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. 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。
   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. 整体方案设计

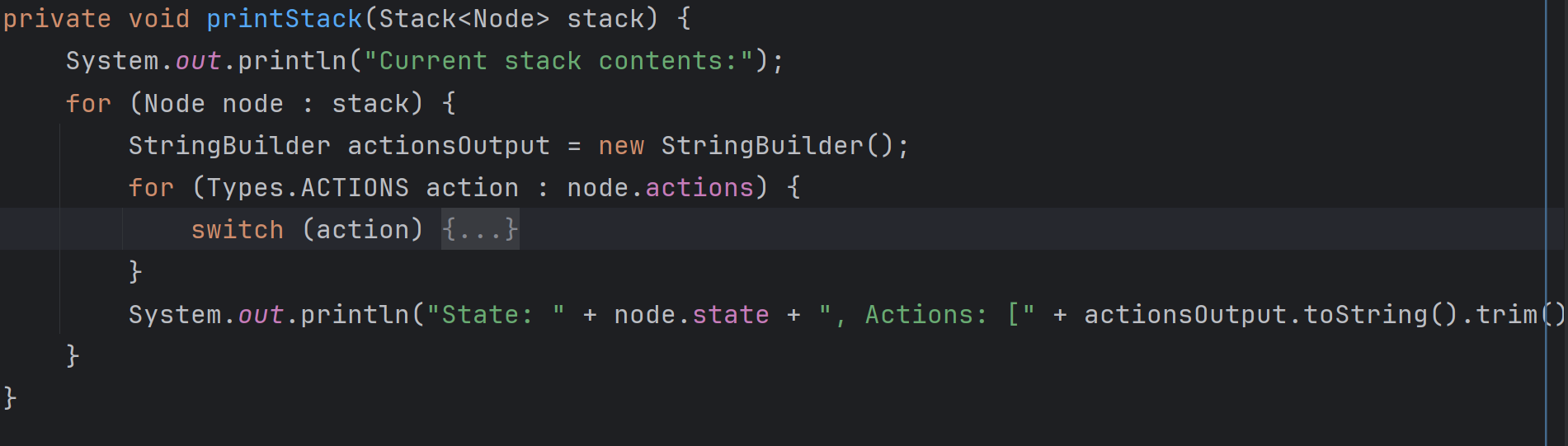
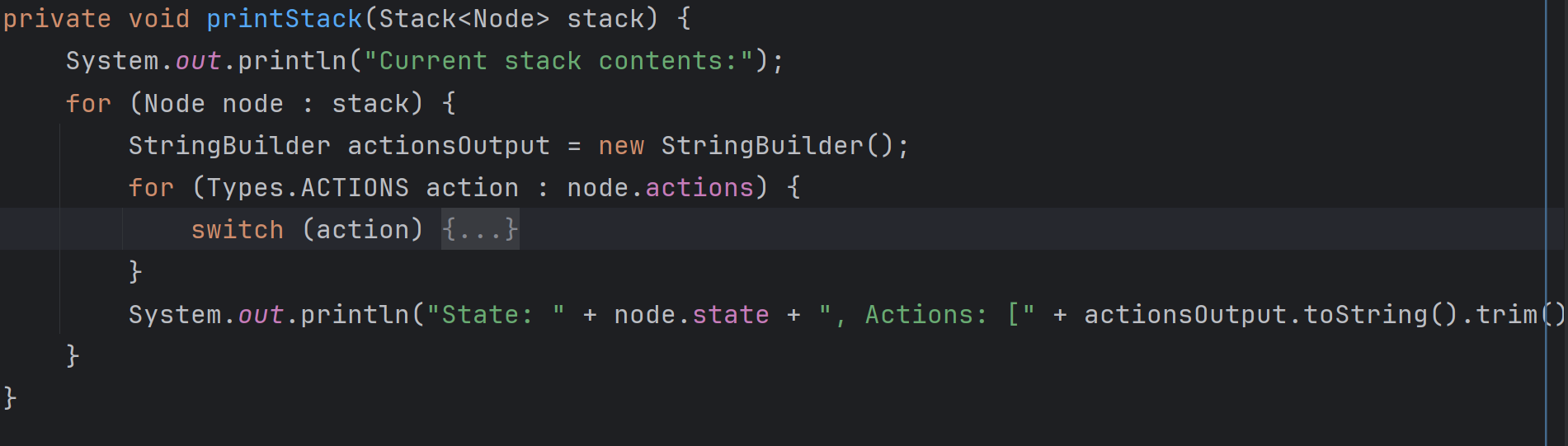
由于没有说明文档，这不得不迫使我RTFSC，并进行梳理，梳理内容大致如下：

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

StateObservation类：

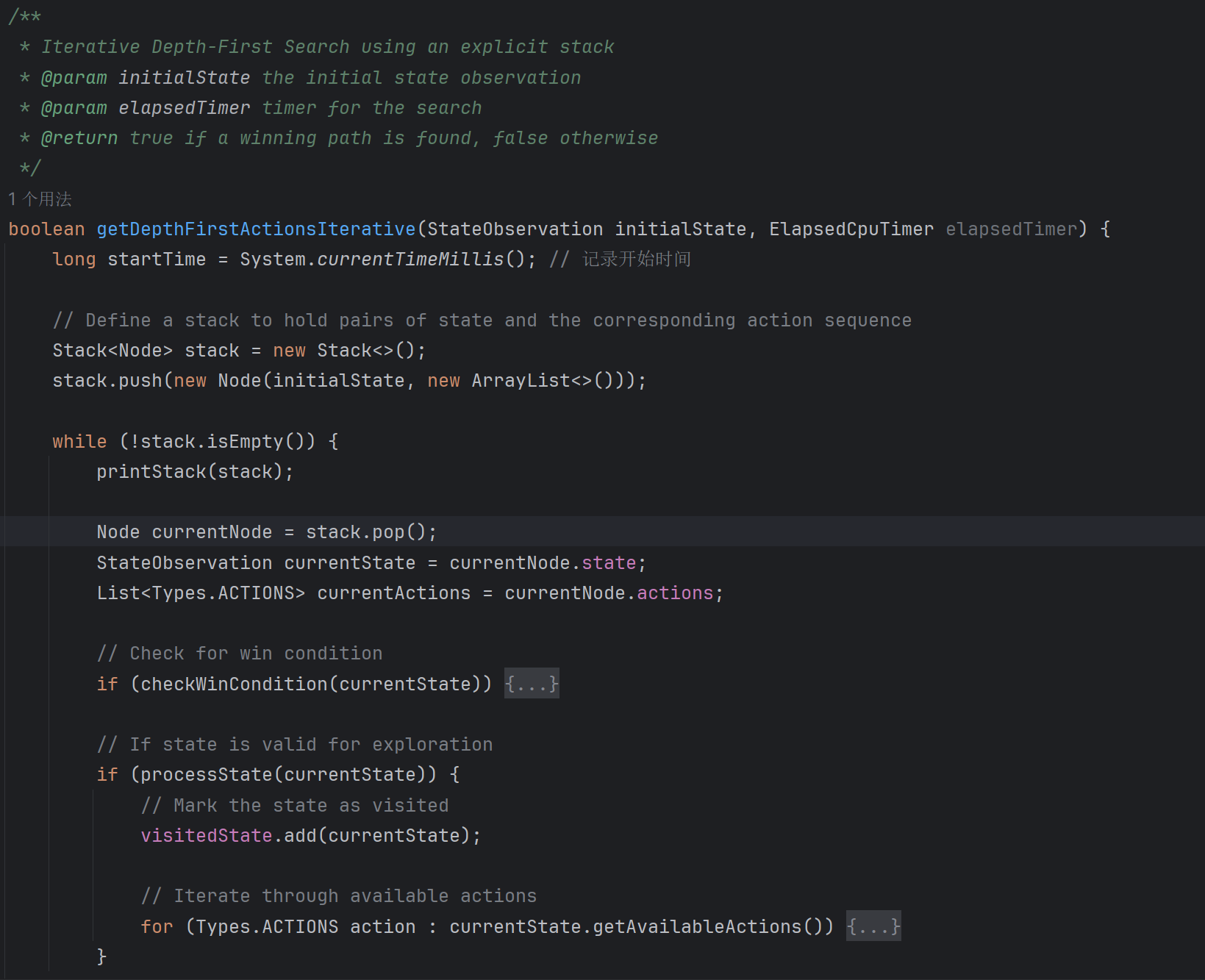
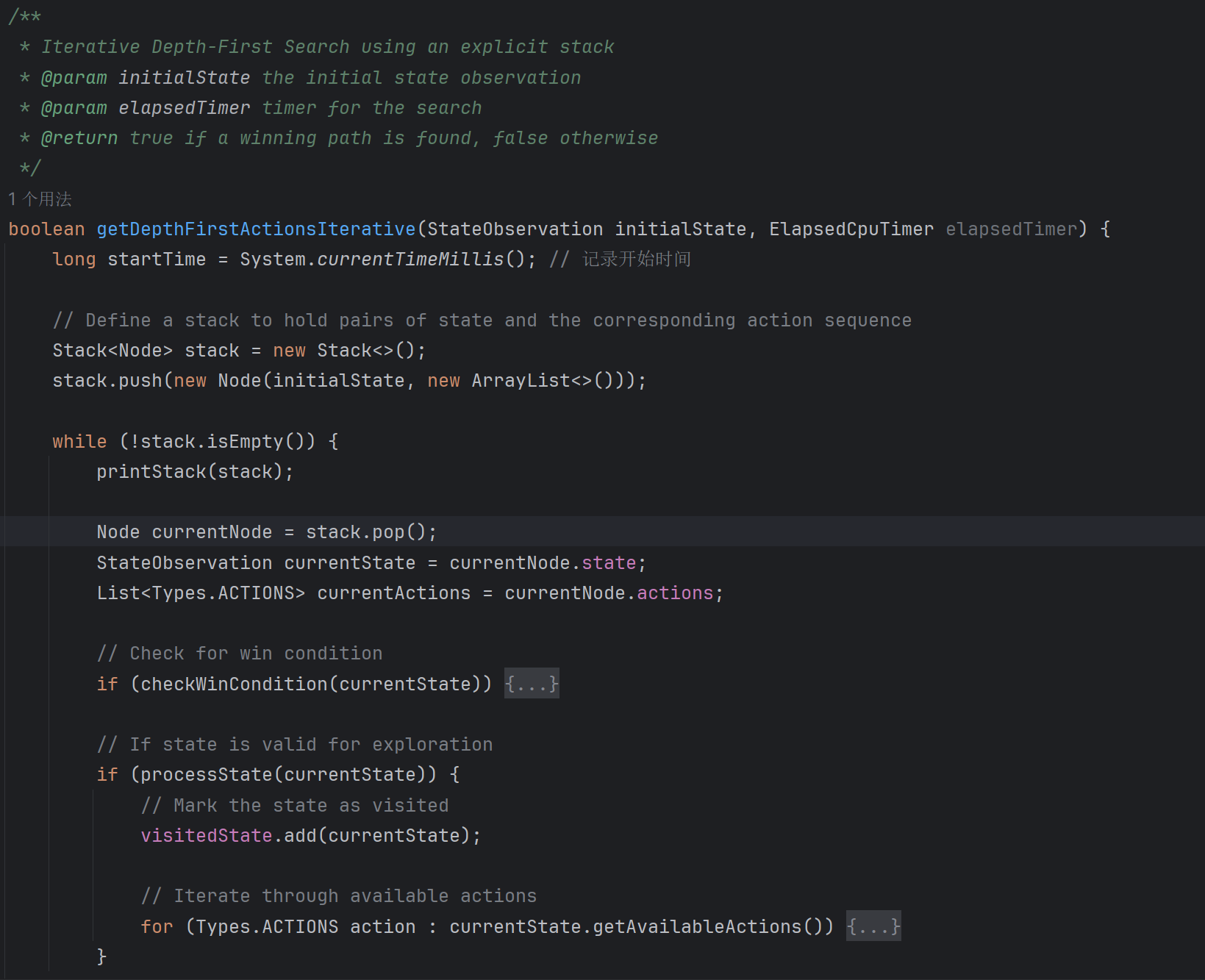
getObservationGrid:功能: 该方法用于获取当前游戏状态的观察网格。具体返回的内容通常包括不同位置的观察数据。

