CLASSICAL PLANNING

# 10 經典規劃

本章中，我們將看到代理人是如何利用問題的結構來構造行動的複雜規劃。

我們已經定義過人工智慧爲對理性行動的研究，這指的是**規劃**——策劃一個行動計劃，以達到自己的目標——其爲人工智慧的重要組成部分。到目前爲止我們已經看到過了規劃代理人的兩個例子：第 3 章中基於搜尋的問題求解代理人和第 7 章中的混合邏輯代理人。在這一章我們介紹一個規劃問題的表示式，以擴展到那些早先方法所無法處理的問題。

第 10.1 節開發一個具表達力但卻小心地予以限制的語言來表示規劃問題。第 10.2 節顯示了前向和後向搜尋演算法是如何能夠利用這種表示，主要透過可以自動根據表示結構得到的精確啓發式(這與第 6 章中爲限制滿足問題構造有效啓發式的方法類似)。第 10.3 節展示一個稱之爲規劃圖的資料結構可以使搜尋規劃更有效率。我們接著描述一些其他規劃的方法並與其他不同的方法相互比較。

這一章談的是單一代理人於完全可觀察的、確定性的、靜態的環境。第 11 與 17 章涵蓋具有數個代理人於部份可觀察的、隨機性的、動態的環境。

## 10.1 經典規劃的定義

第 3 章的問題求解代理人能夠找出能引導至目標狀態的行動順序。但它處理的是原子狀態的表示式，遂也需要好的特定領域啓發式才能有好的效能。第 7 章的混合命題邏輯代理人可在沒有特定領域啓發式之下找出規劃，因爲它根據問題的邏輯架構，使用了領域無關的啓發式。但它依賴基本(無變數)命題的推理，意指當有許多的行動與狀態時，它可能會被淹沒。舉例來說，在 wumpus 世界，向前移動一步的簡單行動得重複執行於所有四個代理人的位向、$T$ 個單位時間和 $n^2$ 個目前位置。

爲應付這一種情況，規劃研究人員決定以**分解表示法**(factored representation)來解決——此表示式將世界的一個狀態被表示成一群變數的組合。我們使用一個叫做 **PDDL** 語言，規劃領域定義語言(Planning Domain Definition Language)，它可以讓我們以一個行動模式表達全部的 $4Tn^2$ 個行動。PDDL 的版本有好幾個，我們選擇其中一個簡單的版本，並改變其語法使之與本書其餘部份一致[1]。我們現在示範 PDDL 是如何描述我們定義一個搜尋問題所需要的四個事件：初始狀態、某個狀態下可採取的行動、行動採取後發生的結果，以及目標測試。

每個**狀態**被表成一個流(fluent)的連言。其中流是基本的、無函數之原子。舉例來說，$Poor \wedge Unknown$ 可能表示了一個不幸的代理人的狀態，而一個包裹運送問題的狀態可能會是 $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$**資料庫語義**被使用：封閉的世界假設意指任何未提及的流都設定爲假，而唯一名稱假設意指 $Truck_1$ 和 $Truck_2$ 是不同的。在一個狀態中下述的流是不被允許的：$At(x, y)$(因爲它是非基本的)，$\neg Poor$(因爲它是一個否定)，$At(Father(Fred), Sydney)$(因爲它使用一個函數符號)。狀態的表示式是經過精心設計的，使得狀態可以被看成是一個可以透過邏輯推理運算來操作的流的連言，或是一個可以透過集合運算來操作的一組流的集合。**集合語義**有時候比較容易處理。

**行動**是由一組行動模式所描述，這組行動模式間接地定義出於問題求解之搜尋所需要的 ACTIONS($s$)和 RESULT($s$, $a$)函數。我們在第 7 章看到，任何用於行動描述的系統必須解決框架問題——說明行動的結果會使得哪些事情出現變化，而哪些事情又會保持不變。經典規劃的問題著重在什麼樣的行動會讓最大多數的事情保持不變。想想一個由一堆放在一個平面上的物件所組成的世界。輕推一個物件的行動會使該物件的位置改變向量 Δ。一個簡潔的行動描述應該只提及 Δ，而不應該提及所有留在原地不動的物件。PDDL 指明一個行動的結果的方式是，它會告知哪些事情發生變化了，而其他所有保持不變的事情則隻字不提。

一個基本(沒有變數)行動集可以用單一**行動模式**來表達。該模式是一個**提升**的表示式——它將推理層次從命題邏輯提升至一階邏輯之受到限制的子集。舉例來說，這是一個自某個位置將一架飛機飛航至另一個位置的行動模式：

$Action(Fly(p, from, to),$

PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$

EFFECT: $\neg At(p, from) \wedge At(p, to))$

該模式由行動名稱、使用於該模式之所有變數的名單、一個**前提**以及一個**效果**所組成。儘管我們還沒有說行動模式如何轉換成邏輯語句，想像變數都已經被全稱量化了。我們可以隨意的選定我們希望實例化之變數的值。舉例來說，這兒有一個基本行動，是將所有變數取代爲值所得到的：

$Action(Fly(P_1, SFO, JFK),$

PRECOND: $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)$

EFFECT: $\neg At(P1, SFO) \wedge At(P_1, JFK))$

一個行動的前提與效果各是文字的連言(正或負的原子語句)。前提定義了該行動在什麼狀態可以被執行，而效果則定義了執行該行動後的結果。一個行動 $a$ 可以於狀態 $s$ 被執行，如果 $s$ 蘊涵 $a$ 的前提。蘊涵也可以用集合語義來表示：$s \models q$ 若且唯若於 $q$ 中的每一個正文字是在 $s$ 中且於 $q$ 中每一個負文字則不是。於形式符號我們說

$$(a \in \text{ACTIONS}(s)) \quad \Leftrightarrow \quad s \models \text{PRECOND}(a)$$

其中任何一個 $a$ 中的變數都已被全稱量化的。例如，

$$\forall p, from, to\ (Fly(p, from, to) \in \text{ACTIONS}(s)) \quad \Leftrightarrow$$

$$s \models (At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to))$$

我們說行動 $a$ 於狀態 $s$ 是**適用的**，如果 $s$ 可以使得前提被滿足。當一個行動模式 $a$ 包含變數，它可以有數個可適用的實例。舉例來說，在圖 10.1 所定義之初始狀態下，*Fly* 行動可以被實例化爲 $Fly(P_1, SFO, JFK)$或是 $Fly(P_2, JFK, SFO)$，兩者於初始狀態都屬於可適用的。如果某個行動 $a$ 有 $v$ 個變數，那麼，於一個具有 $k$ 個名稱唯一之物件的領域中，於最壞情況下尋找可適用的基本行動的話，會耗費 $O(v^k)$時間。

$$\begin{array}{l}
Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK) \\
\quad \wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \\
\quad \wedge Airport(JFK) \wedge Airport(SFO)) \\
Goal(At(C_1, JFK) \wedge At(C_2, SFO)) \\
Action(Load(c, p, a), \\
\quad \text{PRECOND: } At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a) \\
\quad \text{EFFECT: } \neg At(c, a) \wedge In(c, p)) \\
Action(Unload(c, p, a), \\
\quad \text{PRECOND: } In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a) \\
\quad \text{EFFECT: } At(c, a) \wedge \neg In(c, p)) \\
Action(Fly(p, from, to), \\
\quad \text{PRECOND: } At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to) \\
\quad \text{EFFECT: } \neg At(p, from) \wedge At(p, to))
\end{array}$$

**圖 10.1　一個空中運輸規劃問題的 PDDL 描述**

有時候我們希望**命題化**一個 PDDL 問題——以一個基本行動集取代每個行動模式，然後使用一個命題求解器如 SATPL 來尋找一個解答。不過，當 $v$ 與 $k$ 很大的時候，這樣做並不實際。

於狀態 $s$ 下執行行動 $a$ 之**結果**被定義成狀態 $s$，以源起於狀態 $s$ 所形成之那組流來代表，移除了那些於該行動效果中以負文字出現之流[這我們稱爲**刪除表**或 DEL($a$)]，並加入了那些於該行動效果中以正文字出現之流(這我們稱爲**增加表**或 ADD($a$))：

$$\text{RESULT}(s, a) = (s - \text{DEL}(a)) \bigcup \text{ADD}(a) \tag{10.1}$$

舉例來說，對於行動 $Fly(P_1, SFO, JFK)$，我們會移除 $At(P_1, SFO)$並加入 $At(P_1, JFK)$。行動模式要求效果的任何變數也必須出現於前提中。這樣的話，當前提於狀態 $s$ 被滿足的時候，所有的變數會被限制住，且 RESULT($s$, $a$)會因此只是基本原子。換句話說，在 RESULT 運算下基本狀態是封閉的。

也請注意流不是外在地參照到時間，一如於第 7 章者。我們需要用下標代表時間，與如下形式的後繼公理

$$F^{t+1} \quad \Leftrightarrow \quad ActionCausesF^t \ \vee \ (F^t \ \wedge \ \neg ActionCausesNotF^t)$$

於 PDDL，時間與狀態隱含於行動模式中：前提必然會參照到時間 $t$ 且效果會參照到時間 $t + 1$。

一組行動模式看成是一個規劃領域的定義。一個於該領域內的特定問題以加入一個初始狀態以及一個目標而被定義出來。**初始狀態**是基本原子的一個連言(至於所有的狀態，在封閉世界假設之下，意指任何未提及之原子都是假值)。**目標**就如同一個前提：一個可能包含變數之文字連言 (正或負)，如 $At(p, SFO) \wedge Plane(p)$。任何變數都被視爲已被存在量化了，所以這個目標是要有架飛機在 *SFO*。當我們能夠尋找出一系列的行動，使之結束於蘊涵目標之狀態 *s*，則該問題就算被解出。舉例來說，狀態 $Rich \wedge Famous \wedge Miserable$ 蘊涵目標 $Rich \wedge Famous$，且狀態 $Plane(Plane_1) \wedge At(Plane_1, SFO)$ 蘊涵目標 $At(p, SFO) \wedge Plane(p)$。

現在我們已經以搜尋問題來定義規劃：我們有一個初始狀態，一個行動函數，一個結果函數，以及一個目標測試。在探究有效率的搜尋演算法之前，我們將先看看一些範例問題。

### 10.1.1 範例：航空貨物運輸

圖 10.1 顯示了一個航空貨物運輸問題，包括裝載及卸載貨物，以及從一地運送飛到另一地。這個問題可以定義 3 個行動：*Load*(裝載)，*Unload*(卸載)和 *Fly*(飛行)。行動用於兩個述詞：*In*(*c*, *p*)表示貨物 *c* 在飛機 *p* 內，*At*(*x*, *a*)表示物體 *x*(飛機或貨物)在機場 *a*。請注意要確定 *At* 述詞被適切地保持。當一架飛機從某個機場飛到另一個機場的時候，全部貨物都隨同飛機一起走。於一階邏輯，量化飛機內所有的物件會比較容易。但是基本的 PDDL 並沒有一個全稱量詞，所以我們需要一個不同的解決方案。我們使用的方法是說某件貨物當它 *In*(放在)飛機之內時，就不再說它 *At* 任何地方；只有當貨物被卸載後，才能說它 *At* 新的機場。所以 *At* 實際上意指「在給定的位置是可供使用的。」下面的規劃是這個問題的一個解：

$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), UnLoad(C_1, P_1, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_1, JFK, SFO), UnLoad(C_2, P_2, SFO)]$$

最後，存在如 $Fly(P_1, JFK, JFK)$僞行動的問題，這應該是無行動，但是這會出現矛盾的效果 (根據定義，效果將會包括 $At(P_1, JFK) \wedge \neg At(P_1, JFK)$)。通常會忽略這些問題，因爲它們很少導致錯誤的規劃。正確的方法是加入不等式前提，說飛往或飛離某個機場必須是不同的；請參見圖 10.3 中這個問題的另一個範例。

