计算机科学188：人工智能导论
2024年春季 Note 2

作者（其他所有注释）：尼基尔·夏尔马
作者（贝叶斯网络注释）：乔希·胡格和杰基·梁，由王瑞吉编辑
作者（逻辑注释）：亨利·朱，由考佩林编辑
学分（机器学习和逻辑注释）：部分章节改编自教材《人工智能：一种现代方法》。

上次更新时间：2023年8月26日

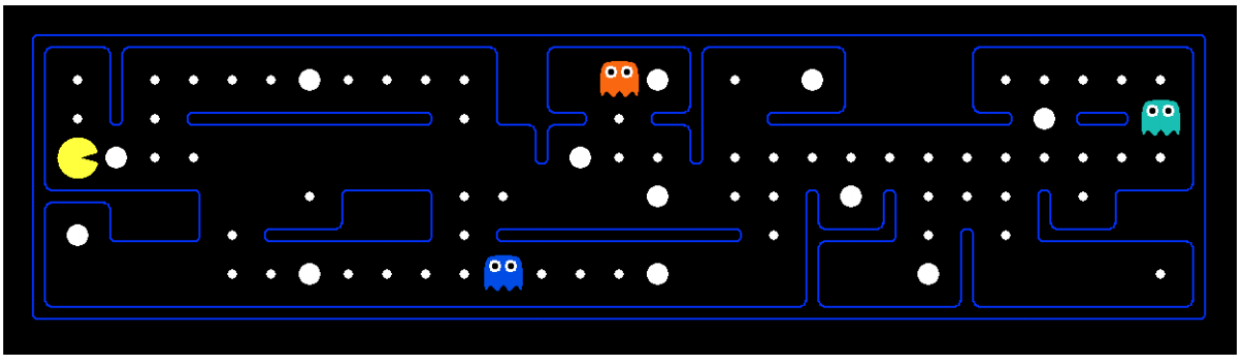
状态空间与搜索问题

为了创建一个理性的规划智能体，我们需要一种方法来数学地表达智能体所处的给定环境。为此，我们必须正式表达一个搜索问题——给定我们智能体的当前状态（它在其环境中的配置），我们如何以最佳方式达到一个满足其目标的新状态？一个搜索问题由以下元素组成：

- 一个状态空间——在给定世界中所有可能状态的集合
- 在每个状态下可用的一组动作
- 一个转移模型——当在当前状态采取特定动作时输出下一个状态
- 动作成本——应用一个动作后从一个状态转移到另一个状态时产生的成本
- 起始状态——智能体最初所处的状态
- 目标测试——一个以状态为输入并确定其是否为目标状态的函数

从根本上说，解决搜索问题首先要考虑起始状态，然后使用动作、转移和成本方法探索状态空间，迭代计算各种状态的子状态，直到到达目标状态，此时我们就确定了从起始状态到目标状态的路径（通常称为计划）。考虑状态的顺序是使用预定策略确定的。我们很快会介绍策略的类型及其用途。

在我们继续探讨如何解决搜索问题之前，重要的是要注意世界状态和搜索状态之间的区别。世界状态包含关于给定状态的所有信息，而搜索状态仅包含规划所需的关于世界的信息（主要是出于空间效率的原因）。为了说明这些概念，我们将引入本课程的标志性激励示例——吃豆人游戏。吃豆人游戏很简单：吃豆人必须在迷宫中导航并吃掉迷宫中的所有（小）食物颗粒，同时不被恶意巡逻的幽灵吃掉。如果吃豆人吃掉其中一个（大）能量豆，他会在一段时间内对幽灵免疫，并获得吃掉幽灵得分的能力。



