第10章 计划、动作和学习

我们已经探讨了在图中查找路径的几种技术,现在来研究这些方法如何被agent用于现实问题中。先回顾一下在第7章第一次考虑图搜索计划方法时所做的假设,并提出一个能满足这种理想假设的agent结构,然后,将讨论如何修改一些搜索算法以减少它们对时间和空间的要求——使它们在提出的体系结构中更有用。最后,讨论了如何学习启发式函数和动作模型。

10.1 感知/计划/动作循环

如第7章所述,基于搜索的规划方法的功效依赖于几个很强的假设。由于以下原因,这些假设常常得不到满足:

1) 知觉过程不可能总是提供环境状态的必需信息(由于噪声或者对重要的特性不敏感)。当两种不同的环境状态引起相同的传感输入时,我们称这种情况为感知混淆(*perceptual aliasing*)。

2) 动作并不总有其模型效果(由于模型不够精确,或者受动器系统在执行动作时偶尔会产生错误)。

3) 可能在环境中有其他的物理过程或其他的agent(例如,在游戏中有对手)。这些过程可能会改变环境以致于干扰agent的动作。

4) 外部作用的存在会引起其他的问题:在构造一个计划期间,环境可能变得与原来的计划不相干。这种困难使得花费太多的时间为一个agent进行计划而变得毫无意义。

5) agent可能在完成一个到达目标状态的搜索之前被要求动作。

6) 即使agent有充分的计算时间,但是计算要求的空间资源不允许搜索进行到目标状态。

有两种主要方法可以用来解决这些困难,同时又能保留基于搜索的计划的主要特征。一种是用概率方法来形式化知觉、环境和受动器的不确定性;另一种办法是用各种附加的假设和近似来消除这些困难的影响。

处理动作的不确定效果的一种正式方法是假定对一定状态下的每一个可执行动作,结果状态由一个已知的概率分布给出。在这种情况下找到合适的动作被称为Markov决策问题(*Markov decision problem*, MDP)[*Puterman*, 1994]。通过假定agent的传感设备在它的状态集上提供一个概率分布,可以解决有缺陷知觉的其他问题。发现动作则被称为局部可见的Markov决策问题(*Partially observable Markov decision problem*, POMDP) [Lovejoy 1991, Monahan 1982, Cassandra, Kaelbling, & Littman 1994]。讨论MDP和POMDP超出了本书的范围,但在介绍了概率干扰后,会在第20章讨论与它们相关的一些技术。

在这里不讨论正式的、基于概率的方法,而是提出一个叫感知/计划/动作(*sense/plan/act*)的结构,在很多应用中它避开了上述的一些复杂性。该结构的基本原理是即使动作偶尔产生了没有预料的结果,或者agent有时不能决定它处于哪一种环境状态下,但是通过保证agent从它的执行环境中得到连续的反馈,这些困难可以被充分地解决。

确保连续反馈的一个方法是计划一个动作序列,只执行这个序列中的第一个动作,感知结果环境状态,重新计算开始节点,然后重复上述过程。这种方式,选择动作的agent被叫做感

知/计划/动作agent。然而为了使这个方法有效，计算一个计划的时间必须比每个动作执行时间要少。图10-1显示了一个感知/计划/动作agent的结构。在良性环境中（容忍几个错误步骤），感知和动作中的错误在感知/计划/动作循环序列中“达到平均数”。

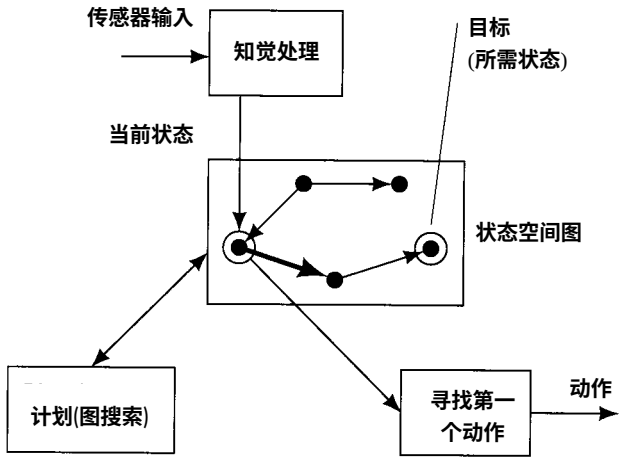


图10-1 一个感知/计划/动作agent的结构

在感知/计划/动作循环中的环境反馈允许解决感知、环境和受动器的一些不确定性。然而，为使反馈有效，必须保证感知和动作一般来说是精确的。在很多应用中，这种假设是现实的。毕竟，提供感觉、感知和受动器特征适合于任务要求是agent设计人员的任务。通常，agent通过比较即刻的感知数据和未展开状态的存储模型能够提高扩展感知精确度（回顾第5章演示的黑板系统的过滤的例子）。

10.2 逼近搜索

下面对以产生计划质量为代价的有限计算或时间资源的搜索算法进行修改，这些计划可能不是最佳的，或者可能不是总能可靠地到达目标状态。即使这个计划不是最优的（甚至也不正确），这些技术的应用也能被合并到感知/计划/动作循环中。定性地讲，只要第一个动作有缩短到达目标距离的趋势（平均情况），经感知/计划/动作循环的多次迭代将最终到达目标。

