# Bait 游戏实验报告

周韧哲（181220076、zhourz@nju.edu.cn）

(南京大学 人工智能学院，南京 210093)

**摘 要:** 完成了 Bait 游戏的四个任务

**关键词:** 深度优先搜索，受限深度优先搜索，A* 算法，蒙特卡洛树搜索

## 1 引言

在 Bait 游戏时间限制内,通过深度优先搜索与受限深度优先搜索通过 level0,完成了 A*算法,理解了蒙特卡洛树搜索在 Bait 游戏中的应用。

## 2 四个任务的具体实现

### 2.1 任务1：深度优先搜索

在 bait 游戏中实现深度优先搜索，我的主要思想是建立一个栈以存储状态，每次从栈中取出一个状态来，执行可执行的动作后判断执行后的状态是否符合要求，若符合则进栈，否则将原状态出栈；执行直到找到成功的一个状态，然后从该状态开始进行回溯，找到第一个动作并返回。

```
private Stack<StateObservation> state_stack = new Stack<~>();
private ArrayList<Types.ACTIONS> action_list = new ArrayList<~>();
private Stack<Types.ACTIONS> success_action = new Stack<~>();
private ArrayList<StateObservation> acted_state = new ArrayList<~>();
private ArrayList<Types.ACTIONS> executable_action = new ArrayList<~>();
```

State_stack 就是上面说的状态栈，action_list 对应的是 state_stack 中进入每一个状态而进行的动作，以便在回溯时找到第一个动作，success_action 存储能够使游戏成功的动作集合，acted_state 存储已经经历过的状态，executable_action 存储现状态下可执行的动作。

```
public void clear(){
    success_action.clear();
    acted_state.clear();
    action_list.clear();
    executable_action.clear();
    state_stack.clear();
}
```

```
public void init(StateObservation stateObs){
    state_stack.push(stateObs);
    action_list.add(null);
    executable_action=stateObs.getAvailableActions();
}
```

Clear()与 init()分别执行清空数据与初始化的工作。

```
private boolean isInActedState(StateObservation stateObs){
    for(StateObservation so:acted_state){
        if(so.equalPosition(stateObs)){
            return true;
        }
    }
    return false;
}
```

IsInActedState()用来判断状态 so 是否在已经经历过的状态中。

```
public Types.ACTIONS DFS(){
    while(true){
        StateObservation state=state_stack.peek();

        if(isInActedState(state)){
            state_stack.pop();
            action_list.remove( index: action_list.size()-1);
            continue;
        }
        acted_state.add(state);

        if (state.isGameOver()){
            if (state.getGameWinner()==Types.WINNER.PLAYER_WINS){
                //回溯
                StateObservation st = state_stack.pop();
                while(!state_stack.isEmpty()){
                    StateObservation state_before = state_stack.pop();
                    Types.ACTIONS act=action_list.get(action_list.size()-1);
                    StateObservation stcopy = state_before.copy();
                    stcopy.advance(act);
                    if (st.equalPosition(stcopy)){
                        success_action.add(act);
                        st=state_before.copy();
                        action_list.remove( index: action_list.size()-1);
                    }
                    else{
                        action_list.remove((action_list.size()-2));
                    }
                }
                return success_action.peek();
            }
            else {
                state_stack.pop();
                action_list.remove( index: action_list.size()-1);
                continue;
            }
        }

        for(Types.ACTIONS action:executable_action){
            StateObservation stCopy = state.copy();
            stCopy.advance(action);
            if (!isInActedState(stCopy) ){
                state_stack.push(stCopy);
                action_list.add(action);
            }
        }
    }
}
```

首先从 state_stack 中取出一个状态作为 state,若该状态已经经历过，则 pop 掉，并把 action_list 中对应的动作删去，继续循环，否则，先将该状态加入 acted_state 中。如果该动作使游戏玩家赢，则可以进行回溯并返回最初执行的动作:此时 state_stack 栈顶的 state 为 PLAYER_WINS 的状态，所以就从该状态开始回溯，若在 state_stack 中它的前一个状态执行 action_list 中 state 对应的动作后结果为 state，则说明它为 state 的前一个状态，更新 state,将该动作加入到 success_action 中；否则，说明它不是 state 的前一个状态，接着 pop 出 state_stack 中的下一个状态进行比较,所以到最后 success_action 中的最后一个元素就是最初 state 为了到达成功 state 而应该执行的第一个动作，在 act()中返回该动作便可。若该状态不使玩家赢,则 pop 掉该状态与对应动作，继续循环。否则，在 for 循环中 copy 该 state 并执行动作，若它不在已经历的状态中，则将这个状态进栈，并将进入这个状态所执行的动作加入到 action_list 中,从而在下一次循环中对这个状态进一步搜索。