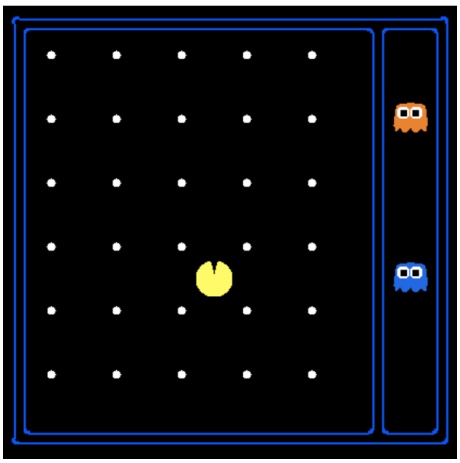
让我们考虑游戏的一种变体，其中迷宫中仅包含吃豆人和食物颗粒。在这种情况下，我们可以提出两个不同的搜索问题：路径规划和吃光所有豆子。路径规划试图最优地解决在迷宫中从位置 (x_1, y_1) 到达位置 (x_2, y_2) 的问题，而吃光所有豆子试图解决在最短时间内消耗迷宫中所有食物颗粒的问题。下面列出了这两个问题的状态、动作、转移模型和目标测试：

- 路径规划
  - 状态：(x, y) 位置
  - 动作：北、南、东、西
  - 转移模型（获取下一个状态）：仅更新位置
  - 目标测试：$(x, y) =$ 是结束状态吗？
- 吃光所有豆子
  - 状态：(x, y) 位置，豆子布尔值
  - 行动：北、南、东、西
  - 转移模型（获取下一个状态）：更新位置和布尔值
  - 目标测试：所有点布尔值是否都为假？

请注意，对于路径规划，状态所包含的信息比吃完全部点的状态要少，因为对于吃完全部点的情况，我们必须维护一个布尔值数组，该数组对应于每个食物颗粒以及在给定状态下它是否已被吃掉。一个世界状态可能还包含更多信息，可能会编码诸如吃豆人走过的总距离或吃豆人在其当前（x，y）位置和点布尔值之上访问过的所有位置等信息。

状态空间大小

在估计解决搜索问题的计算运行时间时，经常出现的一个重要问题是状态空间的大小。这几乎完全是通过基本计数原理来完成的，该原理指出，如果在给定世界中有 n 个可变对象，它们分别可以取 $x_1, x_2, \ldots, x_n$ 个不同的值，那么状态的总数就是 $x_1 \cdot x_2 \cdot \ldots \cdot x_n$。让我们以吃豆人为例来说明这个概念：



假设变量对象及其相应的可能性数量如下：

- 吃豆人位置 - 吃豆人可以处于120个不同的（x，y）位置，并且只有一个吃豆人
- 吃豆人方向 - 可以是北、南、东或西，总共4种可能性
- 幽灵位置 - 有两个幽灵，每个幽灵可以处于12个不同的（x，y）位置
- 食物颗粒配置 - 有30个食物颗粒，每个食物颗粒可以被吃掉或未被吃掉

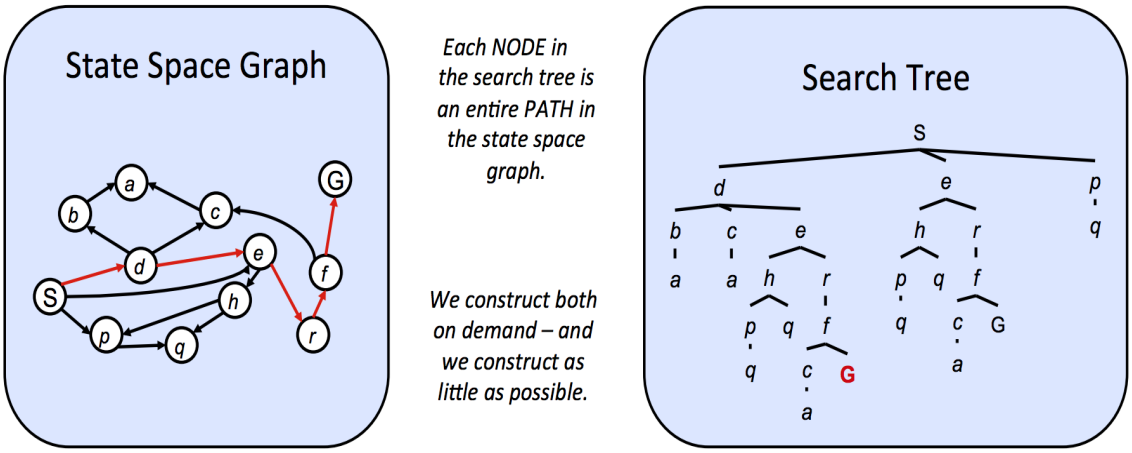
根据基本计数原理，吃豆人有120个位置，吃豆人可以面对4个方向，$12 \cdot 12$ 幽灵配置（每个幽灵有12种），以及 $2 \cdot 2 \cdot \ldots \cdot 2 = 2^{30}$ 食物颗粒配置（30个食物颗粒中的每个都有两个可能的值 - 被吃掉或未被吃掉）。这给我们一个总的状态空间大小为 $120 \cdot 4 \cdot 12^2 \cdot 2^{30}$ 。