* 1. 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：



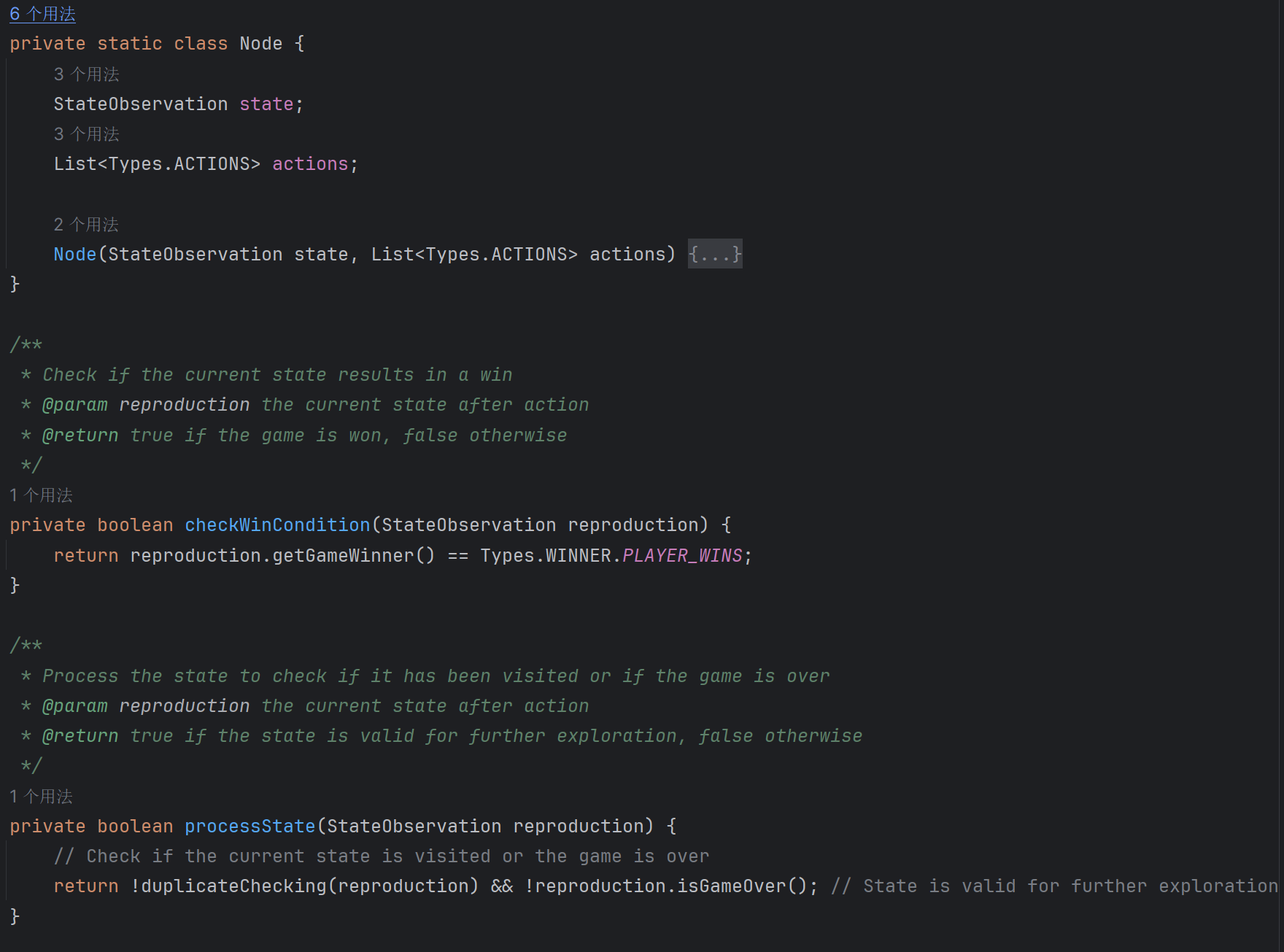
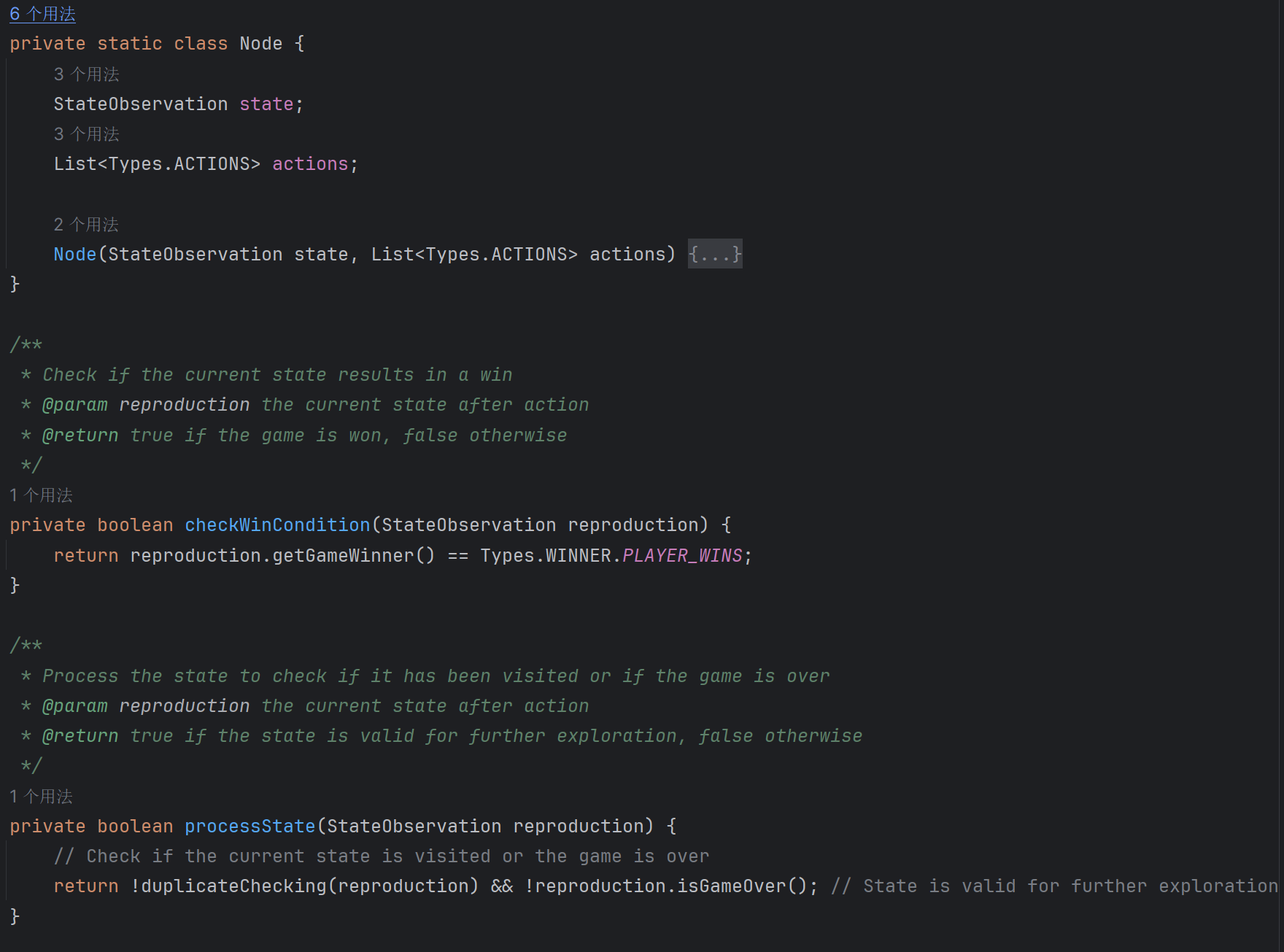
这样就可以随时查看栈的内容，查处可能的搜索问题，并进行改进。

