

---

---

# 作业 1: Bait 游戏实验报告

丁云翔 ( 191250026、191250026@smail.nju.edu.cn )

## 1 任务 1

在 `controllers.DepthFirst.java` 文件中实现了深度优先搜索，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。第一关运行成功

类成员变量：

```
ArrayList<StateObservation> pastState = new ArrayList<>(); // 存储走过的状态
ArrayList<Types.ACTIONS> Actions = new ArrayList<>(); // 存储走过的动作
int now = 0; // act函数中输出动作的数组下标
boolean flag = false; // 是否已搜索到路径
```

`equalTest` 函数：

```
boolean equalTest(StateObservation state){
    for(StateObservation so : pastState){
        if(state.equalPosition(so)){
            return true;
        }
    }
    return false;
} // 判断当前状态之前是否到达过
```

`getDepthFirstActions` 函数：

```

boolean getDepthFirstActions(StateObservation stateObs, ElapsedCpuTimer elapsedTimer){
    pastState.add(stateObs); //将当前状态加入已走过的状态
    for(Types.ACTIONS action : stateObs.getAvailableActions()){ //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，用于模拟施加动作
        stCopy.advance(action); //施加动作
        Actions.add(action); //将当前动作加入已走过的动作
        if(stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS){
            return true; //如果获胜则返回已找到
        }
        if(equalTest(stCopy) || stCopy.isGameOver()){
            Actions.remove( index: Actions.size() - 1); //如果动作施加后的状态之前已经到达过或者游戏失败，则尝试下一个动作
        }
        else{ //动作施加后的是一个新的状态
            if(getDepthFirstActions(stCopy,elapsedTimer)){ //递归进行深度优先搜索
                return true;
            }
            else {
                Actions.remove( index: Actions.size() - 1); //当前动作递归搜索失败，尝试下一个动作
            }
        }
    }
    pastState.remove( index: pastState.size() - 1); //当前局面没有办法成功，删除当前局面
    return false;
}

```

act 函数：

```

public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
    ArrayList<Observation>[] npcPositions = stateObs.getNPCPositions();
    ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
    ArrayList<Observation>[] resourcesPositions = stateObs.getResourcesPositions();
    ArrayList<Observation>[] portalPositions = stateObs.getPortalsPositions();
    grid = stateObs.getObservationGrid();

    /*println(npcPositions, "npc");
    println(fixedPositions, "fix");
    println(movingPositions, "mov");
    println(resourcesPositions, "res");
    println(portalPositions, "por");
    System.out.println(); */

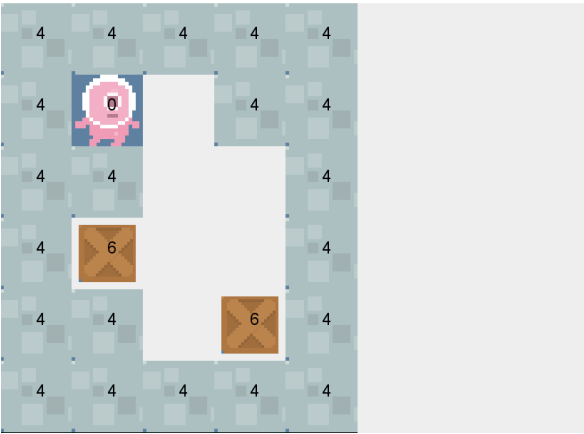
    if(!flag){
        flag = getDepthFirstActions(stateObs, elapsedTimer); //如果没有搜索过则进行搜索，搜索成功后将flag置为true，并且已将路径存储在Actions中
    }

    if(now < Actions.size()){
        now++;
        return Actions.get(now - 1); //搜索到路径后将Actions中的动作依次执行
    }
    return null;
}

```

第一关运行结果：

Java-VGDL: Score:5.0. Tick:10 [Player WINS!]



