Bait实验报告

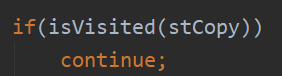
宋磊 181220049 [974534426@qq.com](mailto:974534426@qq.com)

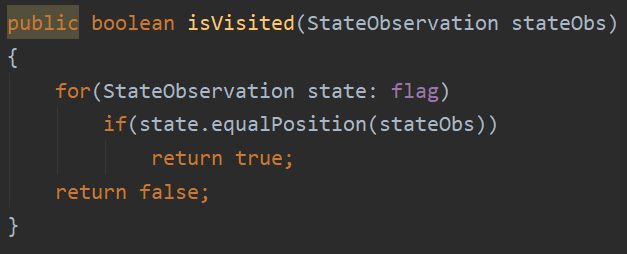
（南京大学 人工智能学院）

1. DepthFirst
   1. 思路

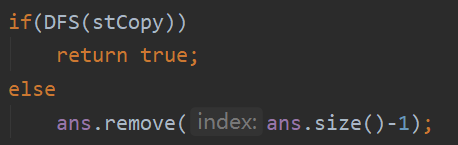
在当前节点通过枚举尝试每个动作，找到解路径，之后记录下来，之后的每次act都按照顺序选择动作并执行，最终完成任务。

如图是对相同状态的判重，避免形成环：





如图维护了一个ArrayList类型的ans解路径，如果从当前节点可以找到路径，则直接返回，否则remove该节点的路径，最后只需要按照ans的顺序执行Action。



1.2 代码结果

代码可以完成前三关，但由于是在使用深度优先搜索找到路径后直接返回，不是对树的完全搜索，找到的不一定是最优路径，例如：在第一关拿到钥匙返回目标时会绕远路，在第二关中，由于箱子位置改变，也会导致state的变化，使得state.equalPosition()方法的到false，所以会走很多在人看起来重复的路，第三关较为良好。

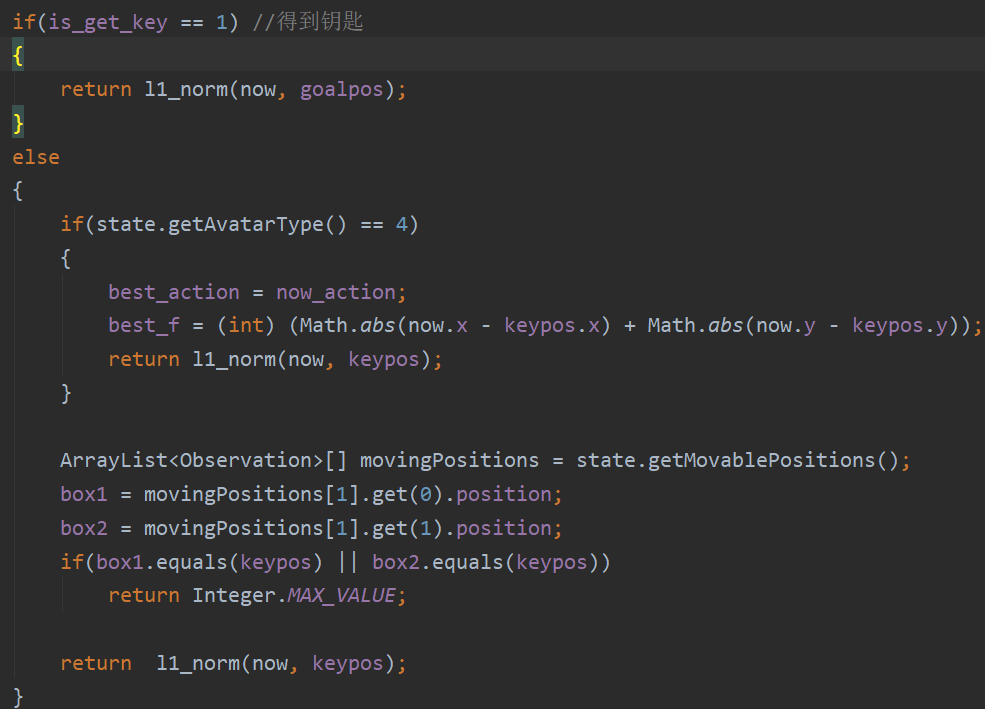
1. limitDepthfirst
   1. 代码解释

2.1.1 整体思路

在第一问的基础上，增加深度的参数，每次递归层数增加时，对应增加深度，达到深度时停止扩展并通过启发式函数判断是否优于当前最优解，并进行更新。与第一问的不同是在每步都进行一次深度受限搜索，得到当前最优Action后执行，下一步重新搜索。