* 1. 具体实现方法
     1. 新建的四个类变量的简单介绍如下：
        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。
        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。
        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`。
        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。
     2. 核心搜索逻辑

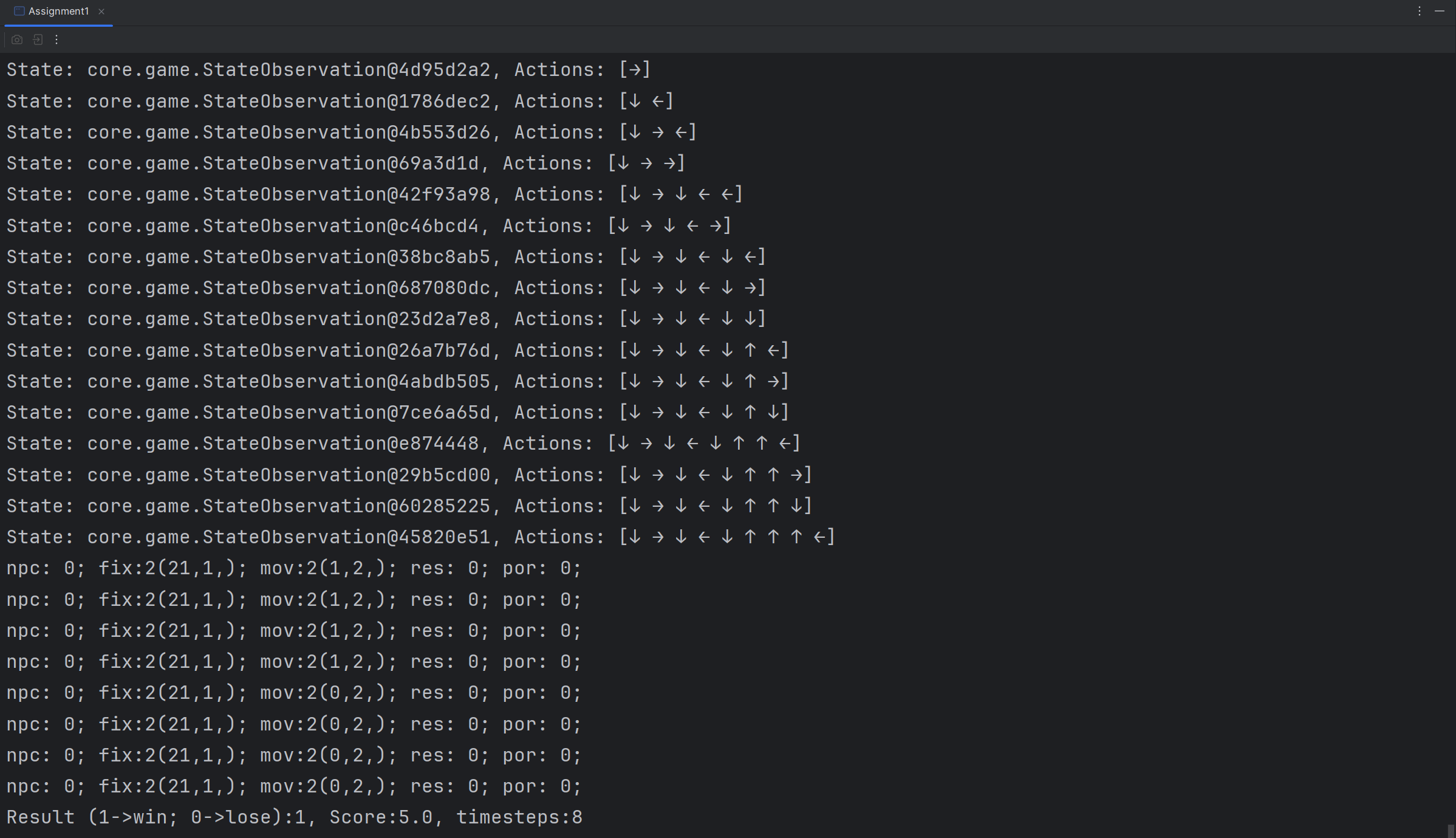
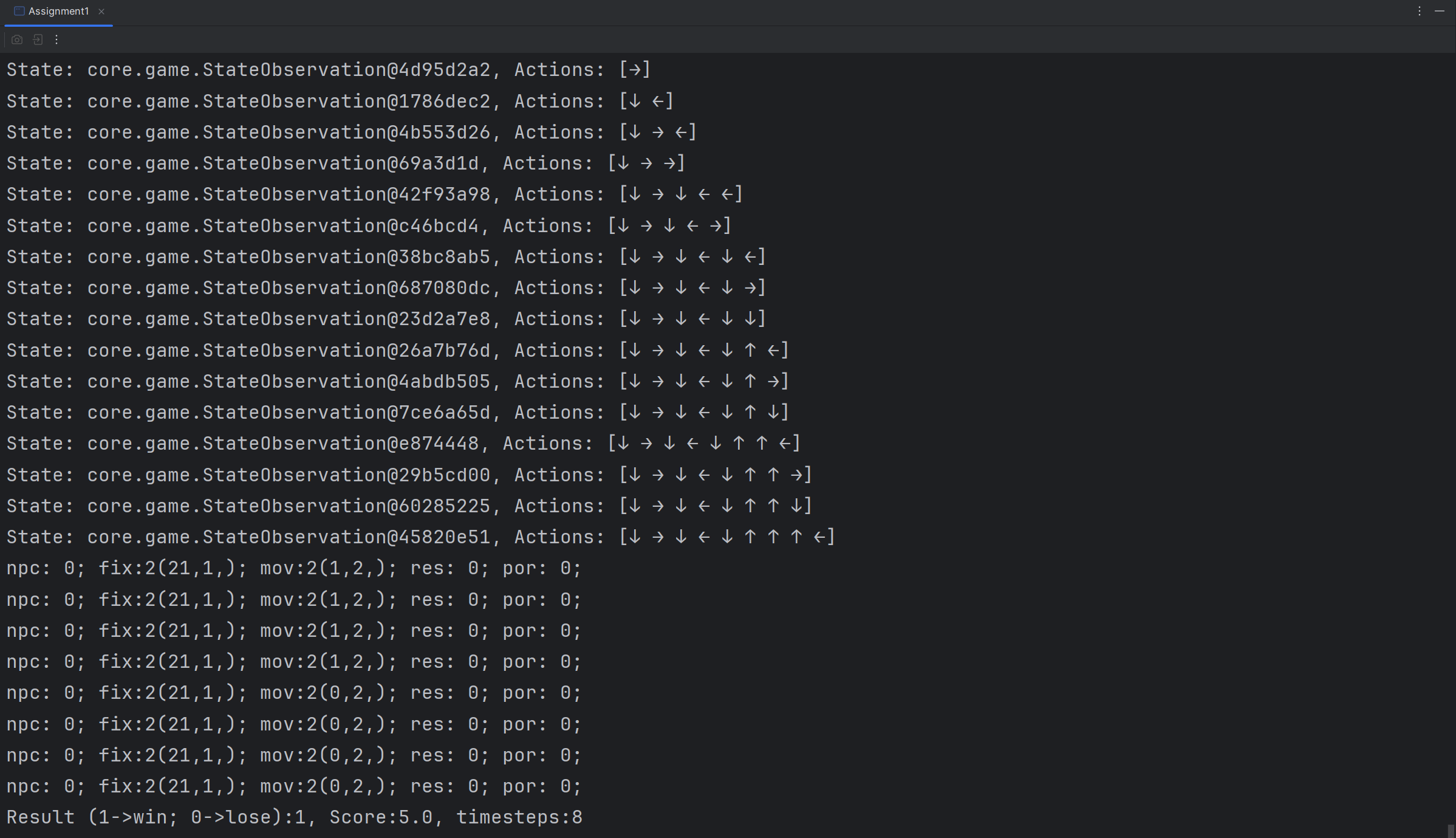


