CS 188 人工智能入门

2024 年春 附注 2

作者（所有其他注释）：尼基尔-夏尔马

作者（贝叶斯网注释）：作者（逻辑笔记）：Josh Hug 和 Jacky Liang，编辑：

Regina Wang：Henry Zhu，编辑：Peyrin Kao

学分（机器学习与逻辑笔记）：部分章节改编自教科书《*人工智能》：现代方法》*。

最后更新2023 年 8 月 26 日

状态空间和搜索问题

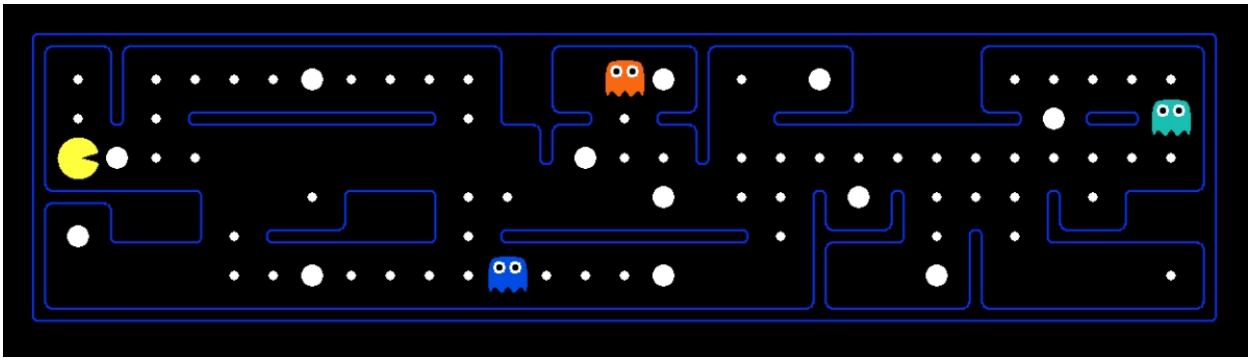
为了创建一个理性的规划代理，我们需要一种方法来数学地表达代理将存在的给定环境。为此，我们必须正式表达一个**搜索问题**--给定代理的当前**状态**（其在环境中的配置），我们如何才能找到一个新状态，以最佳方式满足其目标？搜索问题由以下要素组成：

- **状态空间** - 特定世界中所有可能状态的集合
- 在每个状态下可采取的一系列**行动**
- **转换**模型 - 在当前状态下采取特定行动时输出下一状态
- 行动**成本** - 实施行动后从一个状态移动到另一个状态时产生的**成本**
- **起始状态** - 代理人最初存在的状态
- **目标测试** - 将状态作为输入，并确定其是否为目标状态的函数

从根本上说，要解决搜索问题，首先要考虑起始状态，然后使用动作、转换和代价方法探索状态空间，反复计算各种状态的子状态，直到到达目标状态，这时我们就确定了一条从起始状态到目标状态的路径（通常称为**计划**）。考虑状态的顺序由预先确定的**策略**决定。稍后我们将介绍策略

的类型及其作用。

在我们继续讨论如何解决搜索问题之前，有必要指出**世界态**和**搜索态**之间的区别。世界状态包含给定状态的所有信息，而搜索状态只包含规划所需的世界信息（主要是出于空间效率的考虑）。为了说明这些概念，我们将介绍本课程的标志性激励示例--吃豆子游戏。吃豆子游戏很简单：吃豆人必须在迷宫中穿梭，吃掉迷宫中的所有（小）食物颗粒，而不被恶意巡逻的幽灵吃掉。如果 Pacman 吃下一颗（大）能量颗粒，他就会在一定时间内对幽灵免疫，并获得吃掉幽灵获取积分的能力。