状态空间图与搜索树

既然我们已经确立了状态空间的概念以及完全定义一个状态空间所需的四个组件，那么我们几乎就准备好开始解决搜索问题了。拼图的最后一块是状态空间图和搜索树。

回想一下，图是由一组节点和连接各对节点的一组边定义的。这些边也可能有与之相关的权重。状态空间图是用表示节点的状态构建的，从一个状态到它的子状态存在有向边。这些边表示动作，任何相关的权重表示执行相应动作的成本。通常，状态空间图太大以至于无法存储在内存中（即使是我们上面简单的吃豆人示例也有 $\approx 10^{13}$ 种可能的状态，哎呀！），但在解决问题时从概念上记住它们是有好处的。还需要注意的是，在状态空间图中，每个状态只精确表示一次——根本没有必要多次表示一个状态，并且在试图推理搜索问题时，了解这一点会有很大帮助。

与状态空间图不同，我们接下来感兴趣的结构——搜索树，对一个状态能够出现的次数没有这样的限制。这是因为尽管搜索树也是一类以状态为节点、动作作为状态之间边的图，但每个状态/节点不仅编码状态本身，还编码状态空间图中从起始状态到给定状态的整个路径（或计划）。观察下面的状态空间图和相应的搜索树：



在给定状态空间图中突出显示的路径 $(\mathrm{S} \rightarrow \mathrm{d} \rightarrow \mathrm{e} \rightarrow \mathrm{r} \rightarrow \mathrm{f} \rightarrow \mathrm{G})$ 在相应的搜索树中通过沿着树中从起始状态 S 到突出显示的目标状态 G 的路径来表示。类似地，从起始节点到任何其他节点的每一条路径在搜索树中都由从根节点 S 到与其他节点对应的根节点的某个后代的路径来表示。由于从一个状态到另一个状态通常有多种方式，状态往往会在搜索树中多次出现。因此，搜索树的大小大于或等于其相应的状态空间图。

我们已经确定，即使对于简单问题，状态空间图本身的规模也可能非常大，于是问题来了——如果这些结构太大而无法在内存中表示，我们如何对它们进行有用的计算呢？答案在于我们如何计算当前状态的子状态——我们只存储当前正在处理的状态，并使用相应的getNextState、getAction和getActionCost方法按需计算新的状态。通常，搜索问题是使用搜索树来解决的，在搜索树中，我们非常谨慎地一次只存储少数几个要观察的节点，迭代地用它们的子节点替换节点，直到我们到达目标状态。存在各种方法来决定以何种顺序进行搜索树节点的这种迭代替换，我们现在将介绍这些方法。

盲目搜索

用于找到从起始状态到目标状态的计划的标准协议是维护一个源自搜索树的部分计划的外部前沿。我们通过从前沿中移除与一个部分计划相对应的节点（该节点使用给定策略选择），并将其在前沿上替换为其所有子节点，来不断扩展前沿。用子节点移除并替换前沿上的一个元素，相当于丢弃一个长度为 n 的计划，并考虑源自它的所有长度为 $(n+1)$ 的计划。我们持续这个过程，直到最终从前沿移除一个目标状态，此时我们得出与移除的目标状态相对应的部分计划实际上是一条从起始状态到目标状态的路径。