## 2 任务 2

在 `controllers.LimitedDepthFirst.java` 中实现了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，但不需要一定搜索到通关，而是搜索到一定的深度，根据启发式函数 `distance` 判断局面好坏，决定接下来的路径。第一关运行成功。

类成员变量：

```

ArrayList<StateObservation> pastState = new ArrayList<>(); // 存储走过的状态
ArrayList<Types.ACTIONS> Actions = new ArrayList<>(); // 存储走过的动作
ArrayList<Types.ACTIONS> bestAction = new ArrayList<>(); // 存储搜索过的路径中最优的动作
double bestScore = 10000; // 已走过的路径的最优评分（值越小越好）
boolean hasKey = false; // 是否已经拿到钥匙
int singleSearchDepth = 0; // 当前搜索的深度
int searchDepth = 5; // 规定的受限搜索深度（综合搜索时间和准确性确定）
Vector2d goalpos; // 目标的位置
Vector2d keypos; // 钥匙的位置
double goal_keyDistance; // 钥匙与目标之间的曼哈顿距离

```

`equalTest` 函数：

```

boolean equalTest(StateObservation state){ // 判断当前状态之前是否到达过
    for(StateObservation so : pastState){
        if(state.equalPosition(so)){
            return true;
        }
    }
    return false;
}

```

启发式函数 `distance`：

```

double distance(StateObservation stateObs){ //利用精灵的位置、目标的位置和钥匙的位置构造启发式函数distance
    Vector2d playerpos = stateObs.getAvatarPosition(); //精灵的位置
    if(hasKey){
        return Math.abs(goalpos.x - playerpos.x) + Math.abs(goalpos.y - playerpos.y); //如果已经拿到钥匙，则返回精灵与目标的曼哈顿距离
    }
    else{
        for(StateObservation so : pastState){ //如果在当前搜索的走过的状态中，精灵已经到过钥匙所在的位置，则返回精灵与目标的曼哈顿距离
            if(so.getAvatarPosition().equals(keypos)){
                return Math.abs(goalpos.x - playerpos.x) + Math.abs(goalpos.y - playerpos.y);
            }
        }
    }
    return Math.abs(playerpos.x - keypos.x) + Math.abs(playerpos.y - keypos.y) + goal_keyDistance;
    //否则精灵无钥匙，返回精灵与目标的曼哈顿距离和钥匙与目标的曼哈顿距离的和
}
}

```

### getDepthFirstActions 函数:

```

void getLimitDepthFirstActions(StateObservation stateObs, ElapsedCpuTimer elapsedTimer){
    pastState.add(stateObs); //将当前状态加入已走过的状态中
    singleSearchDepth++; //搜索深度加一
    for(Types.ACTIONS action : stateObs.getAvailableActions()){ //尝试当前局面所有可能的动作
        StateObservation stCopy = stateObs.copy(); //新建一个当前状态的副本，用于模拟施加动作
        stCopy.advance(action); //施加动作
        Actions.add(action); //将当前动作加入已走过的动作
        if(singleSearchDepth == searchDepth){ //如果当前搜索的深度等于受限的深度
            double score = distance(stateObs); //根据启发式函数计算当前局面的评分
            if(score < bestScore){ //如果评分小于之前的最优评分，则更新最优解
                bestAction = (ArrayList<Types.ACTIONS>) Actions.clone();
                bestScore = score;
            }
        }
        else if(stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS){ //如果还没到受限的搜索深度就已经胜利，则根据受限深度与当前搜索深度的差确定评分
            double score = -50 * (searchDepth - singleSearchDepth);
            if(score < bestScore){ //如果评分小于之前的最优评分，则更新最优解
                bestAction = (ArrayList<Types.ACTIONS>) Actions.clone();
                bestScore = score;
            }
        }
        Actions.remove( index: Actions.size() - 1); //因为执行该动作后已胜利，故当前局面的其他动作已无搜索的必要，可以直接返回
        singleSearchDepth--;
        pastState.remove( index: pastState.size() - 1);
        return;
    }
    else if(equalTest(stCopy) || stCopy.isGameOver()){ //如果如果动作施加后的状态之前已经到达过或者游戏失败，不进行操作
    }
    else{ //动作施加后的是一个新的状态
        getLimitDepthFirstActions(stCopy, elapsedTimer); //递归进行深度优先搜索
    }
    Actions.remove( index: Actions.size() - 1); //移除当前施加的动作，尝试另外的动作
}
singleSearchDepth--; //当前状态的搜索结束，返回上一层
pastState.remove( index: pastState.size() - 1);
return;
}
}