使用一个显式栈来储存状态，并进行搜索，并进行入栈出栈的操作。具体来说，对每一个状态进行结束检查和扩展。

其他辅助函数，用于使搜索主函数更加易懂。



* + 1. 运行结果：

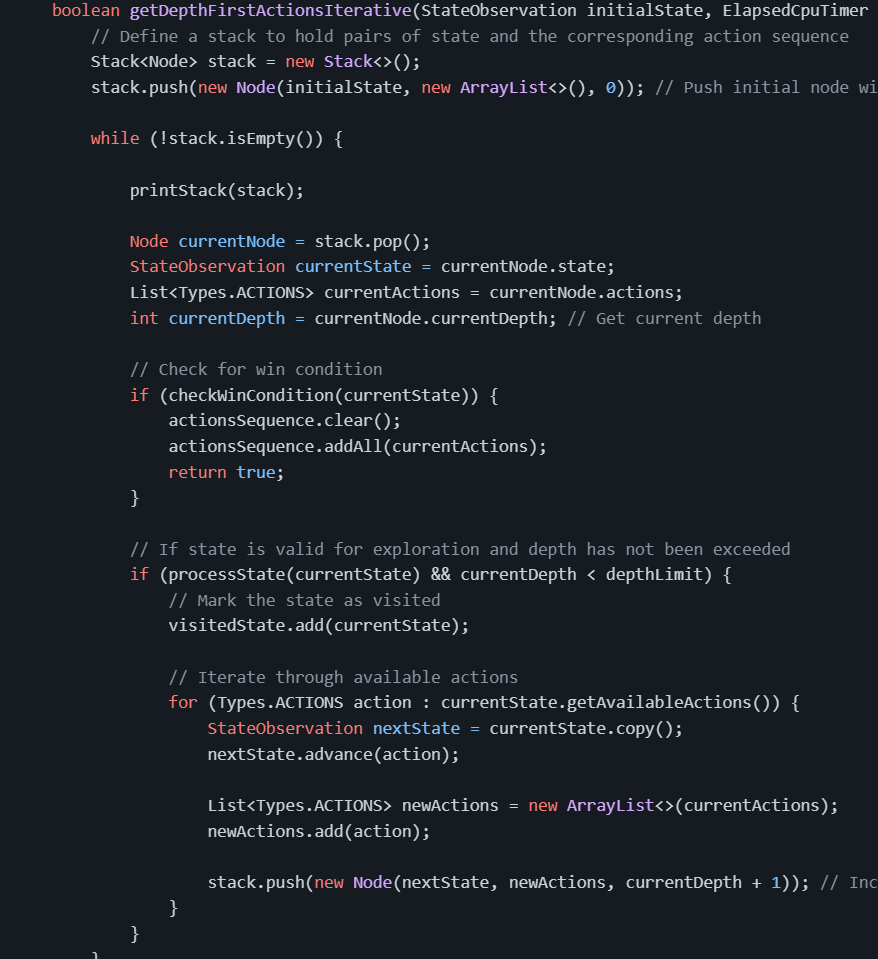
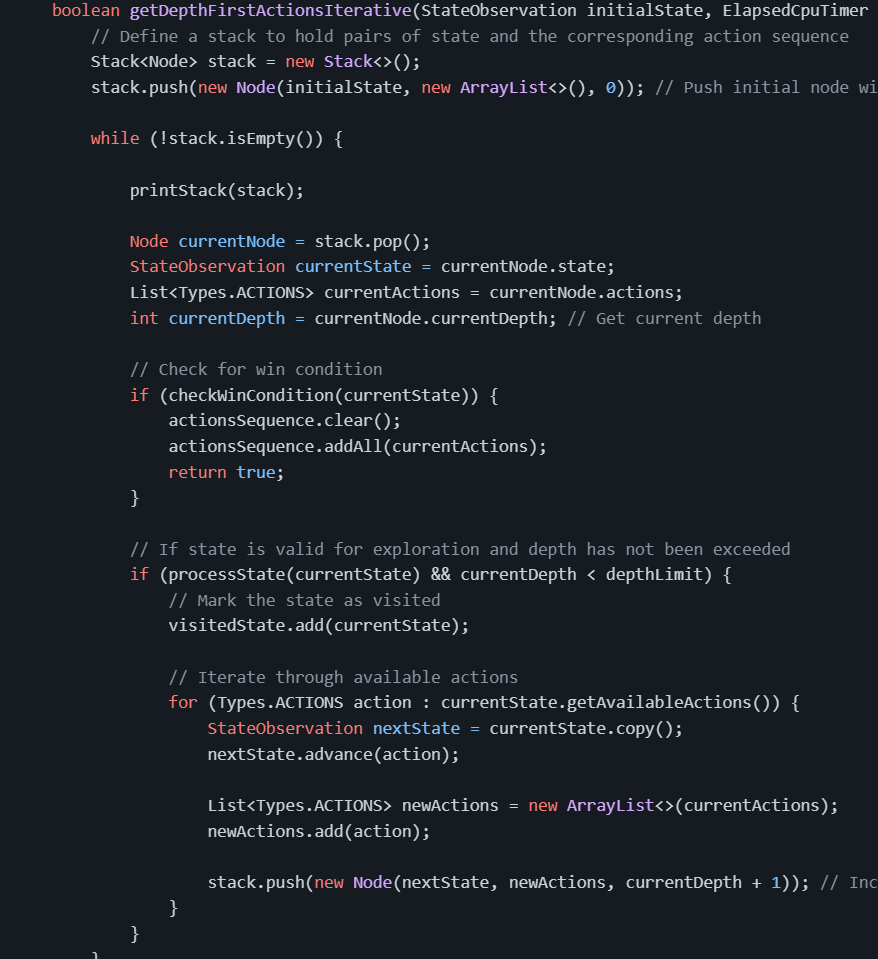


运行结果无问题

* 1. 番外：优化代码
     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：

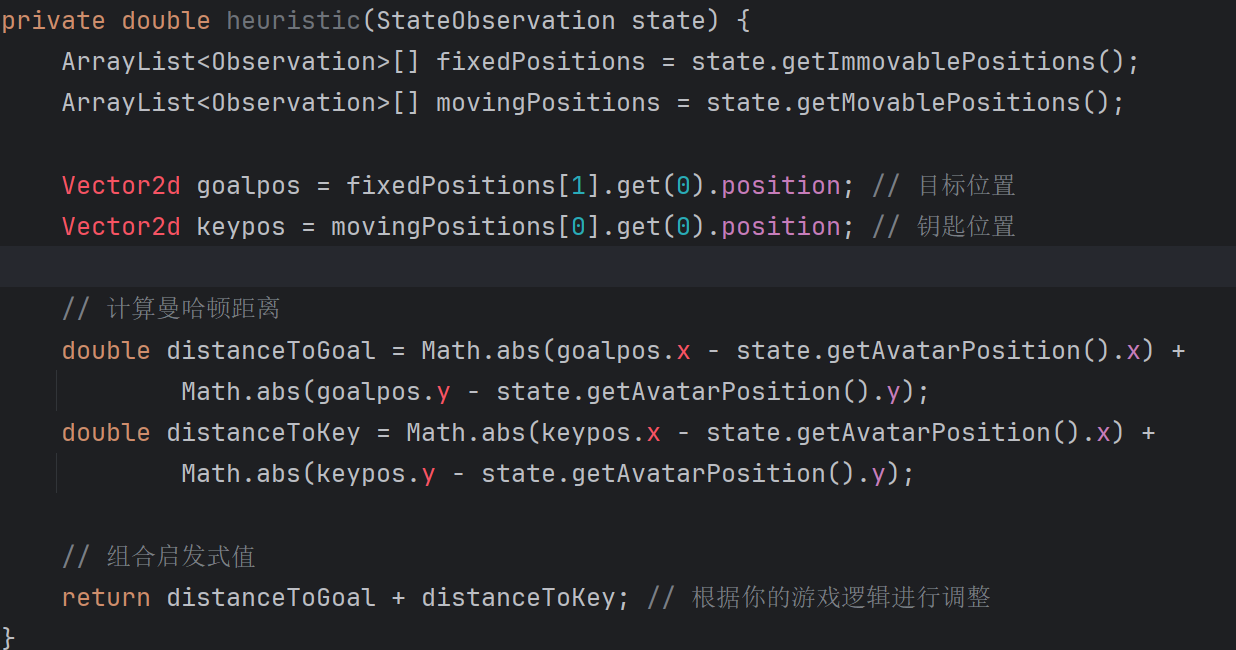
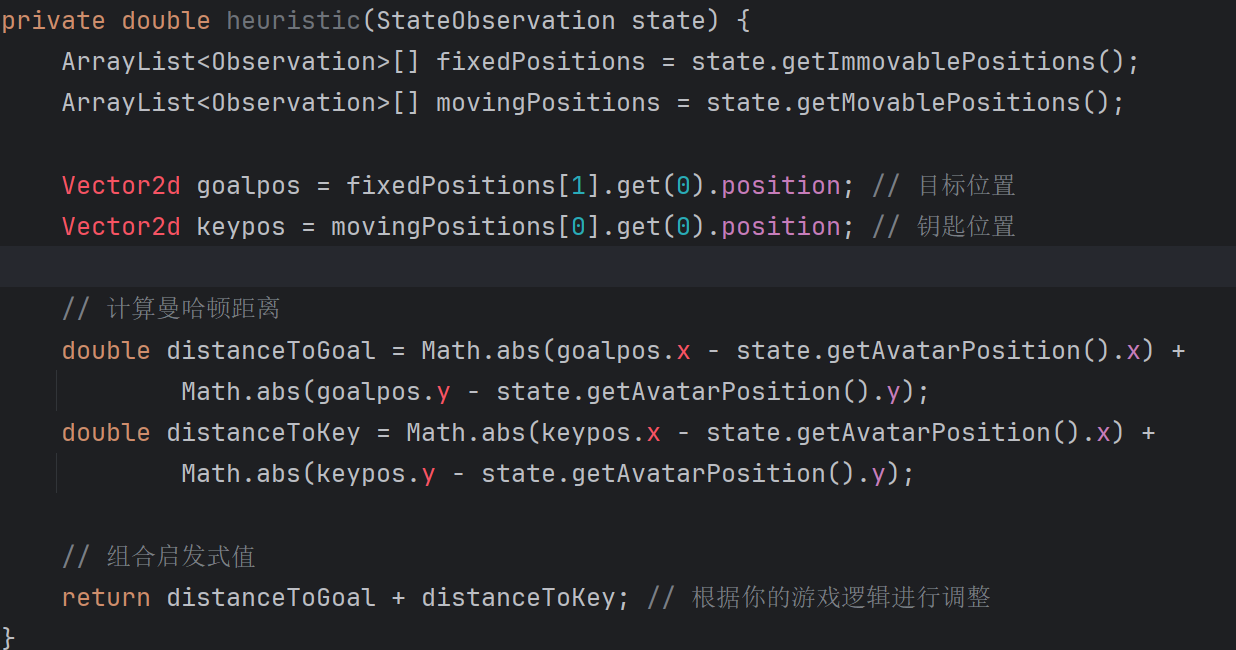
1. 深度受限的深度优先搜索
   1. 第一版：
      1. 此时我试图尝试使用一个简单的深度迭代进行尝试：

