
人工智能导论:作业一实验报告

薛正海(181220063、xuezh@smail.nju.edu.cn)

(南京大学 人工智能学院, 南京 210093)

1 深度优先搜索

树搜索离不开“节点”这一数据结构。本代码中将其抽象, 实现在 `src/controllers/depthfirst/Node.java`, 其中封装了节点状态、节点动作、父节点等数据, 并重写了关键 `clone()` 方法以备使用。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 `unreached_notes` 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、未移动节点(撞墙)或重复节点, 以此决定是否继续展开。当搜索到游戏胜利节点时, 程序跳出循环, 依次根据当前节点的父节点向 `choosed_actions` 栈中压入相应动作, 完成后即可将动作依次出栈, 并在现实中执行之。

在调试程序的过程中, 我印象较深的点有: 1.java 对象的默认拷贝方式只是一个引用, 拷贝对象回随原对象的改变而改变, 产生令人费解的结果, 需经过多次调试方能定位错误。因此, 对象的拷贝需要用特定的方法进行深拷贝; 2.搜索到底是在 `Agent` 的“脑海”中, 还是在现实环境中一步一步试错, 在程序中一定要明确区分。我认为, 这也是 `Model-based planning` 与 `Model-free` 的算法之间的关键区别。

2 深度受限搜索

任务中的深度受限搜索要求每步都进行一次搜索, 利用启发式函数决定当前一步的动作。大致思路与深度优先搜索类似, 但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。相应地, “节点”类中需加入深度信息。

在实现算法的过程中, 我主要遇到了如下问题: 1.启发式函数难以设计, 在第一关中容易出现箱子盖住钥匙的情况; 解决方法: 通过阅读框架手册, 找到描述游戏中各物体位置的函数, 针对性设计启发函数, 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。3.尝试利用深搜做第三关的时候出现明显的 `suboptimality`, 这也是深度优先搜索的一个特点。

3 A*搜索

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的深度, $h(n)$ 为与深度受限搜索中类似的启发函数。深度优先搜索中“栈”的数据结构相应地换成了优先队列, 每次将最优的节点出列。理论上如果 $h(n) \leq h^*(n)$, 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)。考虑到第二关游戏的布局特点, 我在启发函数中额外加入了箱子到钥匙的距离。最终第二关游戏可以被 A*完成, 但似乎出现了一些 `suboptimality`, 这可能与每次不能搜索全部节点有关。

在调试代码的过程中，我发现每次执行一步策略容易导致状态的重复访问，这是因为下一次搜索时智能体已经忘记了之前搜索的经验。因此我令智能体每次搜索完成后，执行多步动作（通常是 5-10 步），加强前后策略的一致性。另外，此游戏包含一定数量的“死胡同”状态，一旦进入，整局游戏相当于提前结束，比如将箱子推到死角。这类状态的启发函数较难设计，我采用了规则+加以极大惩罚的方式防止智能体搜索执行相应状态，但似乎在复杂的环境中难以奏效，因此游戏只进行到了第二关。第三关中由于环境过于复杂，仅覆盖钥匙前的两个洞就需要数十步搜索，在 100timestep 的时间限制下难以搜索到（通常只能搜索 10 层左右）。但是我相信，只要根据环境特点精心设计启发函数，A*算法一定能在给定时间限制内完成搜索。

精心设计启发函数的过程使我想起了强化学习智能体训练过程中奖赏函数的设计。相比而言，奖赏函数要稀疏得多。相应地，强化学习引入一定随机性，用大量试错的方式最大化利用奖赏函数，得到最优解的代价也大一些。

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，同时带有强化学习的身影。其大致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 获得关于已搜索节点的信息并更新原有信息。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` 又包含了这样几个部分：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。下面分别介绍 `root.mctsSearch` 中的三个关键函数。

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 规则在所有未选择的节点中挑选(uct 意义下)最优的。注意代码中还对 `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，与强化学习中的相应方法类似，不再赘述。2.`rollOut()`相对简单，即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。

以上就是我本次实验报告的全部内容，感谢您的阅读！