```

### act 函数:

```

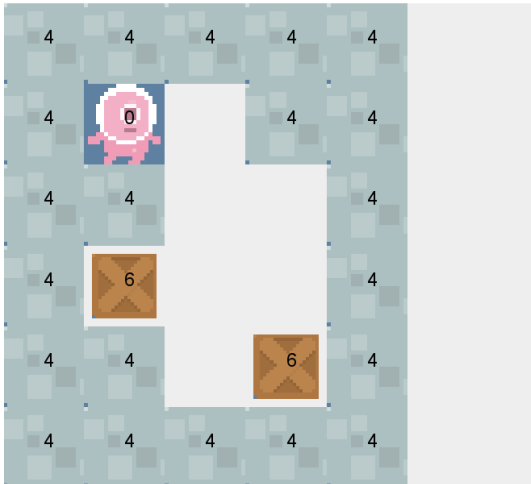
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {

    if(stateObs.getAvatarPosition().equals(keypos)){           // 如果当前状态下精灵在钥匙的位置，则精灵拥有了钥匙
        hasKey = true;
    }
    if(singleSearchDepth == 0 && !hasKey){                     // 初始化目标位置、钥匙位置和钥匙与目标之间的曼哈顿距离
        goalpos = stateObs.getImmovablePositions()[1].get(0).position;
        keypos = stateObs.getMovablePositions()[0].get(0).position;
        goal_keyDistance = Math.abs(goalpos.x - keypos.x) + Math.abs(goalpos.y - keypos.y);
    }
    singleSearchDepth = 0;                                     // 初始化当前搜索深度，最佳评分，动作集以及最佳动作集
    bestScore = 10000;
    Actions = new ArrayList<Types.ACTIONS>();
    bestAction = new ArrayList<Types.ACTIONS>();
    pastState.add(stateObs);                                  // 将当前状态加入已走过的状态中
    getLimitDepthFirstActions(stateObs, elapsedTimer);       // 每一步都进行一次深度受限的深度优先搜索，并将最佳动作集存储在bestAction中
    return bestAction.get(0);                                 // 执行最佳动作集的第一步
}

```

第一关运行结果：

 Java-VGDL: Score:5.0. Tick:8 [Player WINS!]



### 3 任务 3

在任务 2 的基础上，将深度优先搜索换成 A\* 算法，在 `controllers.Astar.Agent.java` 中实现。另外还用到了在 `controllers.Astar.Node.java` 中定义的 `Node` 类。第一、二、三关运行成功。

`Node` 类：

```

public class Node {
    public Node(StateObservation stateObs, double score, ArrayList<Types.ACTIONS> actions, ArrayList<StateObservation> pastState, boolean hasKey) {
        this.stateObs = stateObs.copy();
        this.score = score;
        this.actions = (ArrayList<Types.ACTIONS>) actions.clone();
        this.pastState = (ArrayList<StateObservation>) pastState.clone();
        this.hasKey = hasKey;
    }
    //初始化

    StateObservation stateObs;           //当前节点的状态
    double score;                         //当前节点的评分
    ArrayList<Types.ACTIONS> actions;     //当前节点的已走过的路径
    ArrayList<StateObservation> pastState; //当前节点的已走过的状态
    boolean hasKey;                       //当前节点是否已经拥有钥匙

    public Node parent;
}