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。



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。

* 1. 第二版
     1. 开始设计启发式函数。

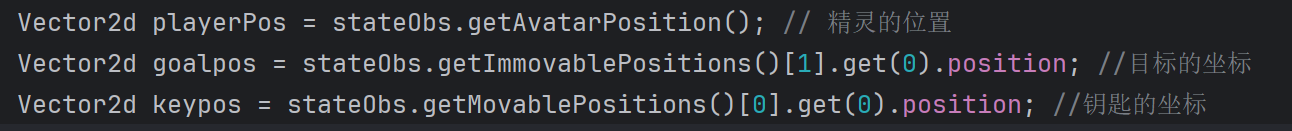
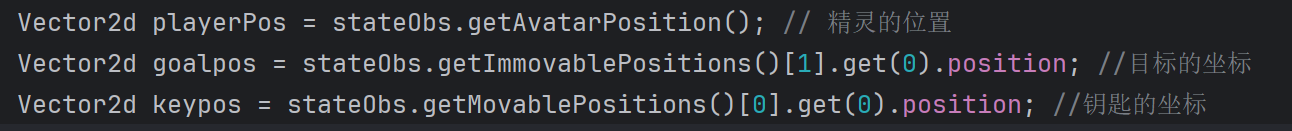


这里是望文生义写的第一版启发式函数，就是单纯的计算了一下曼哈顿距离，很快就出现了问题，由于往终点的权重和往钥匙的权重是一样的，这直接导致了人物在中间位置横跳，对此我一开始想的办法是进行加权：

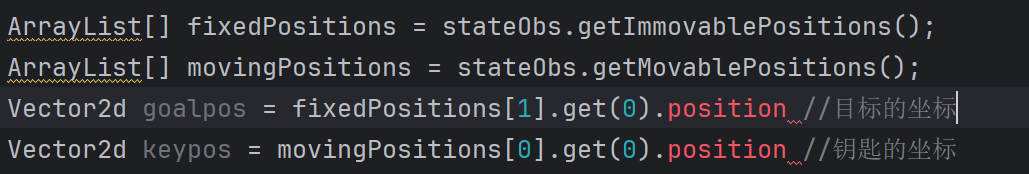
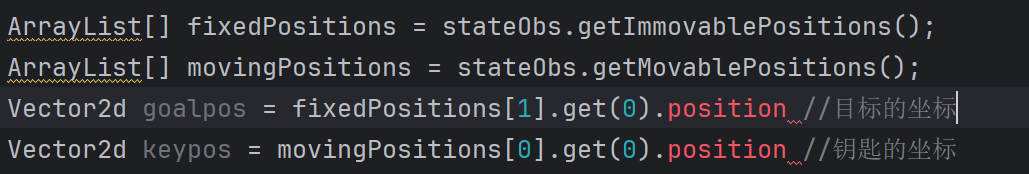


