Bait 游戏实验报告

张运吉 (211300063、 ✉211300063@smail.nju.edu.cn)

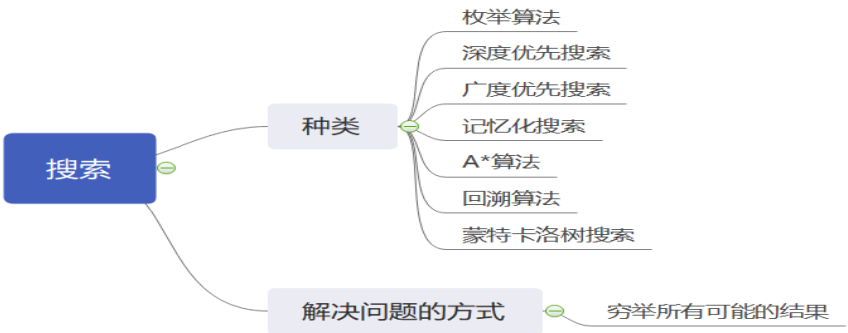
(南京大学人工智能学院，南京 210093)

摘 要: 南京大学 2022 年秋季学期人工智能导论课程作业报告，

关键词: 深度优先搜索、深度限制的 dfs、Asatr 算法、蒙特卡洛树搜索

1 引言

本次实验旨在通过一些小游戏实现课堂上讲的一些搜索算法，游戏使用 GVG-AI 框架开发，GVG-AI 框架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。本次作业使用一种推箱子游戏“Bait”。



2 实验内容

2.1 深度优先搜索

2.1.1 原理

深度优先搜索算法（Depth-First-Search，DFS）是一种用于遍历或搜索树或图的算法。这个算法会尽可能深地搜索树的分支。当节点 v 的所在边都已被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。

2.1.2 具体实现

2.1.2.1 Agent 类成员的定义

```
// 记录搜索的路径
private static ArrayList<Types.ACTIONS> path;
// 判断是否找到路径
private boolean is_found;
```

```
    // 记录搜索过的状态
    private static ArrayList<StateObservation> used_state;
```

2.1.2.2 成员函数的定义

dfs 是寻路的主要方法，实现思路大致是从初始状态开始，逐步向下搜索下，它会依次遍历当前位置可以走的所有状态，并依次判断状态是否重复，游戏是否结束，如果重复则继续搜索其他状态，如果游戏结束则判断赢家是不是 PLAYER，若是则把 is_found 置为 true，然后返回，否则继续搜索其他状态.

进行这两个判断后更新状态，继续往下搜索.

最后需要注意的是如果退出 for 循环之后仍然没有找到路，必须把当前状态从 path 数组中删除，否则得不到正确的 path.

```
    /**
     * 深度优先搜索
     * @param statsObs Observation of the current state.
     */
    public void dfs(StateObservation statsObs) {
        // 如果找到了路径,就不要继续搜索了
        if (is_found)   return;

        ArrayList<Types.ACTIONS> actions
=   statsObs.getAvailableActions();

        for (int i = 0; i < actions.size(); i++) {
            Types.ACTIONS action = actions.get(i);
            StateObservation stCopy = statsObs.copy();
            stCopy.advance(action);

            // 判断状态是否重复
            boolean judge = false;
            for (StateObservation so:used_state) {
                if (stCopy.equalPosition(so)) {
                    judge = true;
                    break;
                }
            }
            if (judge)
                continue;
```

```
            // 判断游戏是否结束
            if (stCopy.isGameOver()) {
                if (stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS)
{
                    path.add(action);
                    is_found = true;
                    return;
                }
                else  continue;
            }

            // 更新状态,继续往下搜索
            used_state.add(stCopy.copy());
            path.add(action);
            dfs(stCopy.copy());

        }

        if (!is_found) {
            if(path.size() >= 1)
                path.remove(path.size() - 1);
        }
    }
```

