

作业1: Bait 游戏

张祎扬 (181840326 181840326@smail.nju.edu.cn)

(南京大学, 匡亚明学院, 南京, 210046)

摘要: 这是一个基于GVGAI游戏框架的编程项目, 主要有四个任务, 分别完成不同的搜索任务。

关键词: 深度优先搜索、深度受限搜索、A*搜索、蒙特卡洛树搜索

1. 引言

在解决搜索问题时, 我们常用几种搜索策略。其中, 深度优先搜索是一种纵向搜索, 它的空间复杂度低, 时间复杂度高; 而深度受限的深度优先搜索则在这一基础上对时间和空间的开销进行了平衡。A*算法看起来要更优秀一些, 设计启发式函数是这其中的关键。蒙特卡洛树搜索则加入了概率, 在大量搜索中平衡均值和不确定性。本次实验就是基于这几种树搜索方法的探索。

2. 实验内容

2.1 任务1

2.1.1 任务内容: 针对第一个关卡, 实现深度优先搜索。

- 在游戏一开始就使用深度优先搜索找到成功的路径通关, 记录下路径, 并在之后每一步按照路径执行动作。
- 在搜索时避免回路, 使用`StateObservation`类的`equalPosition`方法判断状态是否相等。
- 通过`CompetitionParameters.ACTION_TIME`来设置足够的时间来允许搜索。

2.2.2 实验过程及代码实现

首先我观察了作业提供的样例, `sampleRandom.Agent`, 发现它使用了比较重要的类 `StateObservation` 和 `Observation`, 于是我先观察了这两个文件中都有哪些方法。在这一过程中我参考了一个介绍GVG-AI框架的网站[1], 这个网站介绍了一些常用的方法和提供的API接口, 以及如何创建一个新的Agent。

Creating a Controller

You can create a controller for the GVG-AI competition just by creating a Java class that inherits from `core.player.AbstractPlayer.java`. This class must be named `Agent.java` and **its package must be the same as the username** you used to register to this website (this is in order to allow the server to run your controller when you submit). This class must implement two public methods:

- **`public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer)`:** the constructor of the class.
- **`public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer)`:** the function called every game cycle to retrieve an action from the controller.

从这个介绍可以看出, 新的Agent必须包括以上这两种方法, 同时Agent方法需要继承 `core.player.AbstractPlayer.java`. 然后我决定先大致按照 `sampleRandom.Agent` 搭好大致框架, 再修改其中的act方法。

关于代码中的一些变量创建，我有一些初步的想法。为了避免走回路，肯定需要判断走完某一步之后的状态是否重新出现过，这就需要有一个列表来存放所有已经搜索过的状态，还需要一个函数来判断状态是否出现过。另外，由于是深度优先搜索，直接搜索到成功的通关路径，再返回这个路径列表，所以需要有一个列表来存放路径，相应地要有一个整数用来指示列表的元素的下标。