但很快我发现这只不过是在另一个位置上打转，究其原因，是因为无论设置怎么样的比例系数，终究会有一个平衡点，为此，必须分情况讨论：

* + - 1. （奇了怪了）

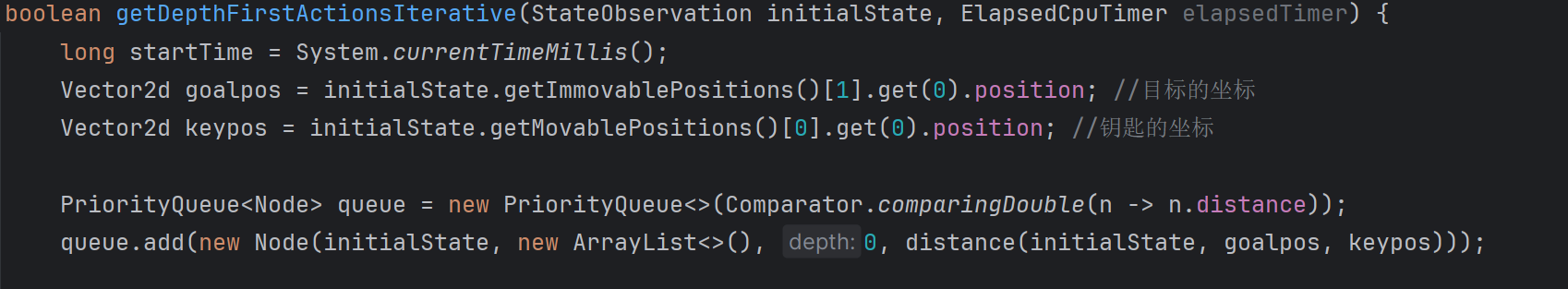


这么写不报错

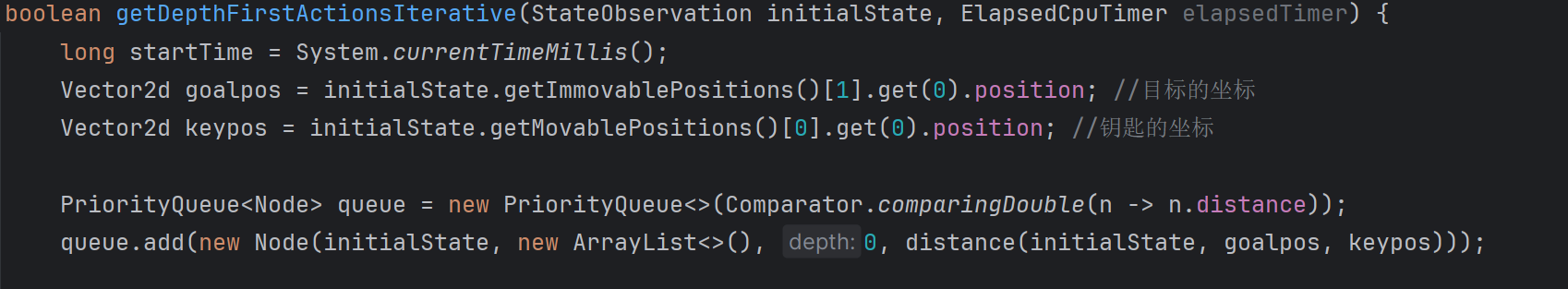


这么写就报错，也不知道是哪地方的问题。

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：



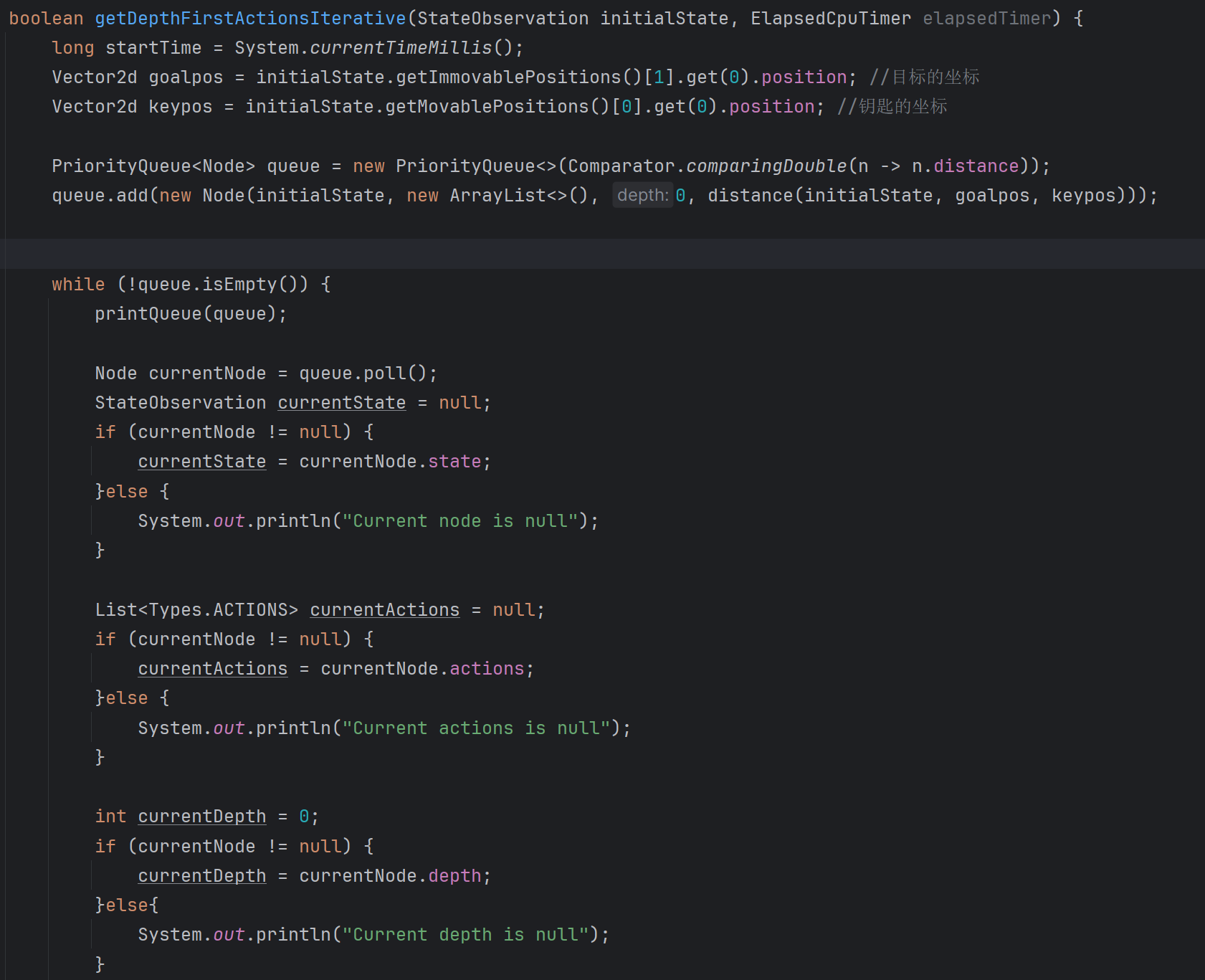
具体实现方法：



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。

## 更改数据结构

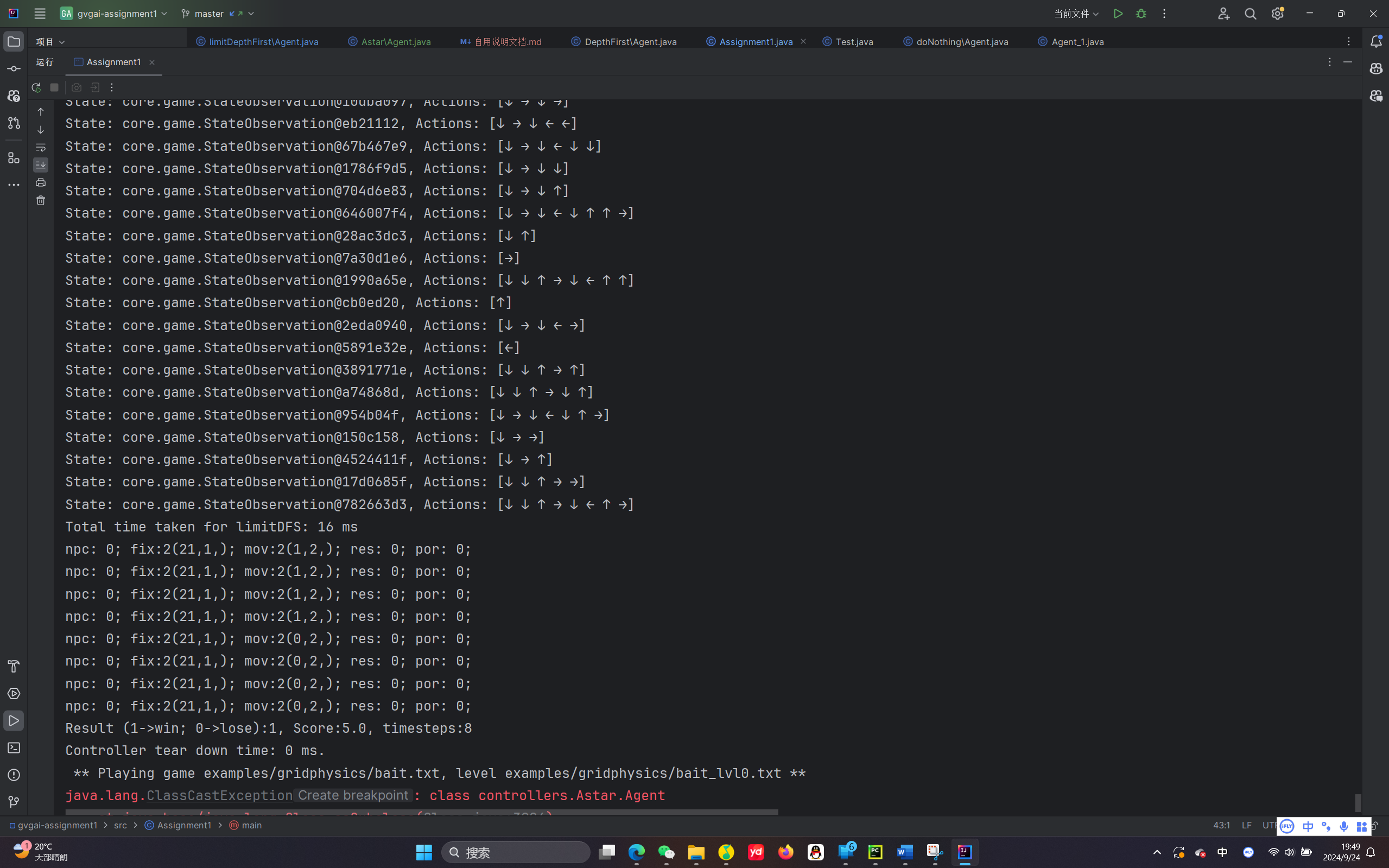
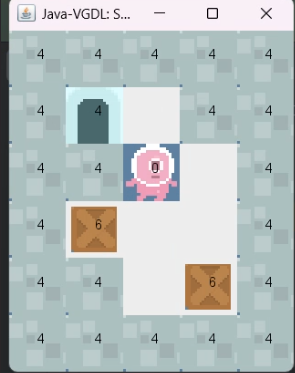
### 现在我们需要每次从序列中取出距离最小点，考虑到这一需求，应该将栈结构改为最小堆结构，此时由于第一个任务我们使用显式栈写法，现在我们只需要对这一数据结构进行更改就行了：