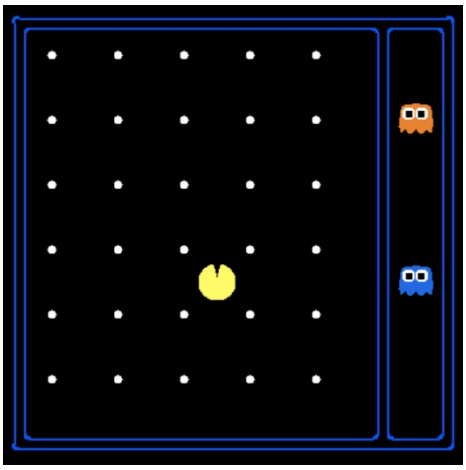
让我们考虑一下游戏的变体，在这个变体中，迷宫里只有吃豆人和食物颗粒。在这种情况下，我们可以提出两个不同的搜索问题：路径问题和吃光所有点的问题。路径问题试图以最优方式从迷宫中的位置（x_1, y_1）到达位置（x_2, y_2），而 "吃掉所有点 "问题则试图在最短时间内吃掉迷宫中的所有食物颗粒。下面列出了这两个问题的状态、行动、转换模型和目标测试：

- **路径**
  - 国家：（x,y）地点
  - 行动北、南、东、西
  - 转换模型（获取下一个状态）：仅更新位置
  - 目标测试：(x,y)=END 吗？
- **吃掉所有圆点**
  - 状态：（x,y）位置，点布尔值
  - 行动北、南、东、西
  - 转换模型（获取下一个状态）：更新位置和布尔值
  - 目标测试：所有点布尔值都是假的吗？

需要注意的是，与 "吃光所有点 "的状态相比，"路径 "的状态包含的信息更少，因为 "吃光所有点 "的状态必须维护一个布尔数组，该数组与每个食物颗粒以及在给定状态下是否被吃掉相对应。世界状态可能包含更多的信息，有可能在当前 (x,y) 位置和点布尔值的基础上，还编码了吃豆人走过的总距离或吃豆人访问过的所有位置等信息。

状态空间大小

在估算求解搜索问题的计算运行时间时，经常会遇到的一个重要问题是状态空间的大小。这几乎只能通过**基本计数原理**来实现，即如果给定世界中有 n 个变量对象，它们可以有 $x_1, x_2, \ldots, x_n$ 则状态总数为 $x_1 \cdot x_2 \cdot \ldots \cdot x_n$。让我们用吃豆子来举例说明这一概念：



假设变量对象及其对应的可能性数如下：

- *吃豆人的位置* - *吃*豆人可以处于 120 个不同的 (x, y) 位置，而且只有一个吃豆人
- *吃*豆子*方向*--可以是北、南、东或西，共有 4 种可能性
- *幽灵位置* - 有两个幽灵，每个幽灵可处于 12 个不同的（x、y）位置
- *食物颗粒配置* - 共有 30 个食物颗粒，每个颗粒都可以吃或不吃

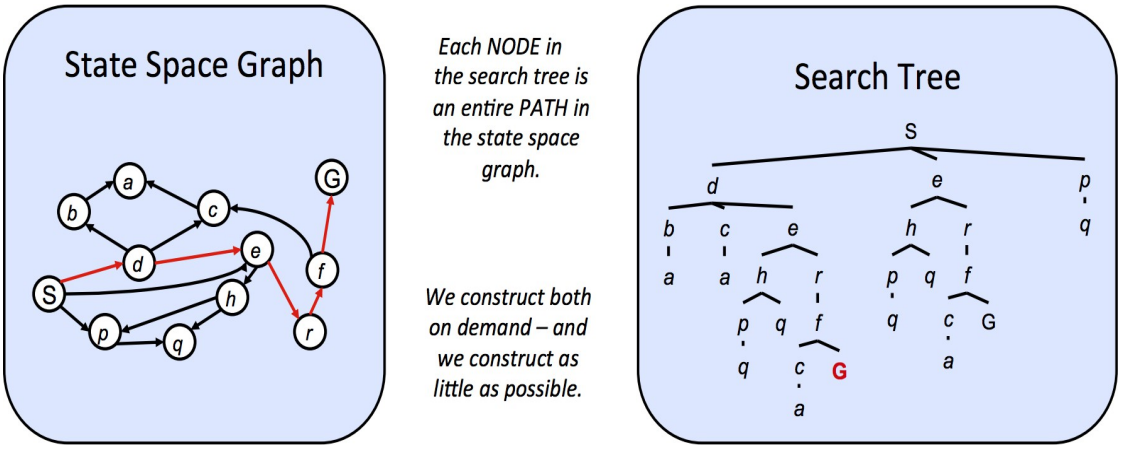
利用基本计数原理，我们可以得出 Pacman 的 120 个位置、Pacman 可能朝向的 4 个方向、12 - 12 个幽灵配置（每个幽灵 12 个）以及 2 - 2 - ...- 2 = 2^{30} 食物颗粒配置（30 个食物颗粒中的每个颗粒都有两种可能的值--吃或不吃）。这样，我们的总状态空间大小为 $120 \cdot 4 \cdot 12^2 \cdot 2^{30}$.

