Assignment 1：Bait游戏实验报告

231300027朱士杭

注意：由于pdf版本的实验报告没法查看gif动图的，推荐前往查看完整内容的Assignment1_Report.md的markdown文件

前言：一些在Agent.java里调用接口的分析说明

StateObservation类（与核心代码关系不大）

1. getObservationGrid():返回观测值的网格。其中包含关卡中的所有观测值，可以通过网格的 x y 坐标访问。每个网格单元的宽度和高度为 getBlockSize() 像素。每个单元包含在该位置的所有观测值的列表。
2. getBlockSize():指明游戏中构成一个块的像素数量
3. getGameWinner():判断游戏是否胜利，若 ==ontology.Types.WINNER.PLAYER_WINS则表示游戏胜利，找到一条路径
4. isGameOver():游戏失败因此结束游戏
5. equalPosition():比较两个状态是否一样。如果一样则返回1
6. getAvailableActions():得到目前状态允许的动作候选集

ArrayList<Types.ACTIONS>类 （设计java基本的数据结构）

1. size():返回ArrayList的大小（元素个数）
2. get(i):返回ArrayList索引为i的元素
3. remove(i):删除ArrayList索引为i的元素
4. add(ele):在ArrayList的末尾添加元素ele

printDebug 和 draw 函数都可以先不用管，因为不会对核心代码产生影响，就当框架摆在那里就可以了

About Task1

核心思路

对于深度优先搜索来说，更宽泛一点对于搜索问题来说，核心关键点有：

1. 初始状态 Initial State
2. 每一步状态 State
3. 转移序列 Transition Sequence（即每一步状态的序列）
4. 动作序列 Actions（即每一步动作的序列）
5. 代价 Cost
6. 目标 Goal

目标是最终得到钥匙并且走到蘑菇，因此初始状态与最终状态已经确定（不妨以目前游戏布局作为状态点，每一次移动都会有一个新的布局状态），只需要寻找到中间的路径即可，而且由于题目并没有要求一定是最短路径，因而不需要进行一些优化，只要找到路径就行。

因此核心思路就有了：

1. 从当前状态当中找出动作候选集Candidates，可以调用getAvailableActions()方法获得
2. 从Candidates当中选取一个动作Action（最好按照顺序选取）
3. 根据动作更新状态State，将该动作加入到动作序列
4. 重复上述步骤，直到达到最终状态 or 超出游戏时间失败 or 到达死胡同没有动作可做
5. 如果到达“死胡同”，则回溯到上一层，删除动作序列中这一步的动作，切换下一个动作继续往下递归，重复步骤4

但是注意在DFS当中需要避免死循环，很有可能本来存在一条合法路径，但是由于没有考虑到状态重复的问题，一直在几个状态之间来回转移导致最终超时游戏失败。因此我们需要引入一个转移序列专门用来存储已经走过的状态集合，一旦施加动作之后的状态曾经出现过，那就换下一个动作去执行，可以有效避免死循环的问题。

另外，动作序列本质上是一个栈Stack的数据结构，非常符合递归对于栈的需求，满足FILO先进后出。

主要的核心代码为（由于注释写的已经很清楚了而且思路也写的非常明白了这里就不再多加赘述了）：

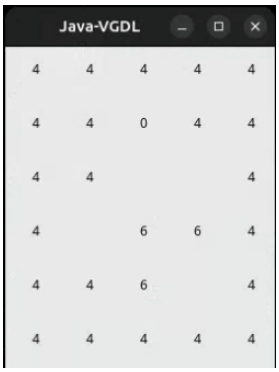
```
//判断当前状态之前是否到达过
boolean test_past_state(StateObservation state){
    for (int i=0; i<pastState.size(); i++) if(state.equalPosition( pastState.get(i) ))return true;
    return false;
}

//核心递归代码！！！
boolean get_DFS_Actions(StateObservation stateObs, ElapsedCpuTimer elapsedTimer){
    pastState.add(stateObs);  //将当前状态加入已走过的状态
    ArrayList<Types.ACTIONS> all_available_actions = stateObs.getAvailableActions();
    for(int i=0;i<all_available_actions.size();i++){
        //尝试当前局面所有可以的动作
        StateObservation stCopy = stateObs.copy();                          //新建一个当前状态的副本，用
        stCopy.advance(all_available_actions.get(i));                                                    //
        Actions.add(all_available_actions.get(i));                                                       //

        if(stCopy.getGameWinner() == WINNER.PLAYER_WINS)  return true;     //如果获胜则返回true表示已找到
        if(test_past_state(stCopy) || stCopy.isGameOver()) Actions.remove(Actions.size() - 1);  //如
        else{                                                               //动作施加后的是一个新的状态
            if(get_DFS_Actions(stCopy,elapsedTimer))  return true;              //递归进行深度优先搜索
            else  Actions.remove(Actions.size() - 1);                  //当前动作递归搜索失败，尝试下一个
        }
    }
    pastState.remove(pastState.size() - 1);                            //当前局面没有办法成功，删除当前局面
    return false;
}
```