如上一章所讲，放宽产生最优计划的要求常会减少找到一个计划的计算代价。可以从两个方面来减少代价。一是能找到到达目标的一条完整路径但不必要求它是最优的；或者是能找到一条局部的路径，它不要求已达到目标节点。一个A*类型的搜索可用于这两种方法。对前者，我们用一个不可接纳的启发式函数；对后者，在到达目标前（用可接纳的或不可接纳的启发式函数）退出搜索。在到达目标前退出搜索是任意时间算法（*anytime algorithm*）[Dean & Boddy 1988， Horvitz 1987]的一个例子。任意时间算法能在任何时刻停止，结果的质量会随着运行时间的增加而改善。在下面的几个部分，我们将详细讨论这些逼近搜索方法。

10.2.1 孤岛驱动搜索

在孤岛驱动(*island-driven*)搜索中，来自问题领域的启发性知识被用于在搜索空间中建立一个“岛节点”序列，假定有好的路径通过这个搜索空间。例如，在计划通过有障碍的地形时，这些岛就是相应的山。假如n_0是开始节点，n_g是目标节点，（n_1，n_2，…，n_k）是这些岛的一个

序列。我们用n_0作为开始节点，n_1作为目标节点，开始一个启发式搜索（用一个同那个目标相适应的启发式函数）。当搜索找到了一条到n_1的路径时，就用n_1作起始点，n_2作目标点开始另一个搜索，等等，直到我们发现了一条到达n_g的路。图10-2显示了孤岛驱动搜索。

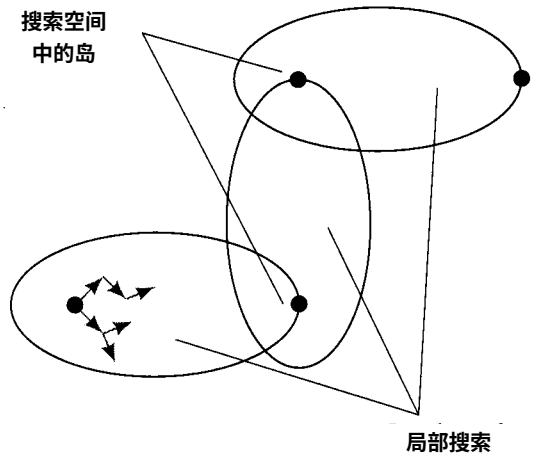


图10-2 孤岛驱动搜索

10.2.2 层次搜索

除了没有显式的岛集合外，层次搜索(*hierarchical search*)非常像孤岛搜索。假定有一些“宏算子”，它们能在一个隐式的岛搜索空间中采取大步骤。一个起始岛（在开始节点附近）和这些宏算子构成了岛的一个隐式的“元级”超大图。首先用一个元(*metalevel*)搜索来搜索这个超大图，直到找到一条宏算子路径，它可以让我们从基级开始节点附近的一个节点到达基级目标节点附近的一个节点。如果已经按照一个基级算子序列定义过宏算子，宏算子可扩展为一条基级算子路径，然后根据基级搜索，这条路径与开始和目标节点相连接。如果没有以基级算子定义的宏算子，我们必须顺着元级搜索中的岛节点路径进行基级搜索。后一种可能性如图10-3所示。

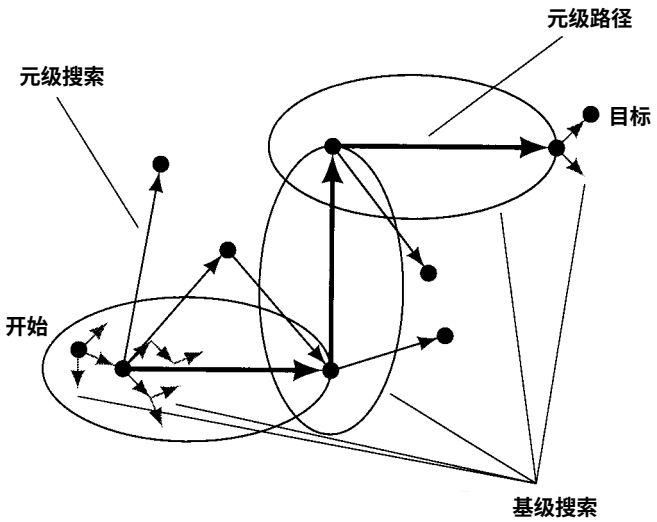


图10-3 分层搜索

作为层次计划的一个例子，考虑在一个网格空间中机器人要将一个方块推到一个给定的目标位置，如图10-4所示。一个可以推动的方块如图所示位于一个单元中，机器人的任务是把目标推到标有*G*的单元中。假定机器人能感知到它的8个相邻单元，能够判断被占据的相邻单元中是一个不可移动的障碍物还是一个可以移动的方块。像第2章一样，假定机器人在它的行列中可移到一个相邻的空白单元中，或者移入一个包含可移动方块的相邻单元中，在这种情况下，方块也向前移动一个单元（除非这种移动被一个不可移动的障碍挡住）。