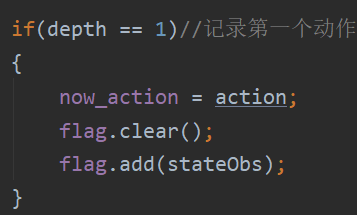
2.1.2 启发式函数

对启发式函数在下面的Astar算法3.2启发式函数设计中进行详细讨论。这里只进行简单分析。通过官网相关资料，发现可以通过state.getAvatarType()判断精灵是否拿到钥匙，简单通过这个函数进行判断，分为三种情况，已经拿到钥匙，对应第一个分支，启发式函数变为当前位置到目标的l1距离。未拿到钥匙但已经找到了通往钥匙的道路，对应第二个分支，则直接向钥匙前进，启发式函数为现在位置到钥匙的距离，未拿到钥匙切未找到通往钥匙的路，对应最后一部分，此时需要判断不会让箱子压到钥匙。

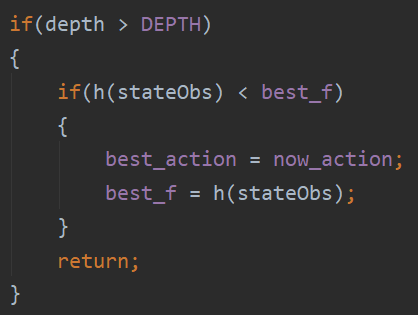


2.1.3 最优动作的选择

对当前最优动作的记录，由于是树状搜索，所以只需要在深度为1时记录now\_action，之后如果在深度较深时发现更优解，一定是由now\_action扩展得到，则将best\_action更新为now\_action，在搜索完成之后执行best\_action。