后来看到任务要求里面好像又说到需要详细介绍对代码的实现，因此这里就补充说明一下吧。首先是总体框架当然是在act函数里面进行实现的。由于在更上一层的框架里面，好像是有一个loop循环调用act函数，每一次都只是返回一个action，因此我们需要维护一个arraylist的动作序列Actions用来存储最终的action；然后就是进行DFS的搜索了，将初始状态stateObs加入到pastState中去存储已经走过的节点（搜索过了就不会再次进行搜索），然后获得当前状态下允许执行的动作合集，然后for循环遍历，在这个循环内部使用递归就可以做到深度优先搜索了。在for循环内部，先copy当前状态模拟动作，然后 判断是否胜利/失败/以前遍历过，如果胜利的话直接返回true结束递归，如果出现问题的话那就结束当前节点的搜索而转移至同一层节点的搜索，直到返回上一级搜索其他的节点。这里需要注意一个点就是有关于Actions行动序列的加入与删除的先后关系，无论搜索到了哪一个节点都将其加入到Actions里面去，如果不符合要求那么再将其从Actions中删除（即拿出的时候判断一下），这样子就可以避免很多逻辑上的问题

最终Task1执行效果如下gif所示：



About Task2

其实Task2并不比Task1困难多少，对于深度限制DFS算法其实只是加了一个深度的限制，避免因为搜索深度过深而导致栈溢出，同时具有DFS与BFS的优点

观察Task1的结果会发现其实这并不是最优的解决路径，因为在返回的过程中其实多绕了2步，因此手动调整限制深度的参数之后再观察最终结果发现最优情况下只需要11步即可完成搜索，比Task1还更优了，可以得出结论深度限制的深度优先搜索从效率与路径角度都是更加优于一般的深度优先搜索的

Task2的核心代码如下图所示，通过比较可以发现，相比与task1中多了depth变量和maxdepth参数用来控制当前的深度，如果当前深度已经超过了限制的最大深度参数，则返回false即不再继续往下搜索，并且删除添加进动作序列的最新动作

```
//判断当前状态之前是否到达过
boolean test_past_state(StateObservation state){
    for (int i=0; i<pastState.size(); i++) if(state.equalPosition( pastState.get(i) ))return true;  /
    return false;
}

//核心递归代码！！！
boolean get_LDFS_Actions(StateObservation stateObs, ElapsedCpuTimer elapsedTimer,int depth){
    if (depth>=max_limit_depth) return false;
    pastState.add(stateObs);  //将当前状态加入已走过的状态
    ArrayList<Types.ACTIONS> all_available_actions = stateObs.getAvailableActions();
    for(int i=0;i<all_available_actions.size();i++){
        //尝试当前局面所有可以的动作
        StateObservation stCopy = stateObs.copy();                          //新建一个当前状态的副本，用于
        stCopy.advance(all_available_actions.get(i));                                                   //旅
        Actions.add(all_available_actions.get(i));                                                      //将

        if(stCopy.getGameWinner() == WINNER.PLAYER_WINS)  return true;     //如果获胜则返回true表示已找到
        if(test_past_state(stCopy) || stCopy.isGameOver()) Actions.remove(Actions.size() - 1);  //如果
        else{                                                               //动作施加后的是一个新的状态
            if(get_LDFS_Actions(stCopy,elapsedTimer,depth+1))  return true;                 //递归进行深度
            else  Actions.remove(Actions.size() - 1);                  //当前动作递归搜索失败，尝试下一个动
        }
    }
    pastState.remove(pastState.size() - 1);                          //当前局面没有办法成功，删除当前局面
    return false;
}
```

最终Task2执行效果如下gif所示：

有关于启发式函数用来判断现在局面好坏的设计我们放在A*算法里面再详细介绍

About Task3

2024.09.24 首先要声明一点，本任务中的A*算法是针对于第1关bait_lvl0.txt而设计的，没有考虑到后面几关当中还有hole的存在，因此确实会存在一些问题，但是只需要一些改进即可，但是尝试过后都失败了，遂无功而返

2024.09.25 但是第二天又打脸了，发现了一个bug之后调试完成，再使用A*算法发现其实是可以成功找到路径的……好当我昨天的话没说

在本任务当中需要用A*算法实现寻路，其实个人认为A*本质上也能算一种贪心算法，只不过其是采用了已有cost代价与预期cost代价之和作为总的代价，在本题当中我们规定，已经走过的路程（即已有的代价）为函数G(x)，其中x表示节点/状态，预期的代价（即Task2当中所说的启发式函数）为函数H(x)，二者相加结果为F(x)函数，表示总代价total cost