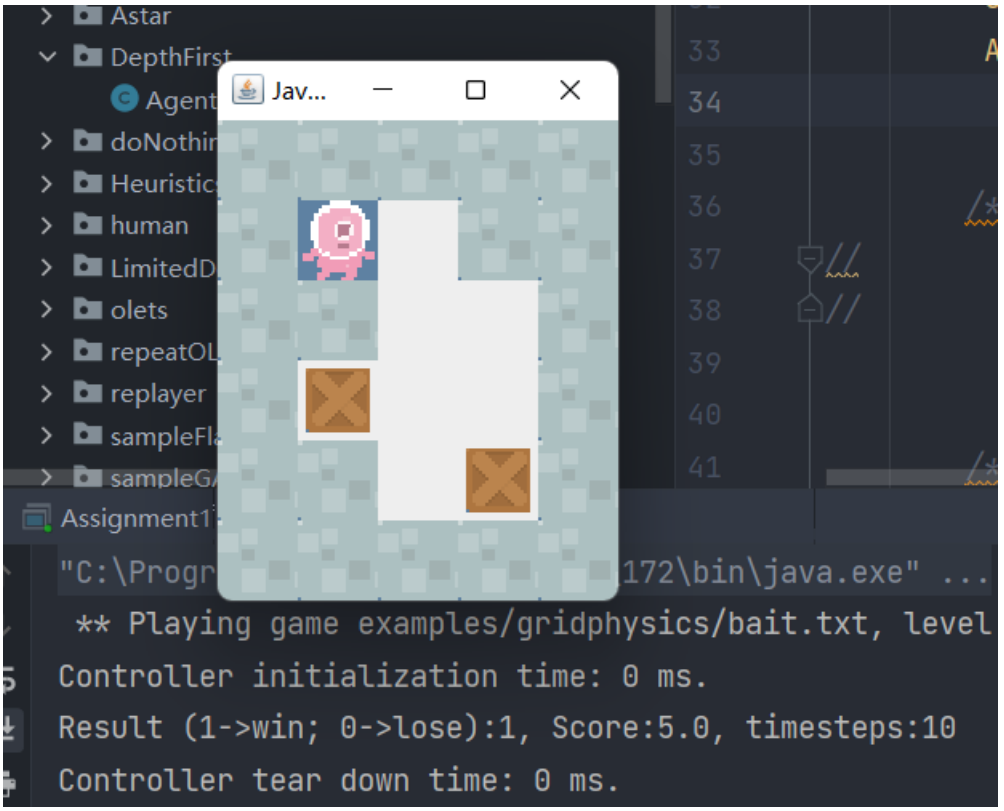
重写 act 函数，一步一步返回 path 中的 action. 这一部分比较简单.

```
    /**
     * Picks an action. This function is called every game step to request
an
     * action from the player.
     * @param stateObs Observation of the current state.
     * @param elapsedTimer Timer when the action returned is due.
     * @return An action for the current state
     */
    @Override
    public Types.ACTIONS act(StateObservation stateObs,
ElapsedCpuTimer elapsedTimer) {

        // 如果没有找到,就先 dfs
```

```
        if(!is_found) {
            dfs(stateObs);
        }
        if (path.size() >= 1) {
            Types.ACTIONS action = path.get(0);
            path.remove(0);
            return action;
        }
        else return null;
    }
```

2.1.3 实现效果

可以成功通过第一关!