```

### Agent 类成员变量:

```

ArrayList<StateObservation> pastState = new ArrayList<>();           //存储所有的走过的状态
ArrayList<StateObservation> targetPastState = new ArrayList<>();     //存储当前节点的走过的状态

Comparator<Node> OrderDistance = Comparator.comparingDouble(o -> o.score);
PriorityQueue<Node> openState = new PriorityQueue<>(OrderDistance); //定义一个按局面评分比较的优先队列, 用来存储还未展开的节点

ArrayList<Types.ACTIONS> Actions = new ArrayList<>();               //存储走过的动作
int now = -1;                                                       //act函数中输出动作的数组下标
Vector2d goalpos;                                                  //目标的位置
Vector2d keypos;                                                   //钥匙的位置
double goal_keyDistance;                                           //钥匙与目标之间的曼哈顿距离
boolean hasKey = false;                                            //是否找到钥匙
int searchDepth = 32;                                              //限制A*算法的搜索深度, 当地图较大时也能得出结果
*/
除第一关外, 后续的其他关卡在ACTION_TIME = 100的条件下均无法完成搜索, 故采用searchDepth参数来限制搜索深度。经过对不同的searchDepth进行测试,
当searchDepth = 28时后续关卡可以通过且用时较少, ACTION_TIME = 1500即可完成搜索, 但通关所需的步数较多; 当searchDepth = 32时通关所需步数较少,
且搜索所需的时长相对较少, ACTION_TIME = 4000即可完成搜索, 所以将searchDepth设置为32。
*/

```

### pastEqualTest 函数:

```

boolean pastEqualTest(StateObservation state){           //判断当前状态之前是否到达过
    for(StateObservation so : pastState){
        if(state.equalPosition(so)){
            return true;
        }
    }
    return false;
}

```

### equalNode 函数:

```

Node equalNode(StateObservation state){                 //如果当前状态在待展开的节点所含的状态中, 返回对应节点, 否则返回null
    for(Node node : openState){
        if(state.equalPosition(node.stateObs)){
            return node;
        }
    }
    return null;
}

```

启发式函数 distance:

```
double distance(StateObservation stateObs, boolean hasKey){ //利用精灵的位置、目标的位置和钥匙的位置以及已走过的步数构造启发式函数distance
    Vector2d playerpos = stateObs.getAvatarPosition(); //精灵的位置
    if(hasKey){
        return Math.abs(goalpos.x - playerpos.x) + Math.abs(goalpos.y - playerpos.y) + (Actions.size() * 50); //如果已经拿到钥匙
    }
    return Math.abs(playerpos.x - keypos.x) + Math.abs(playerpos.y - keypos.y) + goal_keyDistance + (Actions.size() * 50); //如果还没拿到钥匙
}
```