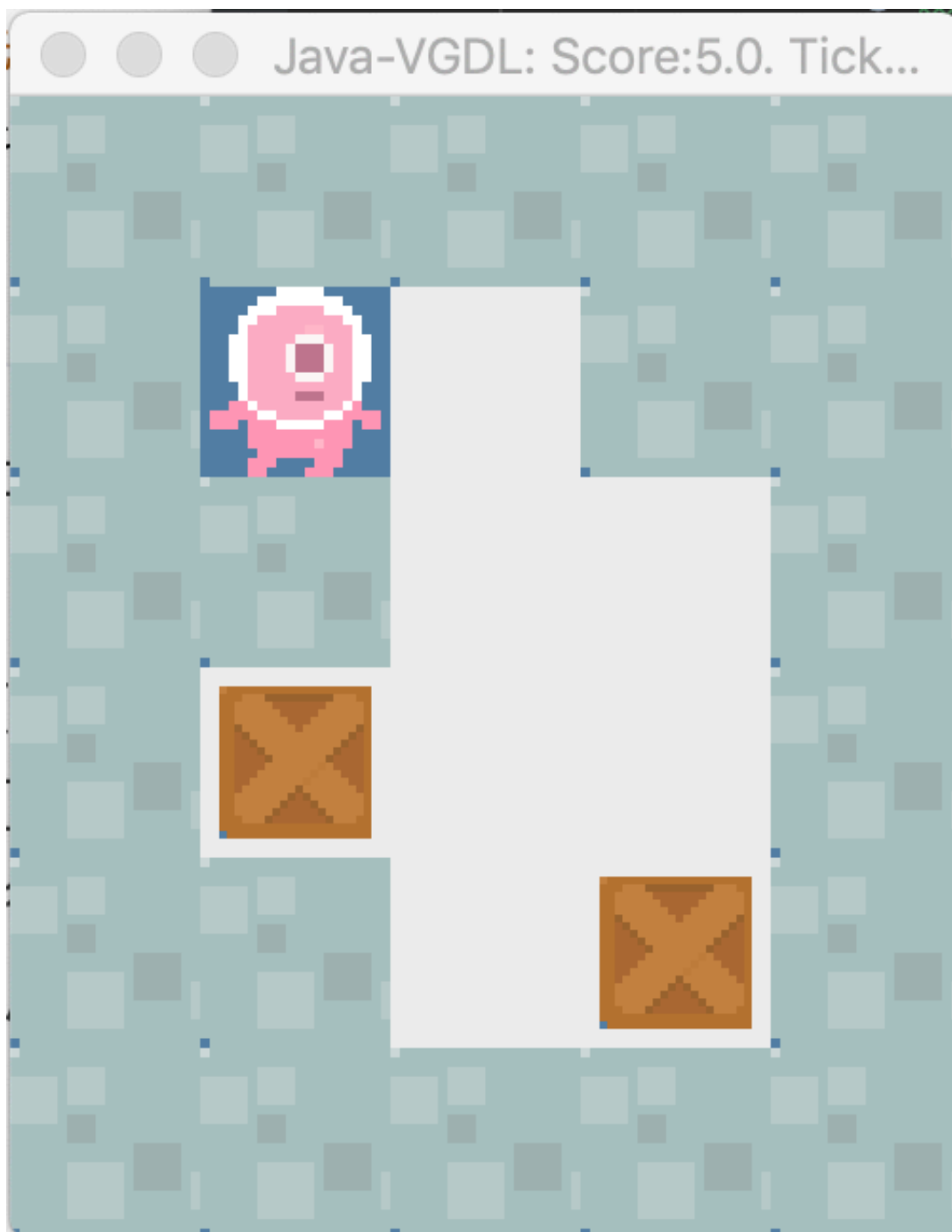
首先我在Agent类的开头对以上设想的几个变量进行了声明，然后在 `public Agent` 中对其进行了初始化。接着编写了一个函数来判断状态是否相等。判断的原理是遍历用来存放所有已经搜索过的状态的列表，使用StateObservation类的equalPosition方法判断状态是否相等，如果相等则返回true，如果遍历完列表后都没有返回true则返回false。

关于搜索函数的主体部分，我的想法是首先要有一个布尔变量来判断是否找到了成功通关的路径，再使用递归的方法，当没有找到成功的路径时，则继续调用自身。伪代码的设想如下：

```
1  boolean success = false;
2  private void depthfirst(StateObservation so){
3      if(success)return;//if the successful path is found then return
4      //if the new so has not been searched yet
5      add the new so to the searched list;//judge if the new so has been
        searched
6      if win{
7          //advanced judgement for the gameover state
8          success = true;
9          return;
10     }
11     //if not win then search again
12     for(action in available actions){
13         add new action to searched path;
14         copy the state and advance action;
15         if the advance state has not been searched yet//if the next state is
        new then search again else quit the search
16         depthfirst();
17         if(!success)//后进先出
18         remove the latest action in the list
19     }
20 }
```