如图是相比于第一问增加的对深度的判断，并更新当前最优解。



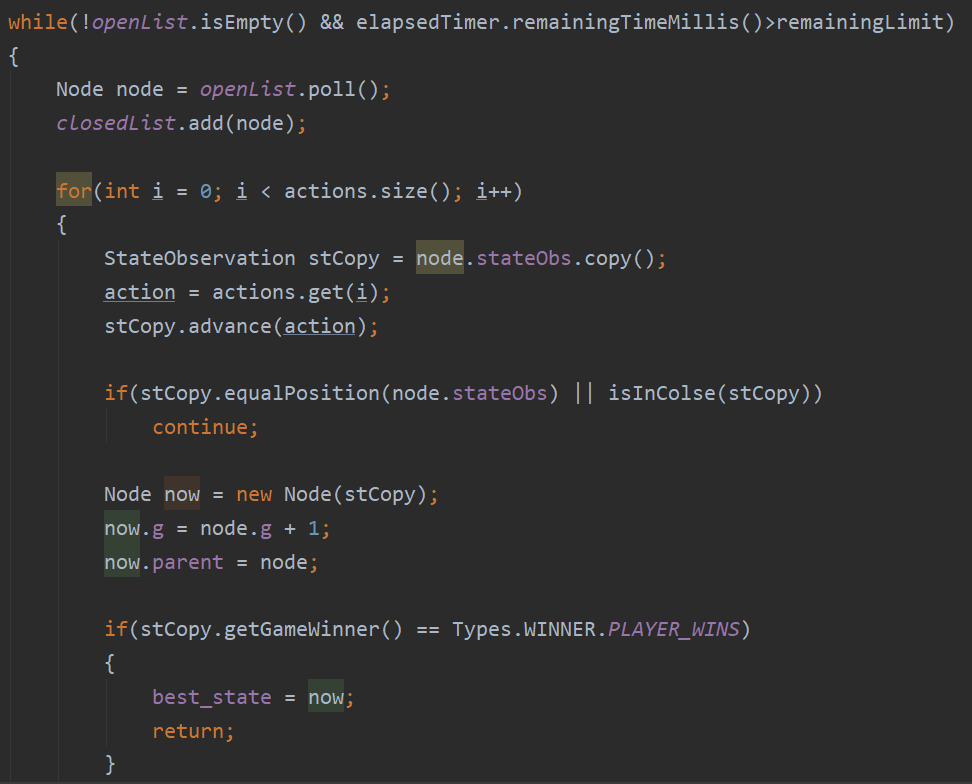
* 1. 代码结果

在depth适当比如3时，通过第一关，但在第二关中需要加深深度，分析认为，在深度为3时，做出的Action不会对局面造成影响（这里说的是向钥匙的靠近，因为没有考虑箱子和洞的位置关系），所以需要对启发式函数进行改良，具体改良见Astar中的启发式函数。

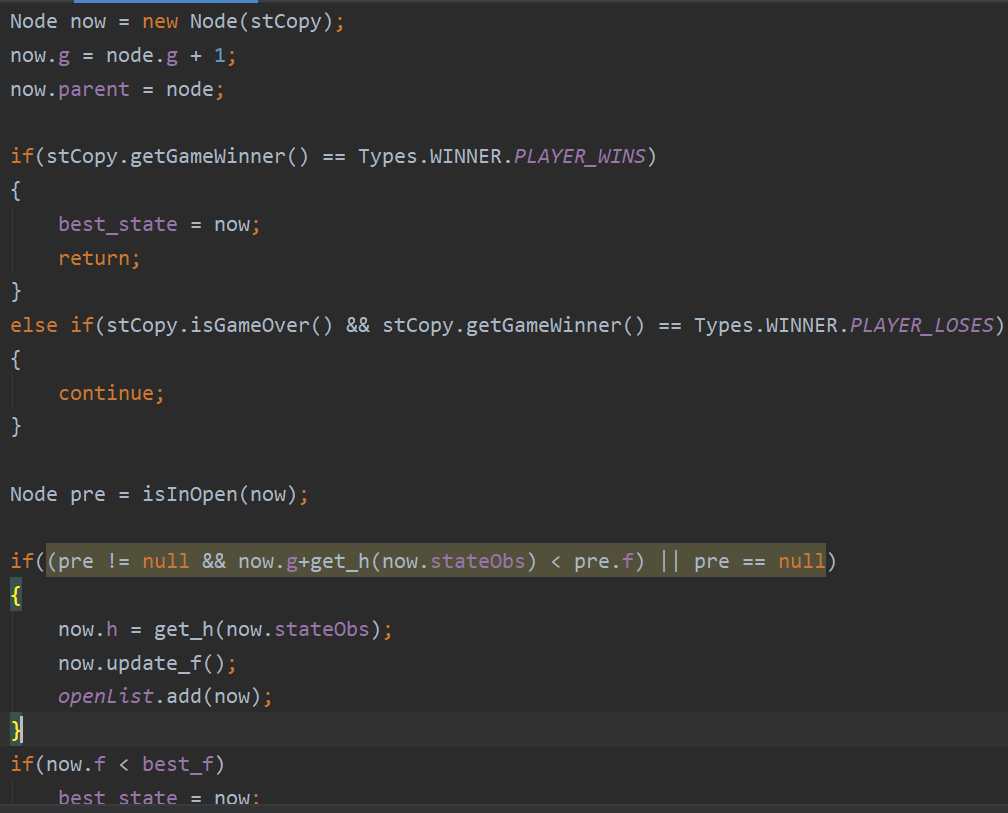
1. Astar
   1. 思路

维护openList和closedList的优先队列，在循环寻找路径时，将openList中优先级最高的元素取出，移入closedList中，表示该点已经访问过，对该点可以执行的动作进行执行，得到新的状态，如果新的状态没有访问（即不在closedList中）并且不在openList中，或是在openList中但代价g值更小，则将代价更小的状态加入openList中，并通过java的priority\_queue自动维护堆性质。