对取节点的方式进行更改：



## 运行尝试：



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。

# A\*算法的实现

## 初步思路设计

### 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：



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。

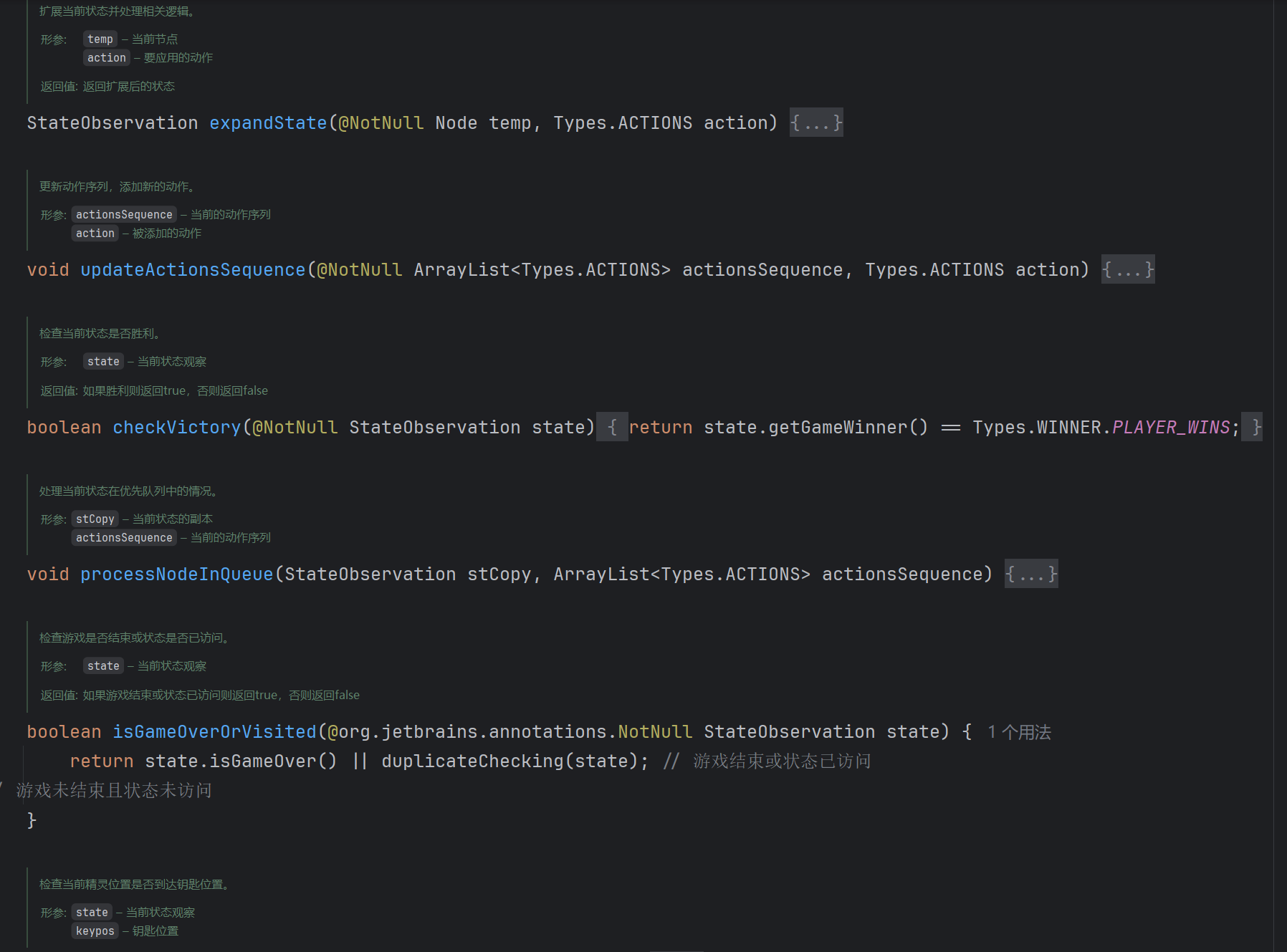
### 扩展节点内容：



### 完成标准A\*算法模板，具体内容



写出一个比较标准的A\*搜索过程，并逐渐完善内部函数：



### 调参：

#### 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：



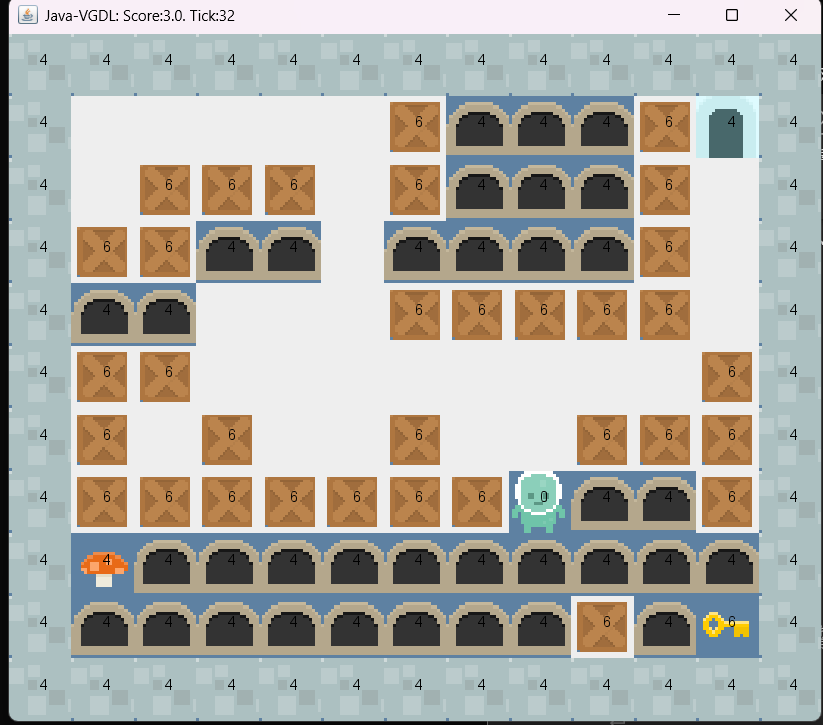
我一开始觉得是搜索深度不够，毕竟这关时间比较充裕，可以考虑增加深度。

增加深度后成功过关，但是第三关直接卡死。

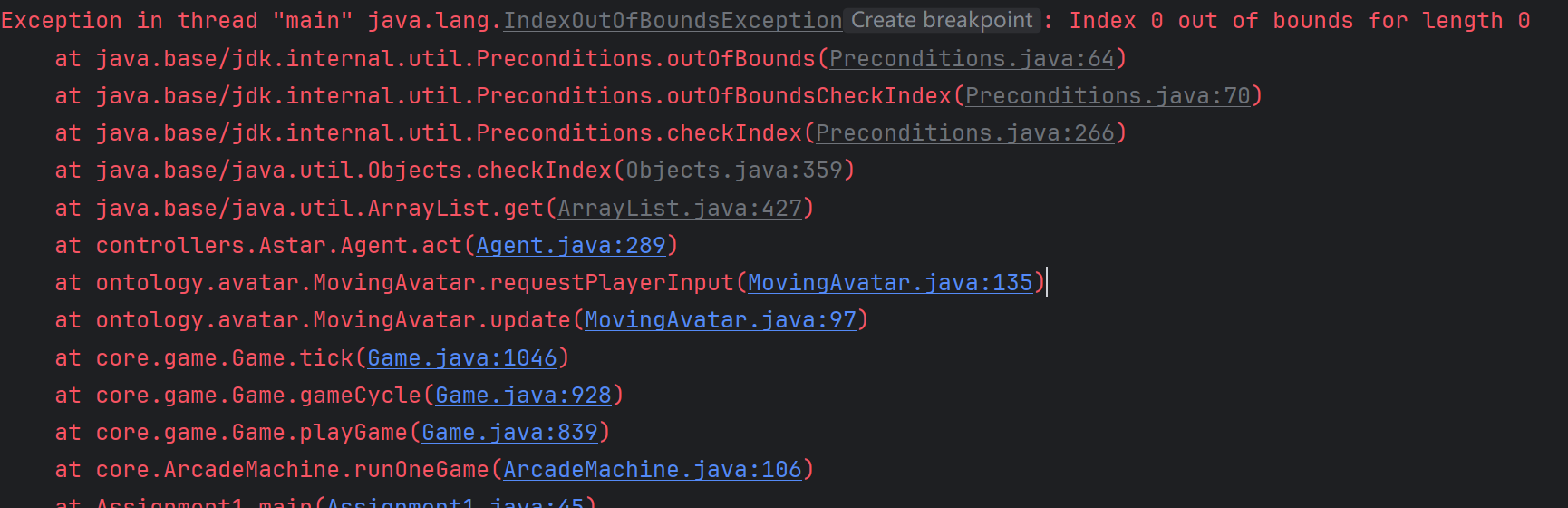
这说明增加深度的方法其实是一种暴力，他会使得整个搜索更偏向于BFS寻找最短路径的情况，这是我们不愿意看到的。

#### 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：



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。

### 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。

# MCTS算法介绍

## MCTS算法基本逻辑：

### 概述：MCTS算法包含一下几个部分：

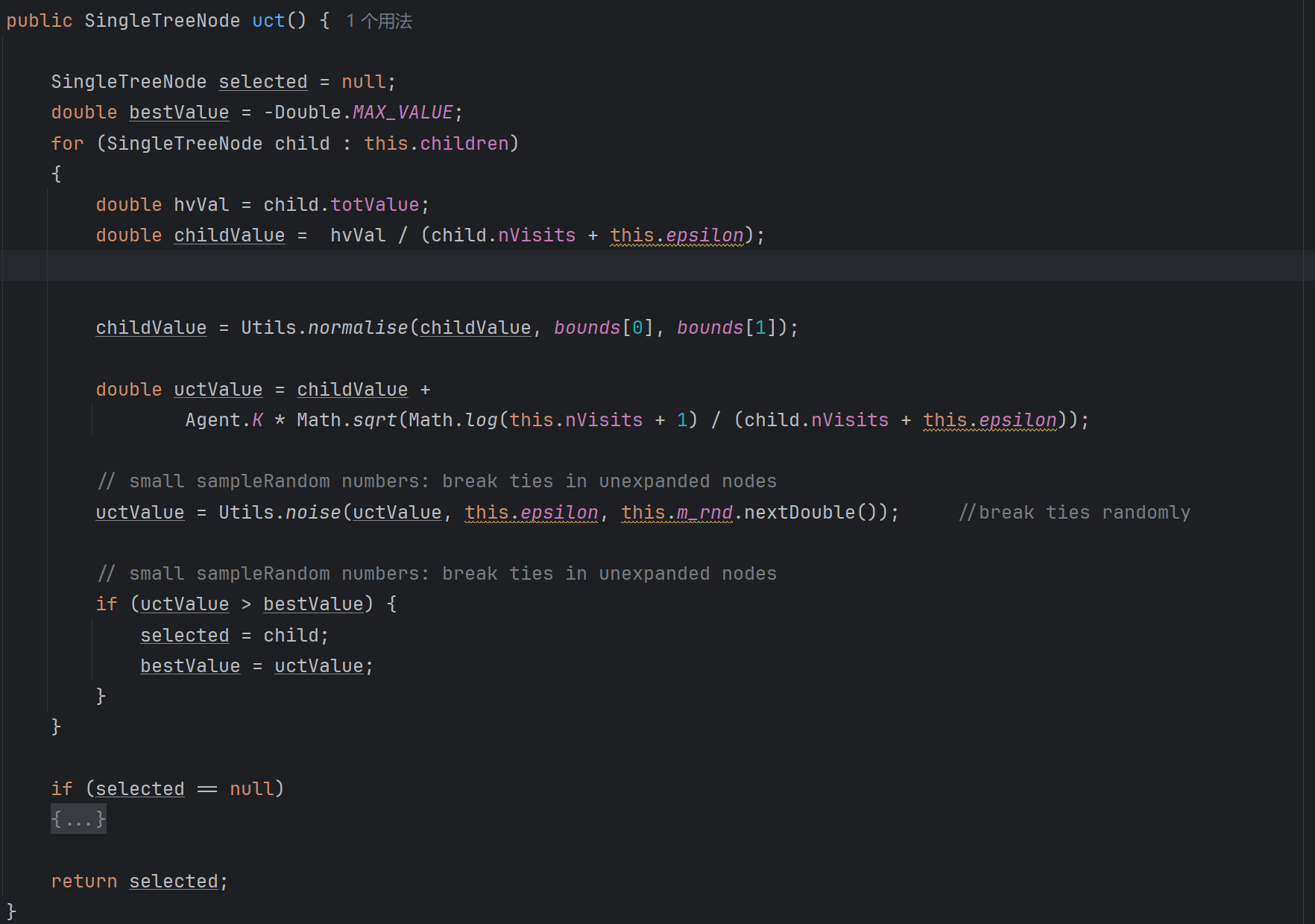
#### 选择（Selection）：从根节点开始，选择一个子节点，直到达到一个尚未完全展开的节点。选择的过程通常使用某种启发式方法，如上置信界（UCB）算法，以平衡探索和利用。

