作业 1: Bait 游戏

孙嘉欣（201240005、201240005@smail）

(南京大学 匡亚明学院大理科班)

关键词: GVG-AI；深度优先搜索;深度受限的深度优先搜索;A*搜索

1 游戏程序结构设计概述

本次作业前三项任务中，基于 GVG-AI 的游戏框架，需分别构建 controllers 包中三种采取不同的搜索方式的 controller。参考了已有的 sampleMCTS 包中的结构（Agent, SingleMCTSPlayer, SingleTreeNode），都构建了 Agent 与 single”xxx”Player 两个类：Agent 类参照了老师课上演示的代码和结构，继承 sampleRandom.Agent，父类中的 debug 和 draw 方法保持不变，用于调试，主要重写 act 方法，通过 act 方法返回下一步的 action，主要负责 Agent 对外交互的功能；single”xxx”Player 类中通过对应的具体算法，负责 action 的搜索；Agent 类包含一个对应 single”xxx”Player 的接口，single”xxx”Player 的搜索结果，由 Agent 逐步对外输出。三类搜索树的节点结构和功能比较相似，因此在任务 1 的 depthfirst 包中完成了 Node 类的全部功能，后两个任务中直接引用。

2 任务 1：深度优先搜索

2.1 Agent类的代码实现

如图 1，Depthfirst.Agent 类的属性中创建了一个 List 用来保存一次性搜索出的全部 action。当 List 为空时，即游戏刚开始，从未搜索过，将初始局面拷贝，由接入 Agent 的 singleDFSPlayer 类调用 DFS 方法一次性搜出通关路径；当 action 列表非空时，按顺序每步取出一个 action，返回单步 action。

```
Types.ACTIONS action = null;

//DfsPlayer searches all actions at the first time.
if (searchedActionList == null) {
    StateObservation stCopy = stateObs.copy();
    searchedActionList = dfsPlayer.DFS(stCopy);
}

//Choose action in turn from searched actions list.
if (searchedActionList.size()>0) {
    action=searchedActionList.remove( index: 0);
}
return action;
```

Fig.1 The core of class depthfist.Agent
图 1 depthfist.Agent 类的代码核心部分

2.2 singleDFSPlayer类的代码实现

singleDFSPlayer 类中主要是 DFS 方法，为深度优先搜索的主体，基本对照课上 DFS 的伪代码实现。另把 Expand 节点展开、isGoal 目标节点判断、Inclosed 判断重复和 getPath 向上回溯路径三个功能作为 Node 类的方法，放在 Node 类中实现。

DFS 方法如图 2 所示，构造栈结构作为 Fringe 搜索空间，LinkedList 结构作为 closed 重复空间（方便插入），创建根节点放入栈（push）Fringe。Fringe 非空时，做循环：取出（pop）栈中节点，将其加入 closed；

经 Node 类内的判定函数 isGoal 判断，若为目标节点，调用 Node 类的 getPath 功能向上回溯路径，得到路径的 action 列表，作为搜索结果直接返回;若非目标节点，调用 Node 类内的 Expand 功能将节点展开，得到所有子节点，for 循环直接遍历其返回的子节点 List，再调用 Node 类内的 in 功能判断每个子节点是否在 closed 的节点列表中，若不在则放入 Fringe 空间。若循环结束未找到目标，返回空的 action 列表。

```
public List<Types.ACTIONS> DFS(StateObservation a_gameState) {
    Stack<Node> fringe = new Stack<>();
    LinkedList<Node> closed = new LinkedList<>();

    //Create root of the tree.
    Node root = new Node(a_gameState,  parent: null,  action: null);
    fringe.push(root);

    while (!fringe.empty()) {
        Node cur = fringe.pop();
        closed.add(cur);

        if (isGoal(cur.state)) {
            System.out.println("FOUND");
            return cur.getPath();
        }

        for (Node child : cur.expand())
        {
            if(!child.in(closed))
                fringe.push(child);
        }
    }
    return new ArrayList<>();
}
```