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
    Types.ACTIONS action = null;
    init(stateObs);
    action=DFS();
    clear();
    return action;
}
```

Act()返回搜索后找到的动作。如此便完成了深度优先搜索算法。

## 2.2 深度受限的深度优先搜索

在 bait 游戏中实现深度受限搜索，我的主要思想是创建一个可比较的 Node 类,在类里实现了启发式函数,建立一个栈以存储节点，每次从栈中取出一个节点来，判断节点的状态与深度并进行评估,若符合要求则加入当前可能成功的一个优先队列节点中;若其状态执行可执行的动作后符合要求，则将此状态作为当前节点 e 的孩子并进栈，否则将原状态出栈；执行直到找到成功的一个状态，然后取得该状态节点的根节点,找到第一个动作并返回。

```
private final int max_depth=4;  //限定的深度
private Stack<Node> node_stack = new Stack<~>();
private ArrayList<StateObservation> acted_state = new ArrayList<~>();
private PriorityQueue<Node> node_queue = new PriorityQueue<~>();
private ArrayList<Types.ACTIONS> executable_action = new ArrayList<~>();
```

Max_depth 为最大搜索深度,node_stack 为上面所说的栈,acted_state 存储已经历过的状态,node_queue 存储有可能成功的节点,executable_action 存储可执行的动作。

```
private void clear(){
    node_stack.clear();
    node_queue.clear();
    acted_state.clear();
    executable_action.clear();
}
```

```
private void init(StateObservation stateObs){
    Node node = new Node(stateObs,  depth: 0);
    node.parent = null;
    node.root_node=null;
    node_stack.push(node);
    executable_action=stateObs.getAvailableActions();
}
```

```
private boolean isInActedState(StateObservation stateObs){
    for(StateObservation so:acted_state){
        if(so.equalPosition(stateObs)){
            return true;
        }
    }
    return false;
}
```

Clear()与 init()分别执行清空数据与初始化的工作 ,isInActedState()用来判断状态 so 是否在已经经历过的状态中。

```java
public class Node implements Comparable<Node>{
    public Node parent;
    public StateObservation state;
    public int depth;
    public double H;
    public Node root_node;
    public Node(StateObservation state, int depth){
        this.state = state.copy();
        this.depth = depth;
        heuristic();
    }
    public void heuristic() {
        if (this.state.getGameWinner()== Types.WINNER.PLAYER_WINS){
            this.H= Double.NEGATIVE_INFINITY;
        }
        else {
            ArrayList[] fixedPositions = state.getImmovablePositions();
            ArrayList[] movingPositions = state.getMovablePositions();
            Vector2d goal_pos = ((Observation)(fixedPositions[1]).get(0)).position;
            Vector2d key_pos = ((Observation)(movingPositions[0].get(0))).position;
            Vector2d avatar_pos = state.getAvatarPosition();
            if (state.getAvatarType()==1){ //没得到钥匙
                this.H=avatar_pos.dist(key_pos)+key_pos.dist(goal_pos);
            }
            else{
                this.H=avatar_pos.dist(goal_pos);
            }
        }
    }
    @Override
    public int compareTo(Node n) { return (int) (this.H-n.H); }
}
```

在类 Node 中,定义了它的 parent,state,depth,H,root_node。Parent 为节点的父节点，state 为节点状态，depth 为节点向下搜索的深度，H 为启发式函数给出的分数评估，root_node 为节点的根节点，用来找出到达此节点状态对应所应该做的第一个动作。启发式函数 heuristic()中,若该 state 下玩家赢了,则 H 为负无穷大;否则,若 Avatar 没得到钥匙,则用它到钥匙的距离与钥匙到目标的距离作为 H,否则用它到目标的距离作为 H，Node 的大小关系定义为 H 的大小关系。