在写主体代码时我基本参考了以上设想的框架完成了 `depthfirst()` 函数，然后在最终的 `act()` 中调用了这个函数。

一点bug：因为深度优先搜索的原则是后进先出，所以需要弹出列表的最后一个元素。本来我类比了python，以为index可以直接取-1，后来发生了一点报错。然后我参考了一些java教程，取列表最后一个元素的代码是 `list.get(list.size()-1)`。运行之后第一关的游戏界面快速闪过，然后发生了报错，观察报错，发现是后面的任务的Agent没有找到。于是我把后面的任务先注释掉，发现Agent成功地通关了。最后的结果如下：



2.2 任务2

2.2.1 任务内容 实现深度受限的深度优先搜索

- 修改为每一步进行一次深度搜索。
- 不需要搜索到通关，而是到一定深度，再设计一个启发式函数判断局面好坏。
- 设置时间较小，需要在设置时间内完成一次决策。

2.2.2 实验过程及代码实现

在这个任务中，有一些地方和以上是相同的，虽然是深度受限的搜索，但是仍然是深度优先的，只不过需要加上一个深度的限制，就是对搜索次数的限制。同样地需要设计函数判断当前的状态是否被重复搜索过。


```

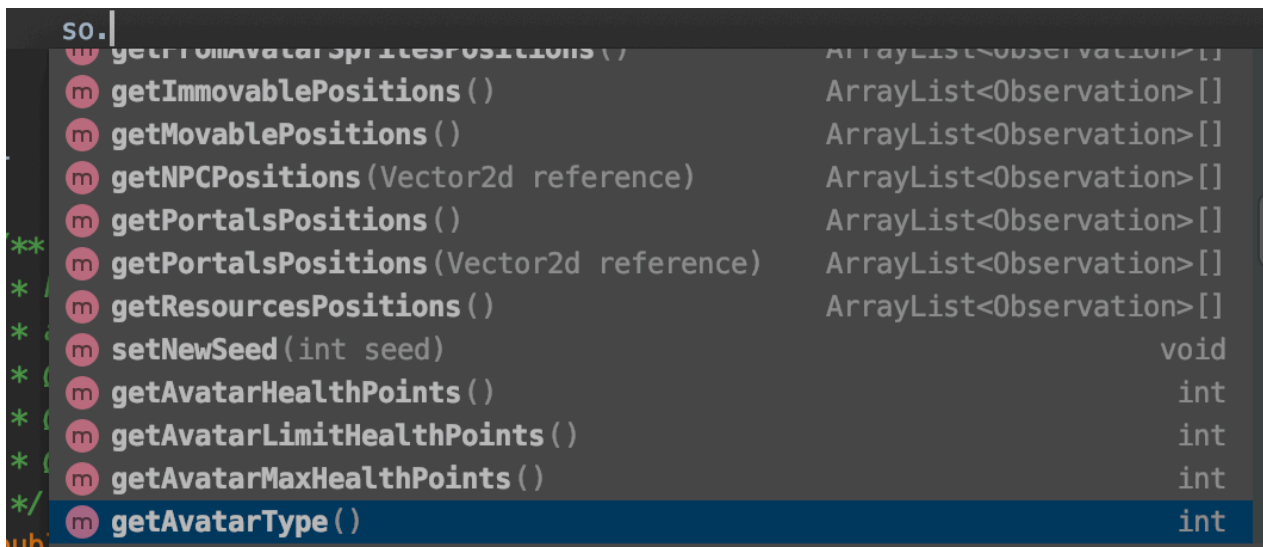
1 private double countHeuristicValue(StateObservation so)
2 {
3     a series of initialization;
4     if the agent has key
5         value = the distance between the agent and the goal;
6     else
7         value = the distance between the agent and the goal+the distance
            between the agent and the key;
8     return value;
9 }

```

在真正开始实现之后我又发现了一个问题。深度优先搜索是直接搜索到解，然后返回所有的动作，再一次执行完毕。而深度受限的深度优先搜索不能保证总能搜索到通关，所以它并不能返回一系列完整的动作，它只能保证每一次选择的动作是基于启发式函数来说最优的动作，并且只返回一个动作，所以我不仅需要有一个列表来记录所有搜索过的状态，还需要有一个列表来记录真实返回过的动作产生的状态，同时在每次返回一个新动作的时候，清空用来记录搜索过的状态的列表。

