人工智能导论作业 1：bait 游戏

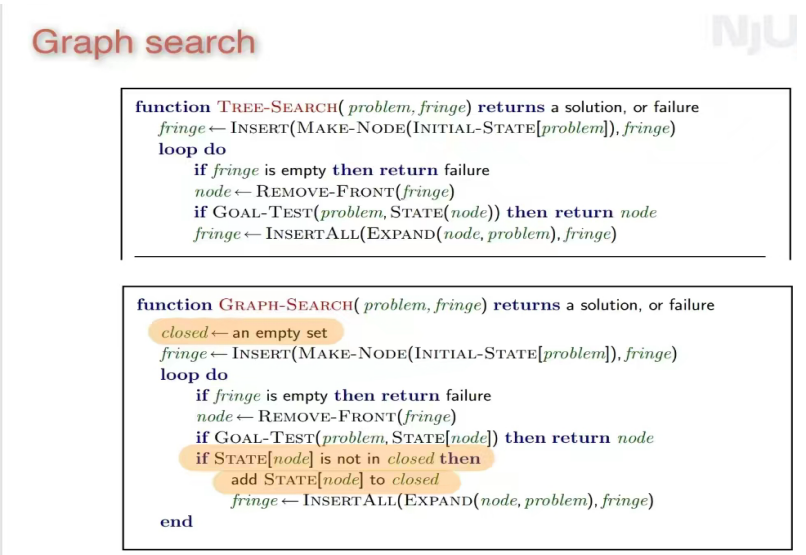
石睿 (211300024、211300024@smail.nju.edu.cn)

(南京大学 人工智能学院 南京 210093)

关键词: 深度优先搜索；深度受限搜索；A*算法；蒙特卡洛树搜索

中图法分类号: TP301 **文献标识码:** A

1 深度优先搜索

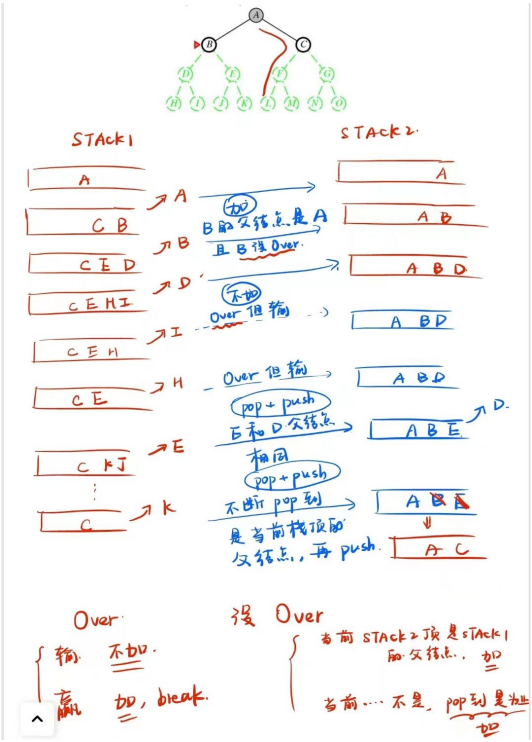


1.1 在 Node.java 文件中

树搜索的通用模板如上图所示，其中class node的方法、属性至关重要，本代码中将其抽象，实现在Node.java之中，其中封装了节点状态(Node_state)、父节点产生本节点的动作(Node_action，根节点是null)、父节点(Node_father)等数据。

同时的Node中实现了Cloneable接口并重写了关键clone()方法，因为在java中不显示提供copy\deepcopy等python中提供的拷贝函数，而在agent脑海中模拟程序运行的时候又不可以使用“=”进行对象的赋值（引

用）。所以为了agent脑海中模拟的时候不影响当前局面状态，使用clone()方法对当前节点进行克隆（浅拷贝）。因为当前Node中属性没有对其他类的引用，所以浅拷贝（(Node)super.clone()）就已经可以（不然的话还需要对Node中对象属性也调用clone函数、或者进行序列化implements Serializable呢！）