假如机器人有一个所处环境的图标模型，该模型由一个与图10-4中的单元方格相似的数组表示，那么，机器人首先能做出一个关于方块如何移动的元级计划——假定方块的移动可以和机器人的移动方式相同。这个计划结果用灰箭头画在图10-4中。然后方块移动的每一步就能展开成一个基级计划。方块移动的第一步是要求机器人找到一条路径到达这样一个单元，它和方块相邻且在方块下一个目标位置的相反一侧。基级计划的结果用黑箭头画在图10-4中。随后的基级计划都是简单的，直到方块必须改变方向，这时，机器人必须找到一条路径到达与方块移动方向相反一侧的单元中，这样一直进行下去。完成在方向上变化的基级计划也在图10-4中[⊖]。

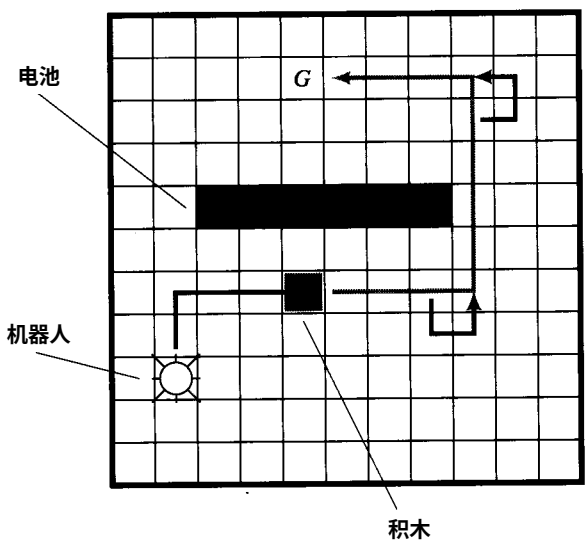


图10-4 推动一个方块

在层次计划中，如果在计划执行期间环境可能变化，仅仅展开元级计划的开始几步是明智的。仅仅展开第一个元级步可以让基级动作去执行，在它执行后，环境反馈可用来开发一个更新的元级计划。

10.2.3 有限范围搜索

在某些问题中，用任何方法搜索发现一条到达目标的路径从计算上讲都是不可能的；而在另外一些问题中，一个动作必须在一个限定的时间内作出选择，而不能在这个时间内搜索到所有到达目标的路径。在这些问题中，用有限的时间和计算量找到一条被认为是在到达目标的好

[⊖] 在一篇关于计划和执行的早期AI论文中谈到了一个非常像图10-4中模拟机器人的任务。[Nilsson & Raphael 1967]。

路径上的节点可能是有用的，尽管该节点并不是目标节点本身。当必须终止搜索时，这个替身节点 n^*在搜索前沿的所有节点中，有最小的 $\hat{f}$ 值。

假定在一个动作被选择前的可用搜索时间允许搜索到深度 d，即所有深度为 d或小于 d的路径都能被搜索到；在该深度的节点 $\mathcal{H}$将被称为范围节点。那么我们的搜索过程将搜索到深度 d，然后选择

$$n^* = \underset{n \in \mathcal{H}}{\operatorname{argmin}} \hat{f}(n)$$

作为目标节点的替代。这个方法叫做有限范围搜索 (*limited-horizon search*)。[Korf 1990]研究了这个算法并称它为最小搜索（*minimin search*）⊖。一个感知/计划/动作系统将在到达n^*的路径上采取第一个动作，感知结果状态，再迭代搜索，一遍一遍地进行下去。我们希望朝着一个拥有最优启发式指标的节点的第一步动作，正好是在朝着目标的路径上。通常，一个 agent没有必要去搜索所有到达目标的路径；因为不确定性，远距离搜索可能是不相关的，不能提供比应用在搜索水平上的启发式函数更好的信息。

有限范围搜索能通过一个处理到深度 d的深度优先搜索而高效地执行。使用单调函数 $\hat{f}$ 评估节点可以极大地减少搜索工作。一旦达到搜索范围的第一个节点 n_1，当 $\hat{f}(n) > \hat{f}(n_1)$，就能在其他节点$n$下终止搜索。节点$n$不可能有一个子孙，它的 $\hat{f}$ 值比 $\hat{f}(n_1)$小，这样在它下面进行搜索就没有意义了（在单调假设下，顺着搜索树上的任何路径上的节点 $\hat{f}$ 值是单调非递减的）。 $\hat{f}(n_1)$被称为α截断值 (*alpha cut-off value*)，在节点n下终止搜索的过程是分支和约束（*branch and bound*）搜索过程的一个实例。并且无论何时当到达其他的搜索范围点 n_2， $\hat{f}(n_2) < \hat{f}(n_1)$，α截断值就降低到 $\hat{f}(n_2)$，它放宽了截断条件。无论何时，当到达的范围节点 $\hat{f}$ 值比当前的α截断值小时，就能以这种方式降低 α截断值。[Korf 1990]把有限范围搜索和各种其他方法进行了比较，包括IDA* 和受时间约束的A*。

有限范围搜索的一种极端形式用 1作为深度约束。开始节点的最近后继被评估，导致执行到达最低 $\hat{f}$ 值的后继的动作。这些 $\hat{f}$ 直和第5章讨论的机器人导航问题中的势函数的值相类似，事实上，一个势函数在机器人导航问题中也是一个合适的 $\hat{f}$ 选择。