实际上，此类算法的大多数实现会在节点对象内部编码有关父节点、到节点的距离以及状态的信息。我们刚刚概述的这个过程称为树搜索，其伪代码如下所示：

```
function TREE-SEARCH(problem, frontier) return a solution or failure
    frontier← INSERT(MAKE-NODE(INITIAL-STATE[problem]), frontier) while not IS-EMPTY(frontier) do
        end
        node ← POP(frontier) if problem.IS-GOAL(node.STATE) then
        end
        return node

        for each child-node in EXPAND(problem, node) do
        end
        add child-node to frontier



return failure
```

伪代码中出现的EXPAND函数返回通过考虑所有可用动作从给定节点可以到达的所有可能节点。该函数的伪代码如下：[1]

```
函数EXPAND(问题, 节点) 产生节点
    s ← 节点.STATE 对于问题.ACTIONS(s) 中的每个动作执行
    end
    s′ ← problem.RESULT(s, action)
    yield NODE(STATE=s′, PARENT=node, ACTION=action)
```

当我们对搜索树中目标状态的位置一无所知时，我们被迫从属于无信息搜索范畴的技术中选择我们的树搜索策略。我们现在将依次介绍三种这样的策略：深度优先搜索、广度优先搜索和一致代价搜索。对于每种策略，还会从以下方面介绍该策略的一些基本属性：

- 每种搜索策略的完备性——如果存在搜索问题的解，在给定无限计算资源的情况下，该策略是否保证能找到它？
- 每种搜索策略的最优性——该策略是否保证找到到达目标状态的最低代价路径？

•分支因子 b - 每次从前沿队列中取出一个前沿节点并将其替换为其子节点时，前沿节点数量的增加量为 $O(b)$。在搜索树的深度 k 处，存在 $O\left(b^k\right)$ 个节点。

•最大深度 m。

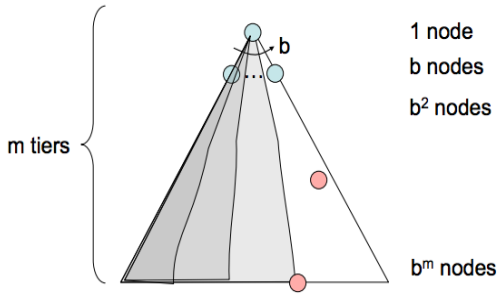
•最浅解的深度 s。

深度优先搜索

•描述 - 深度优先搜索（DFS）是一种探索策略，它总是从起始节点选择最深的前沿节点进行扩展。

[1] 关于产量的定义，请参考教科书的附录B.2。

•前沿表示 - 移除最深层节点并用其孩子节点替换它，这必然意味着这些孩子节点现在是新的最深层节点 - 它们的深度比之前最深层节点的深度大1。这意味着要实现深度优先搜索（DFS），我们需要一种结构，该结构总是赋予最近添加的对象最高优先级。后进先出（LIFO）栈恰好能做到这一点，并且在实现DFS时传统上就是用它来表示前沿。



•完备性 - 深度优先搜索不完备。如果状态空间图中存在环，这必然意味着相应的搜索树深度将是无限的。因此，存在深度优先搜索会忠实地但悲惨地"陷入"在无限大小的搜索树中寻找最深层节点的情况，注定永远找不到解决方案。

•最优性 - 深度优先搜索只是在搜索树中找到"最左边"的解决方案，而不考虑路径成本，因此不是最优的。

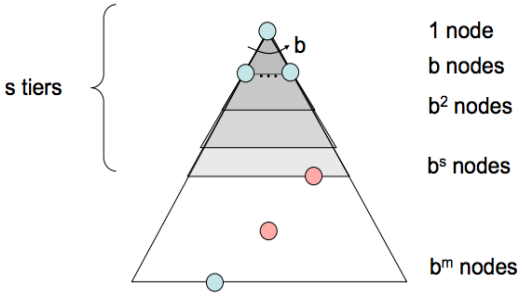
•时间复杂度 - 在最坏情况下，深度优先搜索可能会遍历整个搜索树。因此，对于一棵最大深度为 m 的树，深度优先搜索的运行时间为 $O\left(b^m\right)$ 。