getAStarActions 函数:

```
void getAStarActions(StateObservation stateObs, ElapsedCpuTimer elapsedTimer){
    openState = new PriorityQueue<Node>(OrderDistance); //初始化待展开节点
    pastState = (ArrayList<StateObservation>) targetPastState.clone(); //初始化所有已走过状态为上轮搜索后实际走过的状态
    Actions = new ArrayList<Types.ACTIONS>(); //初始化已走过的动作
    Node startNode = new Node(stateObs,distance(stateObs, hasKey), Actions, targetPastState, hasKey); //新建初始节点并加入openState中
    openState.add(startNode);
    while(!openState.isEmpty()) { //只要还有待展开节点就继续搜索
        Node temp = openState.poll(); //选取评分最优的节点temp
        Actions = (ArrayList<Types.ACTIONS>) temp.actions.clone(); //将Actions初始化为temp节点存储的已走过动作集
        targetPastState = (ArrayList<StateObservation>) temp.pastState.clone(); //将targetPastState初始化为temp节点存储的已走过状态
        pastState.add(temp.stateObs); //将temp节点的状态加入所有已走过的状态
        targetPastState.add(temp.stateObs); //将temp节点的状态加入当前节点的走过的状态
        if(Actions.size() == searchDepth){ //如果达到搜索深度,则返回,按该最优节点存储的Actions执行动作
            return;
        }
        hasKey = temp.hasKey; //初始化hasKey为当前节点的hasKey
        if(!hasKey){ //如果没有钥匙,则判断精灵位置是否与钥匙位置相同,如果是则有钥匙了
            if(temp.stateObs.getAvatarPosition().equals(keypos)){
                hasKey = true;
            }
        }
        for(Types.ACTIONS action : temp.stateObs.getAvailableActions()){ //尝试当前局面所有可能的动作
            StateObservation stCopy = temp.stateObs.copy(); //新建一个当前状态的副本,用于模拟施加动作
            stCopy.advance(action); //施加动作

            Actions.add(action); //将当前动作加入已走过的动作
            if(stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS) { //如果胜利,则返回,按Actions执行动作
                return; //最终的序列步骤在aStarAction中
            }
            if(stCopy.isGameOver() || pastEqualTest(stCopy)) { //如果如果动作施加后的状态之前已经到达过或者游戏失败,则尝试其他动作
                Actions.remove(index: Actions.size() - 1);
                continue;
            }
            Node equalNode = equalNode(stCopy); //搜索openState中与动作施加后的状态相同的节点
            if(equalNode != null){ //如果存在状态相同的节点
                if(distance(stCopy,hasKey) < equalNode.score){ //如果当前走法优于之前的走法,则更新节点
                    openState.remove(equalNode);
                    openState.add(new Node(stCopy,distance(stCopy,hasKey),Actions,targetPastState, hasKey));
                }
                Actions.remove(index: Actions.size() - 1);
            }
            else{ //动作施加后的是一个新状态,加入新状态的节点
                openState.add(new Node(stCopy,distance(stCopy,hasKey),Actions,targetPastState, hasKey));
                Actions.remove(index: Actions.size() - 1);
            }
        }
    }
}
```

act 函数:

```

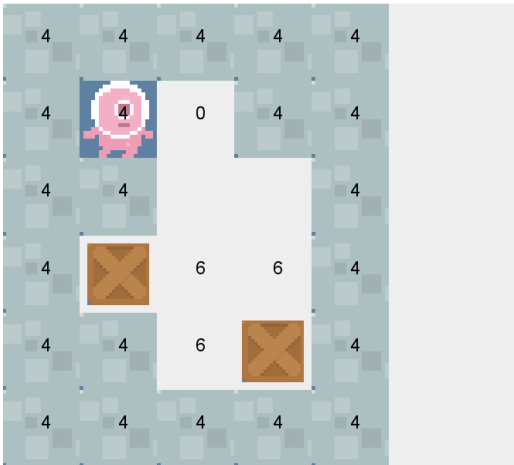
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {

    if(Actions.size() == 0){
        goalpos = stateObs.getImmovablePositions()[1].get(0).position;
        keypos = stateObs.getMovablePositions()[0].get(0).position;
        goal_keyDistance = Math.abs(goalpos.x - keypos.x) + Math.abs(goalpos.y - keypos.y);
    }
    //初始化目标位置、钥匙位置和钥匙与目标之间的曼哈顿距离
    now++;
    //更新下标
    if(now == Actions.size()){
        //如果还未搜索或搜索返回的动作集已经执行完，则基于当前状态继续搜索，并将下标置为0
        getAStarActions(stateObs,elapsedTimer);
        now = 0;
    }
    return Actions.get(now);
    //返回动作
}
}