形式上，设$\sigma(n_0, a)$是agent通过对节点n_0采取动作a期望到达的状态描述，应用与节点 n_0上动作a相对应的算子产生状态描述。在节点n_0选择一个如下动作策略：

$$a = \underset{a}{\operatorname{argmin}} \hat{f}(\sigma(n_0, a)) = \underset{a}{\operatorname{argmin}}[c(a) + \hat{h}(\sigma(n_0, a))]$$

其中$c(a)$是动作的代价。在后面讨论学习 $\hat{f}(\sigma(n_0, a))$方法时，将会使用和上式相似的一个等式。

10.2.4 循环

在存在不确定性和agent依赖逼近计划的所有情况中，用感知 /计划/动作循环可以产生重复的循环。即agent可能会回到前面遇到过的环境状态，重复在那里采用过的动作。当然，这种反复并不意味着agent永远不能达到目标状态。Korf提出了一个计划执行算法叫实时（*real-time*）A*（RTA*），它建立了所有已经遍历过的状态的一个显式图，同时调整这个图中节点的 $\hat{h}$ 值，使它们在到达前面已经遍历过的节点时不会采取动作 [korf 1990]。

⊖ 我们可能会问为什么不搜索到一个给定代价的范围而要搜索到一个给定深度的范围。这是因为搜索需要的时间常常依赖深度而不是代价，因此我们要用深度。

10.2.5 建立反应过程

如第7章开始所述，在一个反应型机器中，设计者已为每一个可能的状态提前计算了合适的到达目标的动作。存储这些和环境状态相对应的动作可能需要大量的内存。另一方面，反应型agent常常比计划型agent反应更快。在某些情况下，提前计算（汇编）一些频繁使用的离线(*offline*)计划，把它们存储为反应例程以便可以在线(*online*)快速产生适当的动作，这样做是有益的。

例如，离线搜索能计划一个以状态空间图中的目标节点为根的生成树(*spanning tree*)，它包含从状态空间中所有（至少很多）节点到达目标的路径。能从目标节点向后搜索产生一个生成树。例如，一个获得目标（块A在块B上，块B在块C上）的积木问题的生成树如图10-5所示，它表示了从所有其他节点到达目标的路径。

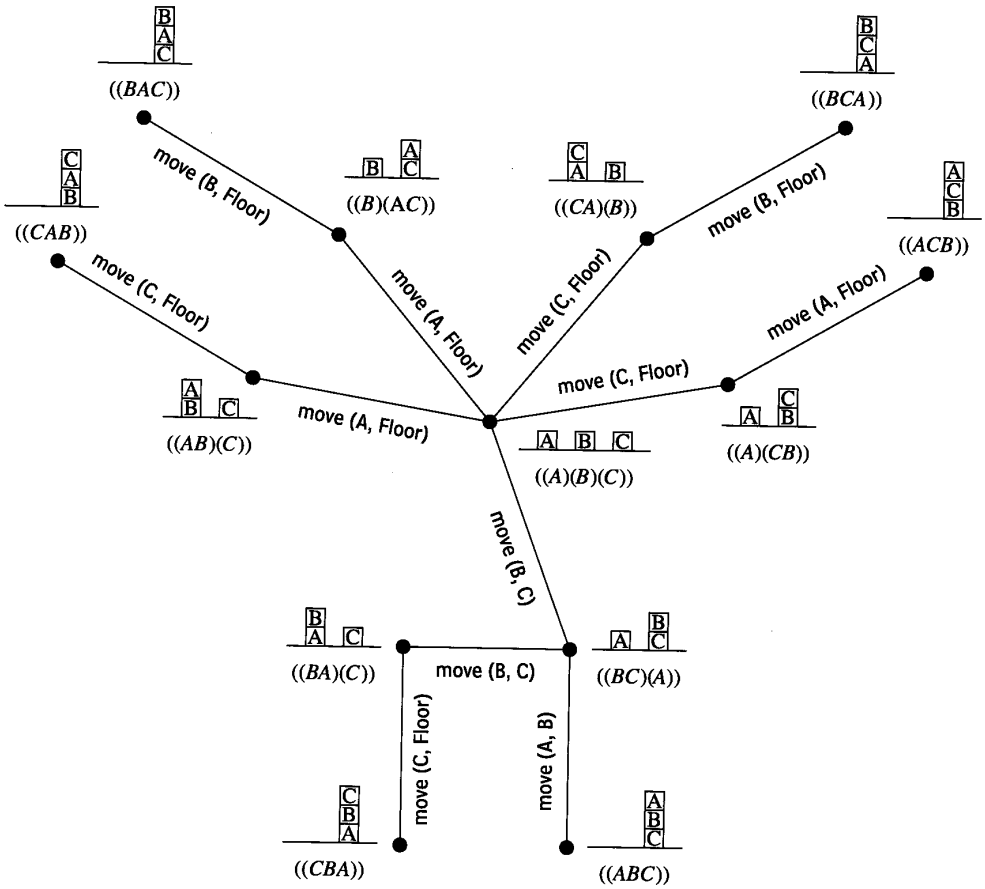


图10-5 堆积木问题的一个生成树

生成树和局部生成树能容易地转换成完全反应型的T-R程序。如果一个反应程序为每一个可能的状态指定了一个动作，就被称为通用计划（*universal plan*）[Schoppers 1987]或动作策略(*action policy*)。在本章的结尾，将讨论动作策略和学习动作策略的方法。即使在一个给定状态下采取的动作并不能总是产生预期的下一个状态，反应型机器仍能处理任何可能产生的状态。

