newAction = stateObs.getAvailableActions();
    for (int i = 0;i < newAction.size();i++) {
        StateObservation stCopy = priorityAstar.state.copy();
        //System.out.println(newAction);
        stCopy.advance(newAction.get(i));
        Astar currentAstar = new Astar(stCopy,newAction.get(i),rootPosition);
        boolean flaga = false;
        boolean flagb = false;
        for (Astar s:Closed) {
            if (stCopy.equalPosition(s.state) == true)
                flaga = true;
        }
        for (Astar s:Open) {
            if (stCopy.equalPosition(s.state) == true)
                flagb = true;
        }
        if (stCopy.equalPosition(priorityAstar.state) == true || flaga == true)
            continue;
        //System.out.println(newAction.get(i));
        if (flagb == false) {
            //System.out.println("11111");
            currentAstar.father = priorityAstar;
            Open.add(currentAstar);
            //System.out.println(Open.size());
        }
        if (flagb == true) {
            for (Astar s:Open) {
                if (stCopy.equalPosition(s.state) == true) {
                    if (s.gcost > currentAstar.gcost) {
                        s.father = priorityAstar;
                        s.gcost = currentAstar.gcost;
                        s.hcost = currentAstar.hcost;
                        s.fcost = currentAstar.hcost;
                    }
                }
            }
        }
        for (Astar s:Open) {
            if (s.state.getGameWinner() == Types.WINNER.PLAYER_WINS) {
                while (s != null) {
                    resultAction.add(s.action);
                    //System.out.println(resultAction);
                    s = s.father;
                }
                finish = true;
            }
        }
    }
} while (finish == false);
```

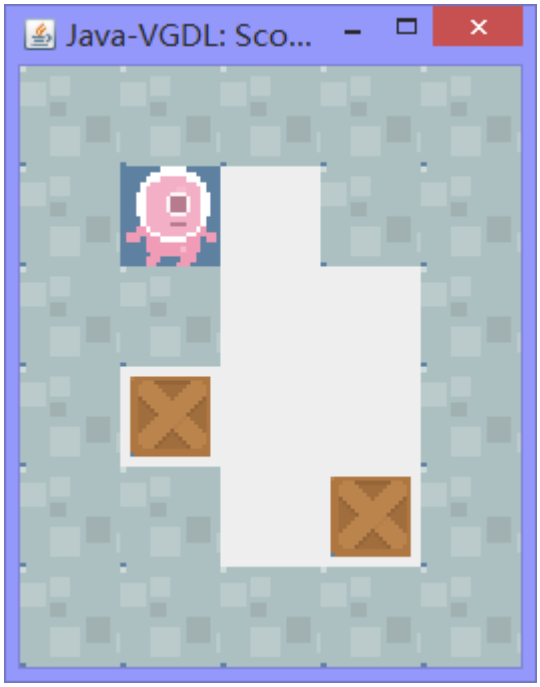
2.4 蒙特卡洛搜索树算法

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）。选择是从根节点开始，递归选择最优子节点直到达到叶子节点。扩展是如果某叶子节点不是终止节点（游戏并没有结束），那么就扩展创建更多的子节点并且选择其中一个。模拟是从刚才选出的这个节点开始运行一个模拟的输出，直到游戏结束。反向传播是用模拟的结果输出更新当前行动的序列。在 Agent.java 中实现的是 act 函数，用于单步执行可行步骤。SingleMCTSPlayer 给出了搜索的框架，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 act 函数实现了如果所有节点都被扩展过，那么选择一个当前最迫切需要扩展的节点，这个决策衡量由公式 $V_i + C \times \sqrt{\frac{\ln N}{n_i}}$ 完成。

Value 给出了每个节点的估值，如果达到了胜利局面，就赋给一个无穷大的值，如果失败，就赋给一个无穷小的值。Backup 即是将模拟的节点全部反向输出更新当前行动的序列。

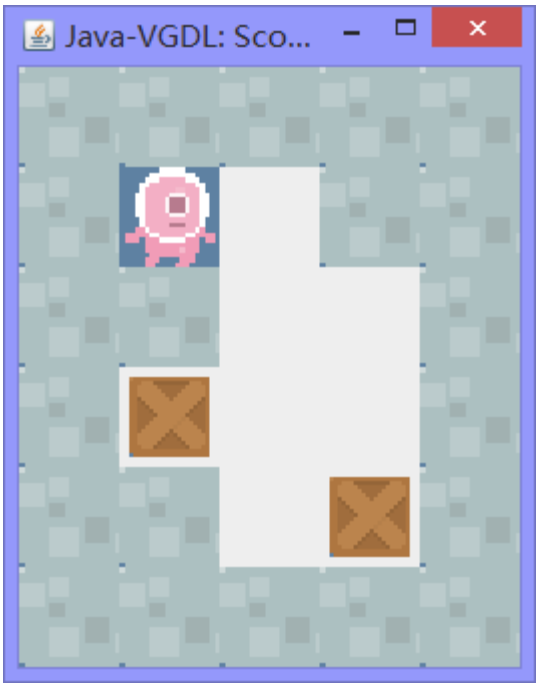
3 实验结果

任务一



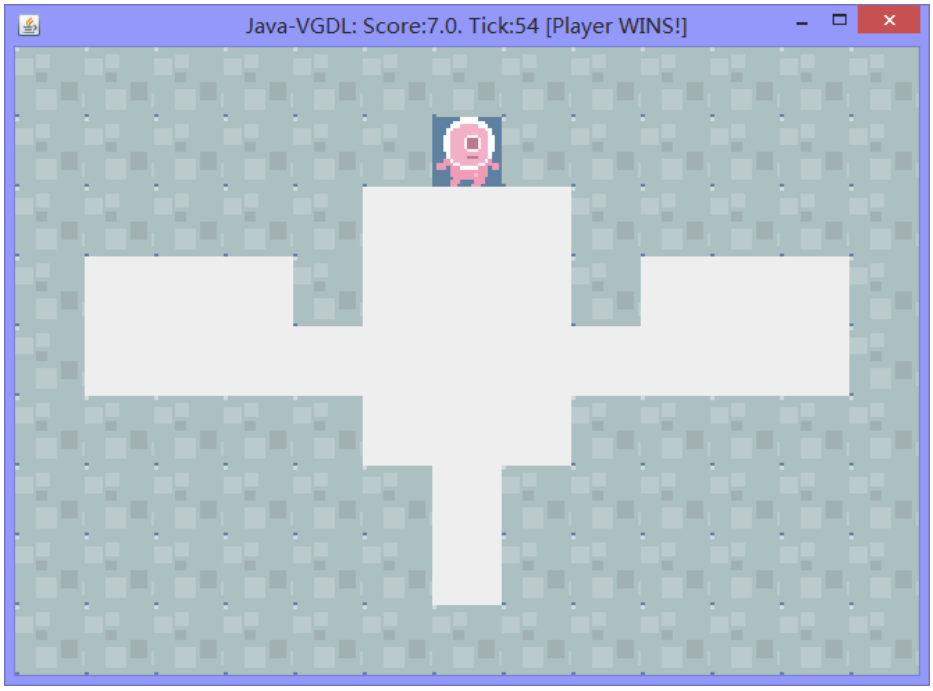