```

第一关运行结果：

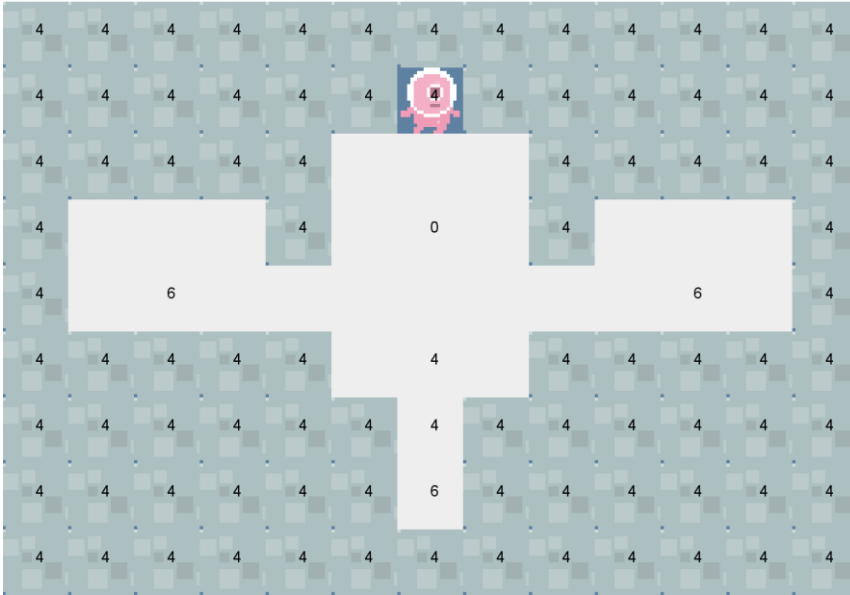
Java-VGDL: Score:5.0. Tick:8 [Player WINS!]



第二关运行结果：

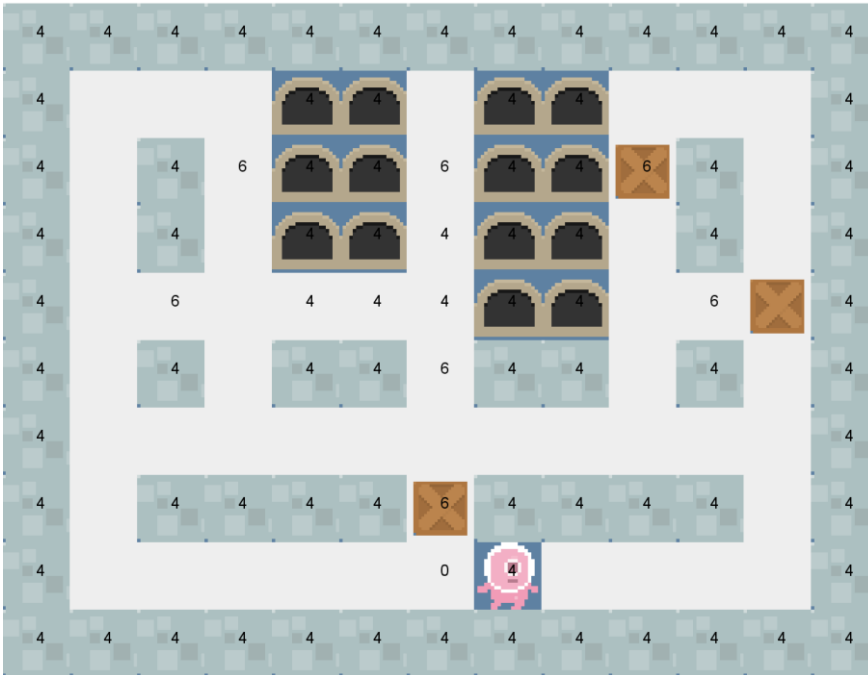


Java-VGDL: Score:7.0. Tick:37 [Player WINS!]



第三关运行结果:

Java-VGDL: Score:9.0. Tick:78 [Player WINS!]