最后在根据以上设想写代码时，我完全复制了任务一中的代码，并且在它的基础上进行修改。

一点问题：当要解决判断精灵是否拥有钥匙的问题时，我发现StateObservation类有这样一个方法 `getAvatarType()`，它返回一个int值，但是我参考了doc文件夹中对该类的说明（一个html文件），发现这里面并没有对这个方法的介绍，我在该类的源码里也没有发现我想要的说明。



```

so.|
m getFromAvatarSpritesPositions() ArrayList<Observation>[]
m getImmovablePositions() ArrayList<Observation>[]
m getMovablePositions() ArrayList<Observation>[]
m getNPCPositions(Vector2d reference) ArrayList<Observation>[]
m getPortalsPositions() ArrayList<Observation>[]
** m getPortalsPositions(Vector2d reference) ArrayList<Observation>[]
* /
* m getResourcesPositions() ArrayList<Observation>[]
* m setNewSeed(int seed) void
* { m getAvatarHealthPoints() int
* { m getAvatarLimitHealthPoints() int
* { m getAvatarMaxHealthPoints() int
*/ m getAvatarType() int
ub

```

然后我参考了GVGAI官网[3]的介绍，发现还是没有这个方法.....

Information about the state of the Avatar:

Some functions can be used to retrieve information of the current state of the avatar. StateObservation provides information about the the avatar (position, resources, speed orientation). See here the most important functions:

State Observation functions: (StateObservation class), querying the Avatar.	
ArrayList<Types.ACTIONS> getAvailableActions()	Returns the actions that are available in this game for the avatar.
Vector2d getAvatarPosition()	Returns the position of the avatar. If the game is finished, this returns Types.NIL.
double getAvatarSpeed()	Returns the speed of the avatar. If the game is finished, this returns 0.
Vector2d getAvatarOrientation()	Returns the orientation of the avatar. If the game is finished, this returns Types.NIL.
HashMap<Integer, Integer> getAvatarResources()	Returns the resources in the avatar's possession. As there can be resources of different nature, each entry is a key-value pair where the key is the resource ID, and the value is the amount of that resource type owned. It should be assumed that there might be other resources available in the game, but the avatar could have none of them. If the avatar has no resources, an empty HashMap is returned.

后来经过我不懈努力的尝试，假如精灵还没拥有钥匙，那么这个方法的返回值就是1。

在大致完成代码时，运行却遇到了一个报错（其实IDEA在编辑代码的时候就有报错了）

```
Vector2d goalpos = fixedPositions[1].get(0).position; //the position of the goal
Vector2d keypos = movingPositions[0].get(0).position; //the position of the key
```

这我就很不得其解了，讲义给的代码为什么会报错！！！结果原来是上面的Arraylist并没有用简括号声明类型.....改完之后整个世界都美好了。

一开始我把LIMIT设置为10，我发现avatar在拿到钥匙之后，在两个箱子周围疯狂徘徊很久，最后回到了目标。然后我一路下调LIMIT，发现情况依然没有什么变化，虽然它像一个智障一样疯狂徘徊，但在经过一段时间后还是可以回到终点的。但是当LIMIT小于等于5以后它就开始疯狂报错了.....

这里我有两个问题想解决，一个是它为什么报错，还有一个是它为什么在两个位置之间疯狂徘徊。但是时间有限，我作业要来不及了，所以有缘再继续吧.....

