

進階搜尋



本章我們放寬上一章的簡化假設，從而更接近真實的世界。

第三章討論了單一類問題：可觀察的，確定性的，已知環境下的解決方案等，是一行動序列。在這一章中，當這些假設放寬時，我們看看會發生什麼。我們從一個非常簡單的例子開始：4.1 節和 4.2 節論及狀態空間中執行純粹的**局部搜尋演算法**，評價並修改當前的狀態而不是系統地從初始狀態開始探索路徑。這些演算法適合那些解答狀態本身的問題，而不涉及其路徑成本。局部搜尋演算法包括受到由統計物理學引起靈感的(**模擬退火法**)和進化生物學啟發的(**基因遺傳演算法**)。

然後，在第 4.3 至 4.4 節，當我們放寬確定性和可觀察性，我們觀察會發生什麼。關鍵的想法是，如果代理人不能準確預測究竟會獲得什麼知覺，那麼它必須考慮在每次**偶發情況下**可顯示知覺時，必須做些什麼。隨著部分可觀察性，代理人還需要跟蹤它可能存在的狀態。

最後，4.5 節研究了**線上搜尋**，在其中代理人所面對的狀態空間是完全未知的，並且必須被探索。

4.1 局部搜尋演算法和最佳化問題

我們談過的搜尋演算法都設計來系統化地探索搜尋空間。此系統性的完成，是透過在記憶體中保留一條或多條路徑，並且沿著路徑的每個點記錄哪些是已經探索過的。當找到目標時，到達目標的路徑同時也構成了這個問題的一個解。然而在許多問題中，到達目標的路徑是無關緊要的。例如，在八皇后問題中(參見第 3.2.1 節)，重要的是最終皇后的佈局，而不是加入皇后的次序。這一類問題包括了許多重要的應用，例如積體電路設計，工廠場地佈局，零工型工廠間排程，自動程式設計，電信網路最佳化，車輛尋徑，文件夾管理等。

如果到目標的路徑是無關的，我們將考慮不同種類的演算法，這類演算法根本不關心路徑。**局部搜尋演算法**從單獨的一個**當前節點/狀態**(而不是多條路徑)出發，通常只移動到與之相鄰的狀態。典型情況下，搜尋的路徑是不保留的。雖然局部搜尋演算法不是系統化的，但是它們有兩個關鍵的優點：(1) 它們只用很少的記憶體——通常容量是一個常數；(2) 它們經常能在不適合系統化演算法的很大或無限的(連續的)狀態空間中找到合理的解。

除了找到目標，局部搜尋演算法對於解決純粹的**最佳化問題**是很有用的，其目標是根據一個**目標函數**找到最佳狀態。許多最佳化問題不適合第三章中介紹的「標準的」搜尋模型。例如，自然界提供了一個目標函數——繁殖適應性——達爾文的進化論可以被視為最佳化的嘗試，但是這個問題並沒有「目標測試」和「路徑成本」。

為了理解局部搜尋，我們發現考慮**狀態空間地形圖**(如圖 4.1 所示)是很有用的。地形圖既有「位置」(用狀態定義)又有「高度」(由啟發成本函數或目標函數的值定義)。如果高度對應於成本，那麼目標是找到最低谷——即一個**全局最小值**；如果高度對應於目標函數，那麼目標是找到最高峰——即一個**全局最大值**(我們可以透過插入一個負號使兩者相互轉換)。局部搜尋演算法探索這個地形圖。如果目標存在，那麼**完備**的局部搜尋演算法總能找到目標；一個**最佳**的搜尋演算法總能找到全局最小值/最大值。

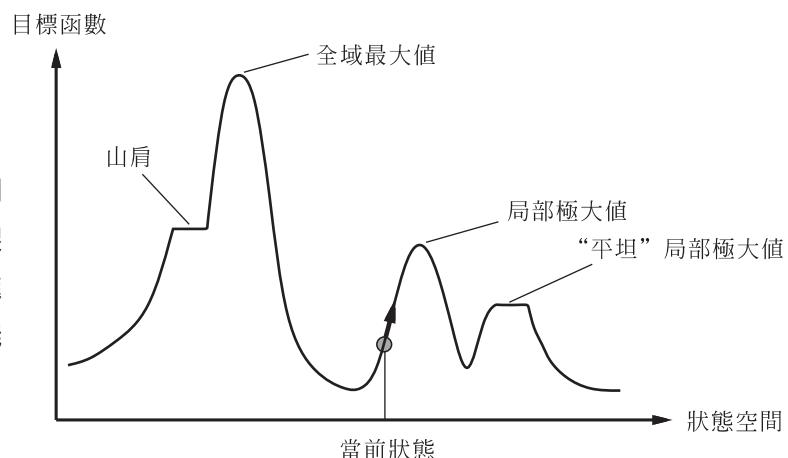


圖 4.1 一維的狀態空間地形圖，高度對應於目標函數。目標是找到全局最大值。如箭頭所示，爬山搜尋法會修正當前狀態並試圖改進它。各種地形特徵在課文中有定義

4.1.1 爬山法搜尋

圖 4.2 顯示了**爬山法(hill-climbing)**搜尋的演算法(**最陡上升**方式)。它是一個向值增加的方向持續移動的簡單迴圈過程——亦即，登高。它將會在到達一個「峰頂」時終止，相鄰狀態中沒有比它更高的值。這個演算法不維護搜尋樹，因此當前節點的資料結構只需要記錄當前狀態與其目標函數值。爬山法不會前瞻與當前狀態不直接相鄰的狀態值。這就像健忘的人在大霧中試圖登頂聖母峰一樣。

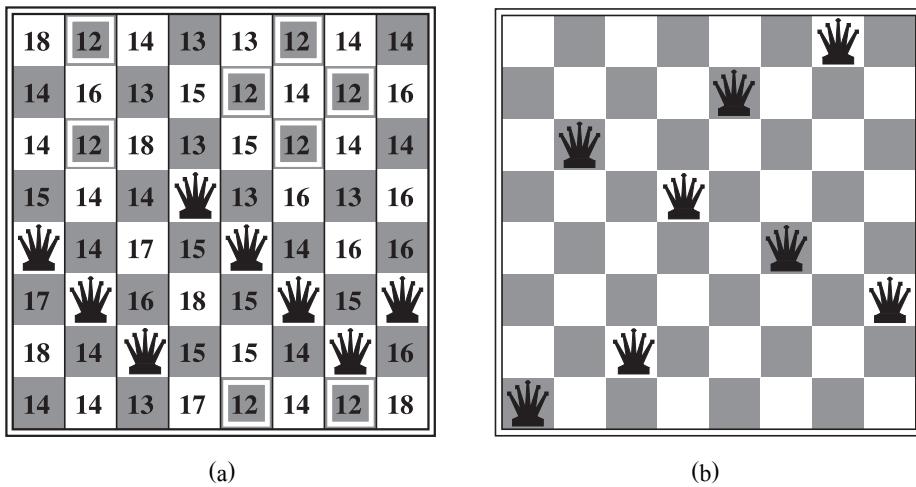
```

function HILL-CLIMBING(problem) returns a state that is a local maximum
    current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor  $\leftarrow$  a highest-valued successor of current
        if neighbor.VALUE  $\leq$  current.VALUE then return current.STATE
        current  $\leftarrow$  neighbor

```

圖 4.2 爬山法搜尋演算法，其為最基本的局部搜尋技術。在每一步中當前的節點都會被最佳鄰節點所代替；按這種方式，鄰節點意味著有最高的值，但是如果使用啟發函數成本估計 h ，我們要找的就是 h 最低的鄰節點

我們將用第 3.2.1 節介紹的八皇后問題舉例說明爬山法演算法。局部搜尋演算法通常使用**完全狀態形式化**，即每個狀態都在棋盤上放八個皇后，每列一個。狀態的後繼者是移動一個皇后到與其同一行的另一個方格時，所產生的所有可能狀態(因此每個狀態有 $8 \times 7 = 56$ 個後繼者)。啓發函數成本 h 是可以彼此攻擊的皇后對的數量，不管中間是否有障礙。該函數的全局最小值是 0，僅在找到完美解時才能得到這個值。圖 4.3(a)顯示了一個 $h = 17$ 的狀態。圖中還顯示了它的所有後繼者的值，最好的後繼者的 $h = 12$ 。爬山法演算法通常在最佳後繼者的集合中隨機選擇一個進行擴展，如果這樣的後繼者多於一個的話。



(a)

(b)

圖 4.3 (a)八皇后問題的一個狀態，其中啓發函數成本估計 $h = 17$ ，方格中顯示的數字表示將這一列中的皇后移到該方格而得到的後繼者的 h 值。最佳移動在圖中已標記出來了。(b)八皇后問題狀態空間中的一個局部極小值；該狀態的 $h = 1$ ，但是它的每個後繼者的 h 值都比它高

爬山法有時稱為**貪婪局部搜尋**，因為它只是選擇相鄰狀態中最好的一個，而不事先考慮之後的下一步往哪個方向走。儘管貪婪是七大罪之一，貪婪演算法卻往往有很好的效果。爬山法能很快朝著解的方向進展，因為它通常很容易改進一個壞的狀態。例如，從圖 4.3(a)中的狀態，它只需要五步就能到達圖 4.3(b)中的狀態，它的 $h = 1$ 基本上很接近於解了。不幸的是，爬山法經常會遇到下面的問題：

- **局部極大值**：局部極大值是一個比它的每個相鄰狀態都高的峰頂，但是比全局最大值要低。爬山法演算法到達局部極大值附近就會被拉向峰頂，然後被卡在局部極大值處而無處可走。圖 4.1 示意性地表現了這種情況。更具體地，圖 4.3(b)中的狀態事實上是一個局部極大值(即成本 h 的局部極小值)；不管移動哪個皇后得到的情況都會比原來差。
- **山脊**：圖 4.4 顯示了山脊的情況。山脊造成一系列的局部極大值，貪婪演算法處理這種情況是很難的。
- **高原**：高原是在狀態空間地形圖上的一塊平坦區域。它可能是一塊平坦的局部極大值，不存在上山的出路，或者是一個山肩，從山肩還有可能取得進展(參見圖 4.1)。爬山法搜尋可能會在高原迷路。

在每種情況下，爬山法演算法都會到達無法取得進展的地點。從一個隨機產生的八皇后問題的狀態開始，最陡上升的爬山法 86%的情況下會被卡住，只有 14%的問題實例能求解。這個演算法速度很快，成功找到最佳解的平均步數是 4 步，被卡住的平均步數是 3 步——對於包含 $8^8 \approx 1$ 千 7 百萬個狀態的狀態空間這是不錯的結果。

圖 4.2 中的演算法如果到達一個高原，最佳後繼者的狀態值和當前狀態值相等時將會停止。如果高原其實是如圖 4.1 中所示的山肩，繼續前進——即側向移動是一個好主意嗎？答案通常是肯定的，但是我們要小心。如果我們在沒有上山移動的情況下總是允許側向移動，那麼當到達一個平坦的局部極大值而不是山肩的時候，演算法會陷入無限迴圈。一種常規的解決辦法是設置允許連續側向移動的次數限制。例如，我們在八皇后問題中允許最多連續側向移動 100 次。這使問題實例的解決率從 14% 提高到了 94%。成功的代價是：演算法對於每個成功搜尋實例的平均步數為大約 21 步，每個失敗實例的平均步數為大約 64 步。

人們發明了爬山法許多變化形式。隨機爬山法在上山移動中隨機選擇下一步；選擇的概率隨著上山移動的陡峭程度而變化。這種演算法常比最陡上升演算法的收斂速度慢不少，但在某些狀態空間地形圖上能找到更好的解。首選爬山法實作了隨機爬山法，採用方式是隨機產生後繼者節點直到產生一優於當前節點的後繼者。這個演算法在有很多後繼者節點的情況下(例如上千個)是個好策略。

到現在為止我們描述的爬山法演算法還是不完備的——它們經常會在目標存在的情況下因為被局部極大值卡住而找不到該目標。隨機重新開始爬山法採納了一條著名的諺語：「如果一開始沒有成功，那麼嘗試，繼續嘗試。」它透過隨機產生的初始狀態^[1]來進行一系列的爬山法搜尋，直到找到目標時停止搜尋。這個演算法是完備的機率接近於 1，原因是它最終會產生一個目標狀態作為初始狀態。如果每次爬山法搜尋成功的概率為 p ，那麼需要重新開始搜尋的期望次數為 $1/p$ 。對於不允許側向移動的八皇后問題實例， $p \approx 0.14$ ，因此我們大概需要 7 次疊代就能找到目標(6 次失敗 1 次成功)。所需步數的期望值為一次成功疊代的搜尋步數加上失敗的搜尋步數與 $(1 - p)/p$ 的乘積，大約是 22 步。當我們允許側向移動時，平均需要疊代約 $1/0.94 \approx 1.06$ 次，平均的數為 $(1 \times 21) + (0.06/0.94) \times 64 \approx 25$ 步。那麼對於八皇后問題，隨機重新開始的爬山法實際上是非常有效的。甚至對於三百萬個皇后，這個方法用不了一分鐘就可以找到解^[2]。

爬山法演算法成功與否大部份取決於狀態空間地形圖的形狀：如果在圖中幾乎沒有局部極大值和高原，隨機重新開始的爬山法將會很快地找到好的解。另一方面，許多實際問題的地形圖看起來就像平坦地板上的一群豪豬，每個豪豬身上刺的尖端還生活著微小的豪豬，乃至無限。NP 難題通常會卡在有指數級數量的局部極大值。儘管如此，經過少數隨機重新開始的搜尋之後還是能找到一個合理的較好的局部極大值的。

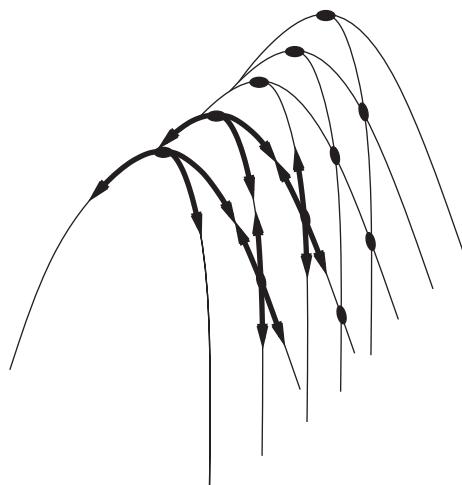


圖 4.4 為什麼山脊會給爬山法帶來困難的圖示。圖中的狀態網格(黑色圓點)疊加在從左到右上升的山脊上，創造了一個彼此不直接相連的局部極大值序列。從每個局部極大值出發，所有的可能行動都是指向下方山方向

4.1.2 模擬退火

爬山法搜尋演算法從來不「下山」，即不會向值比當前節點低的(或成本高的)方向搜尋，它肯定是不完備的，因為它可能停留在局部極大值上。相反地，純粹的隨機行走——就是從後繼者集合中均勻的隨機選取一個後繼者，然後移動到這個選出來的後繼者——是完備的，但是毫無效率可言。因此，試圖把爬山法和隨機行走以某種方式結合起來，同時得到效率和完備性的想法是合理的。**模擬退火(simulated annealing)**就是這樣一個演算法。在冶金中，**退火**是為了增強金屬和玻璃的韌性或硬度而先把它們加熱到高溫再讓它們逐漸冷卻的過程，能使材料結合成一個低能量的結晶態。為了更好地理解模擬退火演算法，讓我們把視線從爬山法轉移到**梯度下降**(也就是成本最小化)，想像使不平表面上的一個乒乓球掉到最深的裂縫中的任務。如果我們只是讓乒乓球在表面上滾動，那麼它會停留在局部極小點。如果我們晃動平面，我們可以使乒乓球彈出局部極小點。技巧是晃動要足夠大讓乒乓球能從局部極小點彈出來，但又不能太大把它從全局最小點趕出來。模擬退火的解決辦法就是開始時使勁搖晃(也就是先高溫加熱)然後慢慢降低搖晃的強度(也就是逐漸降溫)。