状态空间图和搜索树

既然我们已经建立了状态空间的概念，以及完整定义状态空间所需的四个组成部分，那么我们就差不多可以开始解决搜索问题了。最后一块拼图是状态空间图和搜索树。

回顾一下，图形是由一组节点和一组连接不同节点对的边定义的。这些边也可能有相关的权重。**状态空间图由**状态代表节点，从状态到其子节点之间存在有向边。这些边代表行动，相关权重代表执行相应行动的成本。通常情况下，状态空间
图的数据量太大，无法存储在内存中（即使是上面简单的吃豆子例子，也有≈ 10^{13} 种可能的状态，哎呀！），但在解决问题时，我们可以从概念上记住它们。这也是
需要注意的是，在状态空间图中，每个状态都只表示一次，没有必要多次表示一个状态。

与状态空间图不同，我们感兴趣的下一个结构--**搜索树**，对一个状态出现的次数没有这样的限制。这是因为，虽然搜索树也是一类以状态为节点、以操作为状态间边的图，但每个状态/节点所编码的不仅仅是状态本身，而是状态空间图中从起始状态到给定状态的整个路径（或**计划**）。请看下面的状态空间图和相应的搜索树：



给定状态空间图中的高亮路径（S → d → e → r → f → G）在相应的搜索树中表示，即沿着树中从起始状态 S 到高亮目标状态的路径进行搜索

同样，在搜索树中，从起点节点到任何其他节点的每一条路径都由从根 S 到根的某个后代节点的路径来表示。由于从一个状态到另一个状态往往有多种途径，因此状态往往会在搜索树中出现多次。因此，搜索树的大小大于或等于相应的状态空间图。

我们已经确定，即使是简单的问题，状态空间图本身也可能非常庞大，因此问题来了--如果这些结构太大，无法在内存中表示，我们如何才能对它们进行有用的计算呢？答案就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。通常情况下，搜索问题使用搜索树来解决，在搜索树中，我们每次都会非常小心地存储一些需要观察的节点，并用它们的子节点反复替换节点，直到我们到达目标状态。有多种方法可以决定搜索树节点迭代替换的顺序，我们现在就来介绍这些方法。

不知情的搜索

寻找从起始状态到目标状态的计划的标准协议是维护一个由搜索树衍生的部分计划组成的外**边界**。我们**不断扩大**前沿，方法是移除前沿中与部分计划相对应的节点（该节点使用给定**策略**选出），并用其所有子节点替换前沿中的节点。移除前沿上的一个元素并用它的子节点替换它，就相当于丢弃一个长度为 n *的计划*，并将由它产生的所有长度为 $(n + 1)$ 的计划都纳入考虑范围。我们会继续这样做，直到最终将目标状态从前沿中移除，这时我们会得出结论，与被移除的目标状态相对应的部分计划实际上是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都会编码有关父节点、节点间距以及节点对象内部状态的信息。