```java
    public Types.ACTIONS LDFS() {
        while (!node_stack.isEmpty()) {
            Node curr_node = node_stack.pop();
            StateObservation curr_state = curr_node.state.copy();
            int curr_depth = curr_node.depth;

            if (curr_state.isGameOver()) {
                if (curr_state.getGameWinner() == Types.WINNER.PLAYER_WINS) {
                    node_queue.add(curr_node);
                    break;
                }
                else{
                    acted_state.add(curr_state);
                    continue;
                }
            }

            acted_state.add(curr_state);
            if (curr_depth < max_depth) {
                for (Types.ACTIONS action : executable_action) {
                    StateObservation stCopy = curr_state.copy();
                    stCopy.advance(action);
                    if (!isInActedState(stCopy) ){
                        Node node = new Node(stCopy,  depth: curr_depth + 1);
                        node.parent = curr_node;
                        if (curr_node.root_node==null) {
                            node.root_node =node;
                        }
                        else{
                            node.root_node=curr_node.root_node;
                        }
                        node_stack.push(node);
```

```
                    }
                }
            }
        else {
            node_queue.add(curr_node);
        }
    }

    Node node=node_queue.peek();
    Types.ACTIONS action=node.root_node.state.getAvatarLastAction();
    return action;
}
```

首先从 node_stack 中取出一个 node,如果该状态下玩家赢了,则将该 node 加入 node_queue 并跳出循环,此时该 node 的 H 为负无穷大,所以 peek 一下就能从 node_queue 中得到该 node,取得它的根节点,返回第一个动作。如果该状态下游戏结束了但玩家未赢,则加入状态至 acted_state。若游戏未结束,且 node 的深度小于最大深度,则从该 node 向下搜索,执行动作后将下一个节点作为该节点的孩子,同时若当前节点的 root_node 为 null,说明它是初始状态的节点,则其孩子的根节点为孩子本身;否则孩子节点的根节点继承当前节点的根节点,将孩子入栈进入下一步循环继续搜索。若 node 的深度达到了最大深度,则将其加入 node_queue 中,待 node_stack 中所有节点搜索完后取出 H 最小的一个节点,取其根节点返回第一个动作。

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
    Types.ACTIONS action = null;
    init(stateObs);
    action=LDFS();
    clear();
    return action;
}
```

Act()返回搜索后找到的最优动作。如此便完成了受限深度优先搜索算法。

### 2.3 A* 算法

```
public class Node implements Comparable<Node>{
    public double H;
    public double G;
    public double F;
    public Node parent;
    public StateObservation state;

    public Node(StateObservation stateObs){
        this.state = stateObs.copy();
    }

    public void upDate(){
        this.G=this.parent.G+1;
        heuristic();
        this.F=this.G+this.H;
    }

    public int compareTo(Node node){
        return (int)(this.F-node.F);
    }
```

类 Node 定义了 Astar 算法中重要的数据结构，H 为启发式函数给出的预估值，G 为当前的移动耗费值，F=H+G，定义了 Node 的父节点 parent 和其对应的游戏状态 state。初始化 Node 时仅需初始化 state 便可，函数 upDate()用来计算 G,H,F，重写了函数 compareTo()，使 Node 的大小关系定义为 F 的大小关系。

```
public void heuristic() {
    if (this.state.getGameWinner()== Types.WINNER.PLAYER_WINS){
        this.H=Double.NEGATIVE_INFINITY;
    }
    else if (this.state.getGameWinner()== Types.WINNER.PLAYER_WINS){
        this.H=Double.POSITIVE_INFINITY;
    }
    else {
        double K = -10,k=100;
        double box_bias=0,hole_bias=0;
        ArrayList[] fixedPositions = this.state.getImmovablePositions();
        ArrayList[] movingPositions = this.state.getMovablePositions();
        if(fixedPositions.length == 4){
            ArrayList<Observation> hole = fixedPositions[1];
            ArrayList<Observation> box = movingPositions[1];
            for(int i=0;i<box.size();i++){
                box_bias+=1;
            }
            for(int i=0;i<hole.size();i++){
                hole_bias+=1;
            }
        }
        Vector2d goal_pos = ((Observation)(fixedPositions[0]).get(0)).position;
        Vector2d avatar_pos = state.getAvatarPosition();
        if (this.state.getAvatarType() == 1) {
            Vector2d key_pos = ((Observation)(movingPositions[0].get(0))).position;
            this.H=K*this.state.getGameScore()+ key_pos.dist(avatar_pos) + goal_pos.dist(key_pos)+k*hole_bias+k*box_bias;
        }
        else {
            this.H=K*this.state.getGameScore() + goal_pos.dist(avatar_pos)+k*hole_bias+k*box_bias;
        }
    }
```