#### 4 任务 4

阅读 `controllers.sampleMCTS.Agent.java` 控制程序，并介绍其算法。

算法介绍:

首先在创建 Agent 对象时进行初始化, 并创建 MCTSPlayer:

```
public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer)
{
    //Get the actions in a static array.
    ArrayList<Types.ACTIONS> act = so.getAvailableActions();
    actions = new Types.ACTIONS[act.size()];
    for(int i = 0; i < actions.length; ++i)
    {
        actions[i] = act.get(i);
    }
    NUM_ACTIONS = actions.length;

    //Create the player.
    mctsPlayer = new SingleMCTSPlayer(new Random());
}
```

然后调用 act 函数, 并利用 stateObs 对 mctsplayer 对象进行初始化 (创建新的 SingleTreeNode 节点), 然后调用 mctsplayer 的 run 方法获得下一步的动作的下标, 并将动作返回:

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
    ArrayList<Observation> obs[] = stateObs.getFromAvatarSpritesPositions();
    ArrayList<Observation> grid[][] = stateObs.getObservationGrid();

    //Set the state observation object as the new root of the tree.
    mctsPlayer.init(stateObs);

    //Determine the action using MCTS...
    int action = mctsPlayer.run(elapsedTimer);

    //... and return it.
    return actions[action];
}
```

在 run 方法中, 调用新建节点的 mctsSearch 方法, 并用 mostVisitedAction 方法得到最优的动作的下标, 返回该下标, 外层的 act 方法通过该下标返回最优动作:

```
public int run(ElapsedCpuTimer elapsedTimer)
{
    //Do the search within the available time.
    m_root.mctsSearch(elapsedTimer);

    //Determine the best action to take and return it.
    int action = m_root.mostVisitedAction();
    //int action = m_root.bestAction();
    return action;
}
```

在 `mctsSearch` 方法中，在时间允许的范围內不断循环调用三个方法，首先是用 `treePolicy` 方法选取待探索的树节点，然后调用该树节点的 `rollOut` 方法对节点进行探索，并获取到 `delta` 值，然后调用 `backup` 方法回溯，将 `delta` 值加到被选取的树节点的所有父节点的 `totValue` 上，并且所有父节点的 `nVisits` 增加 1，再重复循环，在时间允许的范围內尽可能多地探索节点：

```
public void mctsSearch(ElapsedCpuTimer elapsedTimer) {

    double avgTimeTaken = 0;
    double acumTimeTaken = 0;
    long remaining = elapsedTimer.remainingTimeMillis();
    int numIters = 0;

    int remainingLimit = 5;
    while(remaining > 2*avgTimeTaken && remaining > remainingLimit){
        ElapsedCpuTimer elapsedTimerIteration = new ElapsedCpuTimer();
        SingleTreeNode selected = treePolicy();
        double delta = selected.rollOut();
        backUp(selected, delta);

        numIters++;
        acumTimeTaken += (elapsedTimerIteration.elapsedMillis());

        avgTimeTaken = acumTimeTaken/numIters;
        remaining = elapsedTimer.remainingTimeMillis();
        //System.out.println(elapsedTimerIteration.elapsedMillis() + " --> " + acumTimeTaken + " (" + remaining + ")");
    }
    //System.out.println("-- " + numIters + " -- (" + avgTimeTaken + ")");
}
}
```

`treePolicy` 方法如下，只要游戏没有结束并且搜索深度小于指定的深度，就不断寻找未完全探索的节点，如果找到了，就调用该节点的 `expand` 方法，返回一个随机选取一个动作模拟执行后创建的新节点；否则将把用 `cur` 的 `uct` 方法获得的节点赋给 `cur`，再继续循环，将更多值得展开的节点展开：