### 10.1.2 範例：備用輪胎問題

考慮更換一個洩了氣的輪胎的問題(圖 10.2)。更加準確地說，目標是把一個好的備用輪胎合適地裝配到汽車輪軸上，初始狀態是有一個漏氣的輪胎在輪軸上和一個好的備用輪胎在後備箱內。爲了保持簡單，我們這個問題的版本是一個十分抽象的，沒有難卸的固定螺母或其他複雜因素。只有 4 種行動：從後備箱(trunk)中取出備用輪胎(spare)，從輪軸(axle)卸下漏氣的輪胎(flat)，將備用輪胎裝在輪軸上(putting on)，徹夜將汽車留下(leaving it overnight)無人看管。我們假設汽車存放的環境特別差，這樣將它徹夜留下的效果是輪胎消失。這個問題的一個解答是[*Remove*(*Flat*, *Axle*)，*Remove*(*Spare*, *Trunk*), *PutOn*(*Spare*, *Axle*)]。

$$
\begin{aligned}
&Init(Tire(Flat) \wedge Tire(Spare) \wedge At(Flat, Axle) \wedge At(Spare, Trunk))\\
&Goal(At(Spare, Axle))\\
&Action(Remove(obj, loc),\\
&\quad \text{PRECOND: } At(obj, loc)\\
&\quad \text{EFFECT: } \neg At(obj, loc) \wedge At(obj, Ground))\\
&Action(PutOn(t, Axle),\\
&\quad \text{PRECOND: } Tire(t) \wedge At(t, Ground) \wedge \neg At(Flat, Axle)\\
&\quad \text{EFFECT: } \neg At(t, Ground) \wedge At(t, Axle))\\
&Action(LeaveOvernight,\\
&\quad \text{PRECOND:}\\
&\quad \text{EFFECT: } \neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)\\
&\qquad \wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk))
\end{aligned}
$$

**圖 10.2　簡單的備用輪胎問題**

## 10.1.3　範例：積木世界

一個最著名的規劃領域稱爲**積木世界**。這個領域由一組放在桌子上的立方體形狀的積木組成[2]。積木能夠被疊放，但是只有一塊積木能夠直接放在另一塊的上面。一個機器臂能夠拿起一塊積木並把它移到別的位置，無論是在桌子上還是在另一塊積木上。機械臂每次只能拿起一塊積木，所以它不能拿起一塊上面有其他積木的積木。目標總是建造一堆或多堆的積木，根據哪些積木在其他積木的上面進行指定。例如，目標是把積木 *A* 放到積木 *B* 上並且把積木 *C* 放在積木 *D* 上(見圖 10.4)。

我們用 *On*(*b*, *x*)表示積木 *b* 在積木 *x* 上，其中 *x* 是另一塊積木或是桌子。用 *Move*(*b*, *x*, *y*)來表示將積木 *b* 從積木 *x* 的上面移到積木 *y* 的上面。現在，移動 *b* 的一個前提是沒有其他積木在它的上面。於一階邏輯，這會是$\neg\exists x$ *On*(*x*, *b*)或換個寫法，$\forall x$ $\neg$*On*(*x*, *b*)。基本的 PDDL 不允許量詞，所以取而代之我們引進一個述詞 *Clear*(*x*)，當沒有任何東西於 *x* 時此述詞會爲眞(完整的問題描述於圖 10.3)。

$$
\begin{aligned}
&Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A)\\
&\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C))\\
&Goal(On(A, B) \wedge On(B, C))\\
&Action(Move(b, x, y),\\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge\\
&\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y),\\
&\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))\\
&Action(MoveToTable(b, x),\\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),\\
&\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))
\end{aligned}
$$

**圖 10.3　積木世界中的規劃問題：構造一個三積木塔。**
**一個解答是序列[*MoveToTable*(*C*, *A*), *Move*(*B*, *Table*, *C*), *Move*(*A*, *Table*, *B*)]**

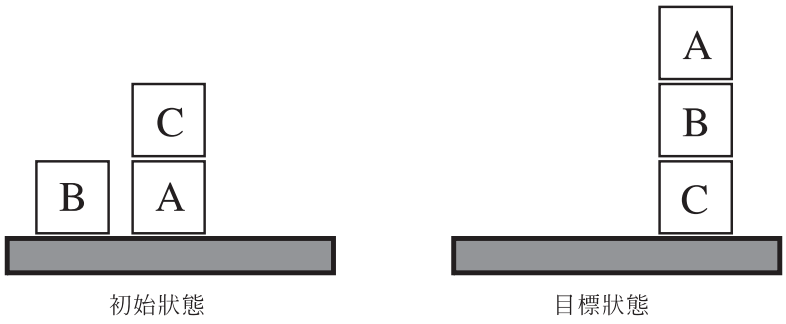


**圖 10.4　於圖 10.3 中積木世界問題之圖**

行動 *Move* 將積木 *b* 從 *x* 移動到 *y*，如果 *b* 和 *y* 上都沒有其他積木。當移動完成後，*x* 上沒有其他積木而 *y* 上不再是空的。第一次嘗試 Move 模式是

*Action*(*Move*(*b*, *x*, *y*),
　PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y)$,
　EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$)

不幸的是，當 *x* 或 *y* 是桌子時，這個行動不能確切地保持 *Clear*。當 $x = Table$ 時，這個行動的效果是 *Clear*(*Table*)，但是桌子不應該變爲上面沒有任何東西；當 $y = Table$ 時，這個行動的前提是 *Clear*(*Table*)，但是當移動一塊積木到桌子上時並不需要桌子上沒有任何東西。爲了修正這個問題，我們做兩件事。首先，我們引入另一個將積木 *b* 從 *x* 移到桌子上的行動：

*Action*(*MoveToTable*(*b*, *x*)),
　PRECOND: $On(b, x) \wedge Clear(b)$,
　EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$)

其次，我們將 *Clear*(*b*)解譯爲「*b* 上存在無物的空間可放一塊積木」。在這種解譯下，*Clear*(*Table*)將永遠是正確的。唯一的問題是沒有什麼可以阻止規劃器使用 *Move*(*b*, *x*, *table*)而不是 *MoveToTable*(*b*, *x*)。我們可以忍受這個問題——它將導致比實際需要大的搜尋空間，但是不會導致錯誤答案——或者我們可以引入述詞 *Block*，並給行動 *Move* 的前提增加 *Block*(*b*)*Block*(*y*)。

## 10.1.4　經典規劃的複雜度

於這個子節我們考慮規劃的複雜度理論，並區分兩個可判定性問題。**PlanSAT** 問的是是否存在任何規劃可以解出一個規劃問題的問題。**Bounded PlanSAT** 問的是是否存在一個長度不長於 k 的解答；這能夠被用於尋找一個最佳的規劃。

第一個結果是這兩個可判定問題都是爲可判定的經典規劃。證明的依據是狀態數目是有限的事實。但是如果我們加入函數符號到該語言，那麼狀態的數目變成是無限的，而 PlanSAT 變成僅僅是半可判定的：一個對於任何可求解的問題會停在正確的答案，但是對於無法求解的問題則不會停止下來之演算法存在。Bounded PlanSAT 問題，即使出現函數符號，仍然是可判定的。對於本節聲言之事的證明，請見 Ghallab 等人(2004)。

PlanSAT 與 Bounded PlanSAT 兩者都是在 PSPACE 之中，這是一個較諸 NP 更大的類別(因此更困難) 並指的是一種能夠被可決定性之圖靈機，於多項式量級的空間下解出的問題。即使如果我們制定相當嚴格的限制，該問題仍然非常的困難舉例來說，如果我們不允許負的效果，這兩種問題仍然是 NP-hard。不過，如果我們也不允許負的前提，PlanSAT 縮減爲 P 類。

這些最壞情況下的結果似乎令人沮喪。我們感到慰藉的是代理人通常不會被要求針對最壞情況問題之實例去尋找規劃，反而是被要求尋找特定領域的規劃 (如具有 $n$ 塊積木的積木世界問題)，這比理論上的最壞情況要簡單得多。對於許多領域(包括積木世界以及空中貨物運輸世界)，Bounded PlanSAT 是 NP-complete，然而 PlanSAT 屬於 P；換言之，最佳的規劃通常很難，但是次佳的規劃有時候卻是很簡單的。想要在比最壞情況問題還容易的情況下做的好，我們將需要好的搜尋啓發式。這是經典規劃形式主義眞正優越之處：它已經減輕非常準確之領域無關啓發式的開發工作，然而以一階邏輯後繼狀態公理爲本的系統在獲致好的啓發式方面的成效並不顯著。

## 10.2 規劃演算法作為狀態空間搜尋

現在我們把注意力轉到規劃演算法上。我們已看到規劃問題的描述如何的定義出一個搜尋問題：我們能夠從初始狀態出發走遍狀態空間，來搜尋一個目標。行動模式之陳述表示式的好處之一是我們也能夠從目標逆向搜尋，搜尋初始狀態。圖 10.5 比較了前向與逆向搜尋。

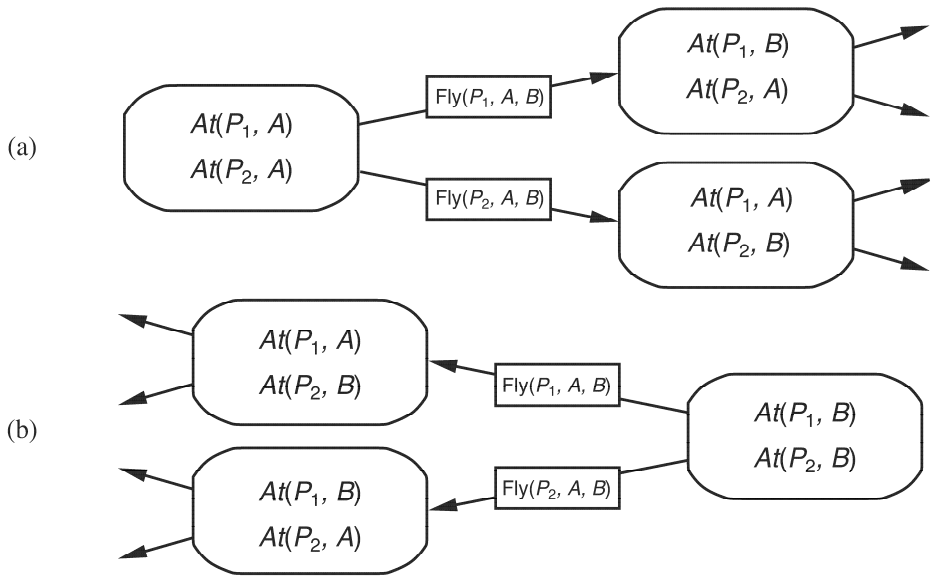


**圖 10.5　一個規劃的兩種搜尋方法**。(a) **對於狀態空間的前向**(progression)**搜尋，自初始狀態開始，並使用問題的行動向前搜尋目標狀態集的一員**。(b) **對一組相關狀態的逆向**(regression)**搜尋，自代表目標的一組狀態開始，並使用相反的行動，逆向搜尋出初始狀態**

### 10.2.1 前向狀態空間搜尋

現在，我們已經展示規劃問題如何映射到搜尋問題，我們能夠以第 3 章任何一個啓發式搜尋演算法或是第 4 章的區域搜尋演算法來解出規劃問題(只要我們持續的追蹤到達目標所使用到的行動)。從規劃搜尋的初期(約 1961 年)直到最近(約 1998 年)，人們認爲前向狀態空間搜尋太無效率而不能實際應用。想要得出箇中理由並非難事。

第一，前向搜尋傾向發掘無關聯的行動。考慮一個購買 AI 書籍的神聖任務：摩登的方法是透過線上書店。假設有一個行動模式 *Buy*(*isbn*)具有效果 *Own*(*isbn*)。ISBNs 是 10 個數字，所以這個行動模式代表 10 億個基本行動。一個茫然無知的前向搜尋演算法將必須開始一一列舉這 10 億個行動以尋找出一個可以走到目標者。

第二，規劃問題常常有非常龐大的狀態空間。考慮一個有 10 個機場，每個機場有 5 架飛機和 20 件貨物的航空貨物問題。目標是將所有的貨物從機場 *A* 運送到機場 *B*。這個問題有一個簡單的解：將 20 件貨物裝載到機場 *A* 的一架飛機上，飛機飛到機場 *B*，卸載所有貨物。找到解可能很困難，因爲平均分支因數是巨大的：50 架飛機中的任何一架可以飛到 9 個其他的機場，200 件包裹中的每一件也能一樣被卸載(如果已經裝載了)或者裝載到機場的任何一架飛機上(如果還沒裝載)。所以於任何狀態下最少時會有 450 個行動(當所有的貨物都在機場但沒有任何飛機)最多時會有 10,450 個行動(當所有的貨物與飛機都在相同的機場)。平均而言，我們說存在大約 2000 個可能的行動，所以達到明顯解的深度的搜尋樹大約有 $2000^{41}$ 個節點。

顯然地，即使這個相當小型的問題實例也沒有一個正確的啓發式。儘管許多的規劃在實際世界的應用仰賴於領域特定的啓發式，不過(如同我們在第 10.2.3 節所見)強的領域無關啓發式能夠被自動地導出；這也使得前向搜尋變得可行。

### 10.2.2 逆向相關聯狀態搜尋

於逆向搜尋我們從目標開始並逆向地實踐行動直到我們找出能抵達初始狀態的一系列步驟。此稱之爲**相關聯狀態**搜尋。因爲我們僅僅考慮與目標(或目前的狀態)有所關聯的行動。如同於信度狀態搜尋(第 4.4 節)，在每一個步驟都會有一組相關聯狀態要考慮，而不僅是單一個狀態。