10.3 学习启发式函数

连续的环境反馈是减少不确定性和弥补一个 agent缺乏对自己动作结果知识了解的一种方法。另外，有用的信息也能从环境中的搜索经验和动作经验中抽取出来。下面将讨论 agent能够更有效地学会计划和动作的不同方法。

10.3.1 显式图

如果agent没有一个好的启发式函数来评估到达目标的代价，有时需要学习这样一个函数。首先解释一个非常简单的适合特定情况的学习过程，在这种情况下可能会存储所有可能节点的一个显式列表。

首先假定 agent对它的动作结果有一个好的模型，并知道从任何节点移到它的后继节点的代价。通过把所有节点的 $\hat{h}$ 值初始化为0来开始这个学习过程，然后启动一个 A*搜索。在扩展 n_i产生后继 $\delta(n_i)$后，如下修改 $\hat{h}(n_i)$

$$\hat{h}(n_i) \leftarrow \min_{n_j \in \delta(n_i)} \left[\hat{h}(n_j) + c(n_i, n_j)\right]$$

$c(n_i, n_j)$是从n_i 到n_j的代价。可以将 $\hat{h}$ 值保存在一个节点表中（因为数量少，可以管理）还假定如果产生一个目标节点 n_g，并知道 $\hat{h}(n_g)=0$。虽然这个学习过程不能提高我们在第一次到达一个目标节点的搜索时达到比相同代价搜索更快的效率，但以后到达相同目标时（从可能的不同开始状态），通过使用学到的 $\hat{h}$ 函数会加快搜索。在n次搜索后，对真正h函数越来越好的评估会从目标节点逐步向后传播（学习算法LRTA*对 $\hat{h}$ 使用相同的更新规则[Korf 1990]）。

如果agent对其动作结果没有模型，通过一个相似的过程， agent能同时学会它们和一个 $\hat{h}$ 函数，虽然学习必须在真正的环境而不是状态空间模型中进行 (当然，这种学习也可能是危险的!)。假定agent有办法区分它真正遍历过的状态，并能对它们进行命名，做一个显式的图或表来表示状态，它能做出它们的估计值和由动作所引起的迁移。也假定如果 agent不知道它们的动作代价，它能学会实施动作的代价。这个过程从一个代表 agent开始状态的单个节点开始。它采取一个动作，也许是一个随机的动作，转变到另一个状态。当它访问到一个状态时，就命名它们，并如下计算它们的 $\hat{h}$ 值：

$$\hat{h}(n_i) \leftarrow \left[\hat{h}(n_j) + c(n_i, n_j)\right]$$

其中，n_i是动作a发生的节点，n_j是结果节点，$c(n_i, n_j)$是动作实施的代价， $\hat{h}(n_j)$是对n_j值的一个评估。如果前面从没访问过n_j，则 $\hat{h}(n_j)$等于0，否则它被存在表中。

无论何时，当agent准备对在图中存有后继节点的节点 n采取动作时，它根据下面的策略选择一个动作：

$$a = \underset{a}{\operatorname{argmin}} \left[\hat{h}(\sigma(n, a)) + c(n_i, \sigma(n, a))\right]$$

$\sigma(n, a)$是对在节点n采取动作a后到达状态的描述。

这个特殊的学习过程从一个随机步开始，最终慢慢到达目标，在随后的试验中向后传播来自更好路径的更好的 $\hat{h}$ 值。因为在节点n选择的动作会到达一个节点，这个节点被估计是在从 n到一个目标的最小代价路径上，因此当更新 $\hat{h}(n)$时没有必要评价n的所有后继。由于模型是逐

渐建立起来的，因此可以把“在环境学习”和“在模型中学习和计划”组合起来。这种组合在[Sutton 1990]的DYNA系统中探讨过。

由于在最佳路径上的节点可能从没被遍历过，因此这种技术会导致agent学到非最佳路径。允许偶然的随机动作(代替那些由学到的策略选择的动作)可以帮助agent学会到达目标的新路径(也许更好)。采用随机动作是一个agent解决“探测与开发权衡”(the exploration (of new paths)versus the exploitation(of already learned knowledge.tradeoff)的一种方式。也有一些其他的方法可以确保所有的节点都被遍历到。

10.3.2 隐式图

现在考虑生成所有节点及其迁移的显式图是不实际的情况。与前面一样，当有动作结果的模型时(即我们有算子，它能把一个状态描述转换成一个后继状态的描述)，我们能执行一个由评估函数导向的搜索过程。典型地讲，这种函数对几个节点可能会产生相同的评估值，因此可以把这样的函数应用到状态描述。然而在一个表中显式保存所有的节点和它们的值可能是不可行的。