2.2 深度受限的dfs

2.2.1 原理

迭代加深的深度有限搜索设定一个最大深度 dmax，开始我们把 dmax 设为 1，然后进行深度受限搜索，如果没有找到答案，则让 dmax 加一，并再次进行深度有限搜索，以此类推直到找到目标。这样既可以避免陷入深度无限的分支，同时还可以找到深度最浅的目标解，从而在每一步代价一致的时候找到最优解，再加上其优越的空间复杂度，因此常常作为首选的无信息搜索策略。

2.2.2 具体实现

2.2.2.1 Agent 成员变量

```
    private int max_depth = 3;
    private int max_score = 1000;

    // 是否找到钥匙

    private boolean is_getkey = false;
    Vector2d keypos;

    // 记录搜索过的状态

    private static ArrayList<StateObservation> used_state = new
ArrayList<>();
```

2.2.2.2 成员函数

因为是搜索深度受到限制，所以每一次搜索最多展开 k 层树(这里假设最大搜索深度为 k)，并不一定能一次找到正确结果，取而代之的是返回一个当前最优解，这么做有一个隐患，当前的最优解不一定能使我们最后走向胜利，因为有可能会产生一些不可逆转的副作用(例如箱子卡在墙角，箱子覆盖钥匙等)，因此设计合理的评估函数，能够准确地给出当前最优解是比较重要的部分.

首先介绍一下评估函数：这里使用老师提示的方法，根据目标，钥匙，玩家三者的距离来设计评估函数，如果玩家找到钥匙，就只计算玩家到目标的距离，否则计算玩家到钥匙和钥匙到目标的距离.

这里实现的时候我遇到一个 bug，就是打注释的那一部分，因为玩家拿到钥匙之后钥匙的坐标就没有了，所以不加判断就计算会跳出 exception，所以我使用一个变量 is_getkey 来判断玩家有没有找到钥匙(虽然后面我发现有更好的方法来判断玩家是否拿到钥匙)，这样就能避免这个错误.

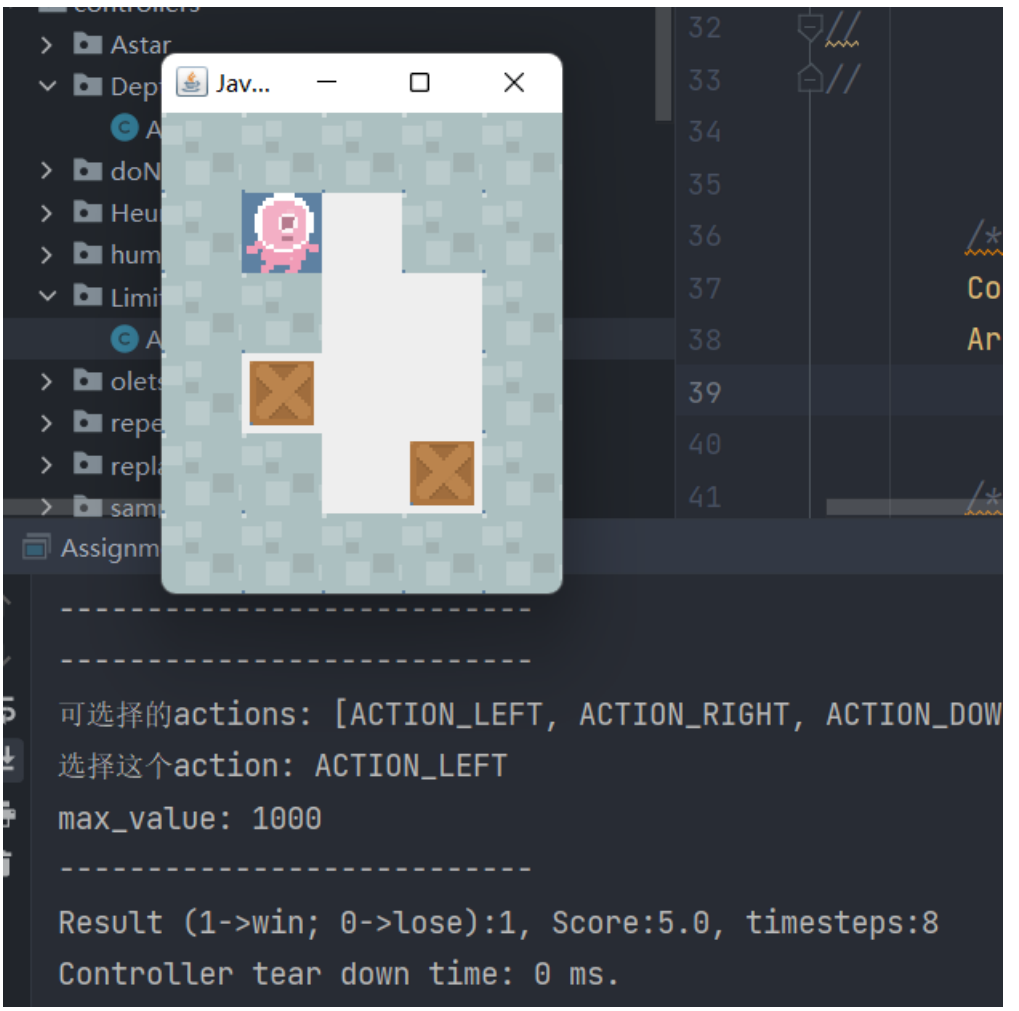
```
private int eval(StateObservation stateObs) {
        ArrayList<Observation>[] fixedPositions =
stateObs.getImmovablePositions();
        ArrayList<Observation>[] movingPositions =
stateObs.getMovablePositions();

        Vector2d goalpos = fixedPositions[1].get(0).position; // 目标

的坐标

        Vector2d playerpos = new

Vector2d(stateObs.getAvatarPosition());// 玩家位置

        //int distance1 = (int)(Math.abs(playerpos.x - goalpos.x) +

Math.abs(playerpos.y - goalpos.y)); // 玩家到目标的距离

        //int distance2 = (int)(Math.abs(goalpos.x - keypos.x) +

Math.abs(goalpos.y - keypos.y)); // 玩家到钥匙的距离

        // 如果找到钥匙,distance 定义为玩家到目标的距离
```

```
        double distance1 = playerpos.dist(goalpos);
        double distance2 = playerpos.dist(keypos);
        if (is_getkey) {
            System.out.println("get key!");
            return max_score - (int)distance1;
        }
        else  {
            return max_score - (int)distance1 - (int)distance2;
        }
```