模擬退火演算法最核心的迴圈(圖 4.5)與爬山法很像。不過它沒有選擇最佳的移動，而是選擇隨機的移動。如果該移動使情況改善，那麼該移動總是被接受。否則，演算法以某個小於 1 的概率接受該移動。這個概率按照該移動的「惡劣狀態」——評價值變壞的梯度 ΔE ——呈指數級下降。這個概率同時也隨「溫度」 T 降低而下降：當一開始溫度高的時候，「壞的」移動更可能被允許發生， T 越低就越不可能發生。如果排程讓 T 下降得足夠慢，這個演算法將找到機率趨近 1 的全局最佳解。

模擬退火演算法首先在 20 世紀 80 年代早期廣泛用於求解 VLSI 佈局問題。現在它已經廣泛地應用於工廠排程和其他大型的最佳化問題。在習題 4.3 中，要求你在八皇后問題中比較它和隨機重新開始爬山法演算法的性能。

```

function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
           schedule, a mapping from time to “temperature”

  current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to  $\infty$  do
    T  $\leftarrow$  schedule(t)
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  next.VALUE - current.VALUE
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 
```

圖 4.5 模擬退火演算法，一個允許下山的隨機爬山法演算法。在退火初期下山移動容易被接受，而隨時間推移下山的次數越來越少。輸入的 *schedule* 決定了為時間函數的溫度 *T* 的值

4.1.3 局部剪枝搜尋

在記憶體中只保存一個節點，看來是對記憶體限制問題的一個極端反應。**局部剪枝搜尋(local beam search)**演算法^[3]記錄 k 個狀態而不是一個。它由 k 個隨機產生的狀態開始。每一步都產生全部 k 個狀態的所有後繼者狀態。如果其中有一個是目標狀態，演算法就停止。否則，它從整個後繼者列表中選擇 k 個最佳的後繼者，然後重複這個過程。

乍看之下， k 個狀態的局部剪枝搜尋只不過是並列地而不是串列地執行 k 個隨機重新開始搜尋。實際上，這兩個演算法是相當不同的。在隨機重新開始搜尋中，每個搜尋過程的執行與其他搜尋過程是獨立的。而在局部剪枝搜尋中，有用的資訊在這並列的搜尋引線之間傳遞。那麼效果上，產生最好的後繼者的狀態，會對其他狀態說：「來這兒吧，這兒的草更綠！」演算法很快放棄沒有成果的搜尋而把資源都用在取得了最大進展的搜尋上。

在其最簡單的形式中，局部剪枝搜尋會受到這 k 個狀態缺乏多樣性的影響——它們將很快聚集到狀態空間中的一小塊區域內，使得搜尋比一個代價高昂的爬山法版本略多一些。這個演算法的一個變化稱為**隨機剪枝搜尋**，類似於隨機爬山法，幫助緩解這個問題。隨機剪枝搜尋不是從候選後繼者集合中選擇最好的 k 個後繼者狀態，而是按一定概率隨機地選擇 k 個後繼者狀態，其中選擇給定後繼者狀態的概率是狀態值的遞增函數。隨機剪枝選擇和自然選擇的過程有些相似性，「狀態」(生物體)根據它的「值」(適應性)產生它的「後繼者」(後代)作為下一代。

4.1.4 基因遺傳演算法

基因遺傳演算法(genetic algorithm，或縮寫為 **GA**)是隨機剪枝搜尋的一個變化形式，它不是透過修改單一狀態而是透過把兩個父代結合來產生後繼者的。它與自然選擇類似，這點和隨機剪枝搜尋是一樣的，除了我們現在處理的是有性繁殖而不是無性繁殖。

像剪枝搜尋一樣，基因遺傳演算法也是從 k 個隨機產生的狀態開始，我們稱作**種群**。每個狀態，或稱**個體**，用一個有限長度的字串表示——通常是 0、1 字串。例如，八皇后問題的狀態必須指定八個皇后的位置，每列有八個位置，那麼一個狀態則需要 $8 \times \log_2 8 = 24$ 位元來表示。或者每個狀態可以由八個數字表示，每個數字的範圍都是從 1 到 8。(我們將在後面看到這兩種不同的編碼形式其表現差異很大。圖 4.6(a)顯示了一個由 4 個表示八皇后狀態的 8 數字字串組成的種群。

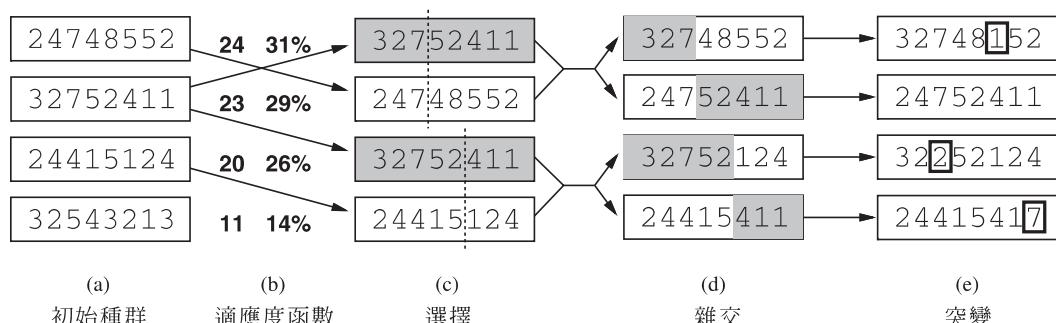


圖 4.6 基因遺傳演算法，圖解說明以數字字串表示八皇后的狀態。(a)是初始種群，(b)是適應度函數，(c)是配對結果。(d)是它們雜交產生的後代，(e)是突變的結果

圖 4.6(b)-(e)顯示了產生下一代狀態的過程。在(b)中，每個狀態都由它的目標函數或適應度函數(在基因遺傳演算法的術語裡)給出評價值(適應值)。對於好的狀態，適應度函數將返回較高的值，因此，對於八皇后問題，我們用不相互攻擊的皇后對的數目來表示，最佳解的適應值是 28。這四個狀態的適應值分別是 24, 23, 20 和 11。在這個特定的基因遺傳演算法形式中，被選擇進行繁殖的概率直接與個體的適應值成正比，其百分比在原始得分旁邊標出。

在(c)中，按照(b)中的機率隨機地選擇兩對進行繁殖。注意其中的一個個體被選擇了兩次而另一個一次也沒選中^[4]。對於要配對的每一對個體，雜交點是在字串中隨機選擇的一個位置。在圖 4.6 中，雜交點在第一對的第三數字之後和在第二對的第五數字之後^[5]。

在(d)中，後代本身是透過父代字串在雜交點上進行雜交而創造出來的。例如，第一對的第一個後代從第一個父代得到了前三數字、從第二個父代那裡得到剩下的數字，而第二個後代從第二個父代得到了前三數字、從第一個父代得到剩下數字。圖 4.7 顯示了這次繁殖過程中涉及的八皇后狀態。這個例子顯示一個事實，如果兩個父代字串差異很大，那麼雜交產生的狀態和每個父代狀態都相差很遠。通常的情況是過程中早期的群體是多樣化的，因此雜交(類似於模擬退火)在搜尋過程的早期階段在狀態空間中常採用較大的步調，而在後來當大多數個體都很相似的時候採用較小的步調。

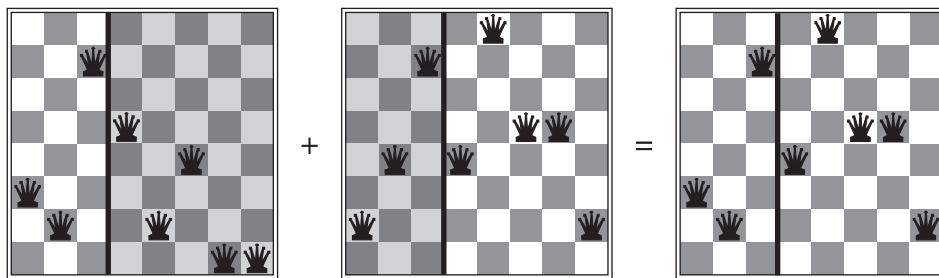


圖 4.7 與圖 4.6(c)中的前兩個父代還有圖 4.6(d)中的第一個後代相對應的八皇后問題的狀態。陰影部分的列在雜交過程中丟失了，非陰影部分的列保留了下來

最後，在(e)中每個位置都會按照一個獨立的小概率隨機地變異。在第一、第三和第四個後代中都有一個數字發生了突變。在八皇后問題中，這相當於隨機地選取一個皇后把它隨機地放到該列的某一個方格裡。圖 4.8 描述了一個實作了所有這些步驟的演算法。

像隨機剪枝搜尋一樣，基因遺傳演算法結合了上山的趨勢和隨機的探索，並在並列搜尋引線之間交換資訊。基因遺傳演算法最主要的優點，如果有的話，還是來自於雜交的操作。不過可以在數學上證明，如果基因編碼的位置在初始的時候就隨機地轉換的話，雜交就沒有優勢了。直觀地講，雜交的優勢在於它能夠將獨立發展出來而能執行有用功能的「磚塊」(多個相對固定的字元構成)結合起來，因此提高了搜尋操作的解析度水準。例如，將前三個皇后分別放在位置 2、4 和 6(互相不攻擊)就組成了一個有用的磚塊，它可以和其他有用的磚塊結合起來構造問題的解。

```

function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
           FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new-population  $\leftarrow$  empty set
    for i = 1 to SIZE(population) do
      x  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      y  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      child  $\leftarrow$  REPRODUCE(x, y)
      if (small random probability) then child  $\leftarrow$  MUTATE(child)
      add child to new-population
    population  $\leftarrow$  new-population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN



---


function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n  $\leftarrow$  LENGTH(x); c  $\leftarrow$  random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))

```

圖 4.8 基因遺傳演算法。這個演算法和圖 4.6 中所示的演算法是一樣的，除了一個變化：在這個更流行的版本中，兩個父代字串的每次配對只產生一個後代而不是兩個

基因遺傳演算法的理論用模式(schema)的想法解釋了這是怎樣運轉的，模式就是其中某些數字未確定的一個子字串。例如，模式 246****描述了所有前三個皇后的位置分別是 2、4、6 的狀態。能匹配這個模式的字串(例如 24613578)被稱作該模式的實例。可以顯示，如果一個模式的實例的平均適應值是超過均值的，那麼種群內這個模式的實例數量就會隨時間增加。顯然，如果鄰近位元相互之間是無關的，那麼這個效果就沒有那麼顯著了，因為只有很少的鄰接磚塊能提供一致的好處。基因遺傳演算法在模式與解的有意義的成份相對應時才工作得最好。例如，如果字串表示的是一個天線，那麼這些模式就表示天線的組成部分，諸如反射器和偏轉儀。一個好的組成部分在各種不同設計下很可能都是好的。這說明基因遺傳演算法的成功運用需要仔細的知識表示工程。

實際上，基因遺傳演算法在最佳化問題上有很廣泛的影響，諸如電路佈局和零工型工廠間排程問題。現在，還不清楚基因遺傳演算法的吸引力是源自其性能，還是源自其美學角度滿足了進化論的形式。為了確認在什麼情況下使用基因遺傳演算法能夠達到好的效果，還有很多研究工作要做。

進化與搜尋

進化論是查理斯·達爾文(Charles Darwin)在《物種起源》(*On the Origin of Species by Means of Natural Selection*, 1859)(編註：直譯為《透過自然選擇方式的物種起源》)一書中提出的。它的核心想法很簡單：變化出現在繁殖過程中，並且將在後代中保存下來，與它們對繁殖適應度的影響呈近似比例。

達爾文的理論並沒有揭示出生物體的特性是怎樣遺傳下來或改變的。控制這個過程的機率規律首先是由一個叫葛列格·孟德爾(Gregor Mendel, 1866)的修道士發現的，他在甜豌豆上進行了實驗。很久以後，Watson 和 Crick(1953)確定了DNA分子的結構及其序列，AGTC(腺嘌呤，鳥嘌呤，胸腺嘧啶，胞嘧啶)。在標準的模型中，基因序列上的某點的突變或者「雜交」(後代的DNA序列是透過雙親DNA序列長片段的合成產生的)都會導致發生變異。

與局部搜尋演算法的相似性我們已經介紹過了；隨機剪枝搜尋演算法與進化演算法之間最主要的區別就是對有性繁殖的利用，在有性繁殖中後代是由多個而不是一個生物體產生的。然而，實際的進化機制比多數基因遺傳演算法要豐富得多。例如，變異就包括DNA的反轉、複製和大段的移動；一些病毒借用一個生物體的DNA，再插入到其他生物體的DNA裡；在基因組裡還有可換位基因，它除了把自己複製成千上萬遍以外不做其他事情。甚至還有些基因破壞不攜帶該基因的細胞以避免可能的配對，從而提高它們本身的複製機率。最重要的是基因本身對基因控制進行編碼，這些編碼決定了基因組是如何繁殖和轉變成生物體的。在基因遺傳演算法中，這些機制是單獨的程式，而不是表現在被處理的字串中。

達爾文的進化論，可能會出現效率低下，一點兒都沒有改進它的搜尋啟發函數而盲目地產生了大約 10^{45} 種生物。然而，比達爾文早50年，另一位偉大的法國自然科學家讓·拉馬克(Jean Lamarck, 1809)提出了一種進化理論，在生物的一生中透過適應獲得的特性將會傳給它的後代。這個過程是很有效率的，但是在自然界裡似乎不太可能發生。很多年之後，詹姆斯·鮑德溫(James Baldwin, 1896)提出了一個更淺顯的類似理論：在生物體存活的時間裡學習到的行為能夠加快進化速度。鮑德溫的理論與拉馬克的不同，而與達爾文的進化論卻是完全一致的，因為它依賴於作用在個體上的選擇壓力，這些個體會在遺傳天性允許的可能行為集合上找到局部最佳。電腦模擬確認「鮑德溫效應」是真實的，「普通的」進化曾經創造出內在性能度量和實際適應性相關的生物體。

4.2 連續空間的局部搜尋

在第二章中，我們解釋了離散和連續環境的區別，指出大多數現實世界環境都是連續的。不過，我們描述過的(除了首選爬山法及模擬退火)演算法沒有一個能夠處理連續狀態及行動空間，因其有無限多個分支因數。這一節將對一些在連續狀態空間上尋找最佳解的局部搜尋技術進行非常簡要的介紹。關於這個主題的文獻很多；許多基本技術起源於17世紀，在牛頓和萊布尼茲發明微積分之後^[6]。我們會在本書的一些地方發現這些技術的應用，包括在學習、視覺和機器人技術這幾章。

4-10 人工智慧 – 現代方法 3/E

我們從一個例子開始。假設我們想在羅馬尼亞建三個新機場，使地圖上(圖 3.2)每個城市到離它最近的機場的距離平方和最小。問題的狀態空間接著就由機場的座標來定義： (x_1, y_1) ， (x_2, y_2) 和 (x_3, y_3) 。這是一個六維空間；我們也可以說狀態空間由六個變數定義。(一般來講，狀態是由一個 n 維的變數向量 \mathbf{x} 來定義的)。在這個狀態空間中移動相當於在地圖上移動一個或多個機場的位置。對於一個特定的狀態一旦計算出最近的城市，目標函數 $f(x_1, y_1, x_2, y_2, x_3, y_3)$ 就很容易計算出來了。令 C_i 為一組城市，其最接近的機場(在當前狀態)為機場 i 。然後，在當前狀態的鄰接狀態，在這裡 C_i 保持不變，我們可得到