使用和第3章的描述相似的方法，在执行搜索过程的同时学习启发式函数。我们先猜想一组子函数，认为它们是启发式函数的组成部分。例如，在8数码问题中，我们可用函数$W(n)$=在错误位置的数码个数，用$P(n)$=每一个数码离“家”的距离之和，还有其他的任何函数，这些函数可能与一个位置离目标的远近有关。然后可以把启发式函数作为这些函数的一个加权组合：

$$\hat{h}(n) = w_1 W(n) + w_2 P(n) + \cdots$$

这里简述两个学习权值的方法。第一种方法，把权值的初始值设为任何我们认为是最好的值，然后使用这些权值构成的$\hat{h}$函数进行搜索。当我们到达一个目标节点n_g时，用最终已知的值$\hat{h}(n_g)=0$，顺着到达目标的路径备份(back up)所有节点n_i的$\hat{h}$值。用这些值作为“训练的例子”，把权值调节为训练例子和加权组合的$\hat{h}$函数的平方差之和的最小值。这个过程在n次搜索中迭代执行。

或者我们能用和第一种方法类似的办法——用每一个节点的扩展来调整$\hat{h}$。在扩展节点n_i产生后继集合$\delta(n_i)$后，调整权值以使得：

$$\hat{h}(n_i) \leftarrow \hat{h}(n_i) + \beta \left(\min_{n_j \in \delta(n_i)} \left[\hat{h}(n_j) + c(n_i, n_j) \right] - \hat{h}(n_i) \right)$$

或重新整理为：

$$\hat{h}(n_i) \leftarrow (1-\beta)\hat{h}(n_i) + \beta \min_{n_j \in \delta(n_i)} \left[\hat{h}(n_j) + c(n_i, n_j) \right]$$

$0<\beta<1$是一个学习率参数，它控制着$\hat{h}(n_i)$向$\min_{n_j \in \delta(n_i)} \left[\hat{h}(n_j) + c(n_i, n_j) \right]$逼近的快慢程度。当$\beta=0$时，没有任何变化。当$\beta=1$时$\hat{h}(n_i)$等于$\min_{n_j \in \delta(n_i)} \left[\hat{h}(n_j) + c(n_i, n_j) \right]$小的$\beta$值使得学习非常慢，然而$\beta$趋于1会使学习过程不稳定且不能收敛。

这种学习方法是时态差分(*temporal difference*)学习的一个实例，时态差分学习由[Sutton 1988]正式提出。权值调节只依赖一个函数的两个时态相邻值[⊖]。Sutton指出并证明了在各种情

⊖ Sutton认为[Samuel 1995]是这个思想的创始人。

况下和该方法的收敛性相关的一些属性。关于时态差分方法，令人感兴趣的是在到达一个目标之前它能在搜索过程中应用(但是直到到达目标时，学习得到的 $\hat{h}$ 值才能和目标相关)。这个过程也必须在几个搜索过程中迭代执行。

我们注意到这种学习技术也能应用于没有动作结果模型的情况，也就是说，这个学习能发生在前面讨论过的现实世界中。走一步(也许是一个随机步或者是根据形成的动作策略选择的步骤)，在这一步的前后评估 $\hat{h}$ 的值，记下这一步的代价，并对权值进行调整。

10.4 奖赏代替目标

在讨论状态空间的搜索策略中，假定agent有一个简单的短期任务，它由一个目标条件描述。目标改变着世界，直到它的图标模型(以数据结构的方式)满足给定的条件。在很多实际问题中，任务并不像刚才描述的那样简单。相反，任务可能是正在进行的。用户按照给 agent一些偶然的正负奖赏(*reward*)来表达他对任务执行的满意程度。 agent的任务是把它收到的奖赏数量最大化(一个简单到达目标的特例也可用这种框架来描述，在这个框架中，当agent到达目标时，给agent一个正的奖赏（只有一次），而每次当agent采取动作时给它一个负的奖赏(根据动作的代价))。

在这种任务环境下，我们要寻找使奖赏最大化的动作策略。对于正在进行还没有终止的任务，存在的一个问题是未来的奖赏可能是无限的，因此难以决定如何使它最大化。一种处理办法是通过一些因子对未来的奖赏打折扣，即 agent宁愿要不久将来的奖赏而不愿遥远的未来奖赏。然后，假定agent在每一个时间步采用一个动作(所谓“采用”一个动作，是指在环境中真正地执行一个动作或者在环境的图搜索模型中应用一个算子)。每个动作使状态描述产生一个改变——这个改变或者是真的被agent感知到的，或者是在模型中应用一个算子计算得到的。

设n代表agent的状态空间图的一个节点，π是节点上的策略函数，节点的值是由节点上的策略描述的动作，设$r(n_i, a)$是agent在节点n_i采用了动作a时agent收到的奖赏。如果这个动作产生了节点n_j，一般地，我们会有$r(n_i, a)= - c(n_i , n_j)+\rho(n_j), \rho(n_j)$ 是到达节点n_j的任何特定的奖赏值。有些策略会比其他的策略产生对未来奖赏更大的折扣，我们寻求一个最优策略 π^*,它能使每个节点的未来折扣奖赏最大化。