我们刚才概述的这一过程被称为**树搜索**，其伪代码如下：

```
函数 TREE-SEARCH（问题、边界）返回解决方案或失败
    frontier← INSERT(MAKE-NODE(INITIAL-STATE[problem])、 前沿) 当 不 IS-EMPTY(frontier) do
        最后
        节点 ← POP（前沿） if problem.IS-GOAL(node.STATE) then
        最后
        返回节点

        for each child-node in EXPAND(problem, node) do
        end
        为边界添加子节点


返回失败
```

伪代码中出现的 EXPAND 函数通过考虑所有可用操作，返回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← node.STATE for each action in problem.ACTIONS(s) do
     end
     s′ ← problem.RESULT(s, action)
    yield NODE(STATE=s′ , PARENT=node, ACTION=action)
```

当我们不知道目标状态在搜索树中的位置时，就不得不从**无信息搜索**技术中选择树搜索策略。下面我们将依次介绍三种这样的搜索策略：**深度优先搜索**、**广度优先搜索**和**均匀成本搜索**。在介绍每种策略的同时，我们还将从以下几个方面介绍策略的一些基本特性：

- 每种搜索策略的**完备性**--如果存在搜索问题的解决方案，该策略是否能保证在计算资源无限的情况下找到它？
- 每种搜索策略的**最优性**--该策略是否能保证找到成本最低的通往目标状态的路径？
- **分支因子** b - 每次前沿节点被删除并替换为其子节点时，前沿节点数量的增加量为 $O(b)$。在搜索树的深度 k 处，存在 $O(b^k)$ 个节点。
- 最大深度 m。
- 最浅解的深度 s。

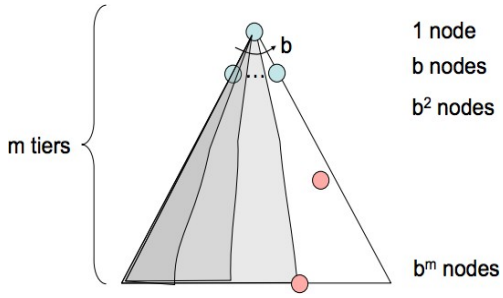
深度优先搜索

- *描述* - 深度优先搜索（DFS）是一种探索策略，它总是选择*最深的*搜索结果，并在搜索过程中不断更新。

  从起始节点扩展到前沿节点。

[1]有关**收益率**的定义，请参阅教科书附录 B.2。

- *前沿表示法*--移除最深节点并用它的子节点替换它在前沿上的位置，必然意味着子节点现在是新的最深节点--它们的深度比前一个最深节点的深度大一个。这就意味着，要实现 DFS，我们需要一种总是给予最近添加的对象最高优先级的结构。后进先出（LIFO）栈正是这样的结构，也是实现 DFS 时传统上用来表示前沿的结构。



- *完整性* - 深度优先搜索并不完整。如果状态空间图中存在循环，这必然意味着相应的搜索树的深度将是无限的。因此，DFS 有可能忠实而又悲惨地 "卡 "在无限大的搜索树中寻找最深的节点，注定永远找不到解决方案。

- *最优性*--深度优先搜索只需找到搜索树中 "最左边 "的解决方案，无需考虑路径成本，因此不是最优的。

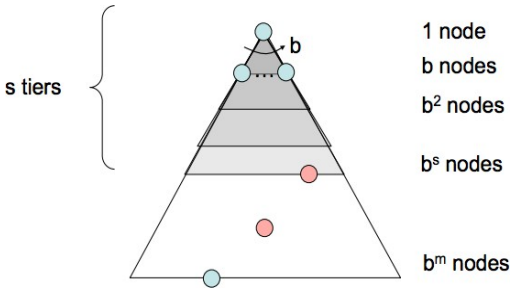
- *时间复杂性*--在最坏的情况下，深度优先搜索可能最终会探索整个搜索树。因此，给定一棵最大深度为 m *的*树，DFS 的运行时间为 $O(b^m)$。