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^3 \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2 \quad (4.1)$$

這個運算式是局部正確的，但不是全局，因為 C_i 組是狀態的(不連續)函數。

一個簡單的避免連續性問題的途徑就是將每個狀態的鄰接狀態離散化。例如，我們一次只能將一個飛機場按照 x 方向或 y 方向移動一個固定的量 $\pm\delta$ 。對於 6 個變數，每個狀態有 12 個可能的後繼者。然後我們就可以應用之前描述過的任何一個局部搜尋演算法。也可以不離散化空間就直接應用隨機爬山法和模擬退火。這些演算法隨機地選擇後繼者，透過隨機產生長度 δ 的向量來完成。

許多方法試圖利用地形圖的梯度來找到最大值。目標函數的梯度是一個向量 ∇f ，它給出了最陡斜面的大小和方向。對於我們的問題，有

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

在一些情況下，我們可以透過解方程式 $\nabla f = 0$ 來找到最大值。(這是可以做到的，例如，如果我們只放一個飛機場；那麼解就是所有城市座標的算術平均值)。然而在很多情況下，這個等式不能用封閉的形式解決。例如，當有三個飛機場時，梯度的運算式依賴於當前狀態下哪些城市距離各個機場最近。這意味著我們只能局部地計算梯度而不能全局地計算。

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_1 - x_c) \quad (4.2)$$

給予局部正確運算式的梯度，我們可以透過下述公式更新當前狀態來執行最陡上升爬山法：

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$$

其中 α 是一個很小的常數，通常被稱為步長。在其他情況下，目標函數可能根本不能用微分形式表示——例如，特定的機場位置集的值要透過執行某個大規模經濟模擬套裝程式來決定。在這些情況下，所謂的經驗梯度可以取決於每個座標上的小增減量引起的反應。在離散化的狀態空間中，經驗梯度搜尋和最陡上升爬山法是一樣的。

在習慣用語「 α 是一個很小的常數」背後，隱藏著非常多調整 α 的不同方法。基本的問題是，如果 α 太小，就需要太多的步驟；如果 α 太大，則搜尋容易錯過最大值。線搜尋技術試圖透過擴展當前的梯度方向——通常透過反覆使 α 加倍——來克服這個困境，直到 f 開始再次下降。發生的點成為新的當前狀態。關於在這點上應該如何選擇新方向有幾個不同學派的想法。

對於許多問題，最有效的演算法是古老的**牛頓-拉夫森**方法。這是一個找到函數的根的一般方法——即求解形態如 $g(x) = 0$ 的方程式。根據牛頓公式計算根 x 的一個新的估計值

$$x \leftarrow x - g(x)/g'(x)$$

為了找到 f 的最大或最小值，我們需要找到梯度為 0 的 \mathbf{x} (亦即 $\nabla f(\mathbf{x}) = \mathbf{0}$)。因此，牛頓公式中的 $g(x)$ 就變成 $\nabla f(\mathbf{x})$ ，並且更新過的方程式可以寫成矩陣向量形式

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x})$$

其中 $\mathbf{H}_f(\mathbf{x})$ 是二階導數的 **Hessian** 矩陣，矩陣中的元素 H_{ij} 由 $\partial^2 f / \partial x_i \partial x_j$ 紿定。對於我們的機場例子，我們可以從公式(4.2)看到 $\mathbf{H}_f(\mathbf{x})$ 是特別簡單：非對角線元素為零，而代表機場 i 的對角線元素剛好是 C_i 城市中數目的兩倍。瞬間的計算顯示，一步更新的移動機場 i 直接到 C_i 的中心，這是公式(4.1)^[7] 中 f 的局部表達最小值。然而對於高維問題，運算及反運算 Hessian 矩陣的 n^2 個元素，成本是昂貴的，因此很多牛頓-拉夫森近似方法被提出來。

局部搜尋演算法在連續狀態空間和離散狀態空間都一樣受到局部極大值、山脊、高原的影響。隨機重新開始和模擬退火都可以使用，並且經常都很有用。然而高維的連續空間，在這種大的地方很容易使這種演算法迷失。

最後一個討論的有用話題是**限制最佳化**。若問題的解須滿足對於每個變數值的某些嚴格限制，則這個最佳化問題是受限制的。例如，在我們的機場選址問題中，我們可以限制機場的位置是在羅馬尼亞境內，並且是在旱地上(而不是在湖中央)。限制最佳化問題的難點取決於限制和目標函數的性質。最著名的一種限制最佳化問題是**線性規劃**問題，其限制必須是線性不等式並且能夠組成一個**凸多邊形集**^[8]，同時目標函數也是線性的。線性規劃中的時間複雜度，顯現於其多項式中的變數數。

線性規劃是最廣為研究和最佳化問題的廣泛使用類型。這是一個一般**凸多邊形最佳化**問題的特殊情況，凸多邊形在限制區域內，允許限制區域成為任何凸多邊形及目標成為任何函數。在某些情況下，凸多邊形最佳化問題也可用多項式求解，並在成千上萬的變數實務中或許是可行的。在機器學習與控制理論中的幾個重要問題可歸結為凸多邊形最佳化問題(見第 20 章)。

4.3 不確定性行動的搜尋

在第三章中我們假設問題的環境是完全可觀察的和確定性的，而且代理人瞭解每個行動的效果。因此，代理人可以準確的計算出從任何行動序列的狀態結果，並且總是知道它所處的狀態。雖然他們當然告訴了代理人初始狀態，但在每個行動之後並沒有提供新的知覺訊息。

當環境為部分可觀察的或不確定性的(或兩者)時候，知覺變得非常有用。在部分可觀察的環境中，每個知覺有助於縮小代理人可能存在的狀態組，從而使得代理人更容易完成其目標。當環境為不確定性時，知覺告訴代理人，已實際發生行動的可能結果。在這兩種情況下，未來的知覺不能事先確定，代理人的未來行動將取決於這些未來的知覺。因此，問題的解並不是一個序列，而是一個**偶發/應變規劃**(也稱為**策略**)，它指定該怎麼做則取決於收到什麼知覺。在本節中，我們檢視不確定性的例子，部分可觀察性則順延到 4.4 節。

4.3.1 不穩定的吸塵器世界

我們使用吸塵器世界作為例子，首次在第二章中介紹，並在第 3.2.1 節定義為搜尋問題。回想一下，它的狀態空間有八個狀態，如圖 4.9 所示。一共有三個行動——*Left*, *Right* 和 *Suck*——目標是清掃完所有的灰塵(狀態 7 和狀態 8)。如果它的環境是可觀察的、確定性的，並且是完全已知的，那麼這問題用第三章的任何一個演算法都能輕易地解決，其解則是一個行動序列。例如，如果初始狀態是 1，那麼行動序列 [*Suck*, *Right*, *Suck*] 就能到達目標狀態 8。

現在假設我們引入不確定性的，一個強大但不穩定的吸塵器形式。在**不穩定吸塵器世界**，*Suck*(吸)行動的工作原理如下：

- 當應用到一個骯髒的方格時，行動清掃此方格，有時也會清掃相鄰方格內的灰塵。
- 當應用到一個乾淨的方格時，行動有時會沉積灰塵在地毯上^[9]。

為了提供一個準確的表述這個問題，我們需要從第三章將轉換模式轉成一般化的概念。不是由返回單一個狀態的結果(RESULT)函數來定義轉換模式，我們使用可返回一組可能結果狀態的結果函數。例如，在不穩定的吸塵器世界，在狀態 1 吸的行動引導到在集合 {5, 7} 中的一個狀態——在右邊方格中的灰塵可能會或可能不會被吸塵了。

我們還需要將問題的解轉成一般化的概念。例如，如果我們在狀態 1 開始，沒有單一行動序列可解決此問題。相反地，我們需要一項如以下的偶發規劃：

[*Suck*, if *State* = 5 then [*Right*, *Suck*] else []] (4.3)

因此，不確定性問題的解可以包含巢狀 **if-then-else** 語句，這意味著它們是樹(tree)，而不是序列。這使得選擇行動可基於執行過程中產生的偶發事件。許多在真實、實體世界的問題都是偶發性問題，因為確切的預測是不可能的。由於這個原因，很多人在走路或者開車時都會睜開眼睛。

4.3.2 AND-OR 搜尋樹

接下來的問題是如何找到偶發不確定性問題的解。如同第三章，我們首先構建搜尋樹，但這裡的樹有不同的特性。在確定性環境中，唯一的分支，是在每個狀態中由代理人自己所選擇引入。我們稱這些節點為 **OR(或)節點**。例如在吸塵器世界，在一個 OR 節點代理人選擇 *Left*(左)或 *Right*(右)或 *Suck*(吸)。在不確定性的環境中，分支也是在每個行動的結果由環境的選擇引入。我們稱這些節

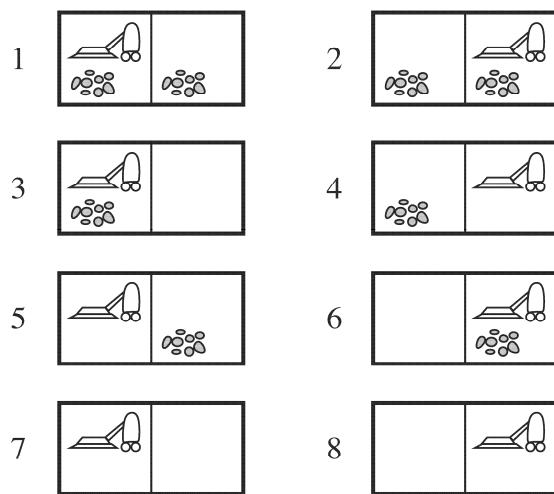
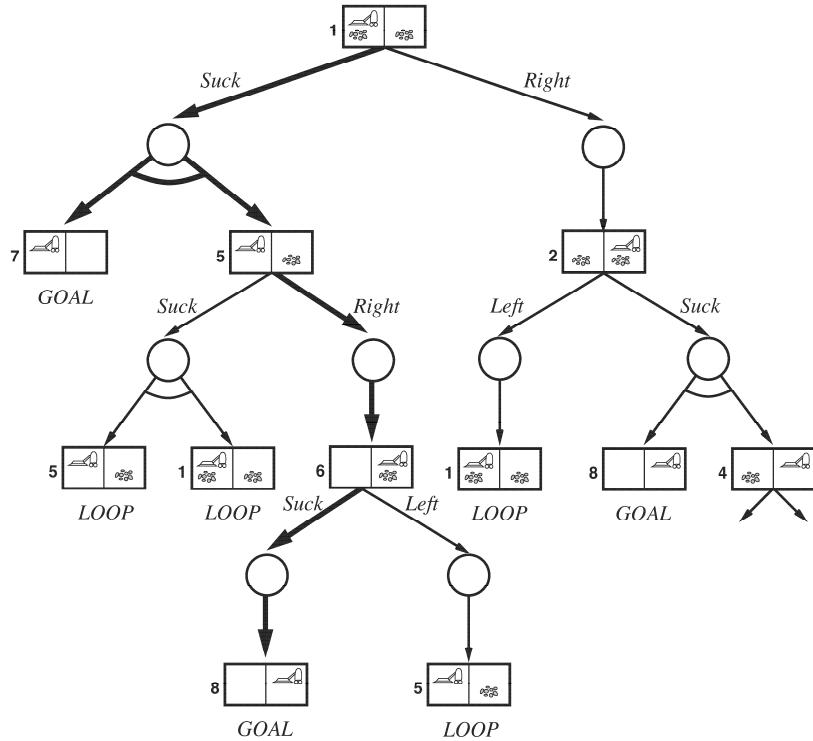


圖 4.9 吸塵器世界的八個可能狀態；狀態 7 和 8 是目標狀態。

點為 **AND(及)節點**。例如，在狀態 1 吸的行動引導到在集合{5, 7}中的一個狀態，因此代理人需要找到一個狀態 5 和狀態 7 的規劃。這兩種類型的節點之間交錯陳列，引導到及-或樹(AND-OR tree)，如圖 4.10 所示。

圖 4.10 不穩定的吸塵器世界搜尋樹的前兩層。狀態節點是某個行動必須被選擇的 OR(或)節點。在顯示為圓圈的 AND 節點處，每個結果都必須被處理，如外射分支的連接弧所示。解用粗線顯示



AND-OR 搜尋問題的解是一棵子樹，(1) 在每個葉子上都有一個目標節點，(2) 在它的每個 OR 節點都指定了一個行動，(3) 在它的每個 AND 節點都包含每個結果分支。圖中用粗線顯示出解，它對應於公式(4.3)的規劃(該規劃使用 if-then-else 符號來處理 AND 分支，但是當一個節點有兩個以上的分支時，此時或許最好使用 CASE 結構)。如圖 3.1 所示，修改代理人基本問題的解，所執行這種偶發的解是很直接的。你也可以考慮不同的代理人設計，其中代理人可以在它找到保證規劃之前就採取行動，並處理一些只在執行過程中出現的偶發事件。這種搜尋和執行的交錯(interleaving)對探索問題(參見第 4.5 節)和賽局問題(參見第五章)也是很有用的。

圖 4.11 紿出了 AND-OR 圖搜尋的一個遞迴的深度優先演算法。演算法的一個關鍵方面是它處理環的方法，環經常出現在不確定性規劃問題中(例如，當一個行動有時沒有效果時，或者當一個非預期的效果可能正確時)。如果當前狀態與從根節點開始的一條路徑上的一個狀態完全相同，那麼它以失敗返回。這並不意味著從當前狀態沒有解；它只是簡單地意味著如果有無循環解，它肯定能從當前狀態的早期具體化到達，所以新的具體化可以被丟棄。用這種檢驗方法，我們確保演算法在每個有限的狀態空間中能終止，因為每條路徑一定到達一個目標、死路或者一個重複狀態。注意演算法沒有檢驗當前狀態是否為某條從根節點開始的其他路徑上的狀態的重複，這對效率而言是非常重要的。習題 4.4 研究了這個問題。

AND-OR 圖也可以用廣度優先或最佳優先方法探索。這個啟發函數的概念必須加以修改，以估計偶發而不是序列解的成本，但進行了可採納的概念，並類似 A*演算法的尋找最佳解。一些提示提供在本章末的參考文獻中。

```

function AND-OR-GRAFH-SEARCH(problem) returns a conditional plan, or failure
    OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
    if problem.GOAL-TEST(state) then return the empty plan
    if state is on path then return failure
    for each action in problem.ACTIONS(state) do
        plan  $\leftarrow$  AND-SEARCH(RESULTS(state, action), problem, [state | path])
        if plan  $\neq$  failure then return [action | plan]
    return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
    for each si in states do
        plani  $\leftarrow$  OR-SEARCH(si, problem, path)
        if plani = failure then return failure
    return [if s1 then plan1 else if s2 then plan2 else ... if sn-1 then plann-1 else plann]

```

圖 4.11 用於不確定性環境產生的 AND-OR 圖的一個搜尋演算法。它回傳一個有條件的規劃，其在所有的環境下到達目標狀態(符號[*x* | *l*]列表，是在 *l* 列表的前面加入目標 *x* 而得)