我們從目標開始，它是一個構成一組狀態描述的文字連言——舉例來說，目標$\neg Poor \wedge Famous$描述那些 *Poor* 爲假，*Famous* 爲眞，以及任何其他流能爲任何值的狀態。如果於某個領域有 $n$ 個基本流，那麼有 $2^n$ 個基本狀態(每一個流能有眞或假)，但是 $3^n$ 組的目標狀態描述(每個流能夠爲正、負或是略而不提)。

一般來說，只有當我們知道如何從一個狀態描述逆向至前導狀態描述的時候逆向搜尋才管用。舉例來說，想要逆向搜尋一個 $n$-皇后問題的解答就很困難，因爲沒有一個簡單的方法可用來描述目標移動一步後的狀態。令人高興的是，PDDL 表示式的設計上就是讓逆向行動變得更容易——如果一個領域能夠以 PDDL 中表示，那麼我們就能夠對之做逆向搜尋。已知基本目標描述 $g$ 以及一個基本行動 $a$，從 $g$ 逆向於 $a$ 讓我們得到的狀態描述 $g'$ 定義如下

$$g' = (g - \text{ADD}(a)) \bigcup Precond(a)$$

也就是說，因該行動而加入之效果在行動之前不必已經爲眞，同時前提在行動之前必須已經成立，否則該行動不可能會已經執行過的。請注意 DEL($a$)並沒有出現在公式中；那是因爲當我們知道 DEL($a$)中的流在該行動之後不再爲眞，我們不知道它們先前是否爲眞，所以無法對它們說什麼了。

想要完整地享用逆向搜尋的優越之處，我們必須處理部份未實例化的行動與狀態，而不僅僅是基本者。舉例來說，假設目標是運送特定的貨物到 SFO：$At(C_2, SFO)$。這提議了行動 $Unload(C_2, p', SFO)$：

$Action(Unload(C_2, p', SFO),$

PRECOND: $In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$

EFFECT: $At(C_2, SFO) \wedge \neg In(C_2, p')$

[請注意我們已經**標準化**了變數名字(在這個情況 $p$ 改成 $p'$)使得如果我們湊巧的於一個規劃中使用相同的行動模式兩次時，不會出現變數名字相互混淆的情況。相同的做法也於第 9 章之一階邏輯推理被採用]。這表示從一架位在 SFO 之未指明的飛機卸下貨物；任何的飛機都行，但是我們現在不說是哪一架。我們能夠利用一階表示式的優越之處：一個單一描述透過間接地量化 $p'$的方式，總結出使用任一架飛機的可能性。逆向的狀態描述是

$$g' = In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$$

最後的問題是判別哪一個行動是逆向實施的候選者。於前向方向，我們選擇**可適用**的行動——哪些是於規劃中可以被用來當做下一步的行動。於逆向搜尋，我們要的是**有關聯**的行動——哪些行動會是於規劃中可以引領至目前之目標狀態的最後一步。

對於一個行動要與目標相關聯，很明顯的必須對目標有所挹助：至少行動的效果之一(不論正或負)必須與目標的某個要素吻合。較不顯而易見的是，該行動必須沒有任何否定目標要素的效果(正或負)。現在，如果目標是 $A \wedge B \wedge C$ 而一個行動有效果 $A \wedge B \wedge \neg C$ 那麼以通俗的意義來說行動與目標是強烈的相關——它讓我們往前邁過了全程的三分之二。但是此處所定義者在技術意義而言是不相關聯的，因爲這個行動不會是一個解答的最後一步——我們總是需要多走一步才能完成 $C$。

已知目標 $At(C_2, SFO)$，以及 $Unload$ 的數個實例是相關聯的：我們可選取任何特定的飛機來卸貨，或是我們可以使用行動 $Unload(C_2, p', SFO)$而不指明哪一架飛機。在不排除任何的解答的情況下，使用代入最一般的量詞(標準化的)行動模式所形成之行動，我們能夠因此降低分支因數。

又另一個例子，考慮目標 $Own$(0136042597)，已知初始狀態爲百億個 ISBN，以及單一行動模式。

$A = Action(Buy(i),$ PRECOND: $ISBN(i),$ $EFFECT$: $Own(i))$

如我們前面曾提過，沒有啓發式的前向搜尋必須開始列舉百億個基本 $Buy$ 行動。但是用逆向搜尋，我們會統一目標 $Own$(0136042597)與(標準化的)效果 $Own(i')$，得到置換式 $\theta = \{i'/0136042597\}$。那麼我們會逆向於行動 $Subst(\theta, A')$已得到前導狀態描述 $ISBN$(0136042597)。這是初始狀態的一部分，因而初始狀態被其所蘊涵，所以我們達成目標了。

我們能夠讓這個更正式。假設一個包含目標文字 $g_i$ 與一個行動模式 $A$ 的目標描述 $g$ 被標準化而產生 $A'$。如果 $A'$有一個效果文字 $e'_j$ 其中 $Unify(g_i, e'_j) = \theta$ 且於此我們定義 $a' = \text{SUBST}(\theta, A')$，且如果於 $a'$中沒有一個是 $g$ 中文字的否定式，那麼 $a'$是一個趨向 $g$ 的相關聯行動。

對於大多數的問題領域，逆向搜尋會將分支因數保持得低於前向搜尋者。不過，逆向搜尋事實上使用的是一組狀態而非個別的狀態，使得它難以得到好的啓發式。這也就是爲什麼目前多數的系統仍偏愛於前向搜尋的主要理由。

### 10.2.3 啓發式規劃

不論是前向或是逆向搜尋，若沒有好的啓發式函數，都不會是有效率的。回想第 3 章提及一個啓發式函數 $h(s)$估計從狀態 $s$ 至目標的距離以及如果我們能夠爲此距離推導一個**可採納**啓發式——一個不會過度估計者——那麼我們就能夠使用 A*搜尋來尋找最佳的解答。一個啓發式能夠藉由定義一個比較容易求解的**鬆弛問題**來導出。這個較容易問題之解答的正確成本及變成原問題的啓發式。

根據定義，並沒有任何辦法可以分析一個原子狀態，也因此它需要人類分析師的些許智巧去針對原子狀態之搜尋問題定義一個好的領域特定啓發式。使用一個分解好的狀態與行動模式表示式來規劃。就可能定義出好的領域無關啓發式，以及自動地對一個已知的問題運用一個好的領域無關啓發式。

將搜尋問題想成是一個圖形，圖形的節點是狀態而其邊是行動。這個問題就是要找出從初始狀態連至目標狀態的一條路徑。有兩個方式讓我們能夠鬆弛這個問題使之更加容易：加入更多的邊到圖上，使得它極爲容易地尋找出一條路徑，或是將數個節點集合在一起，構成一個有較少狀態數目的精簡狀態空間，而因此變得容易搜尋。

我們首先看看增加圖形邊數的啓發式。舉例來說，**忽略前提啓發式**將所有的前提自行動身上取掉。每個行動變成適用於每一狀態，且能以一步達到任何單一目標流(如果有可用的行動——若沒有，問題是不可能的)。這幾乎暗示求解鬆弛問題所需的步驟數目等於未滿足的目標數目—幾乎但是不全然，因爲 (1) 某些行動能完成數個目標且 (2) 某些行動能抵消其他行動的效果。對許多問題，一個準確的啓發式是透過考慮 (1) 並忽略 (2) 而得到。首先，我們拿掉所有的前提與所有的效果，除了那些屬於目標中的文字外，來鬆弛行動。接著，我們計算所需行動的最小數目，而這些行動的正效果的並集能夠滿足目標。這是一個**涵蓋集問題**的實例。有一個次要的惱人問題：涵蓋集問題是 NP 難題。幸運的是，一個簡單的貪婪涵蓋集演算法保證返回一個處於眞正最小值的 log $n$ 倍範圍內的值，其中 $n$ 是目標中的文字個數。不幸的是，貪婪演算法不能保證可採納性。

也可以僅忽略幾個被選定行動的前提。考慮第 3.2 節的華容道遊戲(8 方塊或 15 方塊)。我們能夠將這個寫成具有單一模式 Slide 之方塊規劃問題：

$$
\begin{aligned}
&Action(Slide(t, s_1, s_2),\\
&\quad PRECOND: On(t, s_1) \wedge Tile(t) \wedge Blank(s_2) \wedge Adjacent(s_1, s_2)\\
&\quad EFFECT: On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2))
\end{aligned}
$$

一如 3.6 節所見，若拿掉了前提 $Blank(s_2) \wedge Adjacent(s_1, s_2)$，那麼任何方塊都能夠於一個行動之下移動至任何空位而我們得到 number-of-misplaced-tiles 啓發式。如果我們拿掉了 $Blank(s_2)$那麼我們得到 Manhattan-distance 啓發式。很容易可以看出這些啓發式如何能自動地從行動模式描述被導出。與搜尋問題的原子表示式相較下，操縱模式的簡易性是規劃問題的分解表示式最大優越之處。

另一可能性是**清空刪除列表啓發式**。假設有一瞬間，所有目標及前提僅包含正文字[3]。我們想要創造一個原來問題的鬆弛版本，這會較容易求解，且其中的解答長度會是不錯的啓發式。作法是從所有行動中移掉刪除列表(亦即，從效果中移掉所有負文字)。這使我們能以單調進程朝目標前進——沒有行動能取消另一行動所產生的進程。結果則是，求出此鬆弛問題的最佳解仍屬 NP 難題，但利用爬山法可在多項式時間內找出近似解。圖 10.6 顯示出，使用清空刪除列表啓發式下，兩個規劃問題的狀態空間。點代表狀態，邊代表行動，而點在底平面上的高度代表啓發值。在底平面上的狀態則爲解。在兩問題中，存有一寬路徑到達目標。沒有死路，所以不需要回溯；簡單的爬山搜尋法可輕易找出這些問題的解(雖然可能不是最佳解)。

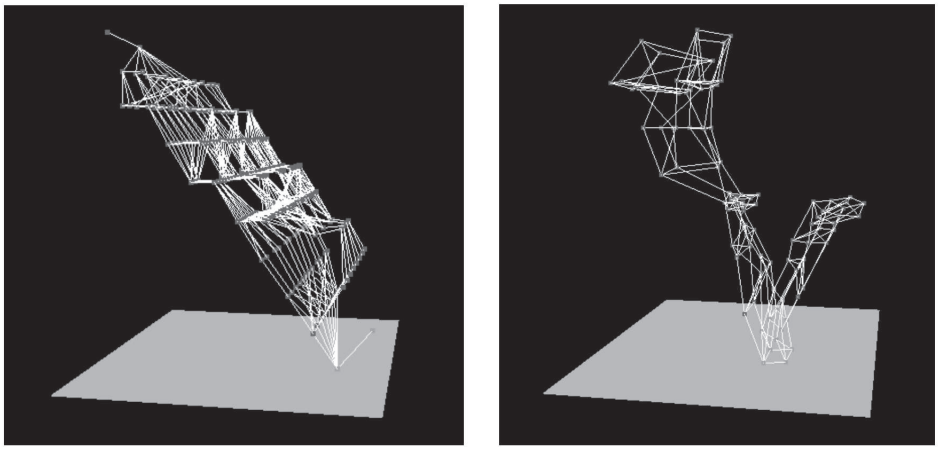

**圖 10.6　由清空刪除表啓發式下的規劃問題的兩個狀態空間。離底面上的高度等於一個狀態的啓發式分數；位在底面的狀態就是目標。沒有局部極小值，所以搜尋目標可非常直接。取自** Hoffmann(2005)

鬆弛問題留給我們一個簡化的——但仍很昂貴——規劃問題只爲了計算啓發函數之值。許多規劃問題有 10100 或更多狀態，而鬆弛行動無助於降低狀態數目。因此，現在關注那些形成**狀態抽象**來減少狀態數目的鬆弛——是一個從問題的基本表示式內的狀態映射到抽象表示式的多對一映射。

狀態抽象化最簡單的形式是清空某些流。舉例來說，考慮一個有 10 座機場、50 架飛機與 200 件貨物的空中貨運問題。每一架飛機能夠出現在 10 座機場中的任一座，且每件貨物能夠能夠在任一架飛機上或是在其中一座機場被卸下來。所以共有 $50^{10} \times 200^{50+10} \approx 10^{155}$ 個狀態。現在考慮一個在該領域的特別問題於此所有貨物只出現在其中的 5 座機場，且所有在給定的機場之貨物都會有相同的送達地點。那麼該問題的一個有用抽象是取掉所有的 *At* 流，除了那些有架飛機及有件貨物在 5 座機場中某一座的情況。現在僅有 $5^{10} \times 5^{5+10} \approx 10^{17}$ 個狀態。這個抽象狀態空間的解答會較原始空間之解答爲短(也因此可以是一個可採納啓發式)，而該抽象解答很容易被擴充爲原問題的解答(透過加入 Load 與 Unload 行動)。

