Bait 游戏实验报告

余孟凡（231240002、231240002@smail.nju.edu..cn）
(南京大学 计算机科学与技术系，南京 210093)

摘 要: 读懂一个既没有说明文档，也没有基本介绍的程序

掌握较为复杂的借口调用方法，真实体会面向对象设计思想。

编写一个基本的 DFS 接口,并逐渐优化。

学习并使用 Astar 算法。

学习大项目的规范化注释。

关键词: 并发性;面向对象;继承反常;渐增式继承;范畴论

1 基础 DFS 搜索

1.1 我们可以从模板开始捋清 DFS 的设计思路： DFS 的核心是控制状态队列，其核心代码的过程是实现控制。因此我们从这段核心思路出发：1.终止条件：当一个节点没有邻居的时候，我们终止继续搜索，返回对当前节点的检查结果。2.我们需要一个储存已经访问过的状态的列表，然后打印当前状态（也可以使用一个列表储存下来），对当前节点进行检查，然后继续搜索所有邻居。

1.2 一个标准的图搜索模板：

```
public class Graph {
    private Map<Integer, List<Integer>> adjList;
    public Graph() {
        adjList = new HashMap<>();
    }
    // Perform depth-first search starting from a given node
    public void dfs(int start) {
        Set<Integer> visited = new HashSet<>();
        dfsHelper(start, visited);
    }
    // Helper method for DFS
    private void dfsHelper(int node, Set<Integer> visited) {
        // Mark the current node as visited
        visited.add(node);
        System.out.print(node + " "); // Process the current node (e.g., print)
        // Traverse all neighbors
        for (int neighbor : adjList.getOrDefault(node, new ArrayList<>())) {
            if (!visited.contains(neighbor)) {
                dfsHelper(neighbor, visited);
            }
        }
    }
}
```

1.3 整体方案设计

由于没有说明文档，这不得不迫使我 RTFSC，并进行梳理，梳理内容大致如下：

Observation 类： 实现了 Comparable 接口，允许比较不同的观察对象。

StateObservation 类：

getObservationGrid:功能：该方法用于获取当前游戏状态的观察网格。具体返回的内容通常包括不同位置的观察数据。

1.4 现在考虑以何种方式实现 DFS：最初考虑在搜索树和图时最常见的递归方法，但是为了调试和方便观察，编写了如下函数来实现显式栈的调试：

```java
private void printStack(Stack<Node> stack) {
    System.out.println("Current stack contents:");
    for (Node node : stack) {
        StringBuilder actionsOutput = new StringBuilder();
        for (Types.ACTIONS action : node.actions) {
            switch (action) {...}
        }
        System.out.println("State: " + node.state + ", Actions: [" + actionsOutput.toString().trim()
    }
}
```


```java
private void printStack(Stack<Node> stack) {
    System.out.println("Current stack contents:");
    for (Node node : stack) {
        StringBuilder actionsOutput = new StringBuilder();
        for (Types.ACTIONS action : node.actions) {
            switch (action) {...}
        }
        System.out.println("State: " + node.state + ", Actions: [" + actionsOutput.toString().trim()
    }
}
```


这样就可以随时查看栈的内容，查处可能的搜索问题，并进行改进。

1.5 具体实现方法

1.5.1 新建的四个类变量的简单介绍如下：

1.5.1.1 actionsSequence:类型：List<Types.ACTIONS> 用于存储代理在探索过程中生成的动作序列。代理将按照这个序列依次执行动作，以达到游戏的目标。

1.5.1.2 visitedState:类型：List<StateObservation> 用于存储已经访问过的状态。通过记录这些状态，代理可以避免重复探索，从而提高搜索效率。

1.5.1.3 isWin:类型：boolean 一个布尔值，用于指示代理是否找到了通往胜利的路径。如果找到了路径，该值将被设置为 `true`。

1.5.1.4 currentActionIndex:类型：int 当前动作序列的索引。它用于跟踪代理在动作序列中的位置，以便在每个游戏步骤中执行下一个动作。

1.5.2 核心搜索逻辑

```java
/**
 * Iterative Depth-First Search using an explicit stack
 * @param initialState the initial state observation
 * @param elapsedTimer timer for the search
 * @return true if a winning path is found, false otherwise
 */
1 个用法
boolean getDepthFirstActionsIterative(StateObservation initialState, ElapsedCpuTimer elapsedTimer) {
    long startTime = System.currentTimeMillis(); // 记录开始时间

    // Define a stack to hold pairs of state and the corresponding action sequence
    Stack<Node> stack = new Stack<>();
    stack.push(new Node(initialState, new ArrayList<>()));

    while (!stack.isEmpty()) {
        printStack(stack);

        Node currentNode = stack.pop();
        StateObservation currentState = currentNode.state;
        List<Types.ACTIONS> currentActions = currentNode.actions;

        // Check for win condition
        if (checkWinCondition(currentState)) {...}

        // If state is valid for exploration
        if (processState(currentState)) {
            // Mark the state as visited
            visitedState.add(currentState);

            // Iterate through available actions
            for (Types.ACTIONS action : currentState.getAvailableActions()) {...}
        }
```

```java
/**
 * Iterative Depth-First Search using an explicit stack
 * @param initialState the initial state observation
 * @param elapsedTimer timer for the search
 * @return true if a winning path is found, false otherwise
 */
1 个用法
boolean getDepthFirstActionsIterative(StateObservation initialState, ElapsedCpuTimer elapsedTimer) {
    long startTime = System.currentTimeMillis(); // 记录开始时间

    // Define a stack to hold pairs of state and the corresponding action sequence
    Stack<Node> stack = new Stack<>();
    stack.push(new Node(initialState, new ArrayList<>()));

    while (!stack.isEmpty()) {
        printStack(stack);

        Node currentNode = stack.pop();
        StateObservation currentState = currentNode.state;
        List<Types.ACTIONS> currentActions = currentNode.actions;

        // Check for win condition
        if (checkWinCondition(currentState)) {...}

        // If state is valid for exploration
        if (processState(currentState)) {
            // Mark the state as visited
            visitedState.add(currentState);

            // Iterate through available actions
            for (Types.ACTIONS action : currentState.getAvailableActions()) {...}
        }
```