1.2 在Agent.java 文件中

程序首先进入主循环，即深度优先搜索的循环。其中每展开一个节点，搜索结果都存储在being_visit（正在搜索的节点） 这个栈中。在从栈中取出节点时，分别判断其是否为终止节点、或重复节点（和已经扩展的节点状态进行匹配，即集合been_visited），以此决定是否需要继续展开。在拿出current_father的过程中，对target_node进行管理，表示从根节点到当前节点所经过的节点们。这两个栈的具体管理如上图所示。

当搜索到游戏最终结束且玩家胜利时，程序跳出循环， 此时target_node栈底是根节点，栈顶是最终状态节点。不断把target_node出栈，把出栈的节点的状态压入sort_actions之中。当target_node栈是空的时候，sort_actions就是从根节点到最终状态节点的一系列动作啦！

可以优化的方面：在子节点加入的时候，并没有进行重复状态检查只在父节点拿出的时候进行了检查，（在A star算法中进行了优化）。

2 深度受限搜索

深度受限搜索要求每步都进行一次搜索

情况1：如果当前这次在depth_max之中没有搜索到最终节点，则利用启发式函数决定当前“根节点”往后一步的动作。

情况2：如果当前这次在depth_max之中搜索到了最终节点，则和深度优先一样，直接返回所有的sort_actions呢

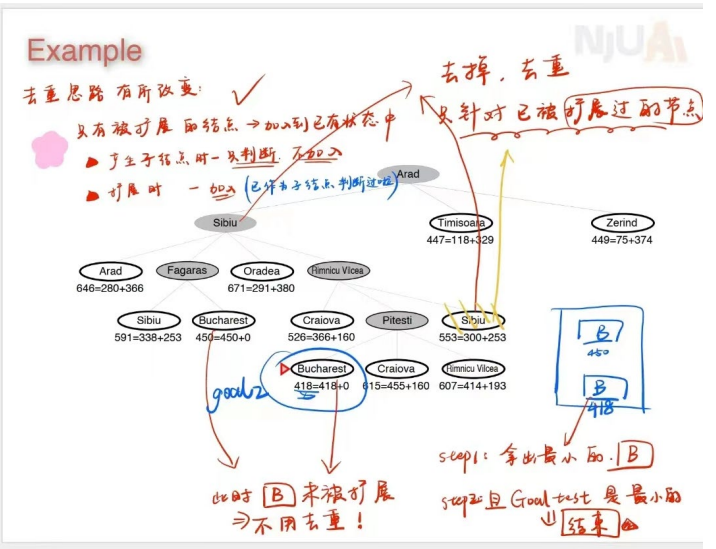
当每步搜索达到一定深度/消耗一定时间后，人为暂停当前搜索过程，根据启发函数选择最优状态，再根据每一节点的父节点一步一步倒退，直到取得最初的动作进行输出。同时，为判断当前节点是否超过了depth_max，主要在“节点”类中加入深度信息。

一些想法：

1、启发式函数在时间允许多次进行深度受限搜索的时候，每一次只返回一步的动作（尝试返回从根节点到当前“最优”节点的全部动作，失败告终）。因为heuristic_function毕竟不是全局最优的节点，只是相对来说和其他节点更优一点。最后的启发函数包括精灵与钥匙的距离，精灵是否有钥匙，精灵与目标的距离；

2、关于如何确定搜索的深度，没有固定的策略，只能当做“超参数”进行调节。本实验在调试的时候，因为减少调试的时间，把搜索深度设置成了3层，但一直找不到bug。最终直接增加搜索深度到10层就可以很快结束啦！【只有在搜索了一定情况之后，heuristic_function才能做出比较准确的一步判断！】

3 A* (A star）算法



A*搜索的公式为f(n)=h(n)+g(n)