4.3.3 嘗試，繼續嘗試

考慮光滑的吸塵器世界，除了有時運動的行動會失敗外，它等同於普通的(非不穩定)吸塵器世界，留下代理人在同一位置。例如，移動狀態 1 的 *Right*(右)到狀態集 {1, 2}。圖 4.12 顯示了部分搜尋圖。清楚地，不再有任何狀態 1 的無循環解，AND-OR-GRAFH-SEARCH 會返回失敗。然而，存在有循環解，即不斷嘗試 *Right* 直到它可行。我們可以透過添加一個用來表示規劃某些部分的標籤並稍後用這個標籤而不是重複規劃本身的方法來表示這個解。這樣，我們的有循環解是

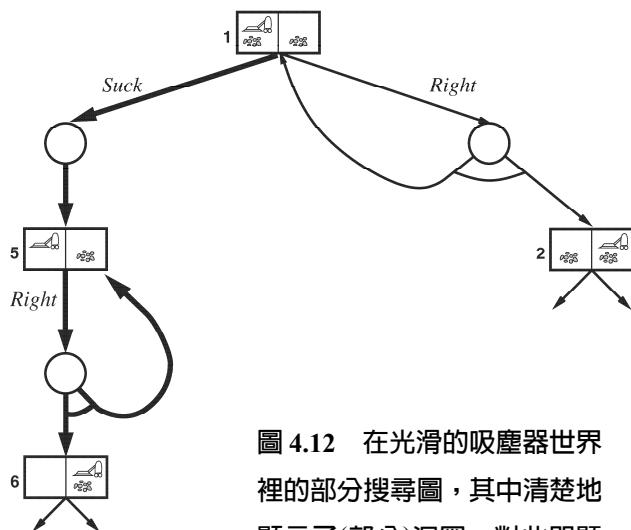


圖 4.12 在光滑的吸塵器世界裡的部分搜尋圖，其中清楚地顯示了(部分)迴圈。對此問題的所有解是循環規劃，因為沒有行動的可靠辦法。

[*Suck, L₁ : Right, if State = 5 then L₁ else Suck*]

(一個用來表示這個規劃的迴圈部分的更好語法是「**while** $State = 5$ **do** Right」)。一般來說一個循環規劃可被視為一種解，它提供每個葉子都是一個目標狀態，且從規劃中的每一個點都可以到達葉子。習題 4.5 論及了 AND-OR-GRAH-SEARCH 所需要的修改。關鍵的實現是狀態空間中返回到狀態 L 的一個迴圈轉換成爲規劃中返回到狀態 L 的子規劃被執行的點的一個迴圈。

給循環解下個定義，代理人執行這樣的解，最終會到達目標，每一個不確定性行動的結果，終究會發生。這種情況合理嗎？這取決於不確定性的原因。如果行動爲擲骰子，那麼它的合理假設，最終將擲出六。如果行動是插入卡片鑰匙到酒店的門鎖，但它第一次並沒有效，然後也許最終會有效，或者鑰匙是錯誤的(或房間是錯誤的！)。經過七、八次嘗試後，多數人會認爲問題是鑰匙，並回到櫃台換一個新的。一種理解這個決定的方法是說，最初丟棄問題的形式化(可觀察的，不確定性的)，是爲支持不同的形式化(部分可觀察的，確定性的)，其失敗是因爲鑰匙不可觀察的性質。在第 13 章我們有更詳細的說明這個課題。

4.4 使用部分觀察的搜尋

我們現在轉到部分可觀察性的問題，其中代理人的知覺並不足以牽制確切的狀態。如同上一節開始時所述，如果代理人是在數個可能的狀態之一——即使是在確定性的環境，則一個行動可能會導致幾個可能的結果之一。解決部分可觀察的問題，需要的關鍵概念是**信度狀態**，代表代理人對其可能存在的實體狀態的當前信念，給予行動序列和到該點的知覺。我們首先從最簡單的情況下研究信度狀態，這是當代理人完全沒有感測器，然後我們加入部分感測以及不確定性的行動。

4.4.1 使用沒有觀察的搜尋

當代理人的知覺沒有提供任何資訊時，即我們所謂的**無感測器問題**，或者有時稱爲**一致性問題**。起初，人們可能會認爲，如果不知道是在什麼狀態，無感測器代理人將沒有希望解決問題，事實上，無感測器問題經常是可解的。此外，無感測器代理人更是出奇地有用的，主要是因爲他們並不依賴於感測器的正常工作。在製造系統，例如，在完全沒有感測下，從一個未知的起始位置使用一行動序列，以正確零件爲導向的許多巧妙方法已被開發。感測的高成本是另一個避免它的原因：例如，醫生經常開廣譜抗生素，而不是使用應變規劃：先進行昂貴的血液測試，然後等著結果出來，再開更特效的抗生素，或是因爲已感染太廣而住院。

我們可以營造一個無感測器的吸塵器世界。假設代理人知道其世界地理，但不知道其位置或灰塵的分佈。在這種情況下，它的初始狀態可以是集合 $\{1, 2, 3, 4, 5, 6, 7, 8\}$ 中的任何元素。現在，如果它嘗試 Right(右)的行動，考慮會發生什麼。這將導致它處在 $\{2, 4, 6, 8\}$ 中的一個狀態——現在代理人有更多的資訊！此外，行動序列[Right, Suck]總是結束在 $\{4, 8\}$ 中的一個狀態。最後，不管起始狀態是什麼，序列[Right, Suck, Left, Suck]保證到達目標狀態 7。我們說代理人可以迫使世界進入狀態 7。

要解決無感測器問題，我們要在信度狀態空間中搜尋，而不是在實體狀態空間中搜尋^[10]。請注意，在信度狀態空間，問題是完全可觀察的，因爲代理人總是知道自己的信度狀態。此外，該解(如果有的話)總是一個行動序列。這是因爲，作爲第三章的一般問題，每個行動之後收到的知覺是完全可預測的——他們總是空的！因此，並沒有偶發規劃。這是真的，即使環境是不確定性的。

4-16 人工智慧－現代方法 3/E

看看信度狀態搜尋問題是如何構建的將會有所助益。假設基本實體問題 P 是由 ACTIONS_P , RESULT_P , GOAL-TEST_P 和 STEP-COST_P 定義。那麼，我們可以定義相應的無感測器問題如下：

- **信度狀態**：整個信度狀態空間包含所有可能的實體狀態集。如果 P 有 N 個狀態，那麼無感測器問題會達到 2^N 個狀態，儘管從初始狀態有許多是不能到達的。
- **初始狀態**：通常為在 P 中的所有狀態集，雖然在某些情況下，代理人有比此具備更多的知識。
- **行動**：這個有點難。假設代理人是在信度狀態 $b = \{s_1, s_2\}$ ，但 $\text{ACTIONS}_P(s_1) \neq \text{ACTIONS}_P(s_2)$ ，則代理人不確定哪些行動是合法的。如果我們假設非法行動對環境並沒有影響，則在當前信度狀態 b ，對任何實體狀態中的行動採取聯集是安全的：

$$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTION}_P(s)$$

另一方面，如果一個非法行動可能導致世界末日，只允許交集是較安全的，即在所有狀態的行動集是合法的。對於吸塵器世界，每個狀態都有相同的合法行動，因此這兩種方法提供了相同的結果。

- **轉移模型**：代理人不知道在信度狀態中的哪個狀態是正確的，所以就它所知，它可能會到達任何從信度狀態中的一個實體狀態採取行動所導致的狀態。對於確定性的行動，狀態集可能到達的是

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ 及 } s \in b\} \quad (4.4)$$

對於確定性的行動， b' 是永遠不會比 b 大。對於不確定性，我們有

$$\begin{aligned} b' &= \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ 及 } s \in b\} \\ &= \bigcup_{s \in b} \text{RESULT}_P(s, a) \end{aligned}$$

這可能比 b 大，如圖 4.13 所示。行動之後產生新的信度狀態的過程稱為預測步驟；符號 $b' = \text{PREDICT}_P(b, a)$ 將派上用場。

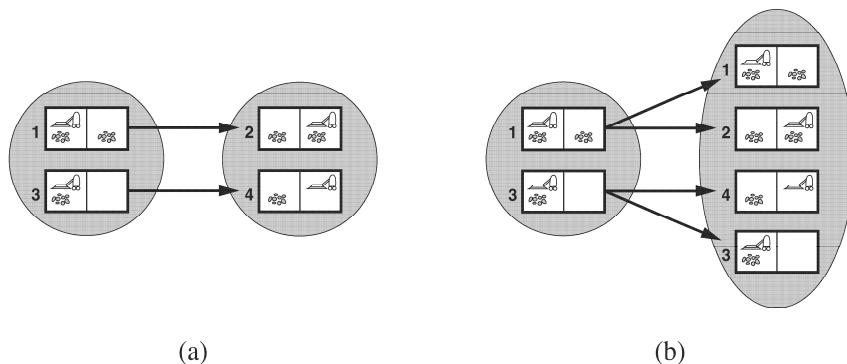


圖 4.13 (a) 以一個確定性的行動，Right，預測無感測器吸塵器世界的下一個信度狀態。(b) 在光滑的無感測器吸塵器世界預測相同的信度狀態和行動

- **目標測試**：代理人需要一個會工作的規劃，這意味著一個信度狀態，只有在所有的實體狀態滿足 $GOAL-TEST_P$ ，才滿足目標。代理人可能會偶然地提前到達目標，但它不會知道它有這樣做。
- **路徑成本**：這個也有點難。若同樣的行動在不同狀態可以有不同的成本，則在一個給定的信度狀態採取行動的成本，可能是幾個值之一(這就產生了一個新類型的問題，我們將在習題 4.8 探討)。現在我們假設一個行動的成本在所有的狀態是一樣的，所以可以直接從基本實體問題轉移。

圖 4.14 顯示了確定性的和無感測器的吸塵器世界中可到達的信度狀態空間。在 $2^8 = 256$ 個可能的信度狀態中，只有 12 個可到達的信度狀態。

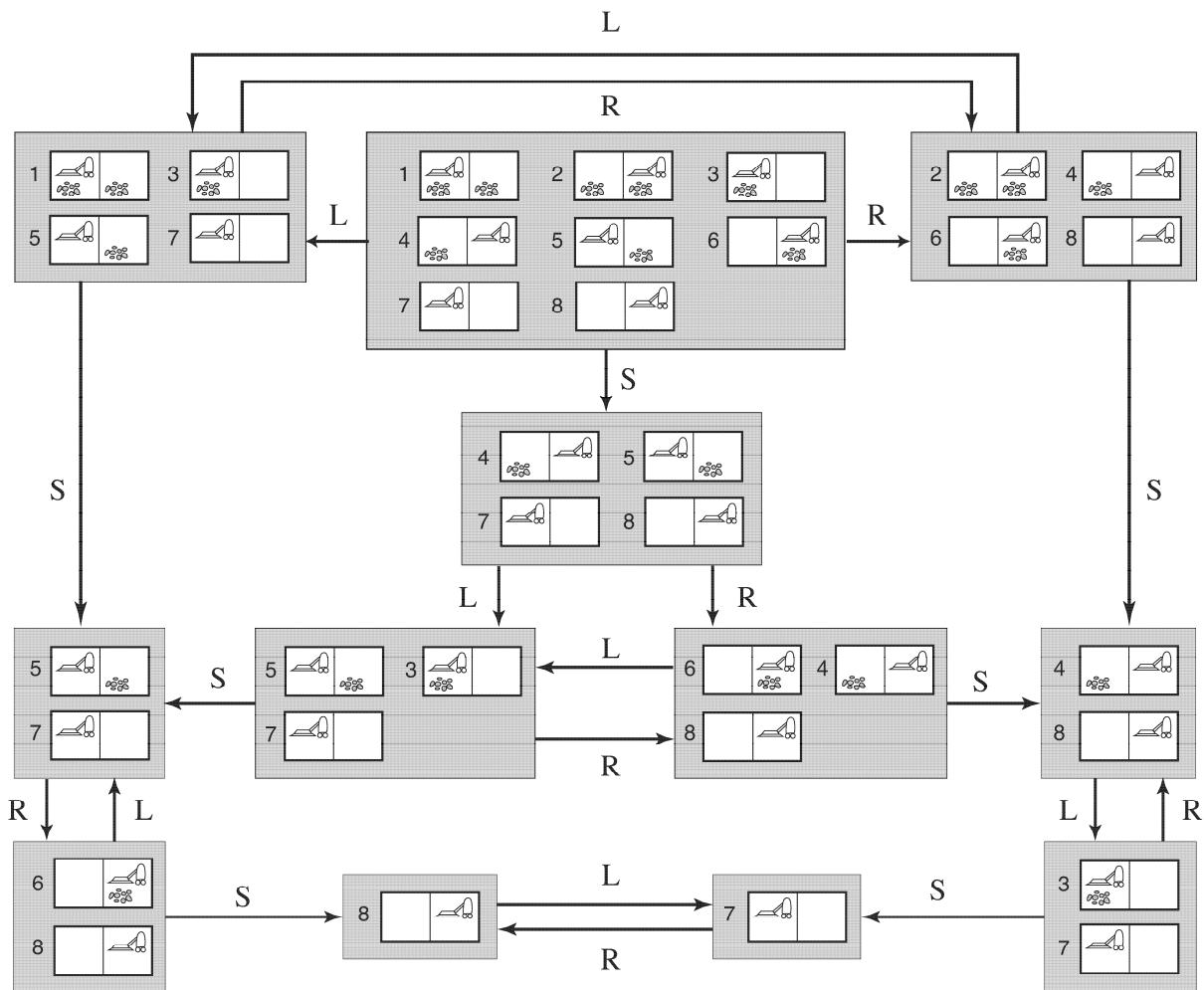


圖 4.14 確定性的、無感測器的吸塵器世界中可到達部分的信度狀態空間。每個陰影方塊對應於一個單獨的信度狀態。在任何給定的點，代理人是在特定的信度狀態，但並不知其實體狀態為何。初始的信度狀態(完全無知)是在上方中央的方塊。動作用有標記的連線表示。為清晰起見，我們省略自我迴圈的連線

前面的定義啓用了，從基本實體問題所定義的自動構建信度狀態問題的形式化。一旦做到這一點，我們可應用第三章的任何搜尋演算法。事實上，我們可以做得更多一點。在「一般」圖的搜尋，新產生的狀態經測試，看看它們是否與現有狀態相同。這種方法對於信度狀態也適用，例如，在圖 4.14，行動序列 [*Suck, Left, Suck*] 從初始狀態開始到達了相同的信度狀態 [*Right, Left, Suck*]，即 {5, 7}。現在，考慮信度狀態由 [*Left*] 達成，即 {1, 3, 5, 7}。顯然，這與 {5, 7} 不同，它是一個超集。很容易證明(習題 4.7)，如果一個行動序列是一個信度狀態 *b* 的解，則它也是任何 *b* 子集的一個解。因此，如果 {5, 7} 已經產生，我們可以丟棄一個到達 {1, 3, 5, 7} 的路徑。相反地，如果 {1, 3, 5, 7} 已經產生並發現可解，那麼任何子集，如 {5, 7}，是保證可解。這額外程度的修剪可大大提高無感測器問題求解的效率。

即使這樣的改善，然而，無感測器問題求解，如同我們所描述的在實務中是很少可行的。其困難是沒有信度狀態空間這麼多的——即使是以指數級的數量大於基本實體狀態空間，大多數情況下，在信度狀態空間和實體狀態空間的分枝因素和解的長度，並非如此不同。真正的困難在於每個信度狀態的大小。例如，對於 10×10 吸塵器世界的最初信度狀態包含有 100×2^{100} 或 10^{32} 左右的實體狀態——太多太多了，就像我們使用原子的表示法，這是一個明確的狀態列表。