具体的寻路过程和 dfs 大差不差，所以我就不赘述了.

2.2.3 实现效果

成功通过第一关!



2.3 AStar算法

2.3.1 原理

Astar 算法是一种启发式搜索算法. 启发式搜索是在状态空间中对每一个搜索的位置进行评估，得到最好

的位置，再从这个位置进行搜索直到目标.这样可以省略大量无谓的搜索路径，提高效率。在启发式搜索中，对位置的估价是十分重要的。采用了不同的估价可以有不同的效果.

A*算法启发函数表示为： f(n)=g(n)+h(n)

f(n) 是从初始状态经由状态 n 到目标状态的代价估计

g(n) 是在状态空间中从初始状态到状态 n 的实际代价

h(n) 是从状态 n 到目标状态的最佳路径的估计代价

2.3.2 具体实现

2.3.2.1 Node 节点类

节点类包含五个主要的成员变量，节点对应的状态 public StateObservation me，父亲节点 public Node father，该节点的 f, g, h 值 public double f_val, h_val, g_val.

成员函数: get_h，这里 h 的评估方法和深度受限的 dfs 的评估方法一样

成员函数: change_father:用于修改节点的 father，并且重新计算 g 值

成员函数 copy: 返回一个和当前节点一样的节点，主要为了防止引用时产生副作用

重写 compareTo 函数: 用来进行 node 类大小的判断

```
// 用来进行 node 类的比较，用在最小优先队列
    @Override
    public int compareTo(Node o) {
        return (int)(this.f_val - o.f_val);
    }


    // copy 节点,防止引用时产生副作用

    public Node copy() {
        return new Node(this.me, this.father);
    }
```

2.3.2.2 Astar 寻路

我理解的 Astar 搜索是 bfs 搜索的一种改进版，在 bfs 基础上增加评估函数，这样可以避免搜索一些没有必要的节点，可以提高效率.