其中g(n)：在游戏中为当前节点走过的距离，即从最初状态走到当前节点状态的总消耗(在Agent中实现的时候g(n)=past_cost(current_node)+node.father_Node.previous_cost)。h(n)为与深度受限搜索中类似的启发函数(h(n)=heuristic_function(stateObs,node))。

3.1 在Node文件中

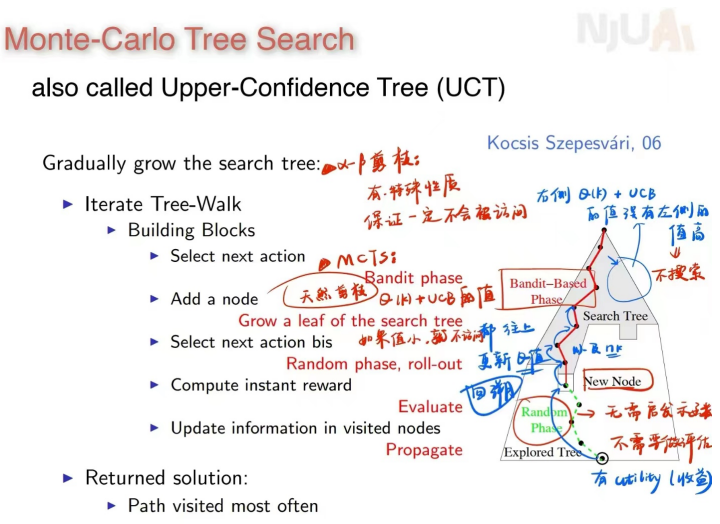
增加了 previous_cost 和 priority 两个属性，优先队列依据 priority 的值从小到大拿出优先级最高的节点，previous_cost 是给子节点计算的 g(n)值做准备呢

3.2 在Agent文件中

深度优先搜索中“栈”的数据结构相应地换成了优先队列(PriorityQueue)，每次将最优的节点（priority值最小的节点）出列。同时因为可能在跳转着搜索，所以之前的target_node保存从根节点到当前节点的方法不再使用。更换成拿到了“正确”的节点(yes_node)，可能是启发式函数提供的节点，也可能是最终玩家获胜状态的节点。从当前“根”节点开始，遍历所有的动作，直到找到当前节点呢【耗时过多】.故不需要维护target_node，只需要最终遍历得到sort_actions就可以啦！

理论上如果h(n)≤h*(n)，A*搜索便可以获得最优解。但实际中，由于启发函数的非最优性，A*搜索耗时过长，超过了每步的决策时限，只能模仿深度受限搜索，不是一次完成全部搜索而是每步进行搜索（如果延长决策时限，可实验验证A*算法的最优性）。

4 蒙特卡洛树搜索



蒙特卡洛树搜索是一类带有随机性的算法，主要思想是搜索一定深度，剩余的节点不再搜索，而是利用随机rollout 方法获得关于已搜索节点的信息并更新原有信息（回溯）。

框架代码给出的实现中，每个timestep 给出动作的关键函数是mctsPlayer.run()，其中包括对节点的蒙特卡洛搜索以及最优节点动作的选取。而root.mctsSearch 又包含以下方法：1.利用treePolicy()的节点展开；2.利用rollOut()的随机采样；3.利用backUp()的反向回溯更新。此外，程序利用了elapsedTimer 进行计时，保证有足够的事件进行一次完整的搜索。

框架中root.mctsSearch 中的三个关键函数的功能如下：

1.treePolicy().函数首先判断当前节点有无未测试的节点动作，如有，依次测试之；若无，则利用uct（置信度上界）规则在所有未选择的节点中挑选(根据uct值)最优的。同时代码中还对 uctValue 加入噪声扰动，防止出现相同uct值。此外，框架代码还提供了epsilon-greedy 的节点选择方法。

2.rollOut() 利用完全随机的策略快速到达游戏的结束，并记录下结束状态的value 值。

3.backUp() 函数即根据此前记录下的value 值依次更新父节点的值信息，以备下一次选择。