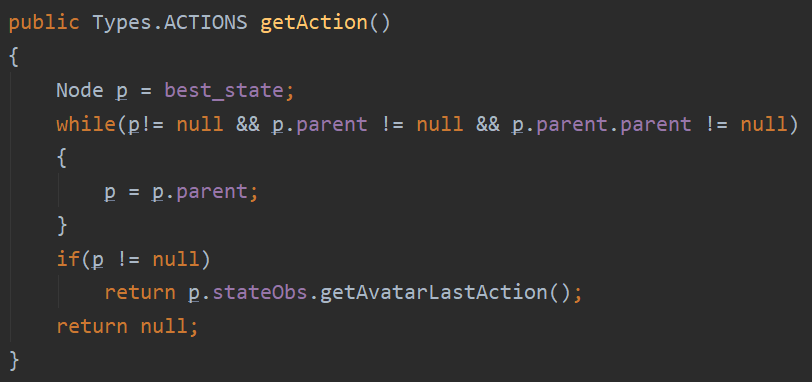
通过不断访问当前最优的状态，得到最优路径。代码的前部为取出最优状态并进行扩展，如图



之后对新状态进行合法性检验，如果位置没有发生变化，或是精灵已经死亡，或该状态已经访问过，则直接退出该次动作，不将其加入openList中。如果是一个合法的状态，并且不在openList中或比原本openList中的同一状态更优，则对openList中状态更新为更优解，如图：



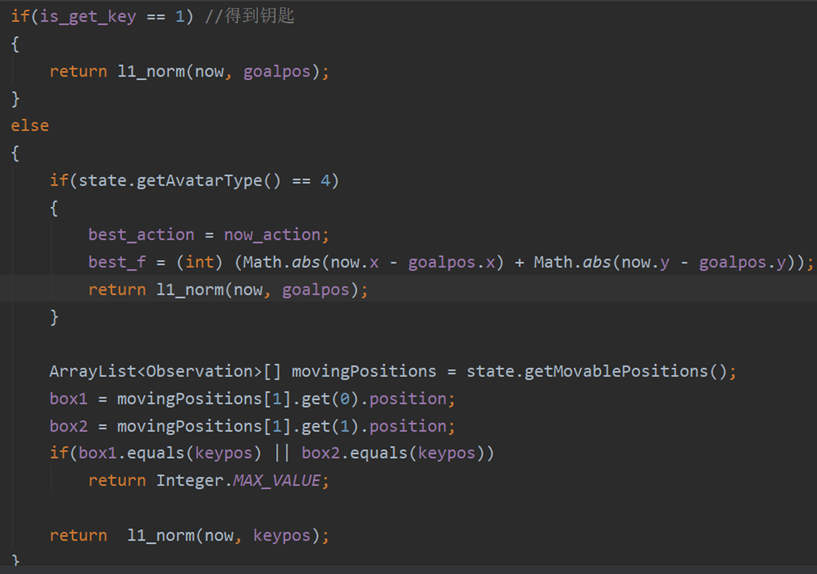
通过后一个动作找到最开始执行的动作的代码对网上进行了参考，与之前同样的思路，动作的执行扩展为一棵四叉树，通过记录其父节点parent，进行回溯，找到最开始执行的动作，并执行。代码中是找到了执行完第一个动作后的状态，通过getAvatarLastAction()函数找到上一个动作，并返回（所以在循环判断条件中要保证p.parent.parent不为空）。