```
 ** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl0.txt **
Controller initialization time: 0 ms.
Result (1->win; 0->lose):1, Score:5.0, timesteps:10
Controller tear down time: 0 ms.
```

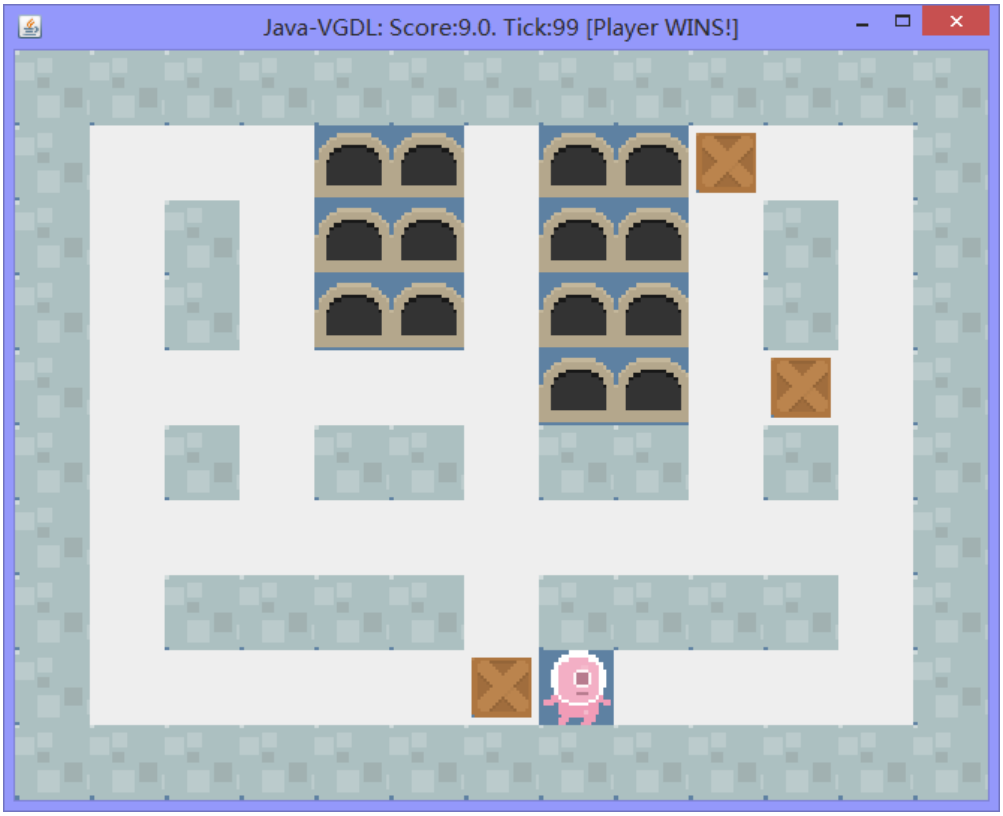
任务二




```
 ** 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






```
 ** 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.
 ** 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:54
Controller tear down time: 0 ms.
 ** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl2.txt **
Controller initialization time: 0 ms.
Result (1->win; 0->lose):1, Score:9.0, timesteps:99
Controller tear down time: 0 ms.
```

References:

[1] http://www.kuqin.com/shuoit/20160219/350769.html

[2] http://www.cnblogs.com/technology/archive/2011/05/26/2058842.html