实现步骤:

1.把起始格添加到开启列表。

2.重复如下的工作：

a) 寻找开启列表中估量代价 F 值最低的格子。我们称它为当前格。

b) 把它切换到关闭列表。

c) 对相邻的 4 格中的每一个进行如下操作

* 如果它不可通过或者已经在关闭列表中，略过它。反之如下。

* 如果它不在开启列表中，把它添加进去。把当前格作为这一格的父节点。记录这一格的 F,G,和 H 值。

* 如果它已经在开启列表中，用 G 值为参考检查新的路径是否更好。更低的 G 值意味着更好的路径。如果是这样，就把这一格的父节点改成当前格，并且重新计算这一格的 G 和 F 值。如果你保持你的开启列表按 F 值排序，改变之后你可能需要重新对开启列表排序。

d) 停止，当你

* 把目标格添加进了关闭列表(注解)，这时候路径被找到，或者
* 没有找到目标格，开启列表已经空了。这时候，路径不存在。

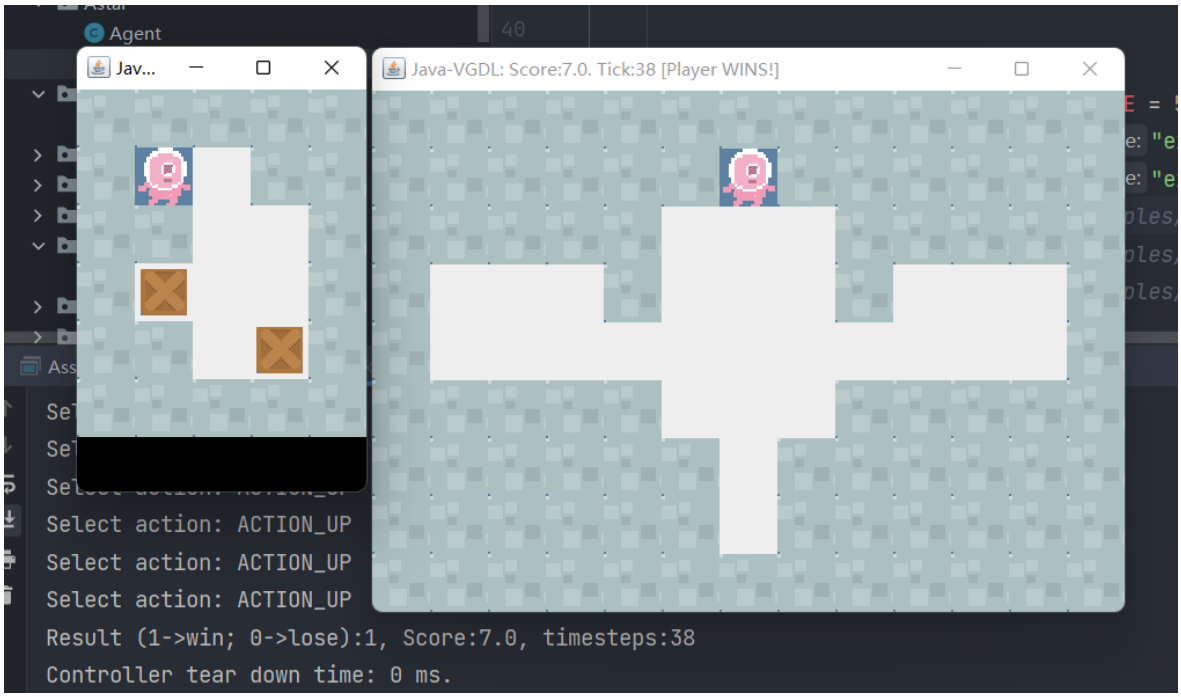
3.保存路径。从目标格开始，沿着每一格的父节点移动直到回到起始格。这就是你的路径。

```
public void AStar(StateObservation stateObs) {

        Node root_node = new Node(stateObs, null);
        this.openList.add(root_node);

        // int depth = 0;
        while (!this.openList.isEmpty()) {
            Node cur_node = this.openList.remove();
            this.closeList.add(cur_node.copy());
            ArrayList<Types.ACTIONS> actions =
cur_node.me.getAvailableActions();
            if (!cur_node.me.isGameOver())
                this.last_node = cur_node;
            for (Types.ACTIONS ac : actions) {
                StateObservation stCopy = cur_node.me.copy();
                stCopy.advance(ac);
                Node son = new Node(stCopy, cur_node);
                // depth += 1;
                // if (depth > this.max_depth) { this.last_node =
cur_node; return; }
                if (stCopy.isGameOver()) {
                    if (stCopy.getGameWinner() ==
Types.WINNER.PLAYER_WINS) {
                        this.last_node = son;
                        this.is_win = true;
                        System.out.println("WIN, SCORE: " +
stCopy.getGameScore());
                        return;
                    } else continue;
                }
                else if (isInClose(son)) continue;
                else if (!isInOpen(son)) this.openList.add(son);
                else if (isInOpen(son)) {
                    double new_g = cur_node.g_val + 50;
                    Node son_copy = getFromOpen(son);
                    if (new_g < son_copy.g_val) {
                        openList.remove(son_copy);
                        openList.add(son);
```

```
                    break;
                }
            }
        }
    }
    System.out.println("No Found!!!");
}
```