•空间复杂度 - 在最坏情况下，深度优先搜索在前沿的每个 m 深度级别上维护 b 个节点。这是因为一旦某个父节点的 b 个子节点被入队，深度优先搜索的性质使得在任何给定时间只能探索这些子节点中任何一个的子树之一。因此，深度优先搜索的空间复杂度为 $O\left(bm\right)$ 。

广度优先搜索

•描述 - 广度优先搜索是一种探索策略，它总是从起始节点选择最浅的前沿节点进行扩展。

•前沿表示 - 如果我们想在访问较深节点之前访问较浅节点，我们必须按照节点的插入顺序访问它们。因此，我们需要一种结构，它输出最早入队的对象来表示我们的前沿。为此，广度优先搜索使用先进先出（FIFO）队列，它正是为此而设计的。



•完备性 - 如果存在解决方案，那么最浅节点 s 的深度必定是有限的，因此广度优先搜索最终必定会搜索到这个深度。所以，它是完备的。

•最优性 - 广度优先搜索通常不是最优的，因为在确定前沿上要替换哪个节点时，它根本没有考虑成本。广度优先搜索保证是最优的特殊情况是所有边的成本都相等，因为这会将广度优先搜索简化为下面要讨论的一致成本搜索的一种特殊情况。

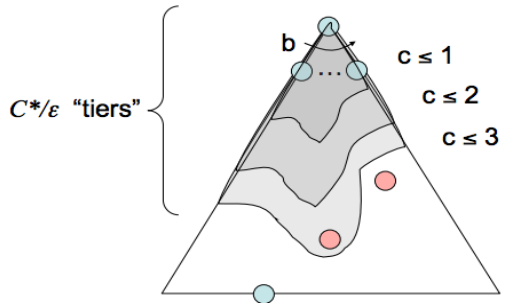
•时间复杂度 - 在最坏情况下，我们必须搜索 $1 + b + b^2 + \ldots + b^s$ 个节点，因为我们要遍历从1到 s 的每个深度的所有节点。因此，时间复杂度是 $O\left(b^s\right)$ 。

•空间复杂度 - 在最坏情况下，前沿包含与最浅解决方案对应的层中的所有节点。由于最浅解决方案位于深度 s 处，所以此深度有 $O\left(b^s\right)$ 个节点。

一致成本搜索

•描述 - 一致代价搜索（UCS）是我们的最后一种策略，它是一种探索策略，总是从起始节点选择代价最低的前沿节点进行扩展。

•前沿表示 - 为了表示UCS的前沿，通常的选择是基于堆的优先队列，其中给定入队节点 v 的优先级是从起始节点到 v 的路径代价，即 v 的反向代价。直观地说，以这种方式构造的优先队列会在我们移除当前最小代价路径并用其子节点替换它时，简单地重新排列自身以按路径代价维持所需的顺序。



•完备性 - 一致代价搜索是完备的。如果目标状态存在，它必定有某个有限长度的最短路径；因此，UCS最终必定会找到这条最短长度的路径。

•最优性 - 如果我们假设所有边的成本都是非负的，那么一致代价搜索（UCS）也是最优的。通过构造，由于我们按照路径成本递增的顺序探索节点，所以我们保证能找到到达目标状态的最低成本路径。一致代价搜索中采用的策略与迪杰斯特拉算法相同，主要区别在于UCS在找到解状态时终止，而不是找到到所有状态的最短路径。请注意，图中存在负边成本会使路径上的节点长度递减，从而破坏我们的最优性保证。（有关处理这种可能性的较慢算法，请参见贝尔曼 - 福特算法）

•时间复杂度 - 我们将最优路径成本定义为 C^* ，将状态空间图中两个节点之间的最小成本定义为 ε 。那么，我们必须大致探索深度从1到 C^*/ε 的所有节点，这导致运行时间为 $O\left(b^{C^*/\varepsilon}\right)$ 。

•空间复杂度 - 大致来说，前沿将包含最便宜解决方案层级的所有节点，因此UCS的空间复杂度估计为 $O\left(b^{C^*/\varepsilon}\right)$ 。

作为关于盲目搜索的临别提示，必须注意的是，上述三种策略本质上是相同的 - 只是在扩展策略上有所不同，它们的相似之处由上述树搜索伪代码体现。