#### 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。

#### 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。

#### 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。

### UCB算法的应用：



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

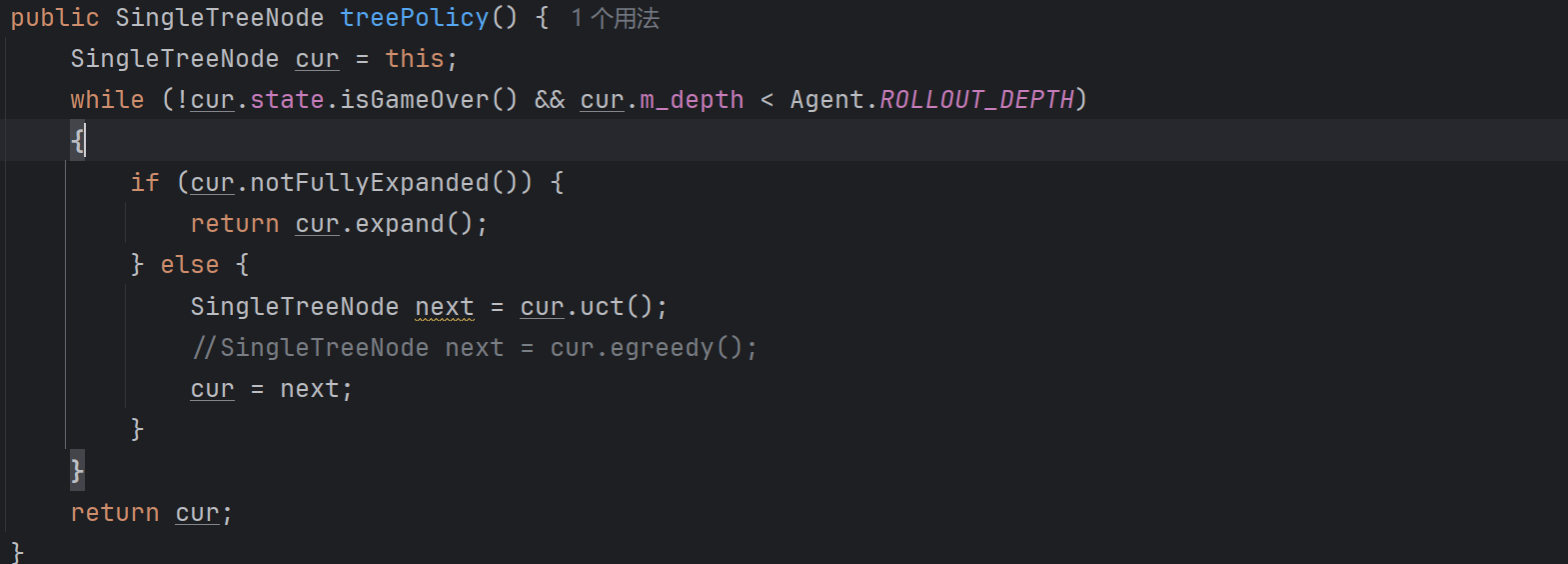


这里 totValue 是节点的总回报值，nVisits 是该节点的访问次数，K 是调节探索的常数，ε 是避免除以零的小常数。

### 扩展过程的处理：

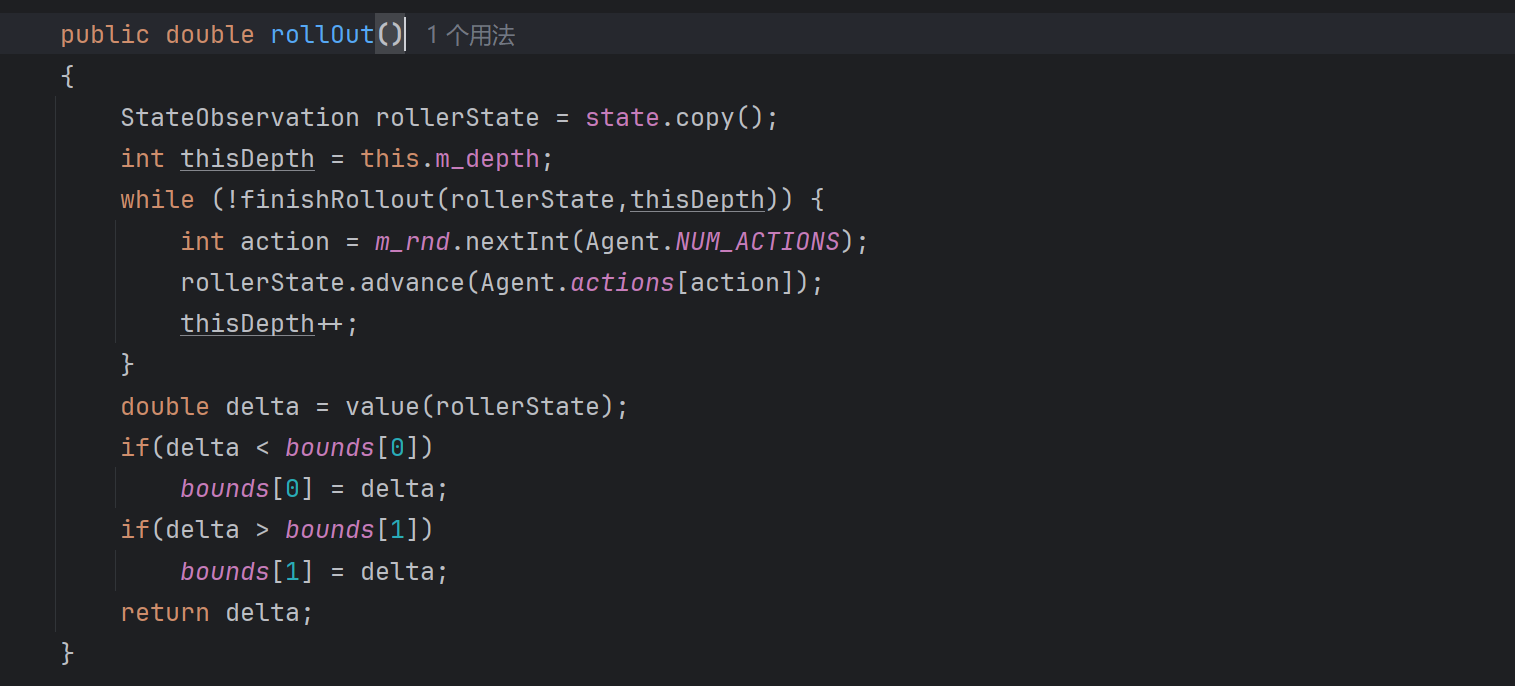


可以看到，该函数寻找一个子节点为null的节点，产生一个新状态，添加至子节点并将其返回。



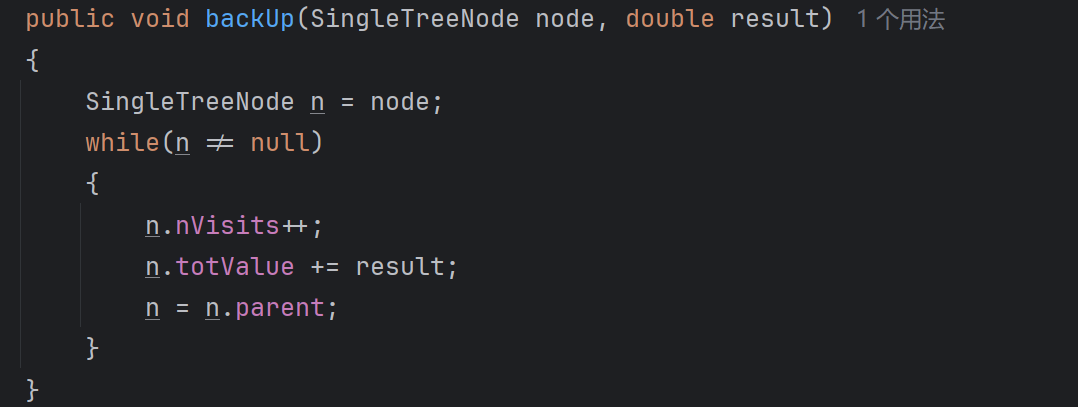
在这里调用了uct和expand函数，用于向树的叶子结点方向遍历直至寻找一个可以扩展的节点。

### 模拟过程：



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。

### 反向传播的实现：



简单增加访问次数和回报次数。

### 主体搜索过程：



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。