2.3.2.3 实现效果

可以通过第一第二关,但不能通过第三关!



2.4 MCTS树搜索

2.4.1 原理

蒙特卡洛树搜索通过随机模拟，获取每一个节点的游戏数据，然后根据游戏数据和节点被访问的次数来决定哪一个是比较好的节点。具体地，一个蒙特卡洛树搜索分为以下步骤：选择，扩展，模拟，回溯。

接下来结合样例代码分享我对蒙特卡洛树搜索的理解。

2.4.2 样例代码解读

首先 Agent 类中有一个 SingleMCTSPlayer 类的成员，这个对象在实例化 Agent 类时通过构造函数创建，通过调用这个成员的 run 方法，可以返回选择的动作。

SingleMCTSPlayer 类中的 run 方法，通过调用 mctsSearch 方法在规定的时间内尽可能地模拟游戏，然后返回被访问次数最多的节点。

```
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;
    }
```

mctsSearch 方法封装在 SingleTreeNode 类里面，主要实现逻辑是，在规定的时间内不断进行以下循环：*首先通过 treePolicy 方法选择一个要扩展的节点（选择，扩展），*然后调用这个节点的 rollOut 方法，rollOut 方法从当前状态开始不断随机选取动作直到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（模拟），delta 值的计算方法主要是看当前胜利的一方，如果游戏没有结束，则返回当前的游戏分数，*通过 backUp 方法，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（回溯）。

```
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
+ ")");
    }
```

接下来介绍 treePolicy 方法：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，调用该节点的 uct 方法，不断往下拓展节点。

```
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，从该公式

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

可以看出，我们的选择会更加倾向于选择那些还没怎么被统计过的孩子节点以及那些评分很高的孩子节点。

```
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;
            }
        }

        if (selected == null)
        {
            throw new RuntimeException("Warning! returning null: " +
bestValue + " : " + this.children.length);
        }

        return selected;
    }
```

rollOut 方法并不复杂，就是一个模拟的过程，从当前状态开始不断随机选择动作知道游戏结束或者搜索深度达到最大。这里就不贴代码了。

在规定的时间内进行足够多次的模拟之后，通过 mostVisitedAction 方法返回被访问次数最多的节点，如果节点的访问次数都一样，就再调用 bestAction 方法，得到平均分值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;
    }
```

3 总结

这次实验主要是实现上课讲的一些搜索算法，自己动手把课堂上讲的算法实现一遍是一件很有成就感的事情，而且经过实践，我更深入理解了这些算法的精髓与一些细节之处.