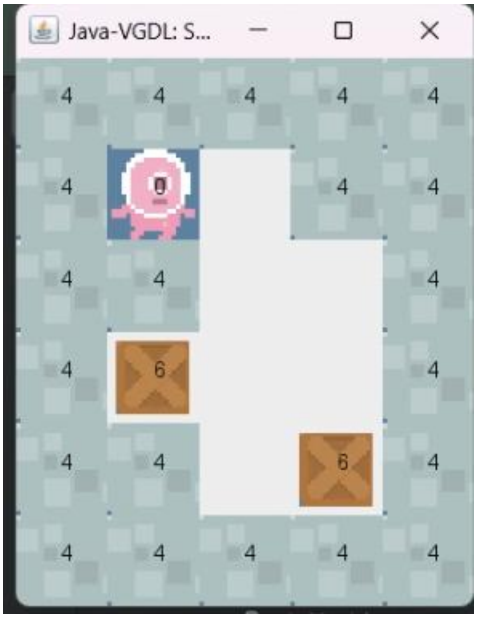
使用一个显式栈来储存状态，并进行搜索，并进行入栈出栈的操作。具体来说，对每一个状态进行结束检查和扩展。

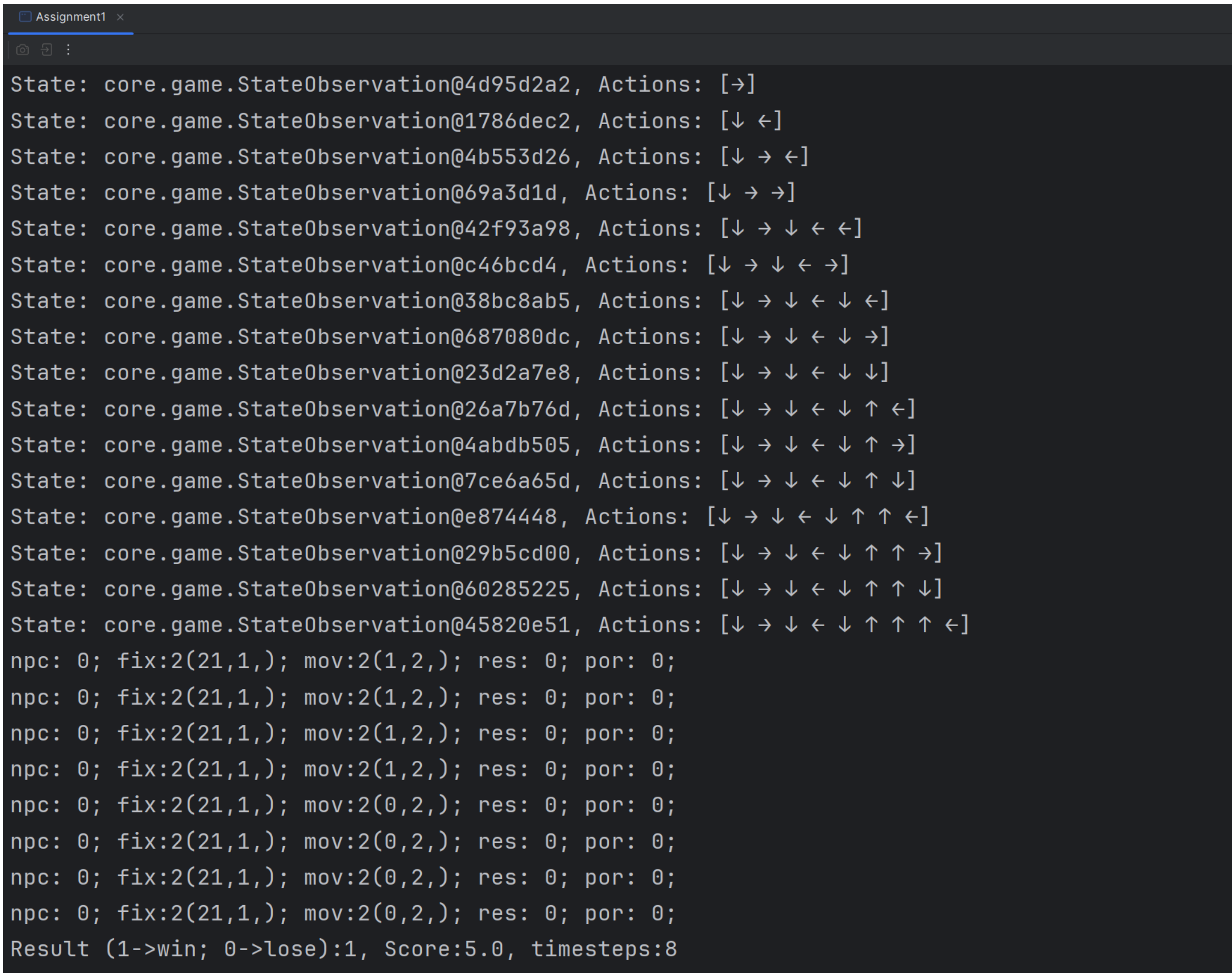
```java
6 个用法
private static class Node {
    3 个用法
    StateObservation state;
    3 个用法
    List<Types.ACTIONS> actions;

    2 个用法
    Node(StateObservation state, List<Types.ACTIONS> actions) {...}
}

/**
 * Check if the current state results in a win
 * @param reproduction the current state after action
 * @return true if the game is won, false otherwise
 */
1 个用法
private boolean checkWinCondition(StateObservation reproduction) {
    return reproduction.getGameWinner() == Types.WINNER.PLAYER_WINS;
}

/**
 * Process the state to check if it has been visited or if the game is over
 * @param reproduction the current state after action
 * @return true if the state is valid for further exploration, false otherwise
 */
1 个用法
private boolean processState(StateObservation reproduction) {
    // Check if the current state is visited or the game is over
    return !duplicateChecking(reproduction) && !reproduction.isGameOver(); // State is valid for further exploration
}
```

```
6 个用法
private static class Node {
    3 个用法
    StateObservation state;
    3 个用法
    List<Types.ACTIONS> actions;

    2 个用法
    Node(StateObservation state, List<Types.ACTIONS> actions) {...}
}

/**
 * Check if the current state results in a win
 * @param reproduction the current state after action
 * @return true if the game is won, false otherwise
 */
1 个用法
private boolean checkWinCondition(StateObservation reproduction) {
    return reproduction.getGameWinner() == Types.WINNER.PLAYER_WINS;
}

/**
 * Process the state to check if it has been visited or if the game is over
 * @param reproduction the current state after action
 * @return true if the state is valid for further exploration, false otherwise
 */
1 个用法
private boolean processState(StateObservation reproduction) {
    // Check if the current state is visited or the game is over
    return !duplicateChecking(reproduction) && !reproduction.isGameOver(); // State is valid for further exploration
}
```