此外，我们还需要维护2个arraylist，一个是openlist一个是closelist。其中openlist表示当前寻路过程中的候选节点，closelist表示已经寻找过的节点。 每一次从openlist候选节点当中寻找一个代价最小的节点，然后将其加入到closelist当中，并寻找接下来的下面动作的节点，直到找到了最终的终点节点。

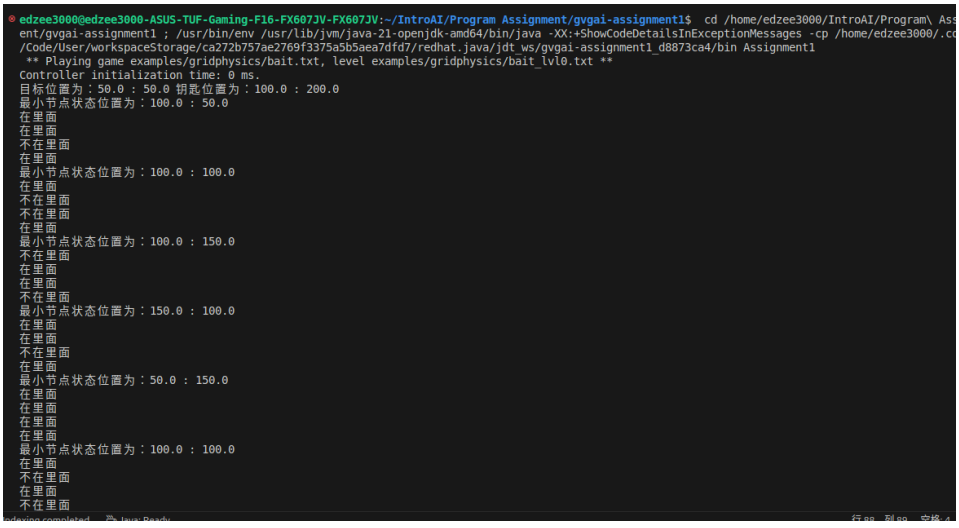
你可能会担心，万一下一个节点之前已经寻找过了呢？很容易证明一点，同样对于一个节点，之前找到过的G一定比之后找到时的G要小，而H又是不变的，因此根本就不用管之后找到的节点，所以我们需要一个closelist，每一次遍历下一次节点的时候都需要判断之前是否已经寻找过了，如果之前已经寻找过了，就直接抛弃不再寻找，这就可以保证算法的一个正确性（不会陷入死循环或者找不到一个比较优的解）

接下来的一个核心问题就是如何确定代价函数F=G+H，对于非启发式函数G来说，不妨让其就是该节点从起始位置出发走了多少步，而对于启发式函数H的设计，不妨让它就是该节点到最终终点的曼哈顿距离。

但是这里有一个问题，包括在写代码的时候也遇到的一个问题就是，当player拿到钥匙与没有拿到钥匙这两种情况的H应该是不一样的，因此这里需要进行分类讨论：

1. 如果player没有拿到钥匙，那么H(x)的值应当为player到key的曼哈顿距离+key到goal的曼哈顿距离
2. 如果player拿到钥匙，那么H(x)应当为player到goal之间的曼哈顿距离

那在写代码的过程中我遇到了什么问题呢？发现不知道为什么在代码运行的过程中最终是找不出来一条允许的路径的，于是乎我就开始进行debug调试，如下图所示：



最后发现一个问题就是每一次player拿到钥匙之后（之前所有的步骤都没有任何问题），已知终点目标位置为(50,50)，结果player走到(100,50)这个位置（也就是起点那个位置）的时候，马上就终止了（？？！！！），非常奇怪，然后发现问题出在这条代码（debug了整整4个小时啊啊啊啊）：