- *空间复杂性*--在最坏的情况下，DFS 会在前沿的 m *个*深度层中的每个深度层维护 b *个*节点。这是因为一旦某个父节点的 b 个子节点被排序，DFS 的本质就只允许在任何给定时间点探索其中一个子节点的子树。因此，DFS 的空间复杂度为 $O(bm)$。

广度优先搜索

- *说明* - 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。

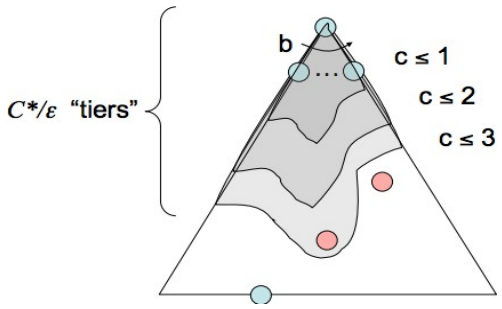
- *前沿表示法*--如果我们想在访问较深节点之前访问较浅的节点，就必须按照节点的插入顺序访问它们。因此，我们需要一种能输出最老排队对象的结构来表示我们的前沿。为此，BFS 使用了先进先出队列（FIFO），它可以完全做到这一点。


s tiers
b
1 node
b nodes
b^2 nodes
b^s nodes
b^m nodes

- *完备性* - 如果存在解，那么最浅节点 s 的深度一定是有限的，所以 BFS 最终一定会搜索到这个深度。因此，它是完整的。
- *最优性* - BFS 通常不是最优的，因为它在决定替换边界上的节点时根本不考虑成本。BFS 保证最优的特殊情况是所有边的成本都相等，因为这将 BFS 简化为统一成本搜索的一种特殊情况，下面将讨论统一成本搜索。
- *时间复杂度* - 我们必须搜索 $1 + b + b^2 + ...$在最坏的情况下，我们必须搜索 $1 + b + b^s$ 个节点，因为我们要搜索从 1 到 s *的*每个深度的所有节点。因此，时间复杂度为 $O(b^s)$。
- *空间复杂度* - 在最坏的情况下，前沿包含最浅解所对应的层中的所有节点。由于最浅解位于深度 s，因此该深度有 $O(b^s)$ 个节点。

统一成本搜索

- *说明* - 统一成本搜索（UCS）是我们的最后一种探索策略，它总是从起始节点选择*成本最低的*前沿节点进行扩展。
- *前沿*表示法--要表示 UCS 的前沿，通常选择基于堆的优先级队列，其中给定排队节点 v 的优先级是起始节点到 v *的*路径成本，或者是 v 的*后向成本*。直观地说，以这种方式构建的优先级队列在我们移除当前最小成本路径并用其子节点替换它时，会简单地重新洗牌，以保持所需的路径成本排序。



- *完备性* - 统一代价搜索是完备的。如果目标状态存在，那么它一定有一条长度有限的最短路径；因此，UCS 最终一定能找到这条最短路径。
- *最优性*--如果我们假设所有边的成本都是非负的，那么 UCS 也是最优的。根据构造，由于我们是按照路径成本递增的顺序探索节点的，因此我们可以保证找到成本最低的通往目标状

态的路径。Uniform Cost Search 算法采用的策略与 Dijkstra 算法相同，主要区别在于 UCS 在找到一个解状态时终止，而不是找到通往所有状态的最短路径。需要注意的是，在我们的图中，负边成本会使路径上的节点长度递减，从而破坏我们对最优性的保证。(有关处理这种可能性的较慢算法，请参见 Bellman-Ford 算法）。

- *时间复杂性* - 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在 1 到 C^*/ε 之间的所有节点，因此运行时间为 $O(b^{C^*/\varepsilon})$。

- *空间复杂度*--粗略地说，前沿将包含最便宜解的所有节点，因此 UCS 的空间复杂度估计为 $O(b^{C^*/\epsilon})$。

作为关于无信息搜索的结束语，必须指出的是，上述三种策略从根本上说是相同的，只是在扩展策略上有所不同，它们的相似之处可以通过上文介绍的树形搜索伪代码来体现。