其他辅助函数，用于使搜索主函数更加易懂。

1.5.3　运行结果：





```
Assignment1
State: core.game.StateObservation@4d95d2a2, Actions: [→]
State: core.game.StateObservation@1786dec2, Actions: [↓ ←]
State: core.game.StateObservation@4b553d26, Actions: [↓ → ←]
State: core.game.StateObservation@69a3d1d, Actions: [↓ → →]
State: core.game.StateObservation@42f93a98, Actions: [↓ → ↓ ← ←]
State: core.game.StateObservation@c46bcd4, Actions: [↓ → ↓ ← →]
State: core.game.StateObservation@38bc8ab5, Actions: [↓ → ↓ ← ↓ ←]
State: core.game.StateObservation@687080dc, Actions: [↓ → ↓ ← ↓ →]
State: core.game.StateObservation@23d2a7e8, Actions: [↓ → ↓ ← ↓ ↓]
State: core.game.StateObservation@26a7b76d, Actions: [↓ → ↓ ← ↓ ↑ ←]
State: core.game.StateObservation@4abdb505, Actions: [↓ → ↓ ← ↓ ↑ →]
State: core.game.StateObservation@7ce6a65d, Actions: [↓ → ↓ ← ↓ ↑ ↓]
State: core.game.StateObservation@e874448, Actions: [↓ → ↓ ← ↓ ↑ ↑ ←]
State: core.game.StateObservation@29b5cd00, Actions: [↓ → ↓ ← ↓ ↑ ↑ →]
State: core.game.StateObservation@60285225, Actions: [↓ → ↓ ← ↓ ↑ ↑ ↓]
State: core.game.StateObservation@45820e51, Actions: [↓ → ↓ ← ↓ ↑ ↑ ↑ ←]
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
Result (1->win; 0->lose):1, Score:5.0, timesteps:8
```

```
State: core.game.StateObservation@4d95d2a2, Actions: [→]
State: core.game.StateObservation@1786dec2, Actions: [↓ ←]
State: core.game.StateObservation@4b553d26, Actions: [↓ → ←]
State: core.game.StateObservation@69a3d1d, Actions: [↓ → →]
State: core.game.StateObservation@42f93a98, Actions: [↓ → ↓ ← ←]
State: core.game.StateObservation@c46bcd4, Actions: [↓ → ↓ ← →]
State: core.game.StateObservation@38bc8ab5, Actions: [↓ → ↓ ← ↓ ←]
State: core.game.StateObservation@687080dc, Actions: [↓ → ↓ ← ↓ →]
State: core.game.StateObservation@23d2a7e8, Actions: [↓ → ↓ ← ↓ ↓]
State: core.game.StateObservation@26a7b76d, Actions: [↓ → ↓ ← ↓ ↑ ←]
State: core.game.StateObservation@4abdb505, Actions: [↓ → ↓ ← ↓ ↑ →]
State: core.game.StateObservation@7ce6a65d, Actions: [↓ → ↓ ← ↓ ↑ ↓]
State: core.game.StateObservation@e874448, Actions: [↓ → ↓ ← ↓ ↑ ↑ ←]
State: core.game.StateObservation@29b5cd00, Actions: [↓ → ↓ ← ↓ ↑ ↑ →]
State: core.game.StateObservation@60285225, Actions: [↓ → ↓ ← ↓ ↑ ↑ ↓]
State: core.game.StateObservation@45820e51, Actions: [↓ → ↓ ← ↓ ↑ ↑ ↑ ←]
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
Result (1->win; 0->lose):1, Score:5.0, timesteps:8
```