於定義啓發式中一個關鍵性的想法是**分解**：將一個問題劃分爲好幾個部分，分別地解出各部份，然後將個部份予以合併。**子目標獨立性**假設所述爲，求解子目標連言的成本可近似爲獨立求解每個子問題的成本總和。子目標獨立假設可能是樂觀的，也可能是悲觀的。當每個子目標的子規劃間存在負相互作用時，它是樂觀的——例如，當一個子規劃的行動刪除了另一個子規劃取得的一個目標時。當子規劃包含冗餘行動時，它是悲觀的，因此也是不可採納的——例如，在合併的規劃中，兩個行動可以被一個單一行動所替代。

假設目標是一組流 $G$，我們將之劃分爲戶無交集的子集合 $G_1, ..., G_n$。我們然後尋找能解出各子目標的規劃 $P_1, ..., P_n$。一個爲達到所有 $G$ 的之規劃的成本估算會是多少？我們能夠將每個 $Cost(P_i)$ 想像成是一個啓發式估算，而我們知道如果我們取出估計值中的最大值來組合出估算值，我們必然會得到一個可採納啓發式。所以 $max_i COST(P_i)$是可採納的，而有時候它是完全正確的：也有可能 $P_1$ 無心插柳地達成所有的 $G_i$。但是於大多數情況，於實踐時該估算結果多偏低。我們能不能換個方式來做，就是將所有的成本加起來？對於許多問題那是一個合理的估計值，但是它並非可採納的。最好的情況是當我們能夠判斷出 $G_i$ 與 $G_j$ 爲**獨立**。如果 $P_i$ 的效果並不會讓所有的前提與 $P_j$ 的目標有任何的改變，那麼估計值 $COST(P_i) + COST(P_j)$會是可採納的，而且較諸取最大值做爲估計值的方式來得準確。我們於第 10.3.1 節展示規劃圖形有助於提供更好的啓發式估計。

明顯的是，在壓縮搜尋空間的做法方面極具潛力的做法是形成抽象。其訣竅在於選取對的抽象並以一種使得總成本——定義一個抽象，執行一個抽象搜尋，然後將該抽象映射原始問題——低於求解原問題的成本的方式來使用。第 3.6.3 節的樣式資料庫技術是很有用的，因爲創造一個樣式資料庫的成本能夠分攤至數個問題實例。

一個有效使用啓發式的系統的範例是 FF，或 FASTFORWARD(Hoffmann，2005)，是一個使用清空刪除表啓發式的前向狀態空間搜尋器，在規劃圖的幫助下估算啓發式(見第 10.3 節)。FF 然後使用爬山搜尋法(略事修改過以便能追蹤該規劃)於該啓發式以尋找一個解答。當它爬到一個平台或是區域最大值的時候——當沒有任何行動會讓一個狀態有更佳的啓發式分數——那麼 FF 使用疊代式深入搜尋法直到它找到一個更好的狀態，或是放棄並重新開始爬山搜尋工作。

## 10.3 規劃圖

我們已經建議過的所有的能夠忍受不準確性的啓發式。這一節說明一個稱爲**規劃圖**的特殊資料結構是如何被用於給出更精確的啓發式估計的。這些啓發式能夠應用到迄今爲止我們看到的任何一種搜尋技術中。另一方面，我們能夠使用一個稱之爲 GRAPHPLAN 的演算法，沿著規劃圖所形成的空間搜尋一個解答。

規劃問題問的是我們是否能夠從一個初始狀態到達目標狀態。假設我們已經知道從初始狀態到後繼狀態，以及它們的後繼者等等所有可能行動的樹。如果我們將這株樹予以適當地索引化，則只需要看看它，我們就能夠立刻回答規劃問題「我們是否能夠從狀態 $S_0$ 達到狀態 $G$」。當然，這株樹的規模是呈指數量級地成長，所以這方法是不實際的。一個近似這株樹而大小爲多項式量級的規劃圖能夠很快地被建構出來。規劃圖無法肯定地回答是否從 $S_0$ 可以抵達 $G$，但是它能夠估計抵達 $G$ 需要走多少個步驟。當它回報說目標並非可達到者的時候，該估計都是正確的且它從不會過度的估計步驟的數目，所以它是一個可採納的啓發式。

一張規劃圖是一種組織成好幾個**階段**的有向圖：首先初始狀態之 S0 階段，由那些代表於 S0 爲眞的各個流之節點所組成；然後 A0 階段是由各個於 S0 中適用於基本行動之節點所組成；那麼隨著 Ai 而更換到 Si 階段；直到我們抵達一個終止條件 (稍後會討論)。

粗略的說，$S_i$ 包含了所有在時間 $i$ 可能成立的文字，視於前個單位時間所執行的行動而定。如果 $P$ 或 $\neg P$ 可能會成立，那麼兩者都會於 $S_i$ 中被表示出來。也粗略的說，$A_i$ 包含了所有會使它們的前提在時間 $i$ 能被滿足的行動。我們說「粗略的說」因爲規劃圖僅記錄各個行動之間可能的負互動的集合；因此，一個文字可能顯示止於階段 $S_j$，而實際上它不會爲眞，一直到後面的階段，如果有的話(一個文字永遠不會出現得太遲)。儘管可能有誤差，一個文字首次出現之乙階段是對從初始狀態達到該文字的困難程度的好估計。

規劃圖只對命題規劃問題起作用——沒有變數的規劃。如在 10.1 節所提，對一組行動模式進行命題化是很直接的。儘管使問題描述的大小增加，規劃圖仍證明爲解決困難規劃問題之有效工具。

圖 10.7 顯示了一個問題，圖 10.8 顯示了它的規劃圖。在階段 $A_i$ 的每個行動會連接到它在 $S_i$ 的前提以及在 $S_{i+1}$ 的效果。所以一個文字會出現是因爲某個行動所造成的，但是我們也希望說一個文字能夠持續，如果沒有任何行動否定它。這會以一個**持續行動**(persistence action)來表示(有時候被稱爲 *no-op*)。對每個正文字和負文字 $C$，我們將具有前提 $C$ 和效果 $C$ 的持續行動添加到問題中。圖 10.8 顯示了一個「實際」行動，$A_0$ 中的 *Eat*(*Cake*)，以及兩個用小方塊畫的持續行動。

階段 $A_0$ 包含了狀態 $S_0$ 中能夠發生的所有行動，但是同樣重要的是它記錄了行動之間的衝突，會防止它們一起發生。圖 10.8 中的灰色線條表示**互斥**(mutual exclusion，或縮寫爲 mutex)連接。例如，*Eat*(*Cake*)與持續行動 *Have*(*Cake*)或¬*Eaten*(*Cake*)是互斥的。不久我們將看到互斥連接是如何計算的。

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
  PRECOND: $Have(Cake)$
  EFFECT: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
  PRECOND: $\neg Have(Cake)$
  EFFECT: $Have(Cake))$

**圖 10.7　「有蛋糕也要吃蛋糕」問題**

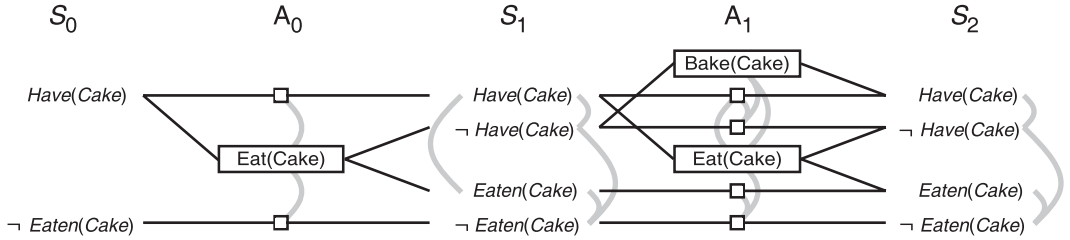


**圖 10.8　到達階段 $S_2$ 的「有蛋糕也要吃蛋糕」問題之規劃圖。矩形表示行動(小矩形表示持續行動)，直線表示前提和效果。互斥連接用灰色曲線表示。未繪出所有互斥連接都畫出來，否則會變得太混雜。一般若兩文字在 $S_i$ 為互斥，那麼那些文字的持續行動在 $A_i$ 會是互斥的，而我們不需要畫出那條互斥連接**

階段 $S_1$ 包含了所有的文字，這些文字能夠由挑取任何於 $A_0$ 中行動的子集合所得到，以及那些指出哪些文字不能一起出現，不論所選的行動爲何，的互斥連接(灰線)。例如，*Have*(*Cake*)和 *Eaten*(*Cake*)是互斥的：取決於 $A_0$ 中的行動選擇，一個或另一個可以稱爲結果，但不是都可以。換句話說，$S_1$ 代表一個信度狀態：一個可能狀態的集合。這個集合的成員是所有使任何子集合成員之間沒有互斥連接之文字的子集合。

我們按這種方式繼續，在狀態階段 $S_i$ 和行動階段 $A_i$ 之間交替，直到我們到達兩個相繼階段等同的階段。這時，我們說圖已經**平整**了。於圖 10.8 中的圖平整於 $S_2$。

我們以一個結構告終，其中每個 $A_i$ 階段包含所有可用於 $S_i$ 中的行動，以及說明了哪些對行動不能被同時執行的限制。每個 $S_i$ 階段包含能夠根據 $A_{i-1}$ 中可能選擇的任何行動得到的所有文字，以及說明哪些對文字不可能出現的限制。認識到規劃圖的建構過程不需要在行動中進行組合搜尋的選擇是重要的。相反，它只是用互斥連接記錄某些選擇的不可能性。

我們現在爲行動和文字定義互斥連接。如果下面 3 個條件中的任何一個成立，那麼一個給定階段的兩個行動之間的互斥關係成立：

- **不一致效果**：
  一個行動否定另一個行動的效果。例如，*Eat*(*Cake*)和 *Have*(*Cake*)的持續有不一致效果，因爲它們在效果 *Have*(*Cake*)上不一致。
- **衝突**：
  一個行動的效果之一是另一個行動的前提之一的否定式。例如，*Eat*(*Cake*)透過否定持續行動 *Have*(*Cake*)的前提而和它衝突。
- **競爭需要**：
  一個行動的前提之一和另一個行動的前提之一互斥。例如，*Bake*(*Cake*)和 *Eat*(*Cake*)是互斥的，因爲它們在前提 *Have*(*Cake*)的值上有競爭。

同一階段的兩個文字，如果一個是另一個的否定式或者能獲得這兩個文字的每一對可能行動都是互斥的，那麼這兩個文字之間的互斥關係成立。這個條件稱爲不一致支持。例如，在 $S_1$ 中，*Have*(*Cake*)和 *Eaten*(*Cake*)是互斥的，因爲獲得持續行動 *Have*(*Cake*)的唯一途徑與獲得 *Eaten*(*Cake*)的唯一途徑即 *Eat*(*Cake*)互斥。在 $S_2$ 中這兩個文字不互斥，因爲有新的途徑可以獲得它們，諸如 *Bake*(*Cake*)與 *Eaten*(*Cake*)的持續，它們不互斥。

規劃圖於規劃問題的大小上屬於多項式量級。對於一個具有 $\ell$ 個文字與一個行動的規劃問題，每個 $S_i$ 沒有多於 $\ell$ 個節點與 $\ell^2$ 個互斥連接，而且每一個 $A_i$ 沒有多於 $a + \ell$ 個節點(包括無行動)，$(a + \ell)^2$ 互斥連接，以及 $2(a\ell + \ell)$個前提與效果連接。因此，整張具有 $n$ 階段的圖其大小爲 $O(n(a + \ell)^2)$。建構出該圖的時間需要相同的複雜度。

### 10.3.1 規劃圖的啓發式估計

一個規劃圖一旦被建構，它就是關於問題的資訊的豐富來源。第一，如果任何目標文字沒有出現在圖的最後一個階段，那麼該問題屬於無法被解出的。第二，我們能夠估計從狀態 $s$ 到達任何目標文字 $g_i$ 的成本，當 $g_i$ 在由初始狀態所建構之規劃圖第一次出現的階段。我們稱這個爲 $g_i$ 的**階段成本**。在圖 10.8 中，*Have*(*Cake*)的階段成本爲 0，*Eaten*(*Cake*)的階段成本爲 1。很容易說明這些估計對單獨目標是可採納的(習題 10.10)。然而這個估計可能不是很好，因爲規劃圖允許每一階段有數個行動，而啓發式只對階段而不對行動數進行計數。由於這個原因，使用**串列規劃圖**計算啓發式是常見的。串列規劃圖堅持在任何給定的時間步只能有一個行動實際發生；這是透過在除了持續行動之外的每對行動間添加互斥連接而實作的。根據串列規劃圖提取的階段成本往往是對實際成本的一個相當合理的估計。