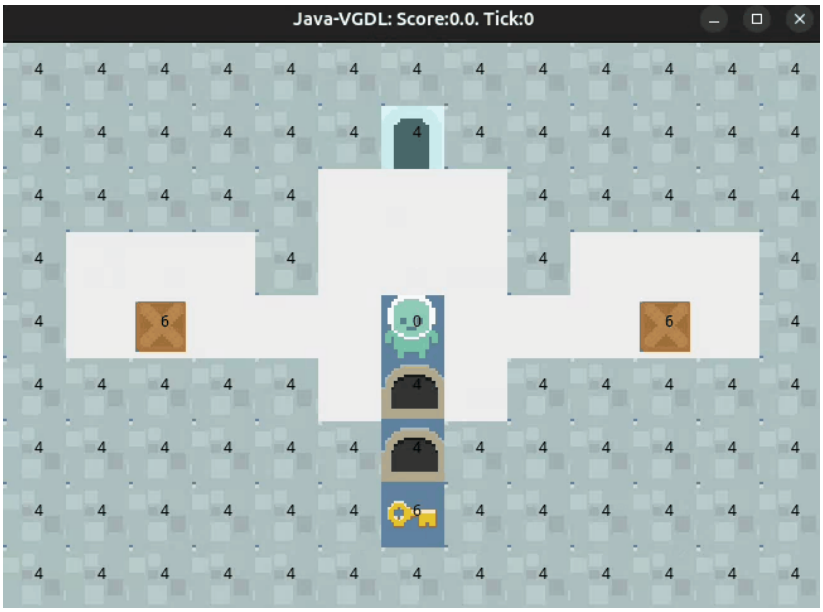
```
if(stCopy.isGameOver()){
    System.out.println("游戏失败");
    close_list.add(newNode);}
```

在这里player当前状态竟然是结束游戏了，然后返回游戏失败返回false表示没有允许的路径，百思不得其解一直都找不出来问题在哪里……

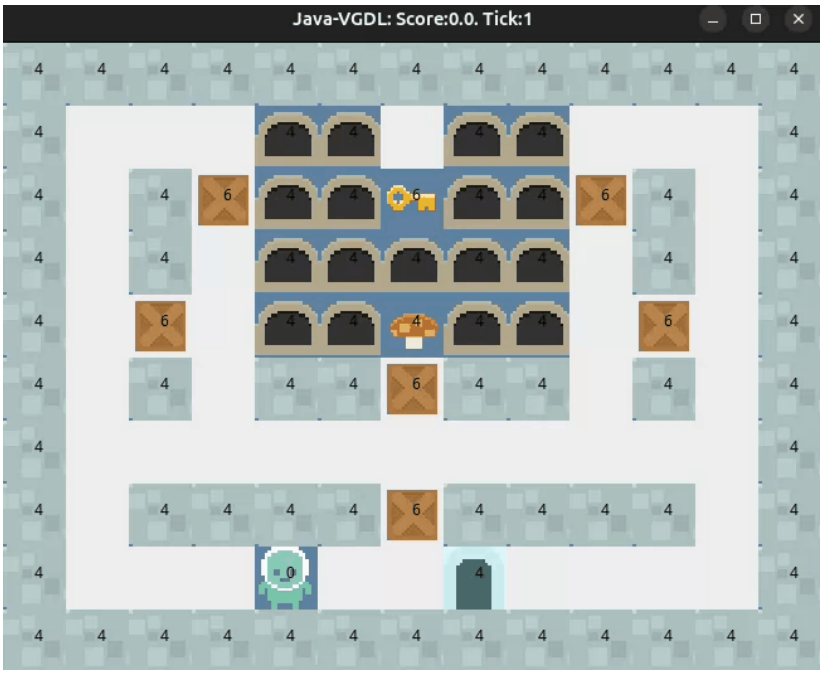
结果第二天突然恍然大悟（其实是在习思想课上突然想到的哈哈哈哈），stCopy.isGameOver()方法会有双重作用，无论游戏成功/失败都会为1，而还有一个if条件判断语句判断其是否胜利竟然放在了这个if判断语句之后了，也就是说真正想判断胜利执行的那些代码将永远不会执行，然后将`if(stCopy.getGameWinner() == WINNER.PLAYER_WINS)`这条语句放到`if(stCopy.isGameOver())`之前就完全可以了，然后测试代码System.out.println输出之后发现存在这条路径。

但是仍然存在一个问题，我发现在Actions动作序列当中竟然第一步是null，然后查看其余步骤发现并没有什么问题，实在想不明白为什么这里会有问题了，于是取个巧吧直接在if判断条件里面加上一句话将这个null的情况过滤掉了，然后测试所有的关卡，发现都没有什么问题（除了运行时间稍微长了一点之外，至于为什么运行时间太长了，其实又是因为自己没有考虑到洞hole和箱子之间的积分关系所以启发式函数就比较简陋了），但是问题不大至少跑出结果了对吧。