有一種解是由更嚴謹描述來表示信度狀態。在英語中，我們可以說，代理人知道「無」的初始狀態；在移動 *Left* 之後，我們可以說，「不在最右邊的欄中，」等等。第七章將介紹在正式代表方案中如何做到這一點。另一種方法是避免標準的搜尋演算法，它以黑盒子來處理信度狀態，就像任何其他問題狀態。相反地，我們可以注意到信度狀態內部，並發展漸增信度狀態搜尋演算法，此演算法一次一個實體狀態地建立解。例如，在無感測器吸塵器世界，最初的信度狀態是 {1, 2, 3, 4, 5, 6, 7, 8}，我們必須找到適用於所有 8 個狀態的一個行動序列。我們可以做到這一點，首先找到一個解，適用於狀態 1，然後我們檢查它是否適用於狀態 2，如果不能，回到狀態 1 找一個不同的解，依此類推。正如一個 AND-OR 搜尋，在每一個分支 AND 節點必須找到一個解，這個演算法必須為在信度狀態的每個狀態找到一個解，不同的是，AND-OR 搜尋在每個分支可以找到不同的解，而漸增信度狀態搜尋必須找到一個解，適用於所有的狀態。

漸增方法主要的優點是，它通常能夠快速檢測故障——當一個信度狀態是無解時，它通常的情況是，一小部分的信度狀態，由最初幾個狀態組成的檢查，也是無解。在某些情況下，這導致了加速與信度狀態的大小成正比，這些信度狀態本身可能被視為大到實體狀態空間本身。

即使是最有效的解的演算法，在沒有解存在時也是沒有多大用處。沒有感測時，很多事是無法做到的。例如，無感測器八方塊遊戲是不可能的。另一方面，一點點的感測即可走一段很長的路。例如，如果只有一個方格是看得見的，則每個八方塊遊戲的實例是可解的——這解包括了按順序移動每個方塊到看得見的方格，然後跟蹤它的位置。

4.4.2 使用觀察的搜尋

對於一般的部分可觀察的問題，我們必須指定環境如何為代理人產生知覺。例如，我們可以定義局部感測吸塵器世界，是代理人有一個位置感測器和一個局部灰塵感測器，但沒有能夠檢測在其他方格中灰塵的感測器。正式問題的規範包括了，在給定的狀態中返回所接收到知覺的 $\text{PERCEPT}(s)$ 函數。(如果感測是不確定性的，那麼我們使用 PERCEPSTS 函數，它返回一組可能的知覺)。例如，在局部感測吸塵器世界裡，狀態 1 的 PERCEPT 是 $[A, \text{Dirty}]$ 。完全可觀察的問題是一個特例，對每一個狀態 s ， $\text{PERCEPT}(s) = s$ ，而無感測器問題也是一個特例，其中 $\text{PERCEPT}(s) = \text{null}$ 。

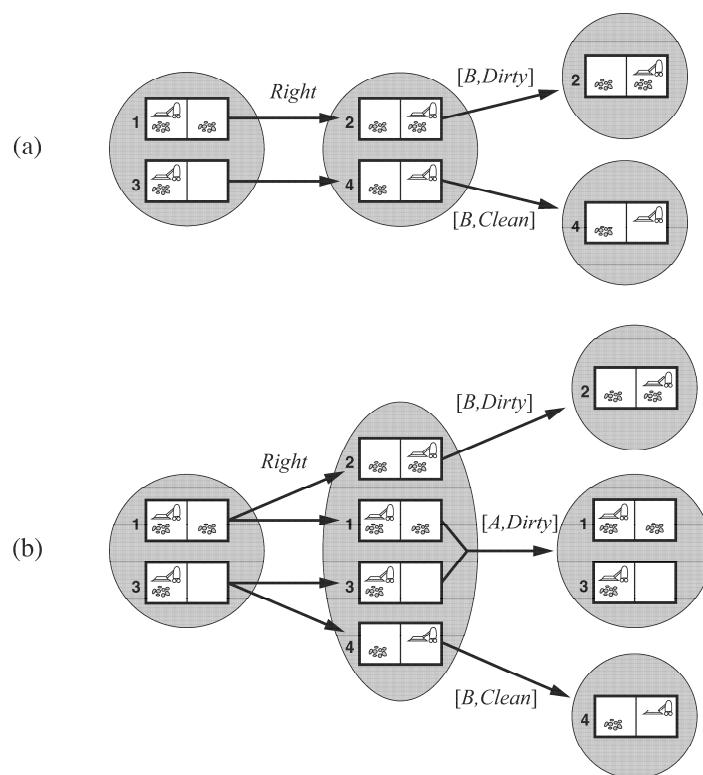


圖 4.15 在局部感測吸塵器世界轉換的兩個例子。(a) 在確定性的世界， Right (右)是應用在初始信度狀態，造成了兩個可能實體狀態的一個新的信度狀態，對於這些狀態，可能的知覺是 $[B, \text{Dirty}]$ 和 $[B, \text{Clean}]$ ，導致兩個信度狀態，每個狀態是一個單元素。(b) 在光滑的世界， Right (右)是適用於初始的信度狀態，給四個實體狀態予一個新的信度狀態，對於這些狀態來說，可能知覺是 $[A, \text{Dirty}]$ ， $[B, \text{Dirty}]$ ， $[B, \text{Clean}]$ ，導致三個信度狀態，如圖所示

當觀察是局部的，它通常的情況是，有數個狀態可以產生任何給定的知覺。例如，知覺 $[A, \text{Dirty}]$ 是由狀態 3 以及狀態 1 所產生。因此，給予這個作為初始知覺，局部感測吸塵器世界的初始信度狀態將是 $\{1, 3\}$ 。就像無感測器問題，ACTIONS、STEP-COST、GOAL-TEST 是從基本實體問題所構建，但轉換模式是比較複雜一點。我們可以想像從一個信度狀態轉換到下一個特定的行動為發生三個階段，如圖 4.15 所示：

- 在預測階段是與無感測器問題一樣的：在信度狀態 b 紿予行動 a ，則預測的信度狀態是 $\hat{b} = \text{PREDICT}(b, a)$ 。
- 觀察預測階段決定了知覺集 o ，它在預測的信度狀態中是可觀察的：

$$\text{POSSIBLE - PERCEPT}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ 及 } s \in \hat{b}\}$$

- 對每一個可能的知覺，更新階段決定了從知覺所導致結果的信度狀態。新的信度狀態 b_o 乃是在可能產生知覺的 \hat{b} 中的狀態集：

$$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ 及 } s \in \hat{b}\}$$

請注意，每個更新的信度狀態 b_o 不能大於預測的信度狀態 \hat{b} ，與無感測器的情況相比，觀察只有助於減少不確定性。此外，對於確定性的感測，信度狀態對各種可能的知覺將是不相交，形成一個原始預測信度狀態的分割。

把這三個階段放在一起，我們得到了從一個給定的行動和隨後可能的知覺，所造成的可能信度狀態：

$$\text{RESULT}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ 及 } o \in \text{POSSIBLE - PERCEPT}(\text{PREDICT}(b, a))\} \quad (4.5)$$

同樣，部分可觀察問題的不確定性，來自於無法準確預測哪些知覺會在行動之後被接收到；在實體環境中的基本不確定性，經由擴大在預測階段的信度狀態可能會貢獻到此，使在觀察階段導致更多的知覺。

4.4.3 求解部分可觀察的問題

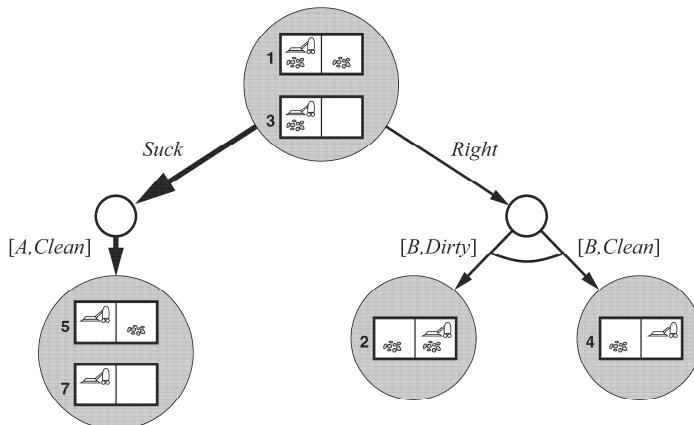
上一節說明了對於不確定性的信度狀態問題，如何從基本實體問題和 PERCEPT 函數得到 RESULTS 函數。給予形式化，圖 4.11 中 AND-OR 搜尋演算法可以直接應用於推導出解。圖 4.16 顯示了局部感測吸塵器世界裡搜尋樹的一部分，假設初始知覺是 $[A, Dirty]$ 。該解是有條件的規劃

$[Suck, Right, \text{if } Bstate = \{6\} \text{ then } Suck \text{ else } []]$

請注意，因為我們提供信度狀態問題到 AND-OR 搜尋演算法中，它返回一個有條件的規劃，此規劃測試了信度狀態而不是測試實際的狀態。這是因為它應該是：在部分可觀察的環境中，代理人將無法執行需要測試實際狀態的解。

如標準搜尋演算法應用於無感測器問題的例子，AND-OR 搜尋演算法以黑盒子來處理信度狀態，就像任何其他狀態。人們可以如同無感測器問題一樣，經由檢查先前產生的信度狀態，即當前狀態的子集或超集，而加以改善。人們也可以推導出漸增搜尋演算法，類似於無感測器問題所描述的，即在黑盒子上提供實質的加速方法。

圖 4.16 在局部感測吸塵器世界問題的 AND-OR 搜尋樹的第一層，Suck(吸)是解的第一步驟



4.4.4 部分可觀察的環境代理人

設計部分可觀察環境的一個問題求解代理人，是很類似於圖 3.1 之簡單的問題求解代理人：代理人制訂一個問題，呼叫一個搜尋演算法(如 AND-OR-GRAH-SEARCH)來解決這個問題，並執行該解。有兩個主要區別。首先，問題的解將是一個有條件的規劃，而不是一個序列，如果第一個步驟是一個 if-then-else 表達式，代理人將需要測試 if 部分的條件，並適當地執行 then-部分或 else-部分。其次，當代理人執行行動並接收知覺時需要維持其信度狀態。這個過程類似於公式 (4.5) 的預測-觀察-更新過程，但實際上較簡單，因為知覺是由環境所給予，而不是由代理人計算出來的。給定一個初始的信度狀態 b ，一個行動 a ，以及一個知覺 o ，新的信度狀態為：

$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o) \quad (4.6)$$

圖 4.17 顯示了信度狀態以局部感測，被保存在幼稚園吸塵器世界裡，其中任何方格在任何時間都有可能變成骯髒的，除非代理人在那一刻正積極地清掃它^[12]。

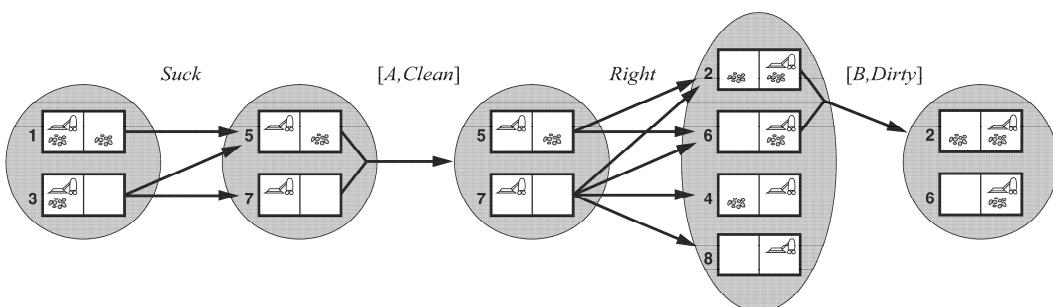


圖 4.17 兩種預測-更新週期的信度狀態，以局部感測被保存在幼稚園吸塵器世界裡

在部分可觀察的環境——其中包括絕大多數的真實世界環境——維持其信度狀態是任何智慧系統的一個核心功能。這個功能以各種名稱進行，包括監測，過濾和狀態評估。公式(4.6)被稱為遞迴狀態估計子，因為它是從前面一個信度狀態，而不是經由檢查整個知覺序列，所計算的新信度狀態。如果代理人不「落後」，則當知覺進來時計算必須盡可能地快。由於環境變得更加複雜，精確的更新計算變得不可行，代理人必須計算近似的信度狀態。注重在環境方面對知覺的影響，也許是當前所關心的。在這個問題上的大部分工作，使用了機率理論的工具，對於隨機、連續狀態等環境已經

進行完成，正如第 15 章所闡釋。在這裡，我們將在一個確定性的感測器和不確定性的行動的離散環境，展示一個例子。

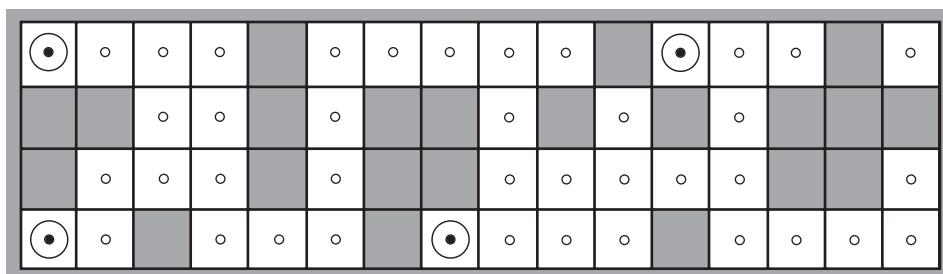
這個例子涉及機器人的定位任務：它是在哪裡工作，給予其世界地圖和知覺及行動的序列。我們的機器人被放置在圖 4.18 迷宮式的環境中。該機器人配備了四個聲納感測器，判斷是否有障礙——在圖中的外牆壁或一個黑色的方格——在每一個指南針的四個方向。我們假設感測器提供了完全正確的數據，而且機器人有一個正確的環境地圖。但不幸的是，機器人的導航系統壞了，所以當它執行一個 *Move* 行動，它隨機的移動到一個相鄰的方格中。該機器人的任務是確定其當前位置。

假設機器人剛剛開機，因此它並不知道它在哪裡。因此其最初的信度狀態 b 由所有位置集組成。機器人再接收 *NSW* 知覺，這意味著存在著北、西、南障礙，並用公式 $b_o = \text{UPDATE}(b)$ 更新，所產生的 4 個位置如圖 4.18(a)所示。您可以檢查此迷宮，並看到這些是產生 *NWS* 知覺僅有的四個位置。

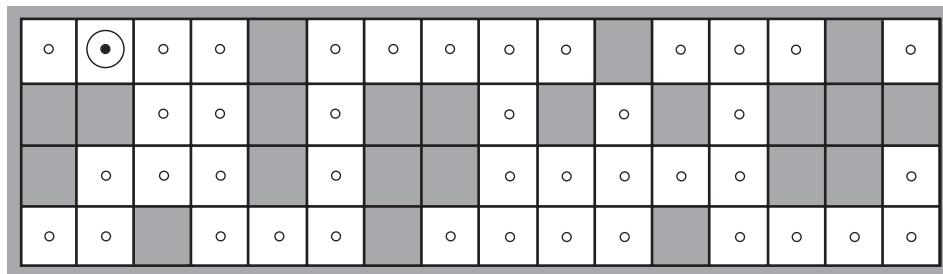
接下來機器人執行一個 *Move* 行動，但結果是不確定性的。新的信度狀態 $b_a = \text{PREDICT}(b_o, \text{Move})$ 包含了距離位置 b_o 一步的所有位置。當第二個知覺 *NS* 到達時，機器人執行 $\text{UPDATE}(b_a, \text{NS})$ ，並發覺信度狀態已經崩潰的下到單一個位置，如圖 4.18(b)所示。這是唯一的位置，其結果可能是

$$\text{UPDATE}(\text{PREDICT}(\text{UPDATE}(b, \text{NSW}), \text{Move}), \text{NS})$$