运行结果无问题

1.6 番外：优化代码

1.6.1 查询资料表明 ArrayList 的查询为 O(n)复杂度，而 HashSet 只需要 O(1)，所以更换数据结构：

2 深度受限的深度优先搜索

2.1 第一版：

2.1.1 此时我试图尝试使用一个简单的深度迭代进行尝试：

每一次搜索只搜索一个固定的深度（从 1 开始），搜索不到加深深度。

```
boolean getDepthFirstActionsIterative(StateObservation initialState, ElapsedCpuTimer
    // Define a stack to hold pairs of state and the corresponding action sequence
    Stack<Node> stack = new Stack<>();
    stack.push(new Node(initialState, new ArrayList<>(), 0)); // Push initial node wi

    while (!stack.isEmpty()) {

        printStack(stack);

        Node currentNode = stack.pop();
        StateObservation currentState = currentNode.state;
        List<Types.ACTIONS> currentActions = currentNode.actions;
        int currentDepth = currentNode.currentDepth; // Get current depth

        // Check for win condition
        if (checkWinCondition(currentState)) {
            actionsSequence.clear();
            actionsSequence.addAll(currentActions);
            return true;
        }

        // If state is valid for exploration and depth has not been exceeded
        if (processState(currentState) && currentDepth < depthLimit) {
            // Mark the state as visited
            visitedState.add(currentState);

            // Iterate through available actions
            for (Types.ACTIONS action : currentState.getAvailableActions()) {
                StateObservation nextState = currentState.copy();
                nextState.advance(action);

                List<Types.ACTIONS> newActions = new ArrayList<>(currentActions);
                newActions.add(action);

                stack.push(new Node(nextState, newActions, currentDepth + 1)); // Inc
            }
        }
```

```java
    boolean getDepthFirstActionsIterative(StateObservation initialState, ElapsedCpuTimer
        // Define a stack to hold pairs of state and the corresponding action sequence
        Stack<Node> stack = new Stack<>();
        stack.push(new Node(initialState, new ArrayList<>(), 0)); // Push initial node wi

        while (!stack.isEmpty()) {

            printStack(stack);

            Node currentNode = stack.pop();
            StateObservation currentState = currentNode.state;
            List<Types.ACTIONS> currentActions = currentNode.actions;
            int currentDepth = currentNode.currentDepth; // Get current depth

            // Check for win condition
            if (checkWinCondition(currentState)) {
                actionsSequence.clear();
                actionsSequence.addAll(currentActions);
                return true;
            }

            // If state is valid for exploration and depth has not been exceeded
            if (processState(currentState) && currentDepth < depthLimit) {
                // Mark the state as visited
                visitedState.add(currentState);

                // Iterate through available actions
                for (Types.ACTIONS action : currentState.getAvailableActions()) {
                    StateObservation nextState = currentState.copy();
                    nextState.advance(action);

                    List<Types.ACTIONS> newActions = new ArrayList<>(currentActions);
                    newActions.add(action);

                    stack.push(new Node(nextState, newActions, currentDepth + 1)); // Inc
                }
            }
```

发现这么搜索其实是严重降低性能的：这样导致了整体的搜索结构变成了类 BFS 样式，但是显著增加了栈开销（这一点可视化非常明显），正好看到了题意的提示，于是转而使用启发式函数。

2.2 第二版

2.2.1 开始设计启发式函数。

```java
private double heuristic(StateObservation state) {
    ArrayList<Observation>[] fixedPositions = state.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = state.getMovablePositions();

    Vector2d goalpos = fixedPositions[1].get(0).position; // 目标位置
    Vector2d keypos = movingPositions[0].get(0).position; // 钥匙位置

    // 计算曼哈顿距离
    double distanceToGoal = Math.abs(goalpos.x - state.getAvatarPosition().x) +
            Math.abs(goalpos.y - state.getAvatarPosition().y);
    double distanceToKey = Math.abs(keypos.x - state.getAvatarPosition().x) +
            Math.abs(keypos.y - state.getAvatarPosition().y);

    // 组合启发式值
    return distanceToGoal + distanceToKey; // 根据你的游戏逻辑进行调整
}
```


```java
private double heuristic(StateObservation state) {
    ArrayList<Observation>[] fixedPositions = state.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = state.getMovablePositions();

    Vector2d goalpos = fixedPositions[1].get(0).position; // 目标位置
    Vector2d keypos = movingPositions[0].get(0).position; // 钥匙位置

    // 计算曼哈顿距离
    double distanceToGoal = Math.abs(goalpos.x - state.getAvatarPosition().x) +
            Math.abs(goalpos.y - state.getAvatarPosition().y);
    double distanceToKey = Math.abs(keypos.x - state.getAvatarPosition().x) +
            Math.abs(keypos.y - state.getAvatarPosition().y);

    // 组合启发式值
    return distanceToGoal + distanceToKey; // 根据你的游戏逻辑进行调整
}
```


这里是望文生义写的第一版启发式函数，就是单纯的计算了一下曼哈顿距离，很快就出现了问题，由于往终点的权重和往钥匙的权重是一样的，这直接导致了人物在中间位置横跳，对此我一开始想的办法是进行加权：

```java
private double heuristic(StateObservation state) {
    ArrayList<Observation>[] fixedPositions = state.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = state.getMovablePositions();

    Vector2d goalpos = fixedPositions[1].get(0).position; // 目标位置
    Vector2d keypos = movingPositions[0].get(0).position; // 钥匙位置

    // 计算曼哈顿距离
    double distanceToGoal = Math.abs(goalpos.x - state.getAvatarPosition().x) +
            Math.abs(goalpos.y - state.getAvatarPosition().y);
    double distanceToKey = Math.abs(keypos.x - state.getAvatarPosition().x) +
            Math.abs(keypos.y - state.getAvatarPosition().y);

    // 组合启发式值
    return distanceToGoal + 3 * distanceToKey; // 根据你的游戏逻辑进行调整
}
```


```java
private double heuristic(StateObservation state) {
    ArrayList<Observation>[] fixedPositions = state.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = state.getMovablePositions();

    Vector2d goalpos = fixedPositions[1].get(0).position; // 目标位置
    Vector2d keypos = movingPositions[0].get(0).position; // 钥匙位置

    // 计算曼哈顿距离
    double distanceToGoal = Math.abs(goalpos.x - state.getAvatarPosition().x) +
            Math.abs(goalpos.y - state.getAvatarPosition().y);
    double distanceToKey = Math.abs(keypos.x - state.getAvatarPosition().x) +
            Math.abs(keypos.y - state.getAvatarPosition().y);

    // 组合启发式值
    return distanceToGoal + 3 * distanceToKey; // 根据你的游戏逻辑进行调整
}
```


但很快我发现这只不过是在另一个位置上打转，究其原因，是因为无论设置怎么样的比例系数，终究会有一个平衡点，为此，必须分情况讨论：

2.2.1.1 （奇了怪了）

```java
Vector2d playerPos = stateObs.getAvatarPosition(); // 精灵的位置
Vector2d goalpos = stateObs.getImmovablePositions()[1].get(0).position; //目标的坐标
Vector2d keypos = stateObs.getMovablePositions()[0].get(0).position; //钥匙的坐标
```

```
Vector2d playerPos = stateObs.getAvatarPosition(); // 精灵的位置
Vector2d goalpos = stateObs.getImmovablePositions()[1].get(0).position; //目标的坐标
Vector2d keypos = stateObs.getMovablePositions()[0].get(0).position; //钥匙的坐标
```