爲了估計目標連言的成本，有 3 種簡單的方法。**最大階段**啓發式簡單地選取任何目標的最大階段成本值；這是可採納的，但是不一定很準確。

**階段總和**啓發式，根據子目標獨立假設，返回目標的階段成本的總和；這是不可採納的，但是在實際中對那些很大程度上可以分解的問題很有效。它比第 10.2 節中的不可滿足目標數啓發式精確得多。對於我們的問題，目標連言 *Have*(*Cake*) ∧ *Eaten*(*Cake*)的階段總和啓發式估計會是 0 + 1 = 1，而正確的答案是 2(由規劃[*Eat*(*Cake*), *Bake*(*Cake*)]達成)。那看起來並不是很糟。一個更嚴重的錯誤是，如果 *Bake*(*Cake*)不在行動集之中，當事實上目標連言爲不可能的時候，該估計會仍然是 1。

最後，**固定階段**啓發式找到一個階段，在這個階段目標連言中的所有文字都出現在規劃圖中，而且其中不存在任何一對是互斥的。這個啓發式對我們的原始問題給出了正確的值 2，對沒有 *Bake*(*Cake*)的問題給出無窮大。它優於最大階段啓發式，在子規劃之間有很多相互作用的任務中產生了極好的效果。當然，它並不完美，舉例來說，它省略掉了三個或更多的文字之間的互動。

作爲產生精確啓發式的工具，我們可把規劃圖視爲一高效可解的鬆弛問題。爲了理解鬆弛問題的特性，我們需要確切理解文字 $g$ 在規劃圖中 $S_i$ 階段出現的含義。理想地，我們希望它能保證存在一個獲得 $g$ 的有 $i$ 個行動階段的規劃，同時希望當 $g$ 不出現時也就沒有這樣的規劃。不幸的是，要得到這個保證與求解原始規劃問題一樣困難。所以規劃圖能夠提供後一半保證(如果 $g$ 不出現，就沒有這樣的規劃)，但是如果 $g$ 眞的出現了，那麼規劃圖能承諾的全部是有一個規劃可能獲得 $g$ 而且沒有「明顯的」缺陷。明顯缺陷是這樣定義的：在某個時刻透過考慮兩個行動或兩個文字就能檢測出的缺陷——換句話說，透過觀察互斥關係。可能有涉及三個、四個或更多行動的複雜缺陷，但是經驗顯示不值得爲了明確這些可能的缺陷而耗費計算努力。這與從限制滿足問題中學到的經驗類似，通常在搜尋解之前計算二元一致性是值得的，但通常計算三元或更高一致性就不值得了(參見 6.2 節)。

一個同樣無法同樣地被規劃圖所認出的不可解的問題的例子之一是積木世界問題，其目標是讓 $A$ 位在 $B$ 之上，$B$ 在 $C$ 之上，且 $C$ 在 $A$ 之上。這是一個不可能的目標；一個塔的底部放在頂部的上面。但是一張規劃圖無法察覺這樣的不可能性，因爲三個子目標中的任何兩個都是可達成的。任何一對文字之間沒有任何互斥，僅僅將三個文字之間視爲一體時才有。想要察覺出這樣的問題是不可能的，我們將必須從頭到尾的搜尋整張規劃圖。

## 10.3.2 GRAPHPLAN 演算法

這一子節說明如何從規劃圖中直接抽取一個規劃，而不只是使用規劃圖來提供啓發式。GRAPHPLAN 演算法(圖 10.9)重複地以 EXPAND-GRAPH 將一個階段加到規劃圖中。一旦所有目標於圖中以非互斥的面貌出現，GRAPHPLAN 呼叫 EXTRACT-SOLUTION 來搜尋一個求解該問題的規劃。如果失敗了，它展延到令一個階段並再試一次，當沒有任何理由再繼續時就以失敗告終。

現在我們來跟蹤 GRAPHPLAN 演算法在第 10.1.2 節中的備用輪胎問題上的操作。該演算法如圖 10.10。GRAPHPLAN 的第一條線將規劃圖初始化爲一階段($S_0$)圖來代表初始狀態。從問題描述之初始狀態的正流如所示，相關聯的負流亦然。沒有顯示出來的是未發生改變的正文字[如 *Tire*(*Spare*)]以及無關聯的負文字。目標文字 *At*(*Spare*, *Axle*)沒有出現在 $S_0$ 中，所有我們不需要呼叫 EXTRACT-SOLUTION——我們確定還沒有解。相反，EXPAND-GRAPH 添加了前提存在於 $S_0$ 階段中的 3 個行動[即除了 *PutOn*(*Spare*, *Axle*)之外的所有行動]，以及一併添加了 $S_0$ 所有文字的持續行動。行動的效果被添加到 $S_1$ 階段。然後 EXPAND-GRAPH 尋找互斥關係並將它們添加到圖中。

```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
    if goals all non-mutex in S_t of graph then
      solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution ≠ failure then return solution
    if graph and nogoods have both leveled off then return failure
    graph ← EXPAND-GRAPH(graph, problem)
```

**圖 10.9** GRAPHPLAN 演算法。GRAPHPLAN 呼叫 EXPAND-GRAPH 以加入一個階段，直到由 EXTRACT-SOLUTIONS 找出一個解答，或是不可能有解答

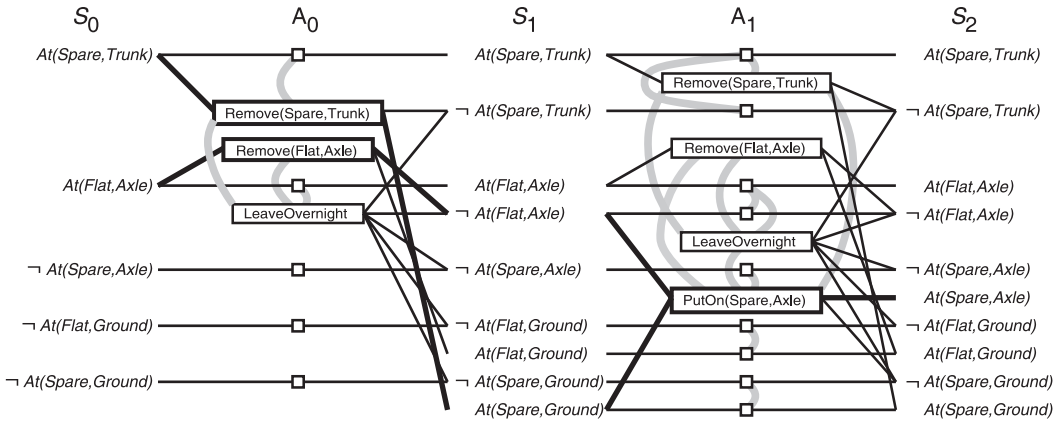


**圖 10.10** 擴展到 $S_2$ 階段之後的備用輪胎問題規劃圖。互斥連接用灰線表示。並無繪出所有連接，否則全畫出來時，圖會變得很混亂。解用粗線和粗輪廓線表示

*At*(*Spare*, *Axle*)仍然沒有在 $S_1$ 中出現，所以我們仍然不需要呼叫 EXTRACT-解答。我們再次呼叫 EXPAND-GRAPH，加上 $A_1$ 與 $S_1$ 而得到如圖 10.10 所示的規劃圖。現在我們有了行動的完全補充，來看一些互斥關係和它們的原因的例子是很值得的：

- **不一致效果：**
  *Remove*(*Spare*, *Trunk*)與 *LeaveOvernight* 互斥，因爲一個有效果 *At*(*Spare*, *Ground*)而另一個有它的否定式。
- **衝突：**
  *Remove*(*Flat*, *Axle*)與 *LeaveOvernight* 互斥，因爲一個具有前提 *At*(*Flat*, *Axle*)，而另一個把它的否定式作爲效果。
- **競爭需要：**
  *PutOn*(*Spare*, *Axle*)與 *Remove*(*Flat*, *Axle*)互斥，因爲一個有 *At*(*Flat*, *Axle*)當做前提而另一個有它的否定式。
- **不一致支持：**
  $S_2$ 中 *At*(*Spare*, *Axle*)與 *At*(*Flat*, *Axle*)互斥，因爲獲得 *At*(*Spare*, *Axle*)的唯一途徑是 *PutOn*(*Spare*, *Axle*)，而其互斥於獲得 *At*(*Flat*, *Axle*)的唯一途徑的持續行動。因此，互斥關係檢測出試圖在同一時間將兩個物件放到同一位置的直接衝突。

這時，當我們返回迴圈的開始時，所有來自目標的文字都出現在 $S_2$ 中，並且其中沒有一個與任何其他的互斥。這意味著可能存在一個解，EXTRACT-SOLUTION 將嘗試去找到它。我們能夠將 EXTRACT-SOLUTION 當成一個布林限制滿足問題(CSP)，其中變數是每個階段的行動，每個變數的值是在規劃之內或規劃之外，而限制是互斥且須滿足各個目標與前提。

另一種做法是，我們能夠定義 EXTRACT-SOLUTION 爲一個逆向搜尋問題，其中於搜尋中的每個狀態包含了一個指向規劃圖某個階段的指標與尚未滿足的目標集。我們定義這個搜尋問題如下：

- 初始狀態是規劃圖的最後階段 $S_n$ 和來自規劃問題的一個目標集。
- 階段 $S_i$ 中，狀態可用的行動是選擇 $A_{i-1}$ 中任何無衝突的行動子集，這些行動的效果能涵蓋狀態中的目標。作爲結果的狀態在 $S_{i-1}$ 階段中，並且把它的目標集作爲被選中的行動集的前提。「無衝突」是指一個行動集中沒有兩個行動是互斥的，也沒有兩個行動的前提是互斥的。
- 目標是到達 $S_0$ 階段的一個狀態，能使所有的目標得到滿足。
- 每個行動的成本是 1。

對這個特殊問題，我們從具有目標 *At*(*Spare*, *Axle*)的 $S_2$ 階段出發。獲得目標集的唯一選擇是 *PutOn*(*Spare*, *Axle*)。這麼做帶我們到具有目標 *At*(*Spare*, *Ground*)與¬*At*(*Flat*, *Axle*)之 $S_1$ 的搜尋狀態。前者僅能透過 *Remove*(*Spare*, *Trunk*)，來達成而後者透過 *Remove*(*Flat*, *Axle*)或是 *LeaveOvernight*。但是 *LeaveOvernight* 與 *Remove*(*Spare*, *Trunk*)互斥，所以僅有的解答是選擇 *Remove*(*Spare*, *Trunk*)與 *Remove*(*Flat*, *Axle*)。這麼做帶我們到具有目標 *At*(*Spare*, *Trunk*)與 *At*(*Flat*, *Axle*)之 $S_0$ 的搜尋狀態。這兩個都出現在狀態中，所以我們有了一個解：在 $A_0$ 階段的行動 *Remove*(*Spare*, *Trunk*)和 *Remove*(*Flat*, *Axle*)，緊接著是 $A_1$ 階段的 *PutOn*(*Spare*, *Axle*)。

若 EXTRACT-SOLUTION 在某一給定的階段尋找某目標集之解答失敗的話，我們會紀錄這對(*level*, *goals*)爲**不良**，一如我們於 CSP 之限制學習(6.4 節)所做的事。無論何時 EXTRACT-SOLUTION 於相同的階段與目標被再次呼叫，我們能夠找出該紀錄爲不良者並立刻回報失敗而非再搜尋一次。我們很快就會看到不良也用於終止測試。

我們知道規劃是 PSPACE 完全的而且建構規劃圖需要多項式時間，所以在最壞情況下一定會出現抽取解是不可操作的情況。因此，在後向搜尋期間我們將需要某個啓發式來引導在行動中進行的選擇。在實踐中一個很好的方法是基於文字階段成本的貪婪演算法。對於任何目標集，我們按下面的順序進行：