隨著不確定性的行動，*PREDICT* 步驟成長了信度狀態，但 *UPDATE* 步驟又縮小了回去——只要知覺提供了一些有用的識別資訊。有時知覺對於局部化並沒有太大的幫助：如果有一個或更多個長的東西向走廊，那麼機器人可以收到長序列的 *NS* 知覺，但永遠不知道它是在走廊的哪裡。



(a) 在 $E_1 = \text{NSW}$ 之後的機器人可能位置



(b) 在 $E_1 = \text{NSW}, E_2 = \text{NS}$ 之後的機器人可能位置

圖 4.18 可能的機器人位置， \odot ，(a) 在一個觀察 $E_1 = \text{NSW}$ 之後；(b) 在第二個觀察 $E_2 = \text{NS}$ 之後。當感測器是無雜訊且轉換模式是準確的，則機器人沒有其他可能的位置，與這兩個觀察序列一致

4.5 線上搜尋代理人和未知環境

至今為止我們一直把目光集中在代理人的離線搜尋演算法。他們在踏上現實世界之前先計算出一個完整的解，然後再執行該解。與之相反，線上搜尋^[13]代理人是透過計算和行動的交叉完成的：首先它先採取一個行動，然後觀察問題的環境並且計算出下一個行動。線上搜尋在動態或半動態的問題領域——對於停留不動或者計算時間太長都會有懲罰的領域——是一個好的想法。線上搜尋也有助於在不確定性的領域，因為它允許代理人集中努力其運算在實際發生的偶發情況，而不是那些可能但也許不會發生的情況。當然，有一個權衡：代理人事先規劃的越多，那麼它自己會發現沒有漿的小溪越少。

線上搜尋對未知環境是一個必要的想法，在那裡的代理人並不知道存在什麼狀態，或做什麼行動。在這種無知的狀態下，代理人面臨了探索問題，必須使用自己的行動作為實驗，學習足以讓其想法是值得的。

線上搜尋的一個範例是放在新建大樓裡的機器人，要求它探索大樓，繪製出一張從 A 到 B 的地圖。逃離迷宮的方法——胸懷抱負的古代英雄所需的知識——也是線上搜尋演算法的一個例子。不過，空間探索不是探索的唯一形式。考慮新生嬰兒：它有許多可能的行動，但是不知道這些行動的後果，而且它只經歷過少數幾個它能到達的可能狀態。嬰兒對世界運轉情況的逐步發現，部分地，也是一個線上搜尋過程。

4.5.1 線上搜尋問題

線上搜尋問題必須透過代理人執行行動解決，而不是純粹的計算過程。我們假設一個確定性的，且完全可觀察的環境(第 17 章放鬆這些假設)，但我們規定，代理人只知道以下幾點：

- $\text{ACTIONS}(s)$ ，傳回狀態 s 下的可能行動的列表；
- 單步成本函數 $c(s, a, s')$ ——注意直到代理人知道行動的結果為狀態 s' 時才能用；
- $\text{GOAL-TEST}(s)$ 。

注意到，代理人不能決定 $\text{RESULT}(s, a)$ ，除非實際在狀態 s 中，且做行動 a 。例如，在圖 4.19 中所示的迷宮問題中代理人不知道從狀態 $(1, 1)$ 採取行動 *Up* 能到達狀態 $(1, 2)$ ；或者當已經到達狀態 $(1, 2)$ 時，不知道行動 *Down* 能回到狀態 $(1, 1)$ 。在某些應用中這種無知程度可以降低——例如，機器人探測器也許知道其移動的行動是如何運轉，只是無知於障礙物的位置。

最後，代理人將使用一個能夠估計從當前狀態到目標狀態的距離的可採納啓發函數 $h(s)$ 。例如，在圖 4.19 中，代理人知道目標的位置並且可以使用曼哈頓距離啓發函數。

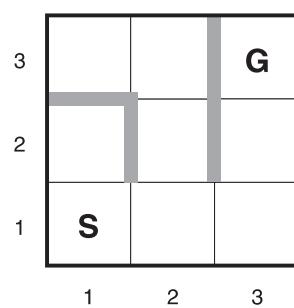


圖 4.19 一個簡單的迷宮問題。代理人從狀態 S 出發要到達狀態 G ，但是代理人對環境一無所知

通常，代理人的目標是用最小的成本到達目標狀態(另一個可能的目標是完全探索整個環境)。成本是指代理人實際行經的路徑的總成本。把這個成本與代理人要走的路徑的成本相比較是常見的，如果它事先瞭解搜尋空間——即知道實際最短的路徑(或最短的完全探索)。在線上演算法的術語中，這被稱為競爭率；我們希望這個值盡可能小。

雖然這聽起來像是一個合理的要求，但是容易看到在某些情況下最好的競爭率是無窮大。例如，如果某些行為是不可逆的——也就是說，它們會導致到一個狀態，而從該狀態則沒有行動可使其回到以前的狀態——那麼線上搜尋有可能偶然進入一個無法到達目標狀態的死狀態。也許你發現「偶然」這個詞並不可信——畢竟也許有演算法剛好不會探索到死路。更精確地，我們說的是沒有演算法能夠在所有的狀態空間中避免死路。考慮圖 4.20(a)所示的兩個有死路的狀態空間。對於一個已經存取過狀態 S 和 A 的線上搜尋演算法而言，這兩個狀態空間看起來是一樣的，所以它必然在這兩個狀態空間中做出相同的決策。因此，它對其中之一就會失敗。這是一個敵對參數的例子——我們可以想像一個對手在代理人探索的時候構造狀態空間，並且它可以隨意放置目標狀態和死狀態。

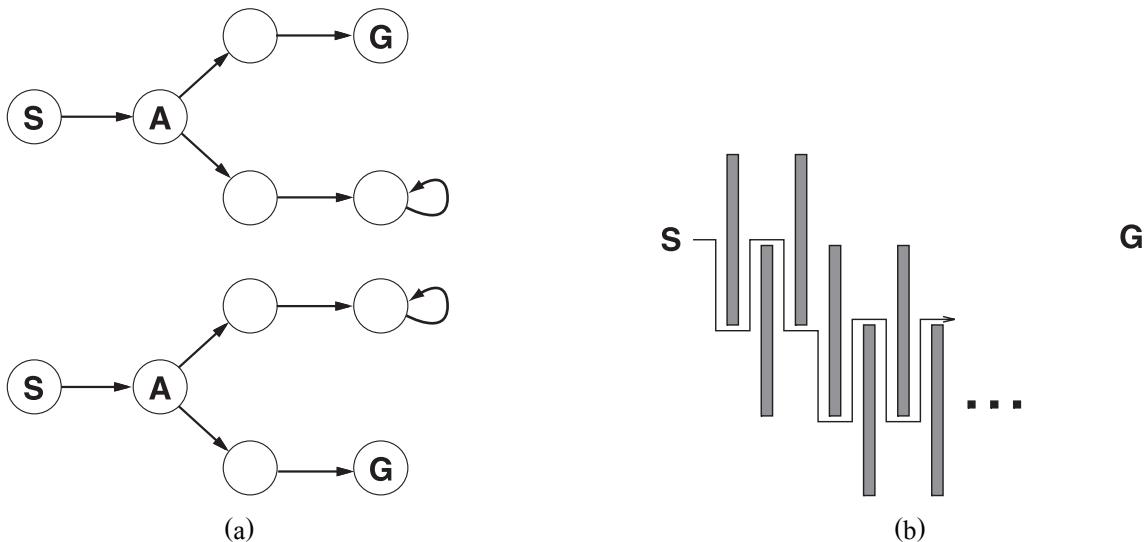


圖 4.20 (a) 兩個能導致線上搜尋代理人陷入死路的狀態空間。任何一個給定的代理人至少在其中一個狀態空間中陷入死路。(b) 能使線上搜尋代理人跟隨到達目標的任意低效率的路徑的二維環境。不管代理人選擇哪條路，對手都用細長的牆封鎖路徑，所以代理人所走的路徑比最佳可能路徑要長很多

死路是機器人探索中的實際困難——樓梯、斜坡、懸崖、單行道和各式各樣自然的地形都提供了不可逆行為的機會。為了取得進展，我們可以簡單地假設狀態空間是可安全探索的——也就是說，從每個可到達的狀態出發都有某些目標狀態是可到達的。具有可逆行動的狀態空間，例如迷宮問題和八方塊遊戲，可以視為無向圖，並且顯然是可安全探索的。

即使在可安全探索的環境裡，如果有無限大成本的路徑，就一定會有無上限的競爭率。這在行動不可逆的環境中很容易顯示出來，不過事實上它在行動可逆的情況下也是成立的，如圖 4.20(b)所示。由於這個原因，通常根據整個狀態空間的大小，而不是僅根據最淺的目標的深度，來描述線上搜尋演算法的性能。

4.5.2 線上搜尋代理人

在每個行動之後，線上代理人都能接收到知覺資訊，告訴它到達了哪個狀態；根據這個資訊，它可以擴展自己的環境地圖。當前的地圖用來決定下一步往哪裡走。這種規劃和行動的交叉意味著線上搜尋演算法，和我們看到的離線搜尋演算法是相當不同的。例如，有些離線演算法(如 A*)有能力在狀態空間的一部分擴展一個節點，然後馬上在空間的另一部分擴展另一個節點，因為節點擴展涉及的是模擬的而不是實際的行動。另一方面，線上演算法只會在它實際佔據的節點發現後繼者。為了避免尋訪整個搜尋樹去擴展下一個節點，按照局部順序擴展節點看來會更好一些。深度優先搜尋就有這個性質，因為(除了回溯的時候)下一個要擴展的節點就是前一個被擴展節點的子節點。

圖 4.21 中顯示了一個線上深度優先搜尋代理人。這個代理人將它的環境地圖儲存在一個表裡， $\text{RESULT}[s, a]$ ，記錄了在狀態 s 執行行動 a 得到的結果狀態。只要從當前狀態出發的某個行動還沒有被探索過，代理人就要嘗試這個行動。難題來自代理人嘗試完一個狀態的所有行動之後。在離線深度優先搜尋中，狀態很簡單地被從佇列中刪除；而在線上搜尋中，代理人不得不實際地回溯。在深度優先搜尋中，這意味著回溯到代理進入當前狀態前最近的那個狀態。這要透過維護一個表來達到，該表列出每個狀態的所有還沒有回溯的前任者狀態。如果代理人已經沒有可回溯的狀態了，那麼它的搜尋就完成了。

```

function ONLINE-DFS-AGENT( $s'$ ) returns an action
  inputs:  $s'$ , a percept that identifies the current state
  persistent:  $result$ , a table indexed by state and action, initially empty
     $untried$ , a table that lists, for each state, the actions not yet tried
     $unbacktracked$ , a table that lists, for each state, the backtracks not yet tried
     $s, a$ , the previous state and action, initially null

  if GOAL-TEST( $s'$ ) then return stop
  if  $s'$  is a new state (not in  $untried$ ) then  $untried[s'] \leftarrow \text{ACTIONS}(s')$ 
  if  $s$  is not null then
     $result[s, a] \leftarrow s'$ 
    add  $s$  to the front of  $unbacktracked[s']$ 
  if  $untried[s']$  is empty then
    if  $unbacktracked[s']$  is empty then return stop
    else  $a \leftarrow$  an action  $b$  such that  $result[s', b] = \text{POP}(unbacktracked[s'])$ 
  else  $a \leftarrow \text{POP}(untried[s'])$ 
   $s \leftarrow s'$ 
  return  $a$ 

```

圖 4.21 使用深度優先探索的一個線上搜尋代理人。該代理人只適用於，其中每個行動都可以由其他一些行動使其「未完成」這樣的狀態空間

我們建議讀者追蹤圖 4.19 中所示的用於迷宮問題的 ONLINE-DFS-AGENT 的過程。很容易看出在最壞的情況下，代理人在狀態空間中最終要通過每個連接剛好兩次。對於探索來說，這是最優的；另一方面，對於尋找目標來說，當一個目標狀態就在初始狀態旁邊的時候，如果代理人要透過很長的旅程才找到目標，它的競爭率就太差了。一個變異的線上疊代深入解決了這個問題；對於可表示為一致搜尋樹的環境，這樣一個代理人的競爭率是一個很小的常數。

因為用了回溯的方法，ONLINE-DFS-AGENT 只能用在行動可逆的狀態空間中。稍微複雜一些的演算法可以在一般的狀態空間中工作，但是並沒有這種具上限競爭率的演算法。

4.5.3 線上局部搜尋

如同深度優先搜尋，爬山法搜尋在節點擴展的時候也有局部性。實際上，因為它在記憶體中只存放一個當前狀態，爬山法搜尋其實就是一個線上搜尋演算法了！不幸的是，它最簡單的形式並沒什麼用，因為它使代理人呆在局部極大值上而無處可去。此外，隨機重新開始演算法是不能用的，因為代理人不能把自己傳送到一個新的狀態。

取代隨機重新開始，可以考慮使用隨機行走來探索環境。隨機行走簡單地從當前狀態隨機選擇可能的行動之一；選擇的時候可以偏向尚未嘗試過的行動。很容易證明倘若狀態空間為有限，隨機行走最終會找到目標或完成探索^[14]。然而，這個過程會很慢。圖 4.22 顯示了一個環境，在其中隨機行走演算法將耗費指數級的步數來找到目標，因為每一步往回走的進展都是向前走的兩倍。當然這個例子是設計出來的，但是許多現實世界的狀態空間的拓撲結構都能造成此類對隨機行走而言的「陷阱」。

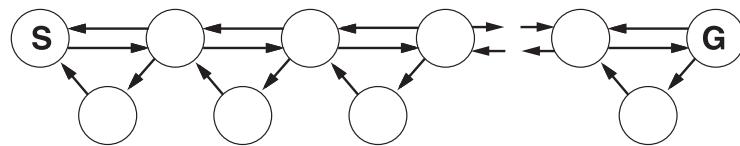


圖 4.22 一個使隨機行走耗費指數級步數才能找到目標的環境

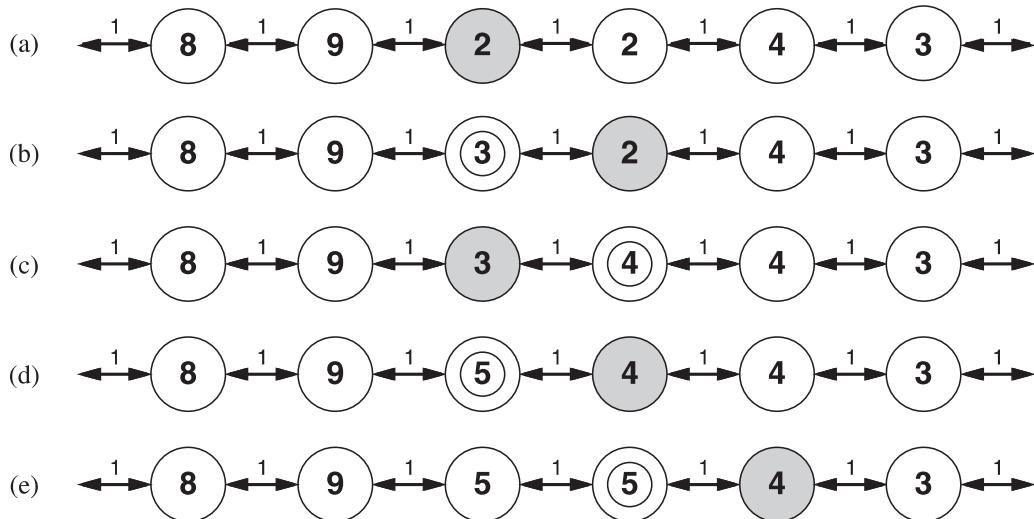


圖 4.23 在一個一維狀態空間上 LRTA* 的五次疊代。每個狀態都標出到達目標的當前成本估計 $H(s)$ ，每條連線則標出其單步成本。陰影狀態表示代理人所在的位置，並將每次疊代的更新成本估計，畫上圓圈標記出來