这么写不报错

```
ArrayList[] fixedPositions = stateObs.getImmovablePositions();
ArrayList[] movingPositions = stateObs.getMovablePositions();
Vector2d goalpos = fixedPositions[1].get(0).position //目标的坐标
Vector2d keypos = movingPositions[0].get(0).position //钥匙的坐标
```

```
ArrayList[] fixedPositions = stateObs.getImmovablePositions();
ArrayList[] movingPositions = stateObs.getMovablePositions();
Vector2d goalpos = fixedPositions[1].get(0).position //目标的坐标
Vector2d keypos = movingPositions[0].get(0).position //钥匙的坐标
```

这么写就报错，也不知道是哪地方的问题。

好吧这样写也是有问题的，问题就在于钥匙被吃掉之后，那个值变成了 null 赋值了一个 Vector2d 类型，直接给程序干崩溃了，所以最后的选择是：

```
boolean getDepthFirstActionsIterative(StateObservation initialState, ElapsedCpuTimer elapsedTimer) {
    long startTime = System.currentTimeMillis();
    Vector2d goalpos = initialState.getImmovablePositions()[1].get(0).position; //目标的坐标
    Vector2d keypos = initialState.getMovablePositions()[0].get(0).position; //钥匙的坐标

    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingDouble(n -> n.distance));
    queue.add(new Node(initialState, new ArrayList<>(), depth: 0, distance(initialState, goalpos, keypos)));
```

具体实现方法：

```
double distance(StateObservation stateObs, Vector2d goalpos, Vector2d keypos) {
    Vector2d playerPos = stateObs.getAvatarPosition(); // 精灵的位置


    // if the player has the key
    if (hasKey) {
        return Math.abs(goalpos.x - playerPos.x) + Math.abs(goalpos.y - playerPos.y);
    }

    // if the player has visited the key
    boolean hasVisitedKey = visitedState.stream()
            .anyMatch(so -> so.getAvatarPosition().equals(keypos));

    //if the player has visited the key
    if (hasVisitedKey) {
        return Math.abs(goalpos.x - playerPos.x) + Math.abs(goalpos.y - playerPos.y);
    }

    // if the player has not visited the key
    return Math.abs(playerPos.x - keypos.x) + Math.abs(playerPos.y - keypos.y) + Math.abs(goalpos.x - keypos.x) + Math.abs(goalp
}
```

```java
boolean getDepthFirstActionsIterative(StateObservation initialState, ElapsedCpuTimer elapsedTimer) {
    long startTime = System.currentTimeMillis();
    Vector2d goalpos = initialState.getImmovablePositions()[1].get(0).position; //目标的坐标
    Vector2d keypos = initialState.getMovablePositions()[0].get(0).position; //钥匙的坐标

    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingDouble(n -> n.distance));
    queue.add(new Node(initialState, new ArrayList<>(), depth: 0, distance(initialState, goalpos, keypos)));
```

在初始化搜索框架时添加，这样一次赋值，每次需要使用时传参就可以了。

2.3 更改数据结构

2.3.1　现在我们需要每次从序列中取出距离最小点，考虑到这一需求，应该将栈结构改为最小堆结构，此时由于第一个任务我们使用显式栈写法，现在我们只需要对这一数据结构进行更改就行了：

```java
boolean getDepthFirstActionsIterative(StateObservation initialState, ElapsedCpuTimer elapsedTimer) {
    long startTime = System.currentTimeMillis();
    Vector2d goalpos = initialState.getImmovablePositions()[1].get(0).position; //目标的坐标
    Vector2d keypos = initialState.getMovablePositions()[0].get(0).position; //钥匙的坐标

    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingDouble(n -> n.distance));
    queue.add(new Node(initialState, new ArrayList<>(), depth: 0, distance(initialState, goalpos, keypos)));


    while (!queue.isEmpty()) {
        printQueue(queue);

        Node currentNode = queue.poll();
        StateObservation currentState = null;
        if (currentNode != null) {
            currentState = currentNode.state;
        }else {
            System.out.println("Current node is null");
        }

        List<Types.ACTIONS> currentActions = null;
        if (currentNode != null) {
            currentActions = currentNode.actions;
        }else {
            System.out.println("Current actions is null");
        }

        int currentDepth = 0;
        if (currentNode != null) {
            currentDepth = currentNode.depth;
        }else{
            System.out.println("Current depth is null");
        }
```

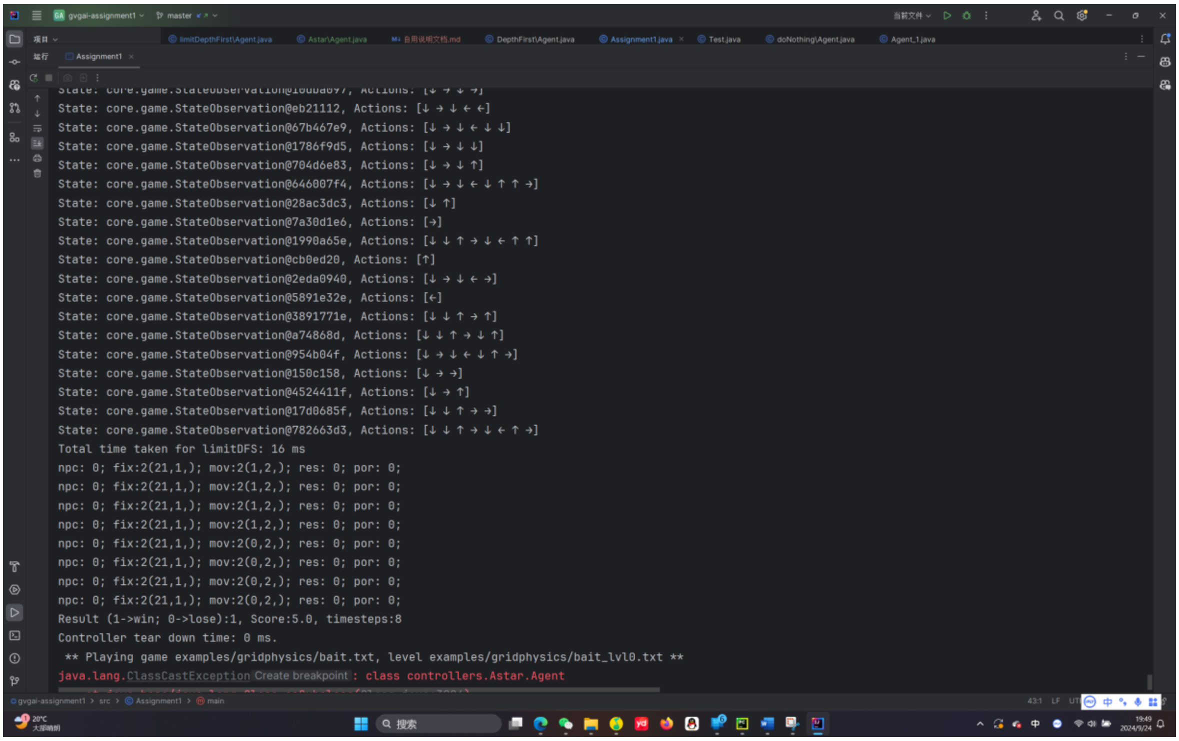
对取节点的方式进行更改：

```java
if (processState(currentState)) {
    visitedState.add(currentState);

    for (Types.ACTIONS action : currentState.getAvailableActions()) {
        StateObservation nextState = currentState.copy();
        nextState.advance(action);

        List<Types.ACTIONS> newActions = new ArrayList<>(currentActions);
        newActions.add(action);

        // 仅当未找到胜利路径时才更新评分
        if (currentDepth + 1 == searchDepth && bestScore != Double.NEGATIVE_INFINITY) {
            double score = -50 * (searchDepth - currentDepth);
            if (score < bestScore) {
                bestScore = score;
                bestAction = new ArrayList<>(newActions);
            }
        }
        queue.add(new Node(nextState, newActions, depth: currentDepth + 1, distance(nextState, goalpos, keypos)));
    }
}
```