启发式函数 heuristic()的主要思想是：如果当前状态下玩家赢了，则预估耗费最小，为负无穷大；否则若玩家输了，则 H 为正无穷大。否则，在游戏进行过程中，我使用当前游戏分数、玩家位置到钥匙与目标的距离、洞和箱子的个数的加权和作为 H，具体为：K 为当前游戏分数的权重，K 取负值，表示游戏分数越高 H 越小则该节点越好；k 为 box_bias 与 hole_bias 的权重，k 为正值，表示 box 和 hole 的数量越少越好，鼓励填洞；若玩家还未拿到钥匙，则加上玩家到钥匙的距离与钥匙到目标的距离；若玩家已拿到钥匙，则加上玩家到目标的距离。

```
private ArrayList<StateObservation> closedList = new ArrayList<>();
private PriorityQueue<Node> openList = new PriorityQueue<>();
private ArrayList<Types.ACTIONS> executable_action = new ArrayList<>();
private Node bestNode;
```

在 Agent.java 中定义 closedList 为关闭列表，存储关闭的状态，定义 openList 为开启列表，存储待探索的节点，executable_action 为某状态下可执行的动作，bestNode 为探索到的最好的节点。

```
public void init() {
    bestNode=null;
    openList.clear();
    closedList.clear();
    executable_action.clear();
}
```

```
public boolean isInClosedList(StateObservation obs) {
    for (StateObservation so : closedList) {
        if (so.equalPosition(obs)) {
            return true;
        }
    }
    return false;
}
```

```
public boolean isInOpenList(StateObservation obs) {
    for (Node node : openList) {
        if (node.state.equalPosition(obs)) {
            return true;
        }
    }
    return false;
}
```

```
public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer) {
    init();
    astarSearch(so);
}
```

Init()初始化各数据结构，isInClosedList()用来判断某状态是否在 closedList 中，isInOpenList()用来判断某状态是否在 openList 中，在 Agent()中，调用 init()与 astarSearch()来探索 bestNode。

```
public void astarSearch(StateObservation stateObs) {
    Node root_node = new Node(stateObs);
    root_node.parent=null;
    root_node.G=0;
    openList.add(root_node);

    while (!openList.isEmpty()) {
        Node curr_node = openList.poll();
        if(!curr_node.state.isGameOver())
        {bestNode=curr_node;}
        executable_action = curr_node.state.getAvailableActions();
        closedList.add(curr_node.state.copy());

        for (Types.ACTIONS action : executable_action) {
            StateObservation stCopy = curr_node.state.copy();
            stCopy.advance(action);

            if (stCopy.isGameOver()){
                if (stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS) {
                    Node newNode=new Node(stCopy);
                    newNode.parent=curr_node;
                    newNode.upDate();
                    bestNode=newNode;
                    return;
                }
                else{
                    continue;
                }
            }
            else if (isInClosedList(stCopy)) {
                continue;
            }
            else if (isInOpenList(stCopy)) {
                for (Node node : openList) {
                    if (node.state.equalPosition(stCopy)) {
                        if (node.G>curr_node.G+1) {
                            openList.remove(node);
                            Node newNode=new Node(stCopy);
                            newNode.parent=curr_node;
                            newNode.upDate();
                            openList.add(newNode);
                            break;
                        }
                    }
                }
            }
            else {
                Node newNode=new Node(stCopy);
                newNode.parent=curr_node;
                newNode.upDate();
                openList.add(newNode);
            }

        }
    }
}
```