提高爬山法演算法的記憶體利用率而不是隨機性是個更有效的方法。基本想法是儲存一個從每個已經存取過的狀態到達目標狀態的成本中的「當前最佳估計」 $H(s)$ 。 $H(s)$ 從啓發函數估計 $h(s)$ 出發，在代理人從狀態空間中獲得經驗時對 $H(s)$ 進行更新。圖 4.23 顯示了一個一維狀態空間的簡單例子。在(a)中，代理人看起來會被平坦的局部極小值卡住，即圖中陰影部分所示狀態。代理人應該根據對鄰接狀態的當前成本估計來選擇到達目標的最佳路徑，而不是停留下來。經過鄰接 s' 到達目標的估計成本，是到達 s' 的成本加上從那裡——即 $c(s, a, s') + H(s')$ ，到達一個目標的估計成本。在例子中，有兩個行動的估計成本為 1 + 9 和 1 + 2，因此看來最好是向右移動。現在，顯然對陰影狀態的成本估計為 2 過於樂觀了。因為最佳移動成本為 1 並且會到達一個離目標狀態至少 2 步的狀態，陰影狀態到目標狀態至少需要 3 步，因此它的 H 需要相對應的更新，如圖 4.23(b)所示。繼續這個過程，代理人將再來回移動兩次，每次都更新 H 並「平滑化」局部極小值，直到它能向右逃離為止。

實作這個方案的代理人被稱為即時學習 A*(LRTA*)代理人，如圖 4.24 所示。像 ONLINE-DFS-AGENT 一樣，它用 *result* 表建造了一個環境地圖。它更新了剛剛離開的狀態的成本估計，然後根據當前的成本估計選擇「顯然最佳的」移動。一個重要的細節是：在狀態 s 從未嘗試過的行動總被假設為能用最小可能成本，也就是 $h(s)$ ，直接到達目標。這種不確定下的樂觀主義鼓勵代理人探索新的、可能更有希望的路徑。

LRTA*代理人在任何有限的、可安全探索的環境中都保證能找到目標。然而不同於 A*，它對於無限的狀態空間不是完備的——有些情況能把它引入無限的歧途。它在最壞情況下可以用 $O(n^2)$ 步探索一個 n 狀態的環境，不過經常做得更好。LRTA*代理人只是龐大的線上代理人家族中的一員，這些線上代理人可以透過指定不同方式的行動選擇規則和更新規則來定義。我們將在 21 章討論原先為針對隨機環境發展的此家族。

```

function LRTA*-AGENT( $s'$ ) returns an action
  inputs:  $s'$ , a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
     $H$ , a table of cost estimates indexed by state, initially empty
     $s, a$ , the previous state and action, initially null

  if GOAL-TEST( $s'$ ) then return stop
  if  $s'$  is a new state (not in  $H$ ) then  $H[s'] \leftarrow h(s')$ 
  if  $s$  is not null
    result[ $s, a$ ]  $\leftarrow s'$ 
     $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)} \text{LRTA}^*-\text{COST}(s, b, \text{result}[s, b], H)$ 
     $a \leftarrow$  an action  $b$  in  $\text{ACTIONS}(s')$  that minimizes  $\text{LRTA}^*-\text{COST}(s', b, \text{result}[s', b], H)$ 
     $s \leftarrow s'$ 
  return  $a$ 

function LRTA*-COST( $s, a, s', H$ ) returns a cost estimate
  if  $s'$  is undefined then return  $h(s)$ 
  else return  $c(s, a, s') + H[s']$ 

```

圖 4.24 LRTA*-AGENT 根據相鄰狀態的值選擇一個行動，代理人在狀態之間移動時更新狀態值

4.5.4 線上搜尋的學習

線上搜尋代理人初始對環境的無知提供了一些學習的機會。首先，代理人僅僅根據它的經歷學習到環境的「地圖」——更精確地說，是每個狀態經過每個行動的結果。(注意確定性環境的假設意味著每個行動經歷一次就足夠了)。其次，局部搜尋代理人利用局部更新規則(和 LRTA*中的情況一樣)可以得到每個狀態更精確的估計值。在第 21 章，我們會看到倘若代理人按照正確的方式探索狀態空間，這些更新最終會收斂到每個狀態的精確值。一旦知道了狀態的精確值，最佳決策就可以簡單地透過移動到最低成本的後繼者而完成——也就是說，那麼純粹的爬山法演算法也是一個最佳策略。

如果你聽從我們的建議，追蹤 ONLINE-DFS-AGENT 在圖 4.19 所示的環境中的行為表現，你將會注意到代理人並不十分聰明。例如，當它已經看到行動 *Up* 能從狀態(1, 1)到狀態(1, 2)時，它仍然不知道行動 *Down* 能回到狀態(1, 1)，或者行動 *Up* 還能從狀態(2, 1)到狀態(2, 2)，從狀態(2, 2)到狀態(2, 3)，等等。通常，我們希望代理人學習到 *Up* 能使 *y* 座標值增加，除非遇到牆；*Down* 能使 *y* 座標值減少，等等。要達到這些必須滿足兩件事情。首先，需要一個對這種一般規則，具形式化和明確的可操作描述，到目前為止，我們把這些資訊隱藏在稱為 RESULT 函數的黑盒子裡。本書的第三部分會討論這個問題。其次，我們需要有演算法能夠根據代理人得到的特定觀察資料來構造合適的一般規則。這些將在第 18 章中論及。

4.6 總結

本章研究了在「傳統」可觀察的、確定性的、離散的環境中，尋找到達目標的最短路徑問題案例之外的一些搜尋演算法。

- 局部搜尋方法諸如爬山法用於完全狀態形式化，它只在記憶體中保留很少數目的節點。已經發展了幾種隨機演算法，包括**模擬退火**，當給定一個合適的冷卻排程安排時，它能夠返回最佳解。
- 許多局部搜尋方法也適用於連續空間的問題。**線性規劃**和**凸多邊形最佳化**問題，遵守狀態空間的一定形狀的限制和目標函數的性質，並認同在實務中往往是非常有效的多項式時間演算法。
- **基因遺傳演算法**是一個保留大量狀態種群的隨機爬山法搜尋。新的狀態透過**突變**和**雜交**產生，雜交把來自種群的狀態對結合在一起。
- 在**不確定性**的環境中，代理人可以應用 AND-OR 搜尋產生**偶發規劃**，無論在執行過程中發生任何結果，它都會到達目標。
- 當環境是部分可觀察的，**信度狀態**表示出該代理人可能存在的狀態集
- 標準的搜尋演算法可以直接應用於信度狀態空間來解決**無感測器問題**，信度狀態 AND-OR 搜尋可以解決部分可觀察的問題。在漸增演算法，信度狀態中透過狀態到狀態的構建解，往往更有效率。
- **探索問題**出現在代理人對環境的狀態和行動一無所知時。對於可安全探索的環境，**線上搜尋**代理人能夠建造一個地圖並且找到可能存在的目標。根據經驗更新啟發函數估計，提供了一種避免局部極小值的有效方法。

● 參考文獻與歷史的註釋

BIBLIOGRAPHICAL AND HISTORICAL NOTES

局部搜尋技術在數學和電腦科學中都有很長的歷史。實際上，牛頓-拉夫森方法(Newton-Raphson method)(牛頓，1671；拉夫森，1690)可以被視為一個很有效的能夠獲得梯度資訊的連續空間上的局部搜尋方法。Brent(1973)的文章是不需要此類資訊的最佳化演算法的經典參考。我們作為局部搜尋演算法提出的剪枝搜尋，源自用於 HARPY 系統(Lowerre，1976)語音識別的有限寬度動態規劃的一個變形。Pearl(1984，第五章)深入分析了一個相關的演算法。

局部搜尋的課題在近幾年來再次復興，是因為它在很多大型限制滿足問題，諸如 n 皇后問題(Minton 等人，1992)和邏輯推理問題(Selman 等人，1992)上取得的令人驚訝的好結果以及與隨機性、多同步搜尋和其他改進的結合。這個被 Christos Papadimitriou 稱為「新時代」演算法的復興，也激發了理論電腦科學家(Koutsoupias 和 Papadimitriou，1992；Aldous 和 Vazirani，1994)的興趣。在作業研究領域，由爬山法演算法變化而來的禁忌搜尋(tabu search)得到廣泛應用(Glover 和 Laguna，1997)。這個演算法維護一個禁忌列表，記錄 k 個已經存取過而且不能再存取的節點；這種演算法在搜尋圖的時候不僅提高了效率，這個列表還可以使演算法避開一些局部極小值。另一個對爬山法演算法的有用改進是 STAGE 演算法(Boyan 和 Moore，1998)。它的想法是根據隨機重新開始的爬山法演算法得到的局部極大值，來獲取整個地形圖的全貌。這個演算法給局部極大值集合找到一個平滑的表面，並且分析計算出表面上的全局最大值。這成為一個新的重新開始點。這個演算法被證實在實務上對於難題是有效的。Gomes 等人(1998)證明了系統性回溯演算法的執行時間經常是重尾分佈(heavy-tailed distribution)，意味著很長執行時間的機率要比執行時間是指數分佈情況下做出的預測值大。當執行時間分佈是重尾，平均來說，隨機重新開始會比單獨執行到完成，更快找到解。

模擬退火首先由 Kirkpatrick 等人(1983)提出，他直接借鑒了 Metropolis 演算法(用於模擬物理現象中的複雜系統(Metropolis 等人，1953)，並被認為是在洛斯阿拉莫斯(Los Alamos)的晚宴聚會上發明的)。模擬退火演算法現在是一個單獨的領域，每年有數百篇文章發表。

在連續空間中尋找最佳解是涉及多個領域的課題，包括最佳化理論，最佳控制理論，以及變分法(calculus of variations)。基本的技術已由 Bishop(1995)，Press 等人(2007)作了良好的闡釋，涵蓋了範圍廣泛的演算法，並提供工作軟體。

如同 Andrew Moore 指出，研究人員已從各式各樣的研究領域取得搜尋和最佳化演算法的靈感：冶金(模擬退火)，生物學(基因遺傳演算法)，經濟學(基於市場的演算法)，昆蟲學(蟻群最佳化)，神經學(神經網路)，動物行為(強化學習)，登山(爬山法)等。

線性規劃(LP)首次由俄羅斯數學家 Leonid Kantorovich(1939)作了系統性地研究。單純演算法(the simplex algorithm)(Dantzig，1949)是電腦的首次應用之一，雖然是最壞情況的指數複雜性，仍然尚在使用。Karmarkar(1984)開發了更有效的內點法，並由 Nesterov 及 Nemirovski(1994)證明，具有多項式複雜性更為一般等級的凸多邊形最佳化問題。引進優良的凸多邊形最佳化，是由 Ben-Tal 和 Nemirovski(2001)和 Boyd 和 Vandenberghe(2004 年)所提供之。

Sewall Wright(1931)建立在適應性地形概念上的工作是基因遺傳演算法發展的重要先驅。20世紀50年代，一些統計學家，包括Box(1957)和Friedman(1959)，用進化技術來解決最佳化問題，但是直到Rechenberg(1965)引入進化策略來解決機翼的最佳化問題，這個方法才流行起來。在20世紀60年代和70年代，約翰·霍蘭德(John Holland, 1975)支持提出的基因遺傳演算法既是一個有用的工具，也是擴展我們對適應性、生物學或其他方面(霍蘭德, 1995)理解的一個方法。人工生命運動(Langton, 1995)使這個想法更進一步，它將基因遺傳演算法產生的結果看作是生物體而不是問題的解。Hinton和Nowlan(1987)以及Ackley和Littman(1991)在澄清鮑德溫效應的含義上做了很多研究。對於進化的一般背景知識，我們強力推薦Smith和Szathmáry(1999)、Ridley(2004)、Carroll(2007)的著作。

從多數基因遺傳演算法與其他演算法(尤其是隨機的爬山法演算法)的比較中都發現基因遺傳演算法收斂得比較慢(O'Reilly和Oppacher, 1994; Mitchell等人, 1996; Juels和Wattenberg, 1996; Baluja, 1997)。這些發現在GA領域中並不是普遍存在的，但是最近把基於種群的搜尋理解為貝葉斯學習(Bayesian learning見第20章)的一種近似形式的努力，也許能夠幫助縮小該領域和它的批判之間的鴻溝(Pelikan等人, 1999)。二次動態系統的理論也能解釋GA的表現(Rabani等人, 1998)。參見Lohn等人(2001)給出的將GA用於天線設計的例子，以及Renner和Ekart(2003)用於電腦輔助設計的例子。

基因遺傳程式設計的領域與遺傳演算法關係非常密切。主要的區別是突變和合併的表示物件是程式而不是字串。程式用表達樹的形式表示；它的表達可以用標準的語言諸如Lisp或者特別設計的表示電路、機器人控制器等等的形式完成。雜交涉及合併子樹而不是合併子串。這種形式的變異保證了後代是形態良好的表示，如果程式是以字串來操作就不會得到這樣好的結果。

最近的基因遺傳程式設計的興趣是由John Koza的研究(Koza, 1992, 1994)引起的，但是它至少能追溯到早期Friedberg(1958)用機器程式碼進行的實驗和Fogel等人(1966)用有限狀態自動機進行的實驗。在用遺傳演算法的時候會遇到關於技術效率的爭論。Koza等人(1999)描述了使用基因遺傳程式設計，來設計電路裝置的實驗。

期刊《進化計算》(*Evolutionary Computation*)和《IEEE 進化計算學報》(*IEEE Transactions on Evolutionary Computations*)收錄基因遺傳演算法和基因遺傳程式設計的論文；這一類論文也會在《複雜系統》(*Complex Systems*)、《自適應行為》(*Adaptive Behavior*)和《人工生命》(*Artificial Life*)中出現。主要的研討會是《遺傳與進化計算會議》(*Genetic and Evolutionary Computation Conference*, GECCO)。基因遺傳演算法很好的概述教科書由Mitchell(1996), Fogel(2000)，以及Langdon和Poli(2002)給出，以及免費的線上圖書由Poli等人(2008)所提供之。

真實環境的不可預測性和部分可觀察性早就被認識到，在使用規劃技術的機器人技術專案上，包括Shakey(Fikes等人, 1972)和FREDDY(Michie, 1974)。這個問題在McDermott(1978a)的有影響力的文章《規劃與行動》(*Planning and Acting*)發表之後受到了更多的關注。