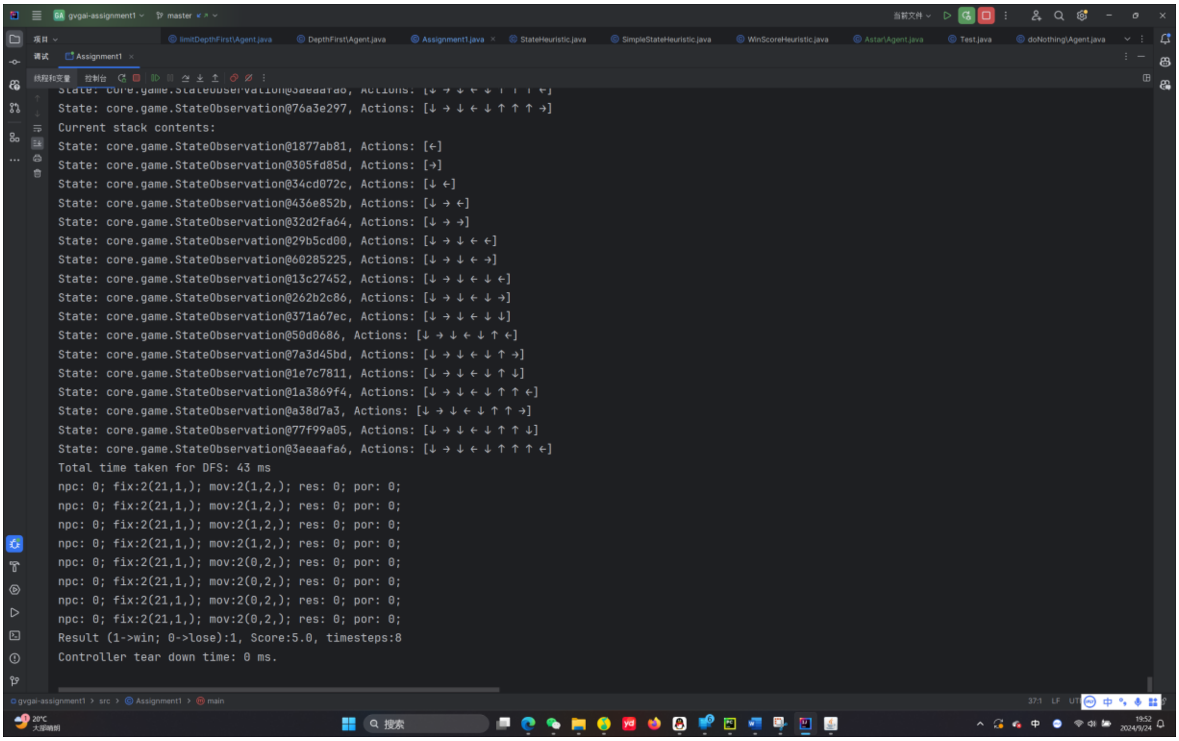
2.4 运行尝试：

Java-VGDL: S...

2.4.1



gvgai-assignment1
master
limitDepthFirst\Agent.java
Astar\Agent.java
自用说明文档.md
DepthFirst\Agent.java
Assignment1.java
Assignment1
State: core.game.StateObservation@eb21112, Actions: [↓ → ↓ ← ←]
State: core.game.StateObservation@67b467e9, Actions: [↓ → ↓ ← ↓ ↓]
State: core.game.StateObservation@1786f9d5, Actions: [↓ → ↓ ↓]
State: core.game.StateObservation@704d6e83, Actions: [↓ → ↓ ↑]
State: core.game.StateObservation@646007f4, Actions: [↓ → ↓ ← ↓ ↑ ↑ →]
State: core.game.StateObservation@28ac3dc3, Actions: [↓ ↑]
State: core.game.StateObservation@7a30d1e6, Actions: [→]
State: core.game.StateObservation@1990a65e, Actions: [↓ ↓ ↑ → ↓ ← ↑ ↑]
State: core.game.StateObservation@cb0ed20, Actions: [↑]
State: core.game.StateObservation@2eda0940, Actions: [↓ → ↓ ← →]
State: core.game.StateObservation@5891e32e, Actions: [←]
State: core.game.StateObservation@3891771e, Actions: [↓ ↓ ↑ → ↑]
State: core.game.StateObservation@a74868d, Actions: [↓ ↓ ↑ → ↓ ↑]
State: core.game.StateObservation@954b04f, Actions: [↓ → ↓ ← ↓ ↑ →]
State: core.game.StateObservation@150c158, Actions: [↓ → →]
State: core.game.StateObservation@4524411f, Actions: [↓ → ↑]
State: core.game.StateObservation@17d0685f, Actions: [↓ ↓ ↑ → →]
State: core.game.StateObservation@782663d3, Actions: [↓ ↓ ↑ → ↓ ← ↑ →]
Total time taken for limitDFS: 16 ms
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(1,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
npc: 0; fix:2(21,1,); mov:2(0,2,); res: 0; por: 0;
Result (1->win; 0->lose):1, Score:5.0, timesteps:8
Controller tear down time: 0 ms.
** Playing game examples/gridphysics/bait.txt, level examples/gridphysics/bait_lvl0.txt **
java.lang.ClassCastException Create breakpoint : class controllers.Astar.Agent
gvgai-assignment1 > src > Assignment1 > main