* 1. 启发式函数设计

3.2.1 第一关

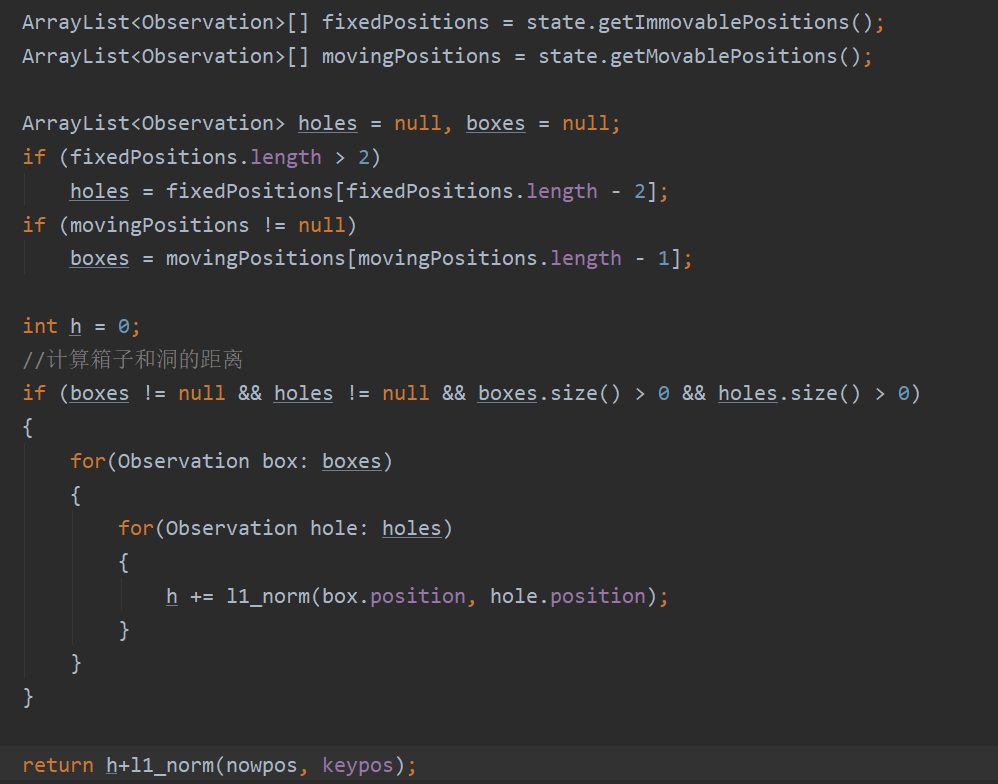
最初设计的启发式函数时最简单的，先分情况讨论是否拿到钥匙，以及箱子不能压到钥匙的简单判断条件，再用l1范数计算距离，但之后发现这样简单的函数只能通过第一关，随后进行了改良。



3.2.2 第二关

第二关过不去的主要原因认为是在之前的函数中，只计算了自身和钥匙的距离，所以精灵只会向钥匙靠近，而不知道将箱子推到洞上，这样肯定是不会通关的，所以加入了钥匙和箱子的相对位置的l1范数，将箱子向洞靠近，也可以得到更小的h，从而让精灵知道用箱子填洞。

代码前半段是通过fixedPosition和movingPosition获得箱子和洞的位置，后面通过二重循环计算每一个箱子和洞的l1范数，记录为h，最后的返回结果中加h，从而使得降低箱子和洞的距离也可以获得更好地评分。



3.2.3 第三关

但这样的代码依然是有问题的，在第二关中表现良好，路径也是最优，但第三关中，精灵总是希望把所有箱子都推到洞上，但其实只需要将左边的推上就可以吃到钥匙和蘑菇，进而完成任务。

第三问遇到的问题是精灵填完左边的洞之后，又走到右边希望填右边的洞，但找不到合适路径，会出现在右边震荡的情况。所以希望限制得分，在得到一定分数时降低h值，进而使其在得到一定分数时会去找钥匙而不是通过填洞获得更低的h，所以在返回值中加入了getGameScore()\*，其中为比例系数，为分数项在总h中占的比重，经过预测在0-200之间，并通过二分法尝试发现取为50时能够比较稳定通过第三关。之所以是比较稳定，是因为在Astar算法中，我会在当前决策实践不足时即下图中后半部分不满足时就跳出，这个会给算法带来一定的随机性，在极少数情况下，会寻路失败。