最终Task3的bait_lvl1关卡执行效果如下gif所示（为什么不放关卡0的结果呢因为跟Task1和Task2里面的结果没有多大差别不想干重复的事儿）：



最终Task3的bait_lvl2关卡执行效果如下gif所示（预计执行时间有2min左右）



下面这个是debug之前的代码，会存在问题因此这里就不再详细叙述了

```
//A*搜索核心代码
boolean AstarSearch(StateObservation stateObs, ElapsedCpuTimer elapsedTimer)
{
    StateObservation init_state=stateObs;
    Node root_node = new Node(init_state,last_node:null,act:null);
    open_list.add(root_node);
    // close_list.add(root_node);//将根节点加入到close list当中去
    // for(Types.ACTIONS action : stateObs.getAvailableActions())
    // {StateObservation stCopy = stateObs.copy();stCopy.advance(action);Node newNode=new Node(stCopy,root_node,action);open_list.add(newNode); }//初始化将可允许的添加进openlist（注意openlist里面是节点）
    // print open_close();
    //从openlist中选出F最小的节点加入到closelist中然后删除，产生新的getAvailableActions对应的状态节点并加入openlist中
    //重复上述操作直到游戏失败or游戏胜利or到了死胡同
    while(!open_list.isEmpty())
    {
        //在openlist中寻找F最小的node并删除，加入到closelist中
        Node min_node=search_min_node();    open_list.remove(min_node);    close_list.add(min_node);
        System.out.println("最小节点状态位置为: "+min_node.state.getAvatarPosition());
        // try{Thread.sleep(1000); } // 放在这里用来中断调试…


        //更新openlist
        for(Types.ACTIONS action :min_node.state.getAvailableActions())
        {StateObservation stCopy = min_node.state.copy();  stCopy.advance(action);  Node newNode=new Node(stCopy,min_node,action);
        if (flag==0 && !newNode.has_key)flag=0; else if(flag==0&&newNode.has_key){System.out.println("拿到钥匙");flag=1;    open_list=new ArrayList<Node>();   open_list.add(newNode);  close_list=new ArrayList<Node>(); break;

        if(flag==0 && stCopy.isGameOver()){System.out.println("游戏失败");//try{Thread.sleep(3000);} // 放在这里用来中断调试
        // catch (InterruptedException e) {// 如果线程在休眠期间被中断，会抛出InterruptedException异常…
            close_list.add(newNode);}


        if(test_in_openlist_and_closelist(stCopy)){//System.out.println("在里面");
        continue;}
        if(!test_in_openlist_and_closelist(stCopy)){//System.out.println("不在里面");
        open_list.add(newNode);}
        //如果找到了终点的话则停止并产生更新全局Arraylist的动作序列（利用链表的思想）
        if(stCopy.getGameWinner() == WINNER.PLAYER_WINS)
        {   System.out.println("游戏胜利");
            Node parent=newNode;
            while(parent!=null){Actions.add(parent.act);    parent=parent.last;}
            Collections.reverse(Actions);
            return true;
        }
        }

    }

    System.out.println("游戏失败");
    try{Thread.sleep(1000); } // 放在这里用来中断调试
        catch (InterruptedException e) {// 如果线程在休眠期间被中断，会抛出InterruptedException异常
            System.err.println("Thread was interrupted.");
            // 可以选择重新设置中断标志
            Thread.currentThread().interrupt();}
    return false;
}
```

这个是debug之后的代码，通过比较可以发现，跟尚敏的代码还是有一定区别的，整体思路就是很简单的，初始化起点节点，然后从openlist中删除，加入到closelist当中去，然后for循环遍历接下来的子节点，每一次都是从openlist当中去寻找最小F(x)代价的节点然后检查其子节点是否结束游戏。大致思路和代码框架其实和深度优先搜索差不了太多，但是唯一不同的是运用到了链表的思想，每一次检查对Actions动态数组不进行操作，而是记录节点，然后通过节点的父节点依次遍历直到回到了初始节点。最后再将这些节点的action动作加入到Actions动作序列当中。

```
//A*搜索核心代码
boolean AstarSearch(StateObservation stateObs, ElapsedCpuTimer elapsedTimer)
{
    StateObservation init_state=stateObs;
    Node root_node = new Node(init_state,last_node:null,act:null);
    open_list.add(root_node);
    // close_list.add(root_node);//将根节点加入到close_list当中去
    // for(Types.ACTIONS action : stateObs.getAvailableActions())
    // {StateObservation stCopy = stateObs.copy();stCopy.advance(action);Node newNode=new Node(stCopy,root_node,action);open_list.add(newNode); }//初始化将可允许的添加进openlist（注意openlist里面是节点）
    // print_open_close();
    //从openlist中选出F最小的节点加入到closelist中然后删除，产生新的getAvailableActions对应的状态节点并加入openlist中
    //重复上述操作直到游戏失败or游戏胜利or到了死胡同
    while(!open_list.isEmpty())
    {
        //在openlist中寻找F最小的node并删除，加入到closelist中
        Node min_node=search_min_node();     open_list.remove(min_node);     close_list.add(min_node);
        System.out.println("最小节点状态位置为: "+min_node.state.getAvatarPosition());
        // try{Thread.sleep(1000); } // 放在这里用来中断调试…


        //更新openlist
        for(Types.ACTIONS action :min_node.state.getAvailableActions())
        {StateObservation stCopy = min_node.state.copy();  stCopy.advance(action);  Node newNode=new Node(stCopy,min_node,action);
        //如果找到了终点的话则停止并产生更新全局Arraylist的动作序列（利用链表的思想）
        if(stCopy.getGameWinner() == WINNER.PLAYER_WINS)//如果游戏胜利的话
        {   System.out.println("游戏胜利");
            Node parent=newNode;
            // System.out.println("最后一步动作为:"+parent.act);
            while(parent!=null && parent.act!=null){System.err.println("每一步动作为: "+parent.act);Actions.add(parent.act);    parent=parent.last;}
            Collections.reverse(Actions);
            return true;
        }
        if (flag==0 && !newNode.has_key)flag=0; else if(flag==0&&newNode.has_key){System.out.println("拿到钥匙");flag=1;    open_list=new ArrayList<Node>();   open_list.add(newNode);  close_list=new ArrayList<Node>(); break;}//

        if(stCopy.isGameOver()){System.out.println("游戏失败");//try{Thread.sleep(3000);} // 放在这里用来中断调试
        // catch (InterruptedException e) {// 如果线程在休眠期间被中断，会抛出InterruptedException异常…
            close_list.add(newNode);}


        if(test_in_openlist_and_closelist(stCopy)){//System.out.println("在里面");
        continue;}
        if(!test_in_openlist_and_closelist(stCopy)){//System.out.println("不在里面");
        open_list.add(newNode);}

        }

    }
```