第二个是第一个任务的搜索结果，可以看到效率提升了三倍，这是因为每次扩展出的四个节点中，我们都选择了可能的最优解进行处理，自然提升了搜索的效率。

3　A*算法的实现

3.1　初步思路设计

3.1.1　其实第二个深度受限里面是用最小堆的思路已经包含了一些 A*的想法，因此我们可以继续尝试完善：成本估计和实际代价的启发式函数：

```java
double g(StateObservation stateObs) { 4 个用法
    return actionsSequence.size() * 180;  //这个参数可以调整，50效果不是很好，40效果比较好，25以下会在箱子推走后卡住，150以上会在另一个方向卡住
}

/**
 * 计算从当前状态到目标状态的启发式成本h(n)。
 *
 * @param stateObs 当前状态观察
 * @param hasKey   指示是否已经找到钥匙
 * @return 当前状态到目标的启发式估计成本
 */
double h(StateObservation stateObs, boolean hasKey) { 4 个用法
    Vector2d playerPos = stateObs.getAvatarPosition(); // 获取精灵位置
    if (hasKey) {
        return Math.abs(goalpos.x - playerPos.x) + Math.abs(goalpos.y - playerPos.y);
    } else {
        double distanceToKey = Math.abs(playerPos.x - keypos.x) + Math.abs(playerPos.y - keypos.y);
        return distanceToKey + Math.abs(goalpos.x - keypos.x) + Math.abs(goalpos.y - keypos.y);
    }
}
```

这里对步数所造成的代价乘上了一个参数，用于平衡深度优先和广度优先。

3.1.2 扩展节点内容：

```java
public class Node {
    public Node(StateObservation state, double h, double g, ArrayList<Types.ACTIONS> actions, 
        this.state = state.copy();
        this.h = h;
        this.g = g;
        this.f = h + g;
        this.actions = new ArrayList<>(actions); // 使用构造函数直接克隆
        this.pastState = new ArrayList<>(pastState); // 使用构造函数直接克隆
        this.hasKey = hasKey;
    } // 初始化

    StateObservation state;        7 个用法
    double g;               1 个用法
    double h;            1 个用法
    double f;    3 个用法
    ArrayList<Types.ACTIONS> actions;    2 个用法
    ArrayList<StateObservation> pastState;   2 个用法
    boolean hasKey;          1 个用法

    public Node parent;
}
```

3.1.3 完成标准 A*算法模板，具体内容

```java
while (!openState.isEmpty()) { // 当仍有待展开节点时继续搜索
    Node temp = openState.poll(); // 获取评分最优的节点
    actionsSequence = new ArrayList<>(temp.actions); // 克隆动作序列
    targetPastState = new ArrayList<>(temp.pastState); // 克隆当前节点的历史状态
    visitedState.add(temp.state); // 添加当前节点状态到已访问状态列表
    targetPastState.add(temp.state); // 更新当前节点的历史状态
    // 达到搜索深度限制，则退出
    if (actionsSequence.size() == searchDepth) {
        return;
    }
    checkForKey(temp.state, keypos); // 检查是否到达钥匙位置
    // 遍历可用动作并扩展状态
    for (Types.ACTIONS action : temp.state.getAvailableActions()) {
        // 使用新的状态扩展函数
        StateObservation transcript = expandState(temp, action); // 扩展状态
        updateActionsSequence(actionsSequence, action); // 添加动作
        // 使用新的胜利检查函数
        if (checkVictory(transcript)) {
            return;
        }
        // 检查游戏是否结束或状态是否已访问
        if (isGameOverOrVisited(transcript)) {
            actionsSequence.remove( index: actionsSequence.size() - 1); // 移除最后一个动作
            continue; // 继续尝试下一个动作
        }
        // 处理当前状态在优先队列中的情况
        processNodeInQueue(transcript, actionsSequence); // 调用处理节点函数
    }
```

写出一个比较标准的 A*搜索过程，并逐渐完善内部函数：

```
扩展当前状态并处理相关逻辑。
形参: temp – 当前节点
      action – 要应用的动作
返回值: 返回扩展后的状态
StateObservation expandState(@NotNull Node temp, Types.ACTIONS action) {...}

更新动作序列，添加新的动作。
形参: actionsSequence – 当前的动作序列
      action – 被添加的动作
void updateActionsSequence(@NotNull ArrayList<Types.ACTIONS> actionsSequence, Types.ACTIONS action) {...}

检查当前状态是否胜利。
形参: state – 当前状态观察
返回值: 如果胜利则返回true，否则返回false
boolean checkVictory(@NotNull StateObservation state) { return state.getGameWinner() == Types.WINNER.PLAYER_WINS; }

处理当前状态在优先队列中的情况。
形参: stCopy – 当前状态的副本
      actionsSequence – 当前的动作序列
void processNodeInQueue(StateObservation stCopy, ArrayList<Types.ACTIONS> actionsSequence) {...}

检查游戏是否结束或状态是否已访问。
形参: state – 当前状态观察
返回值: 如果游戏结束或状态已访问则返回true，否则返回false
boolean isGameOverOrVisited(@org.jetbrains.annotations.NotNull StateObservation state) {  1 个用法
    return state.isGameOver() || duplicateChecking(state); // 游戏结束或状态已访问
游戏未结束且状态未访问
}

检查当前精灵位置是否到达钥匙位置。
形参: state – 当前状态观察
      keypos – 钥匙位置
```