Fig.2　The core DFS of class singleDFSPlayer
图 2　singleDFSPlayer 类的核心部分 DFS

2.3　Node类的代码实现

Node 类在 depthfirst 包中构造完后，后续任务全部直接使用，因此此处的 Node 功能实现比较多，包含一些属性和方法用于后两个任务，如属性 depth（如图 3），任务 1 中无需计算代价和深度，只需保存父节点 parent 和游戏状态 state，为了方便回溯路径中的动作，也保存该节点的动作 action。

```
public Node(StateObservation so,Node parent,Typ
    this.state = so;
    this.parent = parent;
    this.action = action;
    if(parent != null)
        this.depth = parent.depth+1;
    else
        this.depth = 0;
}
```

Fig.3　Construction method of class Node
图 3　Node 类的构造函数

任务 1 中使用到 expand 节点展开、isGoal 目标节点判断、In 判断是否在节点列表中和 getPath 回溯节点路径四个方法（如图 4），参照了老师上课演示的代码。为了避免每次搜索树一模一样，搜索结果唯一，不容易发现问题，在 expand 方法中调用 Collections.shuffle 方法将读取的可能 Action 列表顺序打乱，从而使放入 fringe 的 action 顺序每次游戏搜索时有所不同。isGoal 方法调用节点所含状态的胜者，若玩家胜则为目标节点，若玩家输或无胜者则非目标。getPath 方法中用 LinkedList 结构装获取的路径，从下到上回溯 action 时每次从 List 开头(addFirst)插入，保证顺序可从头依次执行。in 方法中遍历任意 List<Node>，调用 GVG-AI 框架自带的 StateObservation.equalPosition 方法，比较列表中每个节点的状态是否与该节点相同，以此判断是否在其中。

【问题】原框架自带的 equalPosition 方法存在问题，对状态相同的判断有不正确，对重复判断有影响。但重写的代价太大，不想重写，只能勉强使用。

```
    public ArrayList<Node> expand() {
        ArrayList<Node> successors = new ArrayList<>( initialCapacity: 4);
        ArrayList<Types.ACTIONS> actions = this.state.getAvailableActions();

        Collections.shuffle(actions);//随机打乱action,避免每次游戏的行动一样

        for (Types.ACTIONS step : actions) {
            StateObservation nextState = this.state.copy();
            nextState.advance(step);
            successors.add(new Node(nextState,  parent: this, step));
        }
        return successors;
    }

    /**
     * Get action path to reach current state observation.
     * @return ordered actions to execute.
     */
    public LinkedList<Types.ACTIONS> getPath() {
        Node n = this;
        LinkedList<Types.ACTIONS> path= new LinkedList<>();
        while(n.parent != null)
        {
            path.addFirst(n.action);
            n = n.parent;
        }
        return path;
    }
//Indicates if a node is in the given node list.
public boolean in(List<Node> list){
    for (Node i : list) {
        if (this.state.equalPosition(i.state))
            return true;
    }
    return false;
}

//Indicates if node is the goal node.
public boolean isGoal() {
    return this.state.getGameWinner() == Types.WINNER.PLAYER_WINS;
}
```

Fig.4　Four main method of class Node
图 4　Node 类四个主要方法

3　任务 2：深度受限的深度优先搜索

3.1　Agent类的代码实现

LimitedDepthFirst.Agent 每一步进行一次迭代加深搜索，如图 5。参考 sampleRandom 和 sampleMCTS 的代码的循环条件，限制循环在剩余时间足够时，逐次加深搜索深度，以此深度进行深度受限的深度优先搜索。并根据实际运行情况，将最小剩余时间从 5 调大到 10。每次由接入 Agent 的 singleLDFPlayer 类调用 DLS 方法，搜索深度等于循环次数，得到一个当前深度下的最优路径对应的一步行

```
int remainingLimit = 10;
while(remaining > 2*avgTimeTaken && remaining > remainingLimit)
{
    ElapsedCpuTimer elapsedTimerIteration = new ElapsedCpuTimer();

    DLSResult searchResult=ldfPlayer.DLS(stCopy,numIters);
    action = searchResult.action;
    if (!searchResult.cutoff)
        return action;

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

Fig.5　IDS of Agent
图 5 Agent 中的迭代加深搜索

动。若本次循环已经搜索至通关，而非被截断，则不必继续迭代加深，直接返回决策。

DLS 的搜索结果需同时返回是否截断、最优步骤两个信息，故创建一个 Result 类，如图 6，包含这两个属性，以此来返回多个信息。