1. 首先選擇具有最高階段成本的文字。
2. 要達成該文字優先選擇具有簡單之前提的行動。也就是，選擇一個行動，它的前提的階段成本總和(或最大值)是最小的。

### 10.3.3 GRAPHPLAN 的終止

到目前爲止，我們略過了終止問題。這裡我們顯示 GRAPHPLAN 事實上會終止並且於解答不存在時會傳回失敗。

第一件要了解的事是當圖已平整了，爲什麼我們不能停止擴張它。考慮一個在機場 $A$ 有飛機一架與貨物 $n$ 件的空中貨運領域，都是以 $B$ 機場爲目的地：於這個版本的問題，同一個時間內只能有一件貨物放在飛機上。該圖在階段 4 會平整，反映了任何單一件貨物，我們能夠於三個步驟內將之裝載、飛運、並卸載而抵達目的地的事實。但是這並非意謂著一個解答能夠於階段 4 的時候從圖中抽取出來；事實上解答需要 $4n - 1$ 個步驟：對於我們裝載、運送、並卸載的每一件貨物，以及所有貨物，除最後一件之外，我們必須飛回機場 $A$ 以領取下一件貨物。

當圖已經被平整之後，我們必須持續地擴張多久？如果函數 EXTRACT-SOLUTION 尋找解答失敗，那麼至少有一個目標集是無法達成的而會被標示爲不良。所以如果在下一個階段會有較少的不良數是有可能的，那麼我們就應該繼續下去。只要圖本身以及不良兩者都被平整，沒有找到任何解答，我們便能夠以失敗告終，因爲其後的更動會增加解答的可能性是不存在的。

現在我們必須做的就只有證明圖以及 不良必然會平整。第一步要注意到規劃圖的某些特性是單調遞增或者遞減的。「X 單調遞增」的意思是 $i + 1$ 階段的 X 集是階段集合的一個超集合(並不需要嚴格意義上的)。特性如下：

- **文字單調遞增**：
  一旦一個文字在一個給定的階段中出現，它將在所有後繼階段中出現。這是持續行動造成的；一旦一個文字露面，持續行動讓它永遠存在。
- **行動單調遞增**：
  一旦一個行動在一個給定的階段中出現，它將在所有後繼階段中出現。這是文字遞增的一個推論；如果一個行動的前提在一個階段中出現，它們將出現在後繼階段中，因而行動也一樣。

- **互斥單調遞減：**
  如果在給定階段 $A_i$ 的兩個行動是互斥的，那麼它們在所有早先它們共同出現的階段中也是互斥的。這對於文字之間的互斥同樣成立。不見得都會以那些圖所示的方式出現，因爲 那些圖有些簡化：它們既不顯示在階段 $S_i$ 中不成立的文字，也不顯示在階段 $A_i$ 中無法執行的行動。我們可以看到「互斥單調遞減」是正確的，如果你認爲這些不可見的文字和行動與任何事物都互斥的話。

  證明能夠依情況來處理：如果行動 $A$ 和 $B$ 在階段 $A_i$ 是互斥的，它一定是由三種互斥類型中的一種造成的。前兩種，不一致效果和衝突，是行動本身的特性，所以如果行動在階段 $A_i$ 互斥，那麼它們將在每個階段都互斥。第三種情況，競爭需要，依賴於 $S_i$ 階段的條件：階段必須包含一個 $A$ 的前提與一個 $B$ 的前提互斥的情況。現在，這兩個前提可以是互斥的，如果它們是彼此的否定式(這種情況下它們將在每一個階段都互斥)或者獲得其中一個的所有行動與獲得另一個的所有行動互斥。但是我們已經知道可用的行動是單調遞增的，所以透過歸納，互斥肯定是遞減的。
- **不良會呈單調性地減少：**
  如果一個目標集在某個給定的階段是不可達到的，那麼它們於任何前階段是不可達到的。這可以用矛盾法予以證明：如果它們是在某個前階段可達到的，那麼我們只需要加入持續行動使它們在後續的階段是可達到的。

因爲行動與文字呈單調性地增加，且因爲行動與文字的個數是有限的，所以必然會走到一個與前階段有相同的行動與文字個數的階段。因爲互斥與不良減少，且因爲不可能存在比零個互斥或不良還少的情況，必然會有一個階段其與前階段有相同之互斥與不良數的階段。一旦圖已經抵達這樣的狀態，那麼如果其中一個目標漏失或與其他目標互斥，那麼我們能夠停止 GRAPHPLAN 演算法並傳回失敗的訊息。這歸結出證明方式的大致輪廓；進一步的細節請見 Ghallab 等人(2004)。

## 10.4 其他的經典規劃方法

目前最廣爲採用與有效的全面自動規劃方法是：

- 轉譯成一個布林滿足性(SAT)問題
- 前向狀態空間搜尋搭配精心裁剪出來的啓發式(第 10.2 節)
- 使用規劃圖搜尋(第 10.3 節)

於自動規劃的 40 年歷史中，並非僅僅嘗試過這三個方法。圖 10.11 顯示從 1998 年起逐年舉辦之國際規劃競賽中的某些頂尖系統。於這一節我們首先描述轉抑制滿足性問題然後描述三個其他流的方法：規劃如一階邏輯歸納；如限制滿足；以及如規劃調整。

| 年度 | 主題 | 獲勝系統(方法) |
|---|---|---|
| 2008 | 最佳化 | GAMER (模型檢驗，雙向搜尋) |
| 2008 | 滿意性的 | LAMA (快速向下搜尋搭配 FF 啓發式) |
| 2006 | 最佳化 | SATPLAN，MAXPLAN (布林滿足性) |
| 2006 | 滿意性的 | SGPL 一個(前向搜尋；分割爲獨立的子程式)。 |
| 2004 | 最佳化 | SATPLAN (布林滿足性) |
| 2004 | 滿意性的 | FAST DIAGONALLY DOWNWARD (前向搜尋搭配因果圖) |
| 2002 | 自動的 | LPG (區域搜尋，規劃圖被轉換爲 CSP) |
| 2002 | 手寫 | TLPLAN (時序行動邏輯搭配前向搜尋控制邏輯) |
| 2000 | 自動的 | FF (前向搜尋) |
| 2000 | 手寫 | TALPLANNER (時序行動邏輯搭配前向搜尋控制邏輯) |
| 1998 | 自動的 | IPP (規劃圖)；HSP (前向搜尋) |

**圖 10.11 於國際規劃競賽中的某些頂尖效能系統。每一年都有不同的主題：「最佳化」意指規劃器必須產生最短的可能規劃，然而「滿意性的」意指非最佳解答也可以被接受。「手寫」意指領域特定之啓發式是可允許的；「自動的」意指它們是不被允許的**

## 10.4.1 經典規劃成為布林滿足性

於第 7.7.4 節我們看到 SATPLAN 是如何的將表示爲命題邏輯之規劃問題解出。在這裡我們展示如何轉譯一個 PDDL 描述爲一個能夠被 SATPLAN 處理的形式。該轉譯是一連串直截了當的步驟：

- **命題化行動：**
  將各個變數代換爲常數後所形成的基本行動集取代各行動模式。這些基本行動並非轉譯的一部分，但是於後續的步驟會派上用場。
- **定義初始狀態：**
  在問題初始狀態中的每一個流 $F$ 都是斷言 $F^0$，初始狀態沒有提及的流則都是斷言$\neg F$。
- **命題化目標：**
  於目標中的每一個變數，以一個常數的選言取代含有變數之文字。舉例來說，於一個具有物件 $A$，$B$ 與 $C$ 的世界，要求積木 $A$ 在另一塊積木之上，$On(A, x) \wedge Block(x)$的目標，會被取代爲下面這個目標。

$$(On(A, A) \wedge Block(A)) \vee (On(A, B) \wedge Block(B)) \vee (On(A, C) \wedge Block(C))$$

- **加入後繼狀態公理：**
  對每一個流 $F$，加入下述形式的公理

$$F^{t+1} \quad \Leftrightarrow \quad ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$$

  其中 *ActionCausesF* 是一個所有基本行動之加入表中有 $F$ 者的選言，而 *ActionCausesNotF* 是一個所有基本行動之刪除表中有 $F$ 者的選言。
- **加入前提公理：**
  對每一個基本行動 $A$，加入公理 $A^t \quad \Leftrightarrow \quad PRE(A)^t$，也就是說，如果一個行動在時間 $t$ 發生，那麼前提必須已經爲眞。

- **加入行動排斥公理：**
  設說每一個行動都與其他行動涇渭分明。

所得到的轉譯是個讓我們能夠交付予 SATPLAN 去尋找解答的形式。

## 10.4.2 規劃當成一階邏輯演繹：情景演算

PDDL 是一個小心地平衡了語言的表示能力與運算其上之演算法複雜度的語言。但是某些問題仍然很難以 PDDL 來傳達。舉例來說，我們無法於 PDDL 中表示目標「不管那兒有多少件貨物，通通從 $A$ 搬到 $B$」，但是我們能夠使用一個全稱量詞於一階邏輯中作出它。同樣地，一階邏輯能夠精確地傳達全局限制如「在同一時間、不得有四個以上的機器人位在同一位置。」PDDL 僅能夠以前提重複的加諸在與某個動作有關的每一個可能的行動來表現這個限制。

規劃問題的命題邏輯表示也有限度，如時間的符號直接與流繫結在一起的事實。舉例來說，$South^2$ 意指「代理人在時間 2 的時候面向南方。」以此表示，沒有任何方式可以說「如果在時間 1 的時候代理人向右轉，則它將會在時間 2 的時候面向南方，否則它將會面向東方。」一階邏輯使用一個稱之爲**情景演算**的表示式，將線性時間的符號換成分支情況的符號，來讓我們周旋這樣的限制，這個表示式的用法如下：

- 初始狀態被稱之爲一個**情景**(situation)。如果 $s$ 是一個情景而 $a$ 是一個行動，那麼 RESULT($s$, $a$) 也是一個情景。不會有其他的情景。因此，一個情景相應於行動的一個序列，或是歷史。你也能夠將情景想像成付諸行動之後的結果，但是請注意只有在情景的起點與行動都是一樣的時候，那兩個情景才能說是一樣的：(RESULT($s$, $a$) = RESULT($s'$, $a'$)) ⇔ ($s = s' \wedge a = a'$)。行動與情景的某些例子示如圖 10.12。

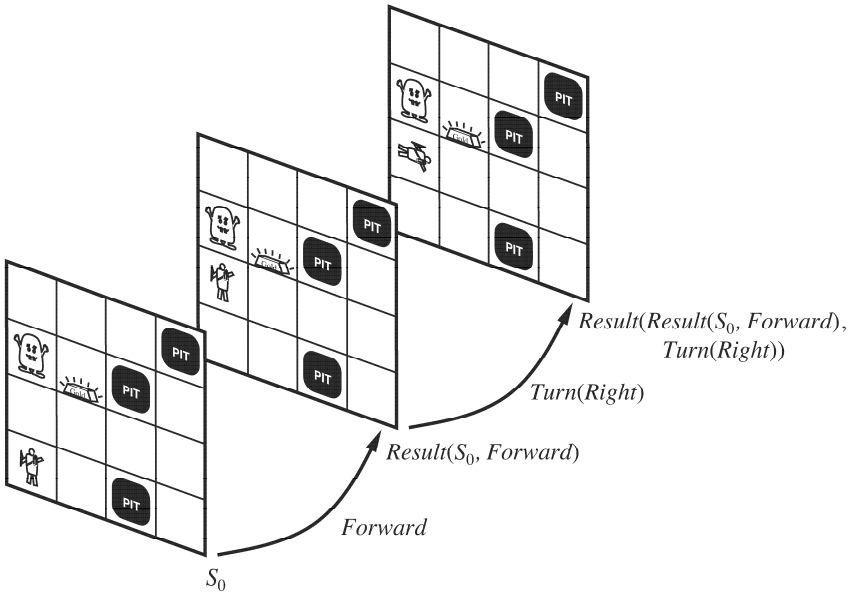


**圖 10.12** Wumpus 世界中的行動結果所呈現的情形

- 一個能夠從某個情景變換到另一個情景的函數或關係屬於一種**流**。慣例上，情景總是流的最後一個參數，舉例來說 $At(x, l, s)$是一個當物件 $x$ 於情景 $s$ 下在位置 $l$ 的時候會爲眞的關係流，而 *Location* 是一個使得與 $At(x, l, s)$有相同的情景 $s$ 時，$Location(x, s) = l$ 成立的函數流。
- 每個行動的前提會以一個**可能性公理**來描述，該前提說明什麼時候行動能夠被付諸實行。它具有形式 $\Phi(s) \Rightarrow Poss(a, s)$其中 $\Phi(s)$是描述前提的某些與 $s$ 有關的公式。wumpus 世界中有一個範例，說是如果代理人是活的而且有一根箭，就有射擊的可能：

  $$Alive(Agent, s) \wedge Have(Agent, Arrow, s) \Rightarrow Poss(Shoot, s)$$

- 每個流以一個**後繼狀態公理**來描述，此公理指出什麼樣的事會發生在流的身上，必須看什麼行動被付諸實行了。這類似於命題邏輯上我們所採取的方法。該公理的形式是

  行動是可能的 ⇒
  (流於所得到的狀態為真 ⇔ 行動的效果使之為真
  ∨它先前已經為真且行動對之無影響)