给定一个策略π，我们能为在状态空间中的每个节点n估算一个值$V^{\pi}(n)$；如果agent从n开始，并采用策略π，$V^{\pi}(n)$是agent得到的总的折扣值。假如我们在节点 n_i，采用动作$\pi(n_i)$，产生了节点n_j，就得到

$$V^{\pi}(n_i) = r[n_i, \pi(n_i)] + \gamma V^{\pi}(n_j)$$

$0<\gamma<1$是折扣因子，它通过在t_i时刻的奖赏值计算在$t=t_{i+1}$时刻的奖赏，$\pi(n_i)$是由节点n_i的策略π描述的动作。对最优策略π^*，我们有:

$$V^{\pi^*}(n_i) = \max_a \left(r[n_i, a] + \gamma V^{\pi^*}(n_j)\right)$$

也就是说，n_i在最优策略下的值是直接奖赏加上最优策略下 n_j 的折扣值(乘以γ)的和的最大值(注意，n_j是动作a的函数，这使$V^{\pi^*}(n_j)$也成为a的函数)。如果知道一个最优策略下的节点值(所谓的最佳值(*optimal value*))，我们就能用下面的方式写最优策略：

$$\pi^*(n_i) = \operatorname{argmax}_a \left(r[n_i, a] + \gamma V^{\pi^*}(n_i)\right)$$

问题是我们一般不知道这些值。但是有一个叫做值迭代(*value iteration*)的学习过程，它将(在一定条件下)收敛到那些最佳值。

值迭代的工作过程如下：开始，我们给每个节点n分配一个随机的估计值(*estimated value*)$\hat{V}(n)$。假如在过程中的某一步到达节点n_i，节点n_i的估计值是$\hat{V}(n_i)$。然后我们选择动作a，它取直接奖赏和后继节点估计值之和的最大值(假定有动作结果模型上的算子，并且这些算子能应用到节点产生的后继节点)。假定动作a将我们带到节点n_j。那么如下更新节点n_i的估计值$\hat{V}(n_i)$：

$$\hat{V}(n_i) \leftarrow (1-\beta)\hat{V}(n_i) + \beta\left[r(n_i, a) + \gamma\hat{V}(n_j)\right]$$

所有其他节点的估计值保持不变。

我们看到这种调整给$\hat{V}(n_i)$一个接近$[r(n_i, a)+\gamma\hat{V}(n_j)]$的增量(依赖β)。在一定程度上，$\hat{V}(n_i)$是$\hat{V}^{\pi^*}(n_j)$的一个好的估计，这个调整有助于使$\hat{V}(n_i)$成为$\hat{V}^{\pi^*}(n_i)$的一个更好估计。

值迭代常常用于动作有随机结果和产生随机奖赏(两者都由概率函数描述)这样一些更普通的情况。尽管如此，倘若$0<\beta<1$，并且我们经常无限地遍历每一个节点，值迭代将收敛到最佳值——可能情况下的希望值(在确定性领域，我们总能用$\beta=1$)，这个结果是令人惊奇的，因为我们正在探索使用(也许一个坏的)最优策略评估的空间，同时我们正试图在一个最优策略下学习节点的值。对值迭代和有关的过程以及它们和动态编程的关系的完整讨论可以参见[Barto, Bradtke, & Singh1995]。

在奖赏依靠早期动作的序列框架中，学习动作策略被称为延迟加强学习(*delayed-reinforcement learning*)。当奖赏被延迟时会出现两个问题。第一，必须确信那些状态——动作对要对奖赏负责。完成此任务就是所谓的时态信用分配问题(*temporal credit assignment problem*)。值迭代是传播信用的一个方法，以便加强合适的动作。第二，当状态空间太大以致于不能存储整个图时，必须用相似的$\hat{V}$值聚集状态。完成此任务被称作结构信用分配问题(*structural credit assignment problem*)。神经网络和其他的学习方法对解决该问题也是有用的。在[Kaelbling, littman, & Moore 1996]中很好地讨论了延迟加强学习。

10.5 补充读物和讨论

感知/计划/动作循环是Agre和Chapman所称的交叉计划(*interleaved planning* [Agre & Chapman 1990])的一个实例。他们比较了交叉计划和他们提出的即席创作。他们的话如下(Agre&Chapman 1990,p30)：

交叉计划和即席创作的不同在于它们对故障的理解。在交叉计划的领域中，一个人会假定事件的正常状态是事件按照计划应该到达的状态。也就是说故障是一个少量现象。在即席创作领域，一个人会假定事情不可能按照计划进行下去。从完全相反的结论中，一个人希望必须继续重新决定该做什么。

但是，负责将感知系统和电机系统设计(记着任务和环境)好，以便“故障”实际上成为次要的，这些难道不是agent设计者的事吗？在T-R树[Nilsson 1994]形式中的计划对处理次要故障是相当健壮的。关于交叉计划和执行的更多内容，参见[Stentz 1995, Stentz & Hebert 1995, Nourbakhsh 1997]。

关于岛搜索问题参见[Chakrabarti, Ghose, & DeSarkar 1986]，对于层次计划建模和分析，分别参见[Korf 1987, Bacchus & Yang 1992]。[Stefik 1995, pp. 259~280]对层次计划提供了一个

非常清晰的说明。

在有限范围搜索中，一个人必须决定“范围”。这种决定必须考虑在附加计算值和已做计算推荐的动作值之间权衡。这个权衡受动作延迟的代价影响。评价更多计算和紧接动作的有关值是元级计算的一个实例。在[Russell & wefald 1991.第5章]中详细地介绍了这个主题。他们的DTA* 算法实现了关于这个主题的一些思想。