astarSearch()为搜索的核心算法。首先定义初始状态为 root_node,其父节点为 null，加入 openList 中，然后开始 while 循环，一步步从 openList 中探索。首先取得当前最优的一个节点，因为 openList 为优先队列，所以执行 poll()就能取得当前最优的节点，并先将它赋值给 bestNode，将它的状态加入 closedList 中。对 curr_node 的状态执行可执行的动作，若玩家赢了，则将新状态节点加入 openList 中，并将 bestNode 赋值为此节点，停止搜索，直接返回。若玩家输了，则继续探索 openList；若 stCopy 在 closedList 中，则继续探索 openList；若 stCopy 在 openList 中，则先找出和它状态相同的那个节点 node,若 node.G(当前的移动耗费值)>curr_node.G+1(新状态节点的 G 值)，则在 openList 中替换掉 node,换为新状态节点；否则，说明 stCopy 为新的待探索节点，直接将其加入 openList 中。

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
    Types.ACTIONS action = null;
    if(bestNode==null){
        astarSearch(stateObs);
        return null;
    }
    Node p=bestNode;
    Node q=p;
    int i=0;
    while(p!= null && p.parent != null && p.parent.parent != null){
        p = p.parent;
        if (i>0){q=q.parent;}
        i++;
    }
    if(p != null && q!=null)
        action=p.state.getAvatarLastAction();
        q.parent=p.parent;
        return action;
```

在 act()函数中,通过对 bestNode 的回溯，找到第一个动作并返回：其中，p 回溯到最初节点的孩子，返回它对应执行的动作,此时 p 为 q 的父节点，将 p 的父节点替换为最初节点，在控制程序下一次调用 act()时，即可返回对应的下一个动作了。如此便完成了 A*算法。

### 2.4 蒙特卡洛树搜索

Agent.java 中创建一个 SingleMCTSPlayer 并在 act()中返回 SingleMCTSPlayer.java 中搜索得到的最优动作,而搜索算法的主要部分在 SingleTreeNode.java 中。

函数 mctsSearch()为蒙特卡洛树搜索算法的框架函数,在 while 循环时间限制内,通过 treePolicy()选择一个叶子节点 selected,然后对该 selected 节点进行 rollOut, 然后将 rollOut 的结果 delta 沿原路径反向传播,更新反向传播路径上所有节点的 nVisits 与 totValue(backup()函数)。

TreePolicy()选择叶子节点的策略是,在游戏未结束且未达到 rollOut 限定深度下,若节点的孩子没有完全扩展则说明它是叶子节点,扩展该节点并返回孩子节点中执行最优动作后的节点作为选择的叶子节点。否则,通过 uct()函数计算子节点的 childValue 与 uctValue,其计算方法分别为,childValue 为 child.totValue / (child.nVisits + this.epsilon),即总价值与访问次数+epsilon(防止除 0)的商,并通过 delta 的 bounds 将其值规范化。uctValue 则为 childValue+Agent.K * Math.sqrt(Math.log(this.nVisits + 1) / (child.nVisits + this.epsilon)),即参数 K 乘以父节点访问次数除以孩子节点访问次数+epsilon 的根号,然后再加上一个随机噪声,以平衡未扩展节点。访问次数少的，totValue 高的子节点 childValue 更高，childValue 称为 exploitation UCT,而+号右侧称为 exploration UCT, 参数 K 控制蒙特卡洛树搜索中 expolitation 和 exploration 组件之间的权衡,做到贪婪与探索的平衡,最后返回一个 uctValue 最高的节点。

RollOut()函数在未完成 rollOut 情况下随机向下探索,若 rollOut 深度达到限定深度或游戏结束则完成 rollOut,并更新 bounds 值,返回得分 delta。

Act()函数返回的动作选择策略在 mostVisitedAction()与 bestAction()中,动作的优先选择是当前节点的孩子节点中访问次数最多的动作;若所有孩子节点中的所有动作访问次数都相同,则调用 bestAction()返回 childValue 最优的一个动作,即 child.totValue / (child.nVisits + this.epsilon)最大的一个动作。可以知道最终结果

更青睐于 expolitation 的结果。

## 3 结束语

本文详细介绍了 Assignment1 中对代码的实现，完成了深度优先搜索，受限深度优先搜索，A* 算法，执行并理解了蒙特卡洛树搜索算法。