2.3 任务3

2.3.1 任务内容 A*算法

- 在任务2的基础上，将深度优先搜索换成A*算法
- 尝试在第二关、第三关中使用A*算法

2.3.2 实验过程及代码实现

根据书[4]上的定义，A*搜索对结点的评估结合了 $g(n)$ ，即到达此结点已经花费的代价，和 $h(n)$ ，从该结点到目标结点所花代价：

$$f(n) = g(n) + h(n)$$

由于 $g(n)$ 是从开始结点到结点 n 的路径代价，而 $h(n)$ 是从结点 n 到目标结点到最小代价路径的估计值，因此

$$f(n) = \text{经过结点 } n \text{ 的最小代价解}$$

所以其实在主函数上，与之前的深度优先搜索差别不大，主要的改动在于启发函数的计算方式改变了。在这一任务中，对每个节点的信息记录更为复杂，所以我决定编写一个 `MyNode` 类，具体参考了任务4中的 `SimpleTreeNode` 类，这样可以方便地查找父结点、子结点，同时记录深度。对于启发式函数的设计，我的想法是，把结点的深度作为到达此结点已经花费的代价，而之前所用的曼哈顿距离是该结点到目标结点话费的最小代价。由此就得到了启发式函数。

在写 `MyNode` 类的时候，我实现了以下功能：`expand()` 用来对某一个结点进行扩展，添加它的子结点。`find_mother()` 用于找到某一个子结点的根结点，这在返回最后的动作的时候非常有用。

原先我还模仿蒙特卡洛树的Agent，实现了一个 `AstarPlayer` 类，后来由于debug耗时巨大，也找不到问题，所以我进行了简化，也把之前放在 `MyNode` 类的Astar搜索方法直接放进了act函数中。

由于搜索时间有限制，所以限制时间的代码和 `SampleRandom` 基本类似，当剩余时间大于一次搜索平均时间的2倍，并且大于最低限制的时候，不断进入循环。在进入循环之前先进行初始化，用当前的状态初始化一个结点，并把它作为根结点，即搜索的开始。然后还有一个列表 `active_nodes` 用于存放当前的活跃结点，即所有边缘的叶结点。在主循环中主要完成以下任务：遍历所有叶结点，找到总代价最小的最优叶结点，然后调用 `expand()` 方法扩展该结点，并将新扩展的子结点添加到叶结点列表中，再将之前找到的最优结点从列表中移除。这样始终保持列表中的是所有最边缘的叶结点。随着搜索的不断推进，在时间耗尽前，搜索的范围将不断扩大。

搜索结束后，遍历所有叶结点，找到代价最小的那个结点，用 `find_mother()` 方法找到它的根结点，从而可以返回一个相应的动作。

很遗憾，在这个任务上我耗费了大量的时间和精力，一遍一遍地确证我的代码的思路，一遍一遍的调试。我想也许是我本身对JAVA语法不够熟悉，因为毕竟是直接上手，很多概念虽然在使用但其实并不清楚。到目前为止，我还是没有通过调试让这个程序跑起来，也就无从验证我写的A*算法的正确性，所以我在以上对我的思路进行了阐释。

我想说的是，实验过程中我确实会遇到很多意想不到的困难。有的bug很低级，但是却让我寻找了很久，但也有的bug的出现真正是因为我的理解有偏差。在完成的任务3的过程中，我经历了数次长达6至8小时的折磨，代码也被一改再改，不断注释，又不断删除注释。在这个过程中，除了意识到自己的菜以外，更多的是学会了保持平和的心态，和无论怎么样都保持耐心的品质。

即便如此，对接下来的每一个任务，我还是充满了信心。

2.4 任务4

经过这周四新课的启发，我知道了这个是蒙特卡洛树搜索。

首先在一开始定义了几个变量。`NUM_ACTIONS` 表示子结点的数量，`ROLLOUT_DEPTH` 是rollout环节完全随机搜索的深度。K是根号2，`actions` 是一个动作数组。

首先阅读SingleTreeNode.java

`SingleTreeNode`中有一些元素，`state` 是当前状态，`parent` 是父结点，`children` 是子结点数组，数组大小为 `NUM_ACTIONS`，`m_rnd` 是随机数，`m_depth` 是当前深度。