[lee & Mahajan 1988]描述了关于学习评估函数的一些方法。延迟加强和时态差分学习方法同随机的动态编程紧密相关。参考[Barto,Bradtke,&Singh 1995,Ross 1988]。基于奖赏学习动作策略的机器人系统的例子可以参考[MahadeVan & Connell 1992]和[Connell & Mahadevan 1993b]中的文章。[Moore &Atkeson 1993]介绍了有效的基于加强存储的方法来控制物理系统。[Montague, et al.1995]介绍了一个基于加强学习的蜜蜂找食物的模型，[Schultz, Dayan, & Montague 1997]描述了灵长类的神经系统是如何实现时态差分学习机制的。

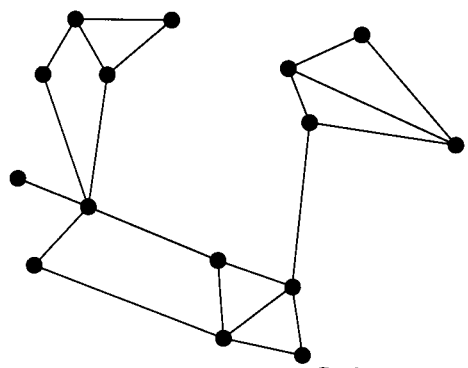
习题

10.1 在孤岛搜索中不是直接找到到达目标的路径，而且一个人首先确定一个岛，它接近于在初始节点和目标节点的中途上。我们首先要找到通过这个岛节点到达目标的一条可接受的路径(先找到从开始节点到该岛节点的一条路径，再找从该岛节点到达目标节点的一条路径)；如果不能找到可接受的途经岛节点的路径，我们只要解决原始问题就行了。假如搜索只能向前进行——即朝着目标节点进行。

- 假定对任何值d，搜索一棵分枝因子为b，深度为d的树需要的时间是kb^d，确定一个合适的岛的时间是c,岛在一条到达目标的可以接受的路径上的可能性是p。找到关于p和c的条件以便用弧岛方法需要的平均时间比普通的广度优先搜索的时间少。
- 给出一个孤岛搜索可能节约时间的搜索问题的例子。

10.2 考虑N个有名称的积木问题，这些积木可以有任意初始的配置，它们必须放在一个任意的目标配置中(忽略积木的水平位置，只考虑它们在堆中的垂直安排)。假如所有的N个积木都在地板上作为一个状态孤岛。根据搜索、存储器、时间要求和解答长度分别讨论这个方法的实现。

10.3 解释对下面的图如何建立一个层次搜索过程。希望从图中的任何节点到标为“目标”的节点找到一条路径(尽管层次搜索对这样小的一个图根本不必要，但在考虑大图中的层次搜索时，这个习题会给你一个锻炼的机会)。



10.4 想像一个agent面对着一个非常大的状态空间。假如状态空间图有b^d个节点，b是平均分枝因子，d是状态空间图中随机选择的两个节点之间的平均路径长度。我们希望比较两种设计策略：一个agent从任意的起始状态到达一个任意的目标状态。一种策略用IDA*在运行时计划一条路径，即当给定目标和初始状态时，agent用IDA*计算一条路径。另一个策略对所有的目标状态和起始状态提前计算和保存所有可能的路径。按照它们的时间和空间复杂度比较这两种策略(对很大的d值，你可以用适当的近似值)。

10.5 你在一个陌生的城市有了一份新工作，目前正和那个城市的一位朋友呆在一起。每天早晨，他驱车送你到城中的一个地铁站，你必须从那儿乘车去工作(那位朋友是城中的递送人员，在你们相处期间他会把你送到很多不同的车站)。地铁站(有有限个站)是一个方形网格布局。其中一个称为中心站，它是你去工作必须到达的一个站。你总能知道你到达了中心站。在每个站，你有四辆火车可以选择：北、东、南、西。每辆火车都是局部的，它只能将你带到网格中一个相邻的站，在那儿你必须下车，再搭乘另一辆车继续。一些相邻站点之间的连接永久地坏掉了，但是你知道从网格中的任何一个站到中心站仍然有一些其他的路径。每到一个站必须付1美元，你每天会从工作中得到100美元。你没有路线图，不知道各个站相对于中心站的任何位置信息。你决定用值迭代方法开发一个在每个站点要乘火车的策略。值迭代似乎是合适的，因为你总是知道你当前所在站点的名字，能从该站点乘坐哪些火车，还有这些火车到达的站点的名字。

- 描述一下这个问题中的值迭代如何工作。需要一个时态折扣因子吗？为什么需要或不需要？如果当你旅行时调整了站点值，如果你用这些值选择火车，在每次旅行中你将在中心站点结束，证明上述问题(提示：你的算法可以调整值，以便在任何试探中你从来不会通过一些不包括中心站点的站点子集进行一个无穷的旅行)。
- 如果你的学习算法不能保证产生最终会产生最佳(最短)路径的值，解释一下为什么不能，并应采取什么办法来保证最佳性。

10.6 考虑下面的状态转换图。从开始状态A有两个可能的动作。动作a到达状态B，并产生一个直接的奖赏0。动作b到达状态C产生直接奖赏1000。一旦到了状态B，只有一个可能的动作。它产生一个+1奖赏并回到B。在状态C，也只有一个可能的动作。它产生一个-1奖赏并回到C。为了采用动作a而不是b，需要什么样的时态折扣因子？