作出明確的使用 AND-OR 樹的首要工作，似乎是 Slagle 的 SAINT 作為符號整合的程式，已在第一章中提到。Amarel(1967)應用到了這個觀念到命題定理證明，在第七章所討論的一個主題，並推出類似 AND-OR-GRAFH-SEARCH 的搜尋演算法。該算法得到了 Nilsson(1971)進一步的發展和形式化，Nilsson 也描述了 AO*，它正如其名稱所示，給予可採納的啓發函數找到最佳的解。Martelli 和 Montanari(1973)進行了 AO*分析和改善。AO*是一個自上而下的演算法，A*的自下而上的一般化是 A*LD(即 A* Lightest Derivation)(Felzenszwalb 和 McAllester，2007)。近幾年對 AND-OR 搜尋的興趣已產生了復甦，具有找到循環解的新演算法(Jimenez 和 Torras，2000；Hansen 和 Zilberstein，2001)和由動態規劃啓發的新技術(Bonet 和 Geffner，2005)。

轉換部分可觀察的問題到信度狀態問題的想法起源於 Astrom(1965)更複雜情況的不確定性機率(見第十七章)。Erdmann 和 Mason(1988)研究了沒有感測器的機器人操作的問題，採用了連續形式的信度狀態搜尋。他們指出，從桌上任意初始位置，經由精心設計的一系列的傾斜行動，使一零件移向東方是可能的。更實用的方法，根據一系列精確的對角線障礙導向，而跨越輸送帶，使用了相同見解的演算法(Wiegley 等人，1996)。

這個信度狀態的做法，已在 Genesereth 和 Nourbakhsh(1993)的無感測和部分可觀察的搜尋問題研究中做了改造。另外的工作是以邏輯為基礎的規劃社區之無感測器問題(Goldman 和 Boddy，1996；Smith 和 Weld，1998)。這項工作強調簡潔陳述信度狀態，如在第 11 章所述。Bonet 和 Geffner(2000)推出第一個有效的啓發式信度狀態搜尋，Bryce 等人(2006)並作了改善。漸增的信度狀態搜尋方法，其解是在每個信度狀態的狀態子集作漸增的構造，已在 Kurien 等人(2002)的規劃文獻中作了研究；一些新的漸增演算法，由 Russell 和 Wolfe(2005)介紹了具不確定性的、部分可觀察的問題。其他隨機、部分可觀察的環境規劃的參考資料，將出現在第 17 章。

探索未知狀態空間的演算法，學者已經研究了很多個世紀。在迷宮中可以沿著左邊走，來完成深度優先搜尋；可以在每個岔路口都做上標記以避免迴圈。在行動不可逆條件下深度優先搜尋會失敗；更一般的探索歐拉圖(Eulerian graph，即每個節點都有相同進入和離開的邊數量的圖)的問題是由 Hierholzer(1873)的演算法解決的。最早對於任意圖探索問題的詳盡的演算法研究是由 Deng 和 Papadimitriou(1990)完成的，他們發展出一個完全通用的演算法，但是證明了探索一般圖的競爭率有可能是無上限的。Papadimitriou 和 Yannakakis(1991)考察了幾何路徑規劃環境(所有行動都是可逆的)中的目標尋徑問題。他們證明了在正方形障礙物情況下一個很小的競爭率是可以達成的，但是在一般矩形障礙物的條件下無法得到有上限的競爭率(參見圖 4.20)。

LRTA*演算法是由 Korf(1990)作為即時搜尋的一部分研究提出來的，在這個問題環境下，代理人在搜尋固定的時間(在雙人遊戲中一般常見的情景)後必須採取行動。**LRTA***演算法實際上是隨機環境下強化學習演算法的一種特殊情況(Barto 等人，1995)。它在不確定條件下的最佳化方針——總是朝向最近的未存取過的狀態——產生了一個探索模式，它的效率在無資訊的情況下比簡單的深度優先搜尋要低(Koenig，2000)。Dasgupta 等人(1994)證明了線上疊代深入搜尋，對於沒有啓發函數資訊的情況下，在一致搜尋樹上尋找目標，具有最優的效率。一些有資訊的 LRTA*主題的變形是透過在圖的已知部分中進行搜尋和更新的不同方法發展出來的(Pemberton 和 Korf，1992)。迄今，在使用啓發函數資訊的情況下，如何以最佳的效率找到目標，還沒有很好的知識。

❖ 習題 EXERCISES

- 4.1** 根據下面的特殊情況給出演算法的名稱：
- $k = 1$ 的局部剪枝搜尋。
 - 具一個初始狀態及沒有限制數目的保留狀態的局部剪枝搜尋。
 - 所有時間 $T = 0$ (以及省略終止試驗)的模擬退火搜尋
 - 所有時間 $T = \infty$ 的模擬退火搜尋。
 - 種群大小為 $N = 1$ 的基因遺傳演算法。
- 4.2** 習題 3.16 考慮無鬆弛每片段都準確地切合假設下的鐵軌建設問題。現在考慮實際的問題，每片段不再準確地切合，但允許多達 10 度的「正確的」對準兩邊的旋轉。說明如何制定此問題，使其可以用模擬退火求解。
- 4.3**  在本題中，我們將探索使用局部搜尋演算法求解習題 3.30 中定義的那種 TSP 問題。
- 實現和測試爬山法，求解 TSP 問題。比較用該演算法得到的結果和透過使用 MST 啓發函數(習題 3.30)的 A*演算法得到的最佳解。
 - 使用基因遺傳演算法而不是爬山法，重複(a)。可以參考 Larrañaga 等人(1999)關於表示法的建議。
- 4.4**  產生大量的八方塊遊戲和八皇后問題的實例，並用以下演算法求解(若可能的話)：爬山法(最陡上升和首選爬山法這兩個變種)，隨機重新開始的爬山法，和模擬退火演算法。度量搜尋成本和問題的解決百分比，並用圖解對比它們和最佳解成本的曲線。評論你的結果。
- 4.5** 圖 4.11 中的 AND-OR-GRAF-SEARCH 演算法只在從根節點到當前狀態的路徑上檢查重複狀態。另外假設演算法儲存每個存取過的狀態並檢查這個列表。(參見圖 3.11 中 BREADTH-FIRST-SEARCH 的一個例子。確定應該儲存的資訊，以及當一個重複狀態被找到時演算法該如何使用儲存的資訊(提示：你至少需要區別先前構造過成功子規劃的狀態和無法找到子規劃的狀態)。說明如何使用標籤，如在 4.3.3 節所定義，以避免有多份的子規劃。
- 4.6**  準確解釋如果不存在無循環規劃，如何修改 AND-OR-GRAF-SEARCH 演算法來產生一個有循環規劃。你將需要處理 3 種情況：標記規劃步驟以便有循環規劃能夠指回規劃的早期部分；修改 OR-SEARCH 以便它在找到一個有循環規劃後繼續尋找無循環規劃；增強規劃表示法以指示一個規劃是否有循環。說明你的演算法在下列情況下如何工作：(a) 光滑的吸塵器世界，(b) 光滑的、不穩定的吸塵器世界。你也許希望用電腦實現檢驗你的結果。
- 4.7** 在第 4.4.1 節我們介紹了信度狀態，以求解無感測器搜尋問題。一個行動的序列，如果它映射在初始信度狀態 b 的每個實體狀態到一個目標狀態，則解決了無感測器問題。假設代理人知道 $h^*(s)$ ，即 b 中的每個狀態 s ，在完全可觀察的問題，解決實體狀態 s 的真正最佳化成本。以這些費用項目，求無感測器問題的可採納啓發式 $h(b)$ ，並證明其為可採納的。對圖 4.14 無感測器吸塵器問題，這種啓發式的準確性，作出評論。A*的執行效果如何？

- 4.8 這個習題探索了在無感測器或部分可觀察環境，的信度狀態之間的子集–超集關係。
- 證明如果一個行動序列是一個信度狀態 b 的一個解，則它也是任何 b 子集的一個解。關於超集 b 可以表示些什麼？
 - 詳細說明如何利用答案(a)的優點，修改無感測器問題的圖解搜尋。
 - 詳細說明，在您於(b)所描述的修改之外，如何修改部分可觀察問題的 AND-OR 搜尋。
- 4.9 第 4.4.1 節中，假定在一個給定的信度狀態中執行任何的實體狀態，則一個給定的行動將具有相同的成本。(這導致了一種具有明確步驟費用的信度狀態搜尋問題)。現在考慮當假設不成立時會發生什麼。最佳化的概念在此背景下是否還有意義，還是需要作修改？還要考慮在信度狀態中執行一個行動，對於「成本」的各種可能的定義，例如，我們可以使用最小的，或最大的實體成本；或是在最低成本的下限和最高成本的上限的成本區間，或只是保留該行動所有可能的費用集。對於這裡的每一種，探討 A*(如果有必要則作修改)是否可以返回最佳解。
- 4.10 考慮無感測器版本的不穩定的吸塵器世界畫出可從最初的信度狀態{1, 2, 3, 4, 5, 6, 7, 8}到達的信度狀態空間，並解釋為什麼這個問題是無法解的。
- 4.11  我們可以將習題 3.7 的導航問題變為如下環境：
- 機器人的知覺是一個相對於代理人的可見頂點位置列表。然而此知覺不包括機器人本身的位置！機器人必須從地圖瞭解自己所處的位置；現在，你可以假設每個位置都有不同的「景觀」。
 - 每個行動是一個描述其依循直線路徑的向量。如果路徑上沒有阻礙，那麼這個行動成功；否則，機器人將停在它的路徑中與第一個障礙物的交會點上。如果代理人傳回一個零運動向量而且處於目標位置(目標是固定並且已知的)，那麼環境就會把代理人傳送到一個隨機的位置(而不是在障礙物裡面)。
 - 代理人每移動一個單位距離，效能指標就給予獎勵 1 分，每次到達目標狀態獎勵 1000 分。
 - 實作這個問題的環境和它的問題求解代理人。在每次傳送後，代理人需要形式化一個新問題，其中包括發現它目前位置。
 - 記錄下你的代理人效能(透過讓代理人在移動的時候產生適當紀錄的方式)並報告它在超過 100 次求解過程中的效能紀錄。
 - 修改環境使得在 30%時間的情況下，會使代理人結束於非預期的目的地(隨機地從其他可見的頂點選擇一個，如果沒有其他可見的頂點就不移動)。這是真實機器人移動錯誤的一個粗劣模型。修改代理人，使得當它檢查到這樣的錯誤時，能夠找出自己在什麼位置，然後想出一個規畫能走回原來所在的位置，再繼續執行舊的規畫。記住有時候走回原來的位置也可能失敗！提供一個代理人成功地克服兩個連續出現的移動錯誤並仍然能夠到達目標的例子。
 - 錯誤發生後，嘗試用另外兩套不同的恢復方案：(1) 在原路徑上前進最接近的頂點；(2) 重新規劃從新的位置到達目標的路徑。比較這三種恢復方案的效能。包含搜尋成本會影響它們的比較結果嗎？

- e. 假設從有些位置得到的景觀是相同的(例如，假設世界被正方形障礙物劃分為網格)。現在代理人面臨的是什麼類型的問題？它的解看來是什麼樣的？
- 4.12** 假設一個代理人在一個如圖 4.19 所示的 3×3 大小的迷宮裡。代理人知道它的初始位置是(3, 3)，目標位置是(1, 1)，四種行動 *Up*(上), *Down*(下), *Left*(左), *Right*(右)通常可以發揮效果，除非遇到有牆阻礙。代理人不知道迷宮內部的牆在哪些地方。在任何給定的狀態，代理人知道合法行動集；它也知道一個狀態是已經存取過的狀態還是新狀態。
- 解譯這個線上搜尋問題如何可以視為在信度狀態空間中的離線搜尋問題，初始的信度狀態包括所有可能的環境佈局。初始信度狀態有多大？信度狀態空間有多大？
 - 在初始狀態可能有多少個不同的感知資訊？
 - 描述這個問題的偶發性規畫的前幾個分支。完整的規畫(大約)有多大？
- 注意這個偶發性規畫是符合給定描述的每個可能環境的解。因此，即使在未知環境下搜尋和執行的交叉也不是嚴格必要的了。
- 4.13**  在本題中，我們將在機器人導航問題中考察爬山法，以圖 3.31 中的環境為例。
- 用爬山法演算法重複習題 4.11。代理人會卡在局部最小值上嗎？可能遇到被凸多邊形障礙物卡住的情況嗎？
 - 構造一個非凸多邊形的環境，代理人在其中會被卡住。
 - 修改爬山法演算法，在決定下一步的時候不用深度為 1 的搜尋，而用深度為 k 的搜尋。它將找到最好的 k 步路徑並且沿著該路徑走一步，然後重複這個過程。
 - 有沒有某個 k 使得新的演算法保證能避免局部極小值？
 - 解譯 LRTA* 是怎樣使新的演算法能夠在這種情況下避免局部極小值的。
- 4.14** 就像 DFS 一樣，對於無限路徑的可逆狀態空間，線上 DFS 是不完備的。例如，假設各狀態是在無限二維網格的點且行動是單位向量 $(1, 0)$, $(0, 1)$, $(-1, 0)$, $(0, -1)$ ，以這個順序作嘗試。證明線上 DFS，從 $(0, 0)$ 開始將不會到達 $(1, -1)$ 。假設代理人是可以觀察的，則除了其目前的狀態，將導致到所有繼承的狀態和行動。撰寫一個演算法，即使是在無限路徑的雙向狀態空間，它仍是完備的。在到達 $(1, -1)$ 時它存取哪些狀態？

本章註腳

- [1] 從一個隱式的狀態空間產生隨機的狀態本身也是一個很難的問題。
- [2] Luby 等人(1993)證明了在某些情況下這樣是最好的：在搜尋一段特定的固定時間之後重新開始一個隨機搜尋演算法；並證明這比起讓搜尋繼續進行要有效率得多。不允許或限制側向移動步數的方法就是此概念的一個例子。
- [3] 局部剪枝搜尋是剪枝搜尋的一個改編，它是基於路徑的演算法。
- [4] 這個選擇規則有很多變化。篩選的辦法是在一個給定的臨界值之下所有的個體都拋棄，這個辦法比隨機選取的收斂速度快(Baum 等人，1995)。

- [5] 這裡有一個編碼問題。如果一個 24 位元的編碼用來代替 8 位數字，那麼雜交點有 $2/3$ 的機會落在一個數字的中間，造成該數字本質上隨意的變異。
- [6] 在閱讀本節的時候，瞭解多元微積分和向量運算的基本知識是有益的。
- [7] 在一般情況下，牛頓-拉夫森更新可以看作是，在 x 處切合一個二次曲面到 f ，然後再直接移動到表面的最小值——它也是 f 的最小值，如果 f 是二次式的話。
- [8] 一組 S 點是凸多邊形，如果連接 S 中任意兩點的線也包含在 S 中。凸多邊形函數是在其「上面」的空間，形成一個凸多邊形集，根據定義，凸多邊形函數沒有局部(相對於全局)極小值。
- [9] 我們假設大多數讀者都會遇到類似的問題，並因此能體諒我們的代理人。我們對那些擁有現代化、有效率的家電用品因而無法融入本教學情境的讀者表示歉意。
- [10] 在完全可觀察的環境中，每個信度狀態只包含一個實體狀態。因此，我們可以查看在第三章中該演算法，作為單元素信度狀態的信度狀態空間的搜尋。
- [11] 在這裡，及整本書，在 \hat{b} 上方的「帽子」指 b 的估計值或預測值。
- [12] 通常向那些不熟悉環境會對小孩子影響的人作道歉。
- [13] 術語「線上」通常用於電腦科學，說明演算法是當其接受到輸入資料時，必須作處理，而不是等到整個輸入資料集都可用以後再處理。
- [14] 隨機行走演算法在無限的一維和二維網格上是完備的。在三維網格上，能返回起點的行走的機率只有 0.3405(Hughes, 1995)。