关于Node类节点的定义如下图所示，主要定义的成员有当前状态，父节点引用，来到该节点对应的action，函数F(x)计算其总代价total cost

```
  //定义一个Node节点类最终通过last进行回溯
  public static class Node{
      public StateObservation state;
      public Node last=null;
      public int cost;
      public boolean has_key=false;
      public Vector2d PlayerPosition;
      public Types.ACTIONS act=null;
      public int level=0;//level表示已经有多少cost了
      public Node(StateObservation state,Node last_node,Types.ACTIONS act){
          this.state=state;
          this.PlayerPosition = state.getAvatarPosition();              //精灵的位置
          if (last_node!=null)
          {this.last=last_node;
          this.level=last_node.level+1;
          this.act=act;
          }
          if(this.state.getAvatarPosition().equals(keypos) || (this.last!=null&&this.last.has_key))
              this.has_key=true;
          this.cost=this.F();
      }
      //定义一个计算cost的函数F(x)=G(x)+H(x)，其中G为已经花费的cost，H为预计还需要花费的cost
      //不妨令预计的花费为人到钥匙的距离+钥匙到终点距离（如果人没有拿到钥匙）；人到终点的距离（如果人拿到了钥匙）
      public int F()
      {
          int G=this.level;//G表示已经花费的cost代价
          int H;
          int distance_player_key= (int)(Math.abs(PlayerPosition.x - keypos.x) + Math.abs(PlayerPosition.y - keypos.y))/block_size;
          int distance_player_goal=(int)(Math.abs(PlayerPosition.x - goalpos.x) + Math.abs(PlayerPosition.y - goalpos.y))/block_size;
          int distance_key_goal=(int)(Math.abs(keypos.x - goalpos.x) + Math.abs(keypos.y - goalpos.y))/block_size;
          if (this.has_key) H=distance_player_goal;//如果已经有钥匙了
          else H=distance_key_goal+distance_player_key;
          return G+H;
      }
  }
```

为了方便调试，也为了让每一次player的位置走动更加方便观察，因此对于关卡1的bait_lvl0地图进行打印，显示结果如下图所示：

```
edzee3000@edzee3000-ASUS-TUF-Gaming-F16-FX607JV-FX607JV:~/IntroAI/Program Assignment/gvgai-assignment1$  cd /home/edzee3000/IntroAI/Program\ Assignment/gvgai-assignment1 ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/edzee3000/.config/Code/User/workspaceStorage/ca272b757ae2769f3375a5b5aea7dfd7/redhat.java/jdt_ws/gvgai-assignment1_d8873ca4/bin Assignment1
 ** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl0.txt **
Controller initialization time: 0 ms.
目标位置为：50.0 : 50.0 钥匙位置为：100.0 : 200.0
最小节点状态位置为：100.0 : 50.0
最小节点状态位置为：100.0 : 100.0
最小节点状态位置为：100.0 : 150.0
最小节点状态位置为：100.0 : 100.0
最小节点状态位置为：50.0 : 150.0
最小节点状态位置为：150.0 : 100.0
最小节点状态位置为：150.0 : 150.0
最小节点状态位置为：100.0 : 150.0
拿到钥匙
最小节点状态位置为：100.0 : 200.0
最小节点状态位置为：100.0 : 150.0
最小节点状态位置为：100.0 : 100.0
最小节点状态位置为：100.0 : 50.0
游戏胜利
是否有路径：true
第0步执行的动作为：null
第1步执行的动作为：ACTION_DOWN
第2步执行的动作为：ACTION_RIGHT
第3步执行的动作为：ACTION_DOWN
第4步执行的动作为：ACTION_LEFT
第5步执行的动作为：ACTION_DOWN
第6步执行的动作为：ACTION_UP
第7步执行的动作为：ACTION_UP
第8步执行的动作为：ACTION_UP
第9步执行的动作为：ACTION_LEFT
Result (1->win; 0->lose):1, Score:5.0, timesteps:9
Controller tear down time: 0 ms.
```