3.1.4　调参：

3.1.4.1　在一开始，我想的是虽然可能会有障碍物，但是两地之间的距离大致还是以曼哈顿距离为主，因此给实际代价的参数设成了 2，结果会在第二关卡住：

我一开始觉得是搜索深度不够，毕竟这关时间比较充裕，可以考虑增加深度。

增加深度后成功过关，但是第三关直接卡死。

这说明增加深度的方法其实是一种暴力，他会使得整个搜索更偏向于 BFS 寻找最短路径的情况，这是我

们不愿意看到的。

3.1.4.2 回调参数：

因此我决定将搜索深度调回 32，增加实际代价的参数，使得整个搜索更偏向 DFS：经过一系列调参，我发现参数在 28-150 之间可以稳定通过第三关，但是第四关无论如何也过不去：

我一开始以为是算力不够了，试图增加深度，但是查看错误信息后发现：

```
Exception in thread "main" java.lang.IndexOutOfBoundsException Create breakpoint : Index 0 out of bounds for length 0
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)
    at java.base/java.util.Objects.checkIndex(Objects.java:359)
    at java.base/java.util.ArrayList.get(ArrayList.java:427)
    at controllers.Astar.Agent.act(Agent.java:289)
    at ontology.avatar.MovingAvatar.requestPlayerInput(MovingAvatar.java:135)
    at ontology.avatar.MovingAvatar.update(MovingAvatar.java:97)
    at core.game.Game.tick(Game.java:1046)
    at core.game.Game.gameCycle(Game.java:928)
    at core.game.Game.playGame(Game.java:839)
    at core.ArcadeMachine.runOneGame(ArcadeMachine.java:106)
```

是堆中已经没有元素了，说明当前局面进入了死局，所有方向均不能通关。

3.1.5 但是 A*算法应该是必定能找到最优解的，哪里出了问题？

根据我的判断，应该是缺乏对坑和箱子的信息判断，导致我单纯使用距离的启发式函数并不是单调的，形成了局部最优的死局。

4 MCTS 算法介绍

4.1 MCTS算法基本逻辑：

4.1.1 概述：MCTS 算法包含一下几个部分：

4.1.1.1 选择（Selection）：从根节点开始，选择一个子节点，直到达到一个尚未完全展开的节点。选择的过程通常使用某种启发式方法，如上置信界（UCB）算法，以平衡探索和利用。

4.1.1.2 扩展（Expansion）：在选择的节点上，生成一个或多个子节点，表示可能的后续状态。

4.1.1.3 模拟（Simulation）：从扩展的节点开始，进行随机模拟，直到达到游戏的终局。这一步骤通常是通过随机选择动作来完成的。

4.1.1.4 反向传播（Backpropagation）：将模拟结果（胜利、失败或平局）反向传播到选择路径上的所有节点，更新它们的胜利次数和访问次数。

4.1.2 UCB 算法的应用：

```
public SingleTreeNode uct() { 1 个用法

    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)
    {...}

    return selected;
}
```

这里的 UCT 函数（Upper Confidence bounds for Trees）如下：

$$\mathrm{UCT} = \frac{\mathrm{totValue}}{nVisits + \epsilon} + K \cdot \sqrt{\frac{\log(nVisits + 1)}{nVisits + \epsilon}}$$

这里 totValue 是节点的总回报值，nVisits 是该节点的访问次数，K 是调节探索的常数，ε 是避免除以零的小常数。

4.1.3 扩展过程的处理：

```
public SingleTreeNode expand() {  1个用法
    int bestAction = 0;
    double bestValue = -1;
    for (int i = 0; i < children.length; i++) {
        double x = m_rnd.nextDouble();
        if (x > bestValue && children[i] == null) {
            bestAction = i;
            bestValue = x;
        }
    }
    StateObservation nextState = state.copy();
    nextState.advance(Agent.actions[bestAction]);
    SingleTreeNode tn = new SingleTreeNode(nextState, parent: this, this.m_rnd);
    children[bestAction] = tn;
    return tn;
}
```

可以看到，该函数寻找一个子节点为 null 的节点，产生一个新状态，添加至子节点并将其返回。

```
public SingleTreeNode treePolicy() {  1个用法
    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;
}
```

在这里调用了 uct 和 expand 函数，用于向树的叶子结点方向遍历直至寻找一个可以扩展的节点。

4.1.4 模拟过程：

```
public double rollOut() 1 个用法
{
    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;
}
```

首先，方法复制当前节点的状态 state，以便在模拟过程中不影响原始状态。在一个循环中，方法随机选择一个可用的动作（通过 m_rnd.nextInt(Agent.NUM_ACTIONS)），并使用 rollerState.advance() 方法更新状态。这个过程会持续进行，直到满足结束条件。模拟过程会检查是否达到游戏结束状态或达到最大深度（通过调用 finishRollout() 方法）。一旦模拟结束，方法会调用 value() 方法计算最终状态的回报值（即游戏得分）。根据计算的回报值，更新当前节点的边界值（bounds），以便在后续的搜索中使用。

4.1.5 反向传播的实现：

```
public void backUp(SingleTreeNode node, double result) 1 个用法
{
    SingleTreeNode n = node;
    while(n != null)
    {
        n.nVisits++;
        n.totValue += result;
        n = n.parent;
    }
}
```

简单增加访问次数和回报次数。

4.1.6 主体搜索过程：

```java
public void mctsSearch(ElapsedCpuTimer elapsedTimer) { 1 个用法

    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 + ")");
}
```

可以看到主循环就是按序执行四个步骤，并将它们加上时间限制。其余两个类的代码为简单的框架代码，具体介绍上文已有，不在赘述了。

致谢 在此,我诚挚地感谢 22 级 AI 王俊童同学，如果不是他的点拨，我的深度受限和 A*算法只怕是要自己磕磕绊绊好久。