  舉例來說，用於關係流 *Holding* 之公理指出代理人執行一個可能的行動之後手上會持有一些金子 $g$ 若且唯若該行動是 $g$ 的 *Grab*，或若代理人已經持有 $g$ 而該行動並沒有將金子放下來：

  $$Poss(a, s) \Rightarrow (Holding(Agent, g, Result(a, s)) \Leftrightarrow a = Grab(g) \vee (Holding(Agent, g, s) \wedge a \neq Release(g)))$$

- 我們需要**唯一行動公理**使得代理人能夠演繹出，舉例來說，$a \neq Release(g)$。對每一對不同的行動名之爲 $A_i$ 與 $A_j$ 我們有一個公理說這些行動是不同的：

  $$A_i(x, \ldots) \neq A_j(y, \ldots)$$

  對每一個名爲 $A_i$ 之行動，我們有一個公理說使用該行動名稱的兩個名稱會是相等的，若且唯若它們的引數都相同：

  $$A_i(x_1, \ldots, x_n) = A_i(y_1, \ldots, y_n) \wedge x_1 = y_1 \wedge \ldots \wedge x_n = y_n$$

- 一個解答是一個滿足目標的情景(也即是一系列的行動)。

於情景演算中許多的工作多投注在定義規劃的形式語義以及拓展先的探索領域。但迄今爲止還沒有任何一個實際的大型規劃程式是植基於情景演算上的邏輯演繹。這部分是因爲於 FOL，要作出有效率的推理是很困難的，但主因爲這個領域還沒有開發出適用於以情景演算所爲之規劃的有效啓發式。

## 10.4.3 規劃當成限制滿足性

我們已見識過限制滿足與布林滿足性有相當大程度的共通性，而且我們已經見識過 CSP 技術對於排程問題是非常有向的，所以並不令人意外的它可以將一個受限制的規劃問題(也就是說，尋找一個長度爲 $k$ 之規劃的問題)當成一個限制滿足性的問題(CSP)來編寫。編寫類似於編寫到一個 SAT 問題(第 10.4.1 節)，但是有一個大大的簡化：在每一個單位時間，我們僅僅需要一個單一變數，$Action^t$，其領域是可能的行動的集合。我們不再需要爲每一個行動準備一個變數，且我們不需要行動排斥公理。也可能編寫規劃圖於 CSP 之中。這是 GP-CSP(Do 及 Kambhampati，2003)所採用的方法。

## 10.4.4 規劃看成是偏序規劃的調整

到目前爲止所有我們已經見識過如何建構完全由線性行動序列所組成之全序規劃。這表示省略掉了許多的子問題其實是彼此獨立無關的事實。一個空中貨運的問題的解答是由全序行動序列所組成，然而如果 30 件貨物於同一個機場被裝載到同一架飛機而且 50 件貨物被裝載另一個機場的另一架飛機，似乎沒有道理給出一個 80 個裝載行動的嚴格線性排序；兩個行動子集合應該被視爲是獨立無關地。

一個替代的做法是以偏序架構來表示規劃：一個規劃是一個行動集以及 *Before*($a_i$, $a_j$)，說的是一個行動先於另一個行動發生，之形式的限制集。於圖 10.13 的下半部，我們看到一個偏序規劃也就是說備胎問題的一個解答。行動是方框而前後順序的限制是箭頭。注意到，*Remove*(*Spare*, *Trunk*)和 *Remove*(*Flat*, *Axle*)可以在任一順序下進行，只要它們在行動 *PutOn*(*Spare*, *Axle*)之前完成。

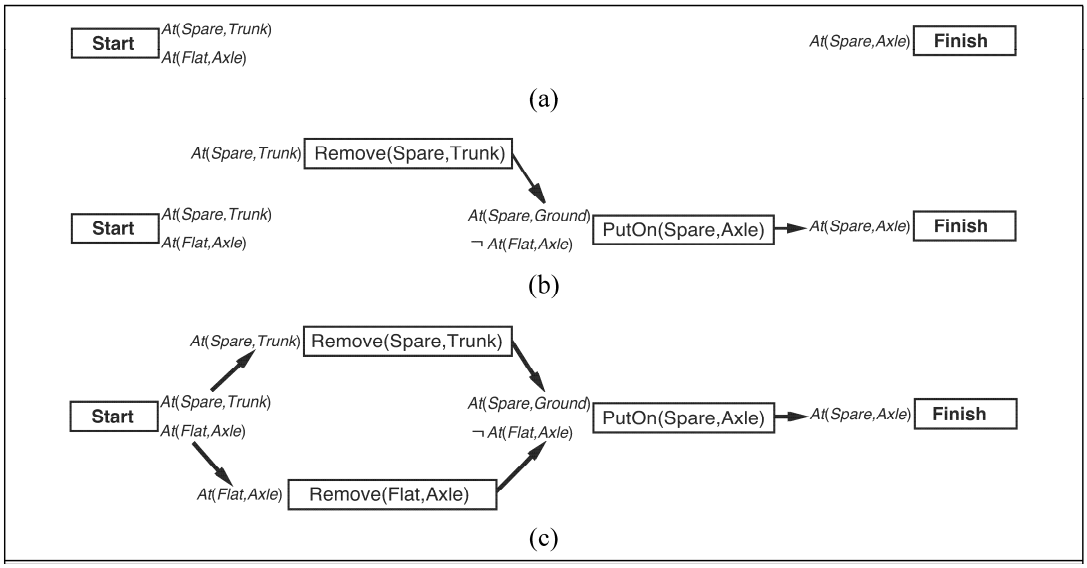


**圖 10.13** (a) **輪胎問題表示為一個空規劃**。(b) **用於輪胎問題的一個不完整偏序規劃。方框代表行動，而箭頭則指出某個行動必須先於另一個行動發生**。(c) **一個完整的偏序解答**

偏序規劃是經由搜尋規劃空間而非狀態空間所製作出來的。我們以空的規劃開始，此規劃由初始狀態與目標所組成，其間沒有任何行動，如圖 10.13 上方所示。搜尋程序然後尋找一個規劃中的**瑕疵**(flaw)，並加入該規劃以補正該瑕疵(或是如果無以補正，該搜尋會回溯並試試別的方法)。一個瑕疵是個使得某部分規劃不能成爲解答的任何東西。舉例來說，於空規劃中的一個瑕疵是沒有一個行動達成 *At*(*Spare*, *Axle*)。補正該瑕疵的一個方法是將行動 *PutOn*(*Spare*, *Axle*)插入該規劃。當然這麼做會引進一些新的瑕疵：新行動的前提是沒有達成的。該搜尋持續的加入該規劃(必要的話會回溯)直到所有的瑕疵被解消，如圖 10.13 的下方。在每一個步驟，我們爲修復瑕疵而使**最少量承諾**爲可能。舉例來說，於加入行動 *Remove*(*Spare*, *Trunk*)我們必須承諾使它發生於 *PutOn*(*Spare*, *Axle*)之前，但是我們並沒有做其他承諾將它放在其他的行動之前或之後。如果於行動模式中有個變數 可以被無限制的遺留下來，我們會這麼做。

於 1980 與 90 年代，偏序規劃看起來似乎是以獨立無關子問題處理規劃問題的最佳途徑——畢竟，它是能代表規劃之獨立分支的僅有方法。另一方面，它的不利之處在於在狀態轉移模型中沒有一個顯而易見的狀態表示。這使得某些計算變得繁瑣。到了 2000，前向搜尋規劃器已經發展出極佳的啓發式，可以讓它們有效率地發掘出獨立的子問題，這個偏序規劃與生俱來的用途。職是之故，偏序規劃器無法匹敵全自動的經典規劃問題。

不過，偏序規劃仍然是一個該領域重要的一部分。對於某些特定的任務，如作業排程，具有領域特定啓發式之偏序規劃仍是首選的技術。許多的這些系統使用高階段次規劃的程式庫，如第 11.2 節所描述者。偏序規劃也常被使用於對人類而言了解該規劃的領域之處。太空船的作業性規劃以及火星漫遊者都是由偏序規劃器所產生的然後在裝載到太空船進行執行之前由人類操作人員檢查。該規劃調整方法使得它類很容易了解規劃演算法都做些什麼事並驗證它們都是正確無誤的。

## 10.5 規劃方法分析

規劃合併我們已經論述過的兩個主要 AI 領域：搜尋和邏輯。也就是，規劃器既能被視爲搜尋解的程式，也能被視爲(構造性地)證明解存在的程式。這兩個領域的想法雜交致使過去 10 年中性能提高了好幾個數量級，並且增加了規劃器在工業應用中的使用。不幸的是，我們還沒有瞭解得很清楚，哪些技術在哪類問題上能夠最好地工作。很有可能的是，新的技術將會出現，並且超越已有的方法。

規劃是最先經歷控制組合爆炸的。如果域內有 $n$ 個命題，那麼就有 $2^n$ 個狀態。一如我們已經見過的，規劃是 PSPACE-hard。在這樣悲觀的情況下，獨立子問題之辨識可以是一個甚具威力的武器。在最佳情況下——問題完全可分解下——我們得到指數級的加速。然而，可分解性被行動之間的負相互作用所破壞。GRAPHPLAN 紀錄了互斥以指出難以互動之處何在。SATPLAN 表示了與互斥關係類似的範圍，不過它是利用通用的 CNF 形式而不是特定的資料結構來完成這個的。前向搜尋試著尋找能涵蓋獨立子問題之樣式(命題的子集合)來啓發式地論述該問題。因爲這個方法是啓發式，所以即使當子問題並非全地獨立的時候，它仍然能夠發生作用。

有時候透過識別那些可以排除的負相互作用而有效地求解問題是可能的。如果一個問題中的子目標存在一個順序，以致規劃器能夠按這個順序獲得它們，而不需要撤銷任何先前獲得的子目標，那麼我們說該問題具有**可串列化子目標**。例如，在積木世界中，如果目標是要建造一個塔[例如，$A$ 在 $B$ 上，$B$ 在 $C$ 上，$C$ 在 *Table*(桌子)上，如圖 10.4 所示)]，那麼子目標是可以由下而上地可串列化的：如果我們首先達成在桌子上的 $C$，當我們達成另一個子目標的時候，我們將永遠不必還原它。使用由下而上技巧的規劃器在積木世界域內可以不需要回溯就能求解任何問題(雖然它並不總能找到最短規劃)。

作爲一個更複雜的例子，對於指揮 NASA 的深空一號太空船的遠端代理人規劃器，已經確定的是涉及指揮航空器的命題可串列化。這可能並不令人感到驚奇，因爲工程師設計的航天器要盡可能地容易控制(服從其他限制)。利用目標的串列化排序，遠端代理人規劃器能夠消除大部分搜尋。這意味著它能足夠快地即時控制航天器，這是在以前被認爲是不可能的。

規劃器如圖 PLAN，SATPLAN，與 FF 已經將規劃的領域向前推進，藉由提高規劃系統的效能層次，藉由澄清所涉及的表示性與組合性議題，以及藉由開發有用的啓發式。不過，有一個問題是這些技術將延伸多遠。似乎更大型問題的進一步的進展不能僅僅分解與命題性的表示，而將需要某種一階與分層表示的綜合體搭配目前使用中的有效率啓發式。

## 10.6 總結

在這一章中，我們在確定性的、完全可觀察的環境中定義了規劃問題。我們描述了用於規劃問題的主要表示方法和一些用來求解它們的演算法方法。需要記住的幾點是：

- 規劃系統是問題求解演算法，它在關於狀態和行動的明確命題(或一階)表示上運轉。這些表示使獲得高效啓發式和強有力且靈活的問題求解演算法成爲可能。
- PDDL，規劃領域定義語言，將初始與目標狀態描寫成文字的連言，且以它們的前提與效果來表示行動。
- 狀態空間搜尋可以前向(**前進**)或者後向(**回歸**)地運轉。可以透過子目標獨立假設和對規劃問題進行各種鬆弛來獲得有效的啓發式。
- **規劃圖**可以從初始狀態出發，漸進地構造。每一層包含一個所有在那個時間步出現的文字和行動的超集合，並且對互斥(或簡稱 **mutex**，文字之間或行動之間不能出現的關係)進行編碼。規劃圖爲狀態空間和偏序規劃器產生有用的啓發式，並能直接用於圖 PLAN 中
- 其他的方法包括於情景演算公理之上的一階演繹；編寫規劃問題爲布林滿足性問題或是爲限制滿足性問題；且直接在偏序規劃空間上進行搜尋。
- 規劃問題的每種主要方法都有它自己的擁護者，對哪一個是最好的還沒有達成一致。這些方法之間的競爭和雜交導致規劃系統的效率獲得顯著提高。