About Task4

首先先大致介绍一下蒙特卡洛树搜索（MCTS）算法， MCTS是一种用于某些类型的问题求解的启发式搜索算法，可以处理高分支因子的情况，并且不需要游戏状态的精确评估函数。 MCTS算法的基本步骤如下：

1. 选择Selection：从根节点开始，按照特定的策略（比如UCT，即上置信界限）选择最有前途的子节点，直到达到一个尚未完全展开（即还有未探索的动作）的节点
2. 扩展Expansion：除非所选节点是一个游戏结束的节点，否则在所选节点上添加一个或多个子节点
3. 模拟Simulation：从新添加的节点开始进行模拟（也称为“rollout”或“playout”），直到游戏结束或达到预定的模拟深度
4. 回溯Backpropagation：将模拟的结果更新到所访问路径上的所有节点

那在这里MCTS算法又是怎么实现的呢？在具体介绍代码实现之前，跟之前一样先明确一些变量含义：

- HUGE_NEGATIVE 和 HUGE_POSITIVE：用于处理游戏结束时的分数。
- epsilon：用于数值稳定性和随机性的一个小值。
- egreedyEpsilon：ε-greedy策略中的概率阈值。
- state：当前节点的游戏状态。
- parent：父节点。
- children：子节点数组。
- totValue：节点的总值，用于评估。
- nVisits：节点被访问的次数。
- m_rnd：随机数生成器。
- m_depth：节点的深度。
- bounds：用于归一化UCT值的边界。

对于SingleTreeNode的构造函数太简单了就不多说了，初始化节点状态、父节点、随机数生成器、子节点数组、总值、访问次数以及设置节点深度。

更重要的其实是SingleTreeNode类当中的MCTS搜索方法，在给定的时间内进行多次迭代，每次迭代中，使用treePolicy选择节点，然后进行模拟（rollOut），并将结果回溯（backUp）。

对于树策略treePolicy选择节点这一步，从当前节点开始，如果游戏未结束且深度小于模拟深度，则进行选择：如果节点未完全展开，则进行扩展——随机选择一个下一步动作，复制当前状态并应用动作，创建新节点并将其添加到子节点数组中并返回扩展的结果；否则，使用UCT策略（一种置信上界公式）选择子节点，使用UCT公式选择子节点，计算每个子节点的UCT值，并选择最大的一个

对于模拟和评估方法，从当前状态开始进行模拟，直到游戏结束或达到模拟深度，使用value方法评估最终状态，然后根据游戏结束状态来评估分数

对于回溯方法backUp,将模拟结果灰度到路径上的所有节点，采用链表的思想`n=n.parent`不断往上回溯直到n!=null，每一次act之后都会再次调用MCTS方法，然后返回较优的动作action，但是选择action的方法又有2种：

```
int action = m_root.mostVisitedAction();
int action = m_root.bestAction();
```

- mostVisitedAction方法是选择子节点中被访问次数最多的动作。 它基于的假设是，被访问得越多的节点，其统计结果越稳定，因此可能是较好的选择。 这种策略倾向于选择那些已经探索得比较充分的路径，可能会忽略那些有潜力但尚未充分探索的路径。

- bestAction方法选择具有最佳值的子节点，通常是根据某种评估标准，如UCT（Upper Confidence bounds applied to Trees）置信上界值或其他值。 它不仅考虑了节点被访问的次数，还考虑了节点的胜率或其他评估指标。 这种策略试图平衡探索和利用，即在已知的好选择和可能更好的未知选择之间做出平衡。