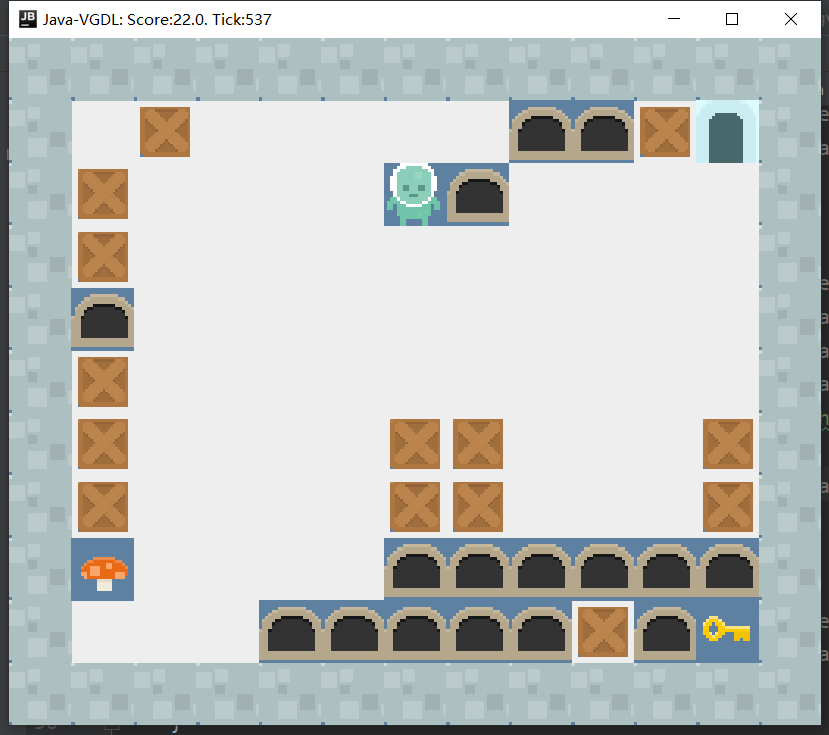
随后得到了第三关的表达式，如图：



3.2.3 第四关及第五关

这里最终只完成了第五关，第四关只能得到较高的分数，但无法完成。

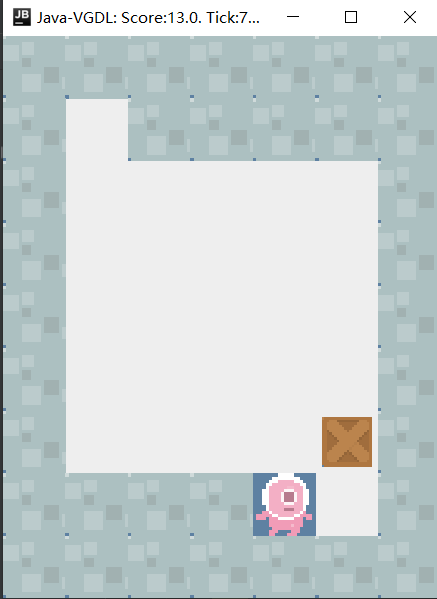
在第四关中，由于之前的启发式函数中计算了每一个箱子和每一个洞的距离，第四关中，精灵会疯狂填洞，如图所示：



分析认为是第三关中添加的出现了问题，在第四关精灵可以得到特别高的分数，得分较高时，该项可能为负，则一定是最好结果，所以精灵会疯狂填洞，在尝试改变比例系数后发现没有明显变化，所以希望在这里给精灵一个惩罚，在分数较高时，让h变高，即让精灵避免只填箱子的情况，所以将减号改为加号，即给了精灵一个类似惩罚项的东西，避免得到较高分数，最终表达式如图：



但发现效果不是很明显，反而是原本无法通过的第五关，在改变为该式并延长搜索时间后，可以通关第五关，并且是完美通关，十分神奇的拍脑袋启发式函数，如图：



* 1. 代码结果

最终通过同一个启发式函数稳定完成了前三关，在对一个正负号改变并延长搜索时间后可以稳定完成第五关，第四关只可以得到较高的分数。但由于在Astar中加入了对时间的判断，防止超时，所有代码有了一些随机性，有时可以最优路径完成，有时会卡在一个地方两三次，有时甚至在走出一步之后直接停止，但整体看结果还是十分稳定，通关情况也十分好。

4 sampleMCTS

4.1 变量分析

epsilon是一个很小的常数，使得在计算为访问过节点的值时，分母不会为0，而是epsilon，使得得到的Value极大，所以会对没有访问过的节点优先进行尝试。

egreedyEpsilon是在egreedy()中使用的常量，大于这个值时会选择当前最好的Action，小于这个值时会随机选择Action，采用的是一种贪婪的方法，但这个函数没有被调用，所以不做详细讨论。