```
public SingleTreeNode treePolicy() {

    SingleTreeNode cur = this;

    while (!cur.state.isGameOver() && cur.m_depth < Agent.ROLLOUT_DEPTH)
    {
        if (cur.notFullyExpanded()) {
            return cur.expand();
        } else {
            SingleTreeNode next = cur.uct();
            //SingleTreeNode next = cur.egreedy();
            cur = next;
        }
    }

    return cur;
}
}
```

`uct` 方法的核心是计算 `uctValue` 的值，从计算公式中可以看出，平均分值越高并且访问次数越少的节点 `uctValue` 越高，越容易被选中，返回的节点在 `treePolicy` 方法中被选取并探索：

```

public SingleTreeNode uct() {

    SingleTreeNode selected = null;
    double bestValue = -Double.MAX_VALUE;
    for (SingleTreeNode child : this.children)
    {
        double hvVal = child.totValue;
        double childValue = hvVal / (child.nVisits + this.epsilon);

        childValue = Utils.normalise(childValue, bounds[0], bounds[1]);

        double uctValue = childValue +
            Agent.K * Math.sqrt(Math.Log(this.nVisits + 1) / (child.nVisits + this.epsilon));

        // small sampleRandom numbers: break ties in unexpanded nodes
        uctValue = Utils.noise(uctValue, this.epsilon, this.m_rnd.nextDouble()); //break ties randomly

        // small sampleRandom numbers: break ties in unexpanded nodes
        if (uctValue > bestValue) {
            selected = child;
            bestValue = uctValue;
        }
    }
}

```

rollOut 方法从当前状态开始不断随机选取动作直达到搜索深度或者游戏结束，然后通过 value 方法获取 delta 值并返回：

```

public double rollOut()
{
    StateObservation rollerState = state.copy();
    int thisDepth = this.m_depth;

    while (!finishRollout(rollerState, thisDepth)) {

        int action = m_rnd.nextInt(Agent.NUM_ACTIONS);
        rollerState.advance(Agent.actions[action]);
        thisDepth++;
    }

    double delta = value(rollerState);

    if(delta < bounds[0])
        bounds[0] = delta;

    if(delta > bounds[1])
        bounds[1] = delta;

    return delta;
}

```

搜索结束后，调用 mostVisitedAction 方法得到最优的（访问次数最多的）节点的下标：

```

public int mostVisitedAction() {
    int selected = -1;
    double bestValue = -Double.MAX_VALUE;
    boolean allEqual = true;
    double first = -1;

    for (int i=0; i<children.length; i++) {

        if(children[i] != null)
        {
            if(first == -1)
                first = children[i].nVisits;
            else if(first != children[i].nVisits)
            {
                allEqual = false;
            }

            double childValue = children[i].nVisits;
            childValue = Utils.noise(childValue, this.epsilon, this.m_rnd.nextDouble()); //break ties randomly
            if (childValue > bestValue) {
                bestValue = childValue;
                selected = i;
            }
        }
    }
}

```

如果所有节点的访问次数都相等，就再调用 `bestAction` 方法，得到平均分最高的节点下标并返回：

```

public int bestAction()
{
    int selected = -1;
    double bestValue = -Double.MAX_VALUE;

    for (int i=0; i<children.length; i++) {

        if(children[i] != null) {
            double childValue = children[i].totValue / (children[i].nVisits + this.epsilon);
            childValue = Utils.noise(childValue, this.epsilon, this.m_rnd.nextDouble()); //break ties randomly
            if (childValue > bestValue) {
                bestValue = childValue;
                selected = i;
            }
        }
    }


    if (selected == -1)
    {
        System.out.println("Unexpected selection!");
        selected = 0;
    }

    return selected;
}

```

所以每一步在 `act` 函数中对 `mctsplayer` 对象进行初始化，并调用对象的 `run` 方法搜索和选取动作，经过这样的不断探索和展开以后，都可以得到一个最优的 `action` 并执行，最终获得胜利。

第一关运行结果:

 Java-VGDL: Score:5.0. Tick:8 [Player WINS!]