但是经过测试之后发现MCTS方法其实并不是非常稳定，虽然其在bait_lvl0关卡中表现非常优异，但是在其他关卡中在一开始就被卡住了，经常在起始点附近来回震荡，导致陷入了死循环，个人分析认为一方面可能是评价函数不是非常完善，应该一开始比较宽松，让player拥有更多的选择权，更偏向于探索，之后越往后面越偏向于精确（怎么有一种模拟退火的思想在里面）；另一方面也有可能是模拟的深度太浅了，在一开始不管往那个方向去最终每个节点几乎都是失败的（因为实际行动链太长了），所以utility基本上都是差不多的，很难有什么明确胜利的指向，但是越往后随着方向更加明确了，MCTS方法应该是越来越好的。

一些实验日志

2024.09.04 第一次阅读整个GVG-AI框架 耗时12h

讲真的第一次硬生生阅读一整个完整的框架太困难了……

首先是一大堆的java类堆叠在一起，而且注释和文档全都是用英文去写的，外加自身英文水平不高以及第一次尝试着去理解这个框架到底是啥，这些debuff放到一起，我看了整整一个晚上才只能说粗略地搞明白自己要写什么东西……太让人头疼啦

我先自己去大致浏览了一遍网上关于java的教程，发现其确实和C/C++语法很像（但是自己从来没有阅读过全是类和对象的代码啊），然后看网页上的manual发现还是看不懂啊，甚至于说我都不知道这个游戏让我写的代码应该写在哪个地方。因为以前从来没有自己独立/合作写过完整的大项目，所有的代码文件全都放在一个.cpp或者.py文件里面，对于java的package管理完全没有概念

后来看了一个半小时完全看不懂只好放弃，等过了两三个小时之后静下心重新去看，然后找找规律，对对应的对象进行跳转之后才明白那个Assignment到底想让我去干嘛了

首先我配置完java环境之后我去尝试run那个Test.java文件，结果发现可以运行，然后通过人的上下左右键的移动实现推箱子，那好接下来去细看Test.java里面的内容发现里面其实核心代码就一条：

```
ArcadeMachine.playOneGame( "examples/gridphysics/bait.txt",
"examples/gridphysics/bait_lvl0.txt", null, new Random().nextInt());
```

好现在问题来了，看不懂……看不懂怎么办？跳转过去看看playOneGame是怎么个事儿，结果发现跳转到了\src\core\ArcadeMachine.java这个文件里面，对应里面会发现有playOneGame，还有一个runOneGame，名字差不多诶，难道有啥区别吗？结合着null和Assignment里面的runOneGame类方法估计是区分人机/人类玩家吧，里面具体实现就先不看了。

要不去网上搜搜？看看有没有人用过GVGAI，有没有什么入门的小技巧，别说还真找到了一个有关于gvgai框架搭建以及controllers编写的介绍，稍微看一看，我感觉重点应该在Agent.java这个程序里面，然后我再去重新查看了一下gvgai-assignment1文件夹下面的东西，发现src目录下面有好多有关于Agent.java的文件，难道它们

都一样，再看一看它们归属的目录，猜一猜估计是用不同的方法对应不同的智能体，然后细看发现有MCTS蒙特卡洛树搜索的sample（实例），还有GA遗传算法的sample，还有OLMCTS离线蒙特卡洛树搜索，而这里我们将要使用的是深度优先DFS算法，估计和它们的目录结构是一样的，回去再看看Assignment里面的内容：

```
String depthfirstController = "controllers.depthfirst.Agent";
String limitdepthfirstController = "controllers.limitdepthfirst.Agent";
String AstarController = "controllers.Astar.Agent";
String sampleMCTSController = "controllers.sampleMCTS.Agent";
```

果然对应的四个任务正好对应的是四个智能体agent对应的目录，看来这个目录还需要自己去创建，还不是提前准备好的（哭笑不得），行吧行吧……

接下来要去做什么呢：看懂controllers.sampleMCTS和controllers.sampleGA两个目录下它们的java脚本是如何写的 然后读懂之后看看自己能不能模仿一下写三个java文件出来（实在不行写在一个java文件里面也行） 读懂完以后先不要急着马上去写代码，看明白是怎么运行之后去学一下java的语言，等熟练掌握语言之后再去写深度优先之类的算法（但凡让我用python去写根本不在怕的 但是用一个全新的java语言去写还是得小心了）

注意：为确保学术诚信，此处进行声明，由于第一次接触Java语言以及GVG-AI框架和这么大的项目，看懂框架并且调用对应接口方法过于困难，仅仅是Task1深度优先搜索就花了3天仍然毫无所获。为了不再浪费时间，其中Task1确实阅读过别人代码（链接在这里），弄明白可以调用哪些接口之后**自己再重新写的**（即写代码过程中是没有看任何资料的），其余所有任务的源代码都是**自己独立完成的**，没有与其余任何人进行交流，也**没有借鉴任何人的代码或者思路**