`uct()` 方法：返回一个树结点。首先把 `bestValue` 设为最小负数，遍历树结点的每一个子结点，设置它们的 `childValue` 和 `uctValue`。其中 `childValue` 就相当于当前的均值 $Q(k)$ ，它是用 `totValue` 除以访问次数，分母上的 `epsilon` 用来处理次数为0的情况。

$$uctValue = Q(k) + \sqrt{\frac{2\ln n}{n_k + \epsilon}}$$

从代码实现可以看出， n 是总次数， n_k 是子结点的访问次数， ϵ 用来控制计算情况，所以 `uctValue` 的两项分别是均值和不确定性，当搜索的总次数越多，均值越有说服力，不确定性越低。该方法的最后比较了当前的 `uctValue` 和 `bestValue`，并且更新 `bestValue` 和 `selected`，返回 `uctValue` 最大的子结点。

`expand()` 方法用于扩展一个子结点。

`notFullyExpanded()` 方法用于判断所有子结点是否都扩展完全。

`treePolicy()` 方法：当游戏没有结束，并且rollout深度没有达到最大限制的时候进入循环。在循环内，如果子结点没有被完全扩展，就继续扩展子结点，否则的话就调用 `uct()` 方法，返回 `uctValue` 最大的子结点。

`finishRollout()` 方法：如果当前深度大于等于限制，则rollout结束。如果游戏结束，则rollout也结束。

`value()` 方法：如果游戏结束，player输了，用当前得分减去 `HUGE_NEGATIVE`；如果player赢了，用当前得分加上 `HUGE_POSITIVE` 作为 `rawScore`。

`rollOut()` 方法：如果rollout没有结束(用 `finishRollout()` 方法判断)，随机选择一个动作并更新当前状态。并令delta为当前状态的rawscore（不超过最大最小double数的限制），返回delta。

`backUp()` 方法：从最后一个子结点开始往回回溯并更新 `totValue` 值和visit的次数。

`mostVisitedAction()` 方法返回访问最多次的结点的索引。

`mctSearch()` 方法的参数是 `ElapsedCpuTimer elapsedTimer`，里面设置了每一次搜索的平均时间，花费的总时间，剩下的时间，和迭代次数。当剩余的时间大于两倍的搜索平均时间并且剩余时间大于限制的时候，进入搜索循环。在搜索循环中，调用 `treePolicy()` 方法选中一个子结点，再从这个子结点往下rollout，返回delta值，最后调用 `backUp()` 方法从最后一个结点往上更新这条路径上所有的结点的参数。这就完成了一次蒙特卡洛树搜索，即bandit和rollout的结合。

接着阅读SingleMCTSPPlayer.java

首先设置了根结点 `m_root` 和随机数 `m_rnd`，`init()` 方法对其初始化。`run()` 方法中首先对根结点调用 `mctSearch()` 方法，然后调用 `mostVisited()` 方法，并返回action值（索引）。

最后阅读Agent.java

在 `Agent()` 类中，对一系列动作数组和数组的长度进行了初始化。在act函数中，首先初始化了 `mctsPlayer`，然后调用run方法返回一个最佳action的索引，最后返回动作列表中该索引的动作。

总结：蒙特卡洛树算法主要由两部分组成，第一部分是用bandit选择一次子结点（概率），第二部分是从这个选择的子结点随机rollout下去，并不断更新路径上每一个结点的信息。根据公式

$$value = Q(k) + \sqrt{\frac{2\ln n}{n_k}}$$

可以求的均值和不确定性之和（这其中又加入了噪声来平衡数据），最终经过大量搜索之后，选择最佳的动作。

3.结束语

本文详细记录了我在完成任务一过程中的代码思路 and 心路历程，无论什么时候回头看，它都将是一笔宝贵的财富。

致谢：

感谢一些同学在我因作业进展不顺利时给予我的鼓励和安慰，让我一直坚持尽自己最大的努力完成。

References:

[1][<http://www.gvgai.net>](<http://www.gvgai.net/>)

[2]《人工智能：一种现代的方法》3.5 有信息（启发式）的搜索策略

[3]<http://gvgai.net/forwardModel.php>

[4]《人工智能：一种现代的方法》3.5.2 A*搜索：缩小总评估代价