State，parent，children是自身的状态，以及树的相关变量。

Totalvalue和nvisits是这个节点的当前的值和访问次数，用于计算后面的uctValue。

m\_rnd是随机数发生器，在计算uctValue时用于添加噪声，以及扩展节点时会使用。

m\_depth为当前节点的深度，在Agent文件中有一个ROLLOUT\_DEPTH，是最大深度，防止树无限向下延伸。

4.2 代码分析

4.2.1 主要思路分析

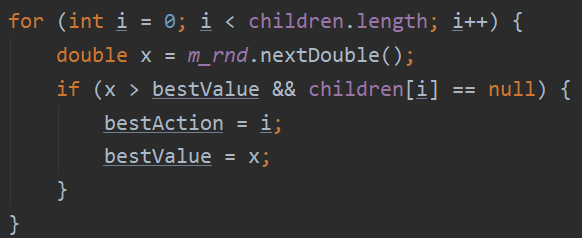
mctsSearch()是执行的核心，第一步是Selection，对应代码中的treePolicy()函数，就是在树中找到一个最好的节点进行探索，优先选择未探索过的节点，如果都探索过，则选择值最大的节点。第二步是Expansion，对应treePolicy()中调用的expand()函数，一般是随机执行一个未与之前子节点重复的操作，在代码中通过对每个节点生成一个随机数，选择随机数最大的节点进行执行来达到随机选择的目的。第三步是Simulation，对应Rollout()函数，就是在前面新Expansion出来的节点开始进行游戏，直到到达游戏结束状态，这样可以得到这个expansion出来的节点的得分是多少，Rollout()过程中采取的Action是随机的，从而实现快速达到游戏结束的目的。第四步是Backpropagation，对应backUp()函数，将新扩展的节点的得分进行回溯，即对所有祖先节点的得分和访问次数进行累加，更新totalvalue和nvisits，方便后面计算uctValue。执行完成后通过mostVisitedAction()函数执行act进行游戏。

重复上述操作，通过不断的模拟游戏得到大部分节点的uctValue，然后下次模拟的时候根据uctValue值对exploration和exploitation进行平衡，选择最好的节点，该算法适用于搜索空间巨大，不能通过简单的搜索枚举找到最优解的情况，MCTS在这种情况下能更大概率找到更好的Action。

4.2.2 函数分析

treePolicy()是选择Action的函数，循环的判断条件使得搜索的深度不会超过ROLLOUT\_DEPTH，不会在书中无限向下搜索，之后如果当前节点没有完全扩展，则进行扩展，否则沿树向下搜索直到找到一个未完全扩展的节点。

Expand()函数是对节点的扩展，对为扩展出的每一个节点生成一个随机数，挑取最大的一个作为下一个扩展的节点，达到随机选择扩展节点的目的，如图。

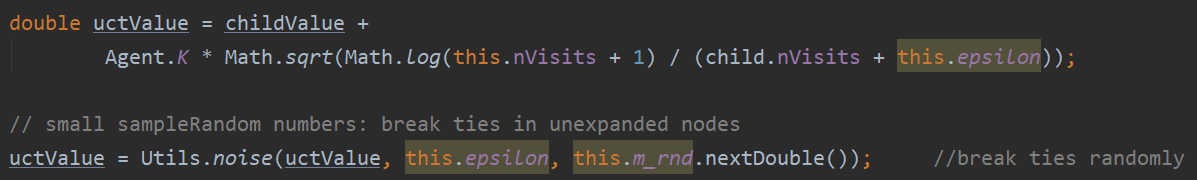


Uct()函数是计算uct值，核心代码如下图，公式如下

*)*

通过这个公式计算自身的值，childValue为之前模拟中的平均估值，后面一项使得节点在被访问次数较少时会优先访问，最后的为噪声。





Rollout()函数在随机选择Action模拟游戏，在很短的时间内结束游戏，得到预测值，并通过backUp()函数进行更新。如图为Rollout随机选择行动的函数。