## ❍ 參考文獻與歷史的註釋 BIBLIOGRAPHICAL AND HISTORICAL NOTES

人工智慧規劃起源於狀態空間搜尋、定理證明和控制理論的研究，及機器人技術、排程和其他領域的實際需要。STRIPS(Fikes 和尼爾森，1971)，第一個主流規劃系統，形象地說明了這些領域的相互作用。STRIPS 是作爲 SRI 的 Shakey 機器人專案軟體的規劃部分而設計的。它的整體控制結構以 GPS 爲模型，GPS 就是通用問題求解器(General 問題求解器，紐厄爾和西蒙，1961)，一個使用手段目標分析(means-ends analysis)的狀態空間搜尋系統。Bylander(1992)證明了簡單 STRIPS 規劃是 PSPACE 完全的。Fikes 和尼爾森(1993)對 STRIPS 專案進行了歷史性的回顧，並對它與新近的規劃成就的聯繫進行了綜述。

STRIPS 所使用的表示語言已經遠較它的演算性方法來得有影響力；我們所稱的「經典」語言貼近於 STRIPS 所使用者。行動描述語言或稱 ADL(Pedanault，1986)放鬆了 STRIPS 語言中的一些限制，使得對更多實際問題的編碼成爲可能。Nobel(2000)探索了將 ADL 編譯成 STRIPS 的方案。問題領域描述語言，或 PDDL(Ghallab 等人，1998)，面世時是個用於表示規劃問題的電腦可剖析、標準化的語法，自 1998 年起已經被國際規劃競賽採用爲標準語言。存在數個延伸版本；最新的版本是，PDDL 3.0，納入了規劃限制與偏好(Gerevini 及 Long，2005)。

20 世紀 70 年代早期的規劃器通常在全序行動序列上工作。問題分解透過計算每個子目標的子規劃並按照某種順序對子規劃進行排列來實作。這種被 Sacerdoti(1975)稱爲**線性規劃**的方法很快就被發現是不完備的。它不能求解一些非常簡單的問題，諸如 Sussman 不規則問題(參見習題 10.7)，這是 Allen Brown 在 HACKER 系統的實驗中發現的(Sussman，1975)。一個完備的規劃器必須允許來自不同子規劃的行動在一個單一序列中**交叉**。可串列化子目標(Korf，1987)的概念正好對應於無交叉規劃器在其中是完備的問題集。

交叉問題的一解決方案是目標回歸規劃，一種記錄了哪些步驟滿足全序規劃的技術，從而避免子目標間的衝突。這被 Waldinger(1975)引入，並被 Warren(1974)的 WARPLAN 使用。WARPLAN 也值得注意，它是第一個用邏輯程式語言(Prolog)編寫的規劃器，也是使用邏輯程式設計有時能獲得顯著節省的最佳例子之一：WARPLAN 只有 100 行程式碼，是當時可比較的規劃器規模的很小部分。

偏序規劃的基礎想法包括衝突探測(Tate，1975a)和對從衝突中獲得的條件的保護(Sussman，1975)。偏序規劃(當時稱爲**任務網路**)的建構是由 NOAH 規劃器(Sacerdoti，1975，1977)和 Tate(1975b，1977)的 NONLIN 系統開創的。

偏序規劃在下個 20 年的研究中獨領風騷，然而第一個正式問世的是 TWEAK(Chapman，1987)，是一個可以讓各式各樣的規劃問題之完整性與難解性的證明(NP 難題和不可判定性)變得十分簡單的規劃器。Chapman 的工作得出對完備偏序規劃器的簡潔描述(McAllester 及 Rosenblitt，1991)，然後引領至廣爲流傳的實作 SNLP(Soderland 及 Weld，1991)與 UCPOP(Penberthy 及 Weld，1992)。偏序規劃於 1990 年代末期因爲出現了更快速的方法而逐漸失去吸引力。Nguyen 和 Kambhampati(2001)提議重新考慮是值得的：用從規劃圖中得到的精確啓發式，他們的 REPOP 規劃器比圖 PLAN 提高了很多，並可以與最快的狀態空間規劃器相抗衡。

對狀態空間規劃重新燃起興趣是由 Drew McDermott 的 UNPOP 程式(1996)起了帶頭的作用，它是第一個主張清空刪除表啓發式，UNPOP 之名是對當時一窩蜂地傾心於偏序規劃的現象的回擊；McDermott 懷疑其他的方法並沒有得到該有的關注。Bonet 和 Geffner 的啓發式搜尋規劃器(HSP)及其後來的衍生體(Bonet 和 Geffner，1999)第一次將狀態空間搜尋應用於大規模規劃問題。HSP 是順方向搜尋而 HSPR(Bonet 與 Geffner，1999) searches 則是逆方向搜尋。迄今爲止最成功的狀態空間搜尋器是 FF(Hoffmann，2001；Hoffmann 與 Nebel，2001；Hoffmann，2005)，2000 年 AIPS 規劃競賽的優勝者。FASTDOWNWARD(Helmert，2006)是一個前向狀態空間搜尋規劃器，它預先處理行動模式爲一個替代性的表示如此可使某些限制更明白。FASTDOWNWARD(Helmert 與 Richter，2004；Helmert，2006)贏得 2004 年的規劃競賽，且 LAMA(Richter 與 Westphal，2008)，一個植基於 FASTDOWNWARD 搭配改良過後的啓發式的規劃器，贏得 2008 年的競賽。

Bylander(1994)與 Ghallab 等人(2004)討論數個規劃問題之衍生問題的計算性複雜度。Helmert(2003)證明出許多標準的性能測試問題的複雜度範圍，Hoffmann(2005)分析了清空刪除表啓發式的搜尋空間。Caprara 等人(1995)就義大利鐵路的排程作業之集合-涵蓋問題對的啓發式進行了討論。Edelkamp(2009)與 Haslum 等人(2007)描述如何爲規劃啓發式建構資料庫。正如我們於第 3 章曾經提到過的，Felner 等人(2004)於滑塊謎題上使用樣式資料庫而出現令人振奮的結果，這能夠被視爲一個規劃領域，但是 Hoffmann 等人(2006)證明了經典規劃問題上的某些抽象限制。

Avrim Blum 和 Merrick Furst(1995，1997)用他們的圖 PLAN 系統使規劃領域獲得新生，它比當時的偏序規劃器快好幾個數量級。其他圖規劃系統，例如 IPP(Koehler 等人，1997)、STAN(Fox 和 Long，1998)和 SGP(Weld 等人，1998)，也隨即出現。一個非常類似於規劃圖的資料結構稍早些時候被 Ghallab 和 Laruelle(1994)開發出來，他們的 IXTET 偏序規劃器用它獲得精確啓發式以引導搜尋。Nguyen 等人(2001)徹底地分析推導自規劃圖的啓發式。我們對規劃圖的討論是部分基於這項工作和 Subbarao Kambhampati(Bryce 及 Kambhampati，2007)的講稿及文章。正如本章中提到的，能從不同途徑把規劃圖用於引導解的搜尋。2002 年 AIPS 規劃競賽的獲勝者 LPG(Gerevini 和 Serina，2002)受 WALKSAT 的啓發式，用局部搜尋技術來搜尋規劃圖。

將情景演算方用法於規劃是由 John McCarthy(1963)所引介。我們這裡所展示的版本是由 Ray Reiter(1991，2001)所倡議的。

Kautz 等人(1996)研究了對行動模式的各種命題方式，發現最緊緻的形式並不必然導致最快解答時間。Ernst 等人(1997)進行了一個系統的分析，他們還開發了一個能從 PDDL 問題產生命題表示的自動「編譯器」。Kautz 和 Selman(1998)開發的 BLACKBOX 規劃器結合了圖 PLAN 和 SATPLAN 的想法。CPLAN，一個植基於限制滿足性的規劃器，由 van Beek 及 Chen(1999)所論述。

最近，人們對用**二元決策表**來進行規劃表示產生了興趣，二元決策表是對在硬體同位領域中已經有廣泛研究的有限自動機的簡潔描述(Clarke 和 Grumberg，1987；McMillan，1993)。有技術能夠證明二元決策表的特性，包括成爲規劃問題的解的特性。Cimatti 等人(1998)提出了一個基於該方法的規劃器。其他表示也有所使用；例如，Vossen 等人(2001)綜述了規劃問題中整數程式設計的使用。

仍然沒有統一的評判，但是現在對各種規劃方法有了一些有趣的比較。Helmert(2001)分析了幾類規劃問題，顯示基於限制的方法，諸如圖 PLAN 和 SATPLAN，對 NP 難題領域是最好的，而基於搜尋的方法在無需回溯就能找到可行解的領域中做得更好。圖 PLAN 和 SATPLAN 在具有很多物件的領域遇到了麻煩，因爲這意味著它們必須建立許多行動。在某些情況下，問題可以透過動態產生命題化行動而被延遲或避免；只有當需要的時候，而不是在搜尋開始前將它們全部初始化。

《*Readings in Planning*》(Allen 等人，1990)是一本對於此領域早期文獻的完整選集。Weld(1994，1999)提供 1990 年代中關於規劃演算法的兩篇優秀綜述。從這兩篇綜述中來看過去 5 年的變化是有意思的：第一篇集中於偏序規劃上，而第二篇介紹了圖 PLAN 和 SATPLAN。《*Automated Planning*》(Ghallab 等人，2004)是一本對於規劃的各面向極佳教科書。LaValle 的教科書《*Planning Algorithms*》(2006)同時納入了經典與隨機規劃，尤其於機器人的動作規劃方面著墨甚多。

自從規劃搜尋出現以來，它已經成爲人工智慧的中心問題之一，關於規劃的論文是主流人工智慧期刊和會議的重要來源。也有專門的會議，例如「人工智慧規劃系統國際會議」(International Conference on AI Planning Systems，AIPS)，「空間規劃和排程國際專題研討會」(International Workshop on Planning and Scheduilng for Space)以及「歐洲規劃會議」(European Conference on Planning)。

## ❖ 習題 EXERCISES

**10.1** 描述問題求解與規劃之間的不同和相似之處。

**10.2** 從圖 10.1 已知行動模式與初始狀態，於下之所述之狀態時，所有 *Fly*(*p*, *from*, *to*)的可運用之具體實例是什麼？

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(JFK) \wedge Airport(SFO)\text{？}$$

**10.3** 猴子和香蕉問題是關於實驗室的一隻猴子面對掛在天花板上的一些夠不到的香蕉的問題。一個箱子是可用的，如果猴子爬上箱子，它就可以夠到香蕉。起初，猴子位於 *A*，香蕉位於 *B* 而箱子位於 *C*。猴子和箱子的高度是 *Low*，但是如果猴子爬到箱子上面，它的高度就跟香蕉一樣是 *High*。猴子可用的行動包括從一個位置走到另一個位置的 *Go*，將物件從一個地方推到另一個地方的 *Push*，爬上一個物件的 *ClimbUp* 或爬下一個物件的 *ClimbDown*，抓住一個物件的 *Grasp* 或放開一個物件的 *UnGrasp*。如果猴子與物件在相同的位置在相同的高度，則 *Grasp* 的結果是猴子拿著該物件。

**a.** 寫下初始狀態描述。

**b.** 寫六個行動模式。

**c.** 假設猴子想透過摘取香蕉卻把箱子留在最初的位置來愚弄溜號去喝茶的科學家。在情景演算語言中把這個作爲普通目標寫下來(即沒有假設箱子必須位於 *C*)。這個目標能被 STRIPS 風格的系統解決嗎？

**d.** 對推(push)的規則可能是不正確的，因爲如果物件太重的話，當 *Push* 運算元應用的時候它的位置將保持不變。修改你的問題描述來解決重物的問題。

**10.4** 最初的 STRIPS 程式是設計用來控制機器人 Shakey。圖 10.14 顯示了一個版本的由 4 個沿走廊排列的房間組成的 Shakey 世界，其中每個房間有一扇門和一個電燈開關。Shakey 世界中的行動包括從一個地方移動到另一個地方，推可移動物體(例如箱子)，爬上或爬下剛性物體(例如箱子)及打開和關上電燈開關。機器人本身無法爬上箱子或切換開關，但是規劃器能夠找到並列印出超過機器人能力的規劃。Shakey 的 6 種行動如下：

- *Go*(*x*, *y*, *r*)，這要求 *Shakey* 是 *At x* 且 *x* 和 *y* 是同一房間 *r* 內的位置。按照慣例，兩個房間之間的門是在它們內部的。
- 在同一房間內將箱子 *b* 從位置 *x* 推到 *y*：*Push*(*b*, *x*, *y*, *r*)。我們需要述詞 *Box* 和箱子的常數。
- 