```
public static class DLSResult {
    boolean cutoff;
    Types.ACTIONS action;

    public DLSResult(boolean cutoff, Types.ACTIONS action){
        this.cutoff=cutoff;
        this.action=action;
    }
}
```

Fig.6 Class DLSResult to return two messages
图 6 用于传递两个结果的 Result 类

3.2 singleLDFPlayer类的代码实现

singleLDFPlayer 类包括 DLS 方法（深度受限的深度优先搜索的主体）、启发式函数（评判局面）和曼哈顿距离计算函数。

3.2.1 DLS 方法的实现

DLS 方法（如图 7）接受当前局面和该次搜索最大深度两个参数，由于无论是否搜索到目标，就算由于时间有限 cutoff 也需返回一步决策，所以 DLS 方法返回有限搜索深度下的最佳一步，和是否 cutoff 两个信息，通过构造的 DLSResult 类同时返回。此时最佳的一步要么是目标节点对应的一步 action，要么应是被截断时，最大深度节点中最佳局面对应的一步 action，局面好坏由启发式函数判定，因为深度小于最大深度的节点要么还可以往下走，要么已经游戏失败。

```
public Agent.DLSResult DLS(StateObservation a_gameState, int limit) {
    Stack<Node> fringe = new Stack<>();
    LinkedList<Node> closed = new LinkedList<>();
    Node optNode= null;

    //Create root of the tree.
    Node root = new Node(a_gameState,  parent: null,  action: null);
    fringe.push(root);

    while (!fringe.empty()) {
        Node cur = fringe.pop();
        closed.add(cur);

        if (cur.isGoal()) {
            return new Agent.DLSResult( cutoff: false, cur.getNext());
        }
        else if(cur.depth==limit){
              System.out.println("cutoff");
            if(optNode==null|| heuristic(optNode.state)> heuristic(cur.state))
                optNode=cur;
        }
        else {
            for (Node child : cur.expand()) {
                if (!child.in(closed))
                    fringe.push(child);
            }
        }
    }

    assert optNode != null;
    return new Agent.DLSResult( cutoff: true, optNode.getNext());
}
```

Fig.7 The core DLS of class singleLDFPlayer
图 7 singleLDFPlayer 类的核心部分 DLS

【问题】启发式函数对局面的判断不一定准确，且未搜索到终点，暂时的最佳步骤不一定是最终的最佳步骤，但对于第一关足够。

DLS 方法与 DFS 方法类似，只做了如下修改：经 Node 类内的判定函数 isGoal 判断，若为目标节点，不调用 Node 类的 getPath 功能回溯全部路径，而是调用 Node 类的 getNext 功能（如图 8），回溯到对应的第一层节点，只获得对应的第一步 action，返回包含一步 action 和未被截断两个属性的 Result 类;若非目标节点，再判断是否到达最大深度，若到深度，通过 optNode 变量记录下最佳节点（若变量为空，则第一次遇到截断，记录下当下节点；若非空，通过启发式函数比较两节点的局面，保存代价更小、局面更好的节点）；若既不是目标节点，也未到深度，即可继续向下搜索，调用 Node 类内的 Expand 功能将节点展开，将不在 closed 的子节点放入 Fringe 空间。循环结束未返回，则搜索完有限深度下的全部节点未找到目标，optNode 变量应记录下最大深度节点中的相对最佳节点（即使由于前面的错误“最佳”判断已无法胜利，仍会义无反顾地走向死亡），返回包含 optNode 节点对应的一步 action 和截断两个属性的 Result 类。

```
public Types.ACTIONS getNext() {
    Node n = this;
    Node next=n;
    while(n.parent != null)
    {
        next=n;
        n = n.parent;
    }
    return next.action;
}
```

Fig.8　getNext method of class Node
图 8　Node 类的 getNex 方法 t

3.2.2　启发式函数的实现

启发式函数（如图 9）以曼哈顿距离为指标，以 Avatar 距离通关的可能步数作为局面的行动代价。由于游戏坐标以 50 为单位，曼哈顿距离的计算中，横纵坐标距离差的绝对值在求和后再除以单位长度 50，以保证代价的数值不会很大。此时的代价基于步数，为整数，当 Avatar 到达目标时，代价为 0。

```
public int heuristic(StateObservation stateObs){
    if(stateObs.getGameWinner()== Types.WINNER.PLAYER_LOSES) //Avatar消失,无法获得位置
        return Integer.MAX_VALUE;

    int cost;

    ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
    Vector2d goalpos = fixedPositions[1].get(0).position; //目标的坐标（第一关）
    Vector2d Avatarpos =stateObs.getAvatarPosition();//Avatar的坐标

    int Avataritype=stateObs.getAvatarType();//Avatar的状态 without key-1 with key-4
    if (Avataritype==1){  //not get key
        Vector2d keypos = movingPositions[0].get(0).position; //钥匙的坐标（第一关）
        cost=ManhattanDis(goalpos,keypos)+ManhattanDis(keypos,Avatarpos);//Avatar到钥匙+钥匙到目标

        //箱子挡住钥匙
        for(Observation box :movingPositions[1]){        //箱子的坐标（若存在）（第一关）
            if(box.position==keypos){
                cost+=1;
                break;
            }
        }
    }
    else //get key
        cost=ManhattanDis(goalpos,Avatarpos);//Avatar到目标

    return cost;
}
```

Fig.9　Heuristic function in Task 2
图 9　任务二中的启发式函数

若 Avatar 遇到洞死亡，则 Avatar 对象消失，无法获得 Avatar 的位置，此时游戏失败，将代价视为无限大，直接返回整型最大值。非死亡正常状态下，针对第一关的地图，找到目标对象的存储位置，获取目标

坐标。需要根据 Avatar 的状态，确定是否拿到了钥匙。因为若 Avatar 已获得钥匙，则钥匙对象消失，无法获得钥匙位置，以原位置试图获得钥匙的位置会出现数组下标越界的报错，此时只需直接以 Avatar 到目标的曼哈顿距离为局面的代价。若 Avatar 未获取钥匙（Avatar 的 itype=1），则 Avatar 需先拿到钥匙，再到目标，局面代价为 Avatar 到钥匙的曼哈顿距离与钥匙到目标的曼哈顿距离之和，此外又遍历了所有箱子的位置，若箱子的位置与钥匙位置相同，则箱子覆盖了钥匙，需先将箱子推开再获取钥匙，即代价+1。

【问题】代价预估比实际偏小了很多。但经过尝试，若以其他更接近的方式计算，需读取墙、洞、蘑菇等物体的坐标后，另外比较很多处的位置，要遍历很多位置列表，时间代价很大，容易时间超限。因此这里干脆就简单处理，用最简单的方法评估。

4 任务 3：A*算法

4.1 A*算法的代码实现

Agent 类的代码很简单，让 Agent 直接把计时器传给 singleAstarPlayer 类，在 singleAstarPlayer 类内设计了一个计时器接口，由 singleAstarPlayer 类在有限时间里使用 A*算法搜索节点，返回当前搜索到的最佳局面所对应的一步行为。主要工作由 singleAstarPlayer 类的 aStar 方法完成。

```
public Types.ACTIONS aStar(StateObservation a_gameState) {
    ArrayList<Node> fringe = new ArrayList<>();
    LinkedList<Node> closed = new LinkedList<>();

    //Create root of the tree.
    Node root = new Node(a_gameState,  parent: null,  action: null);
    fringe.add(root);
    pastList.add(root);         //加入到已经经过的状态列表

    int remainingLimit = 8;
    while (!fringe.isEmpty()) {

        //pick a node by A*
        fringe.sort(Comparator.comparingInt(this::f));
        Node cur = fringe.remove( index: 0);
        closed.add(cur);

        if (cur.isGoal()||searchTimer.remainingTimeMillis()<=remainingLimit)
            return cur.getNext();
        else {
            for (Node child : cur.expand())
                if (!child.in(pastList)&&!repeated(closed,child))//既未前次走过，也未本次搜过
                    fringe.add(child);
        }
    }
    //System.out.println("NO FOUND");
    return null;
```

Fig.10 A*search
图 10 A*搜索

A*算法对一般树搜索方法做了如下修改（如图 10）：由于后一次搜索时，若不知道前面已经走过的局面，就会出现左右反复横跳的情况，因此要彻底不走回头路，将已经经过的状态记录下来（在 singleAstarPlayer 类中创建 LinkedList 结构的 pastList 保存每次调用搜索时树的根节点，即拷贝下的当前状态）；循环取节点时，先将节点列表根据对应的代价评估函数 f（到达该节点的实际代价与该节点距离重点的代价估值之和，即节点深度与启发式函数对节点局面的估值之和）排序，取出第一个即代价估值最小的节点；若该节点为目标节点或剩余时间小于设定的最小剩余时间（根据实际测试结果，设置为 8）则返回该节点对应的下一步 action，时间截止时所拿出的节点为当下 fringe 中代价最小的节点，就是当前搜索的最佳节点；否则继续搜索，将节点展开，此时尽管一个子节点保存的局面已经在 closed 中出现，但可能是沿着代价更小的路径到达该状态，故以新的标准对比子节点和 closed 中的节点（repeated 函数，如图 11），

```java
/**
 * Indicates if node is repeating one. If state same but depth(cost) less,
 * it isn't repeated and shouldn't be excluded.
 * @param list closed list
 * @param node node for evaluation .
 */
public boolean repeated(List<Node> list, Node node){
    for (Node i : list) {
        if (node.state.equalPosition(i.state) && node.depth>=i.depth)
            return true;
    }
    return false;
}
```

Fig.11 Method to judge if a node in the closed
图 11 判断是否在 closed 中的函数

将局面相同，但代价更小（深度更小）的节点，应算作新状态被加入 fringe，此外还需不在 pastList 中，排除过去几步已经走过的状态（此时因为已经走过故即使代价更小也没法重做行动），且为了减少 fringe 的排序负担，在此处再排除已经失败的子节点（调用 Node 类的失败判定方法 isLost）。

【问题】当前的最优解不一定为最终的最优路径，由于不走回头路，一旦一步走错就可能无法纠错，而一错到底。每次取节点时排序使时间复杂度随空间复杂度飙升。没有想到如何选取节点代价更小。

4.2 启发式函数的实现

基于任务 2 中的启发式函数设计，为了适应多关卡，没有针对地图设定读取固定位置的目标位置，而是遍历固定物体列表，根据读取物体的类型，找到目标 goal 类型（itype=7）对应的物体。再干脆就简单处理，同样用分别用 Avatar 到钥匙的曼哈顿距离与钥匙到目标的曼哈顿距离之和，以及 Avatar 到目标的曼哈顿距离表示有无状态下的代价。

```java
Vector2d goalpos =null;

for (ArrayList<Observation> itemlist:fixedPositions) {
    if (!itemlist.isEmpty() && itemlist.get(0).itype == 7)
        goalpos = itemlist.get(0).position; //目标的坐标
}

Vector2d Avatarpos =stateObs.getAvatarPosition();//Avatar的坐标

int Avataritype=stateObs.getAvatarType();//Avatar的状态
if (Avataritype==1) //not get key
{
    Vector2d keypos = movingPositions[0].get(0).position; //钥匙的坐标
    return ManhattanDis(goalpos,keypos)+ManhattanDis(keypos,Avatarpos);
}
else //get key
    return ManhattanDis(goalpos,Avatarpos);
```

Fig.12 Heuristic function in A*
图 12 A*中的启发式函数

【问题】启发式函数设计得很简单，估计的 h(x)比实际值 h(x)*小很多，箱子、洞、蘑菇的影响根本没有考虑在内，使 A*算法实际除了第一关根本跑不过后面几关，难得情况下根本不关注箱子的盲走可以刚好碰到箱子，成功通关。优化方案是采取更好、更准确的启发式函数和提高搜索效率，争取在搜索得更快更多，但我推箱子技术有限，我想不出如何在不大量读取坐标、比较洞与钥匙、目标相对位置的情况下更好地拍脑袋。

5 任务 4：MCTS 算法介绍

5.1 算法介绍

controllers.sampleMCTS.Agent.java 使用的算法为 MCTS 蒙特卡罗树搜索。MCTS 仍依靠启发式函数，但不是简单的直接搜索，而是基于随机模拟的搜索，用随机得到的概率，评判局面好坏。MCTS 对子节点的选择基于 UCB（Upper Confidence Bounds）算法，配置探索和利用不同的权重：在选择子节点时会优先考虑没有探索过的节点，如果都探索过则根据得分，得分与这个子节点的胜率和探索次数都有关，与平均得分高正相关（认为它比其他节点更值得利用）与选中次数负相关（因为其他选择次数少的节点更值得探索）。

结合代码和网络资料，可见 MCTS 有四个主要阶段：（1）选择 Select：从根节点开始，通过计算每个节点的子节点的 UCB 值，递归选择 UCB 值最优的子节点，直到到达一个没有被完全展开的节点；（2）

扩展 Expand：如果前面选中的子节点不是一个终止节点（游戏终止或到达设置的深度）且那么就随机选择一步可能的行动，创建一个新的子节点；模拟试运行 Rollout：从新 Expend 出来的节点开始随机向后模拟游戏，（即一路都随机选择行动），直到游戏结束或到达深度限制；反向传播 Backup：用模拟的结果得分，更新经过的所有节点的价值和访问次数。

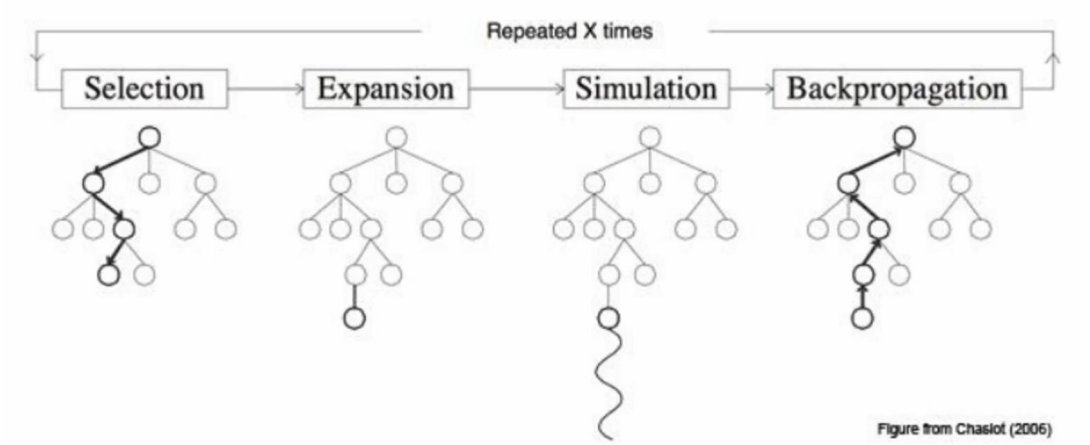

Fig.12　four phases of MCTS
图 12　MCTS 的 4 个阶段

搜索结束之后，从根节点的子节点（即下一步行动）中选择被访问次数最高的节点，若访问次数相同，就选择平均价值最高的。

5.2　代码的实现

Agent 类的构造函数首先将所有可能的 Action 装入静态列表，方便获取；再创建一个接入新的随机生成器的 SingleMCTSPlayer 类，名为 mctsPlayer，用来作为 MCTS 算法的行使对象。Act 方法中，先利用当前局面的拷贝将接入的 SingleMCTSPlayer 类初始化，即用当前局面创建 mctsPlayer 中树的根节点，；再让 mctsPlayer 调用 run 方法，发挥它运行 MCTS、得到最佳行动的功能，先调用根节点的方法进行 MCTS，再从搭建好的搜索树中选择最优子节点的对应行动，返回一个对应的行动对应下标，从而通过列表下标得到相应的一步行动。

MCTS 的具体四个阶段通过节点的功能完成：代码中 tree policy 连接了 Select 和 Expend 两个阶段；uct 完成了 select 对节点的选择，UCT 公式中的常数被设置为 K=sqrt(2)，并将子节点价值归一化后再带入公式；子节点选择中（搜索中和搜索后）对 UCT 值、访问次数和平均得分的计算都再通过引入一个极小量 epsilon，产生随机噪音的方法打破节点间数值的平局；value 函数通过游戏的得分和胜负评价局面得分，rollout 方法中并会用 rollout 的得分更新归一化的数值边界。

References:

[1]　AI 如何下棋？直观了解蒙特卡洛树搜索 MCTS！！！_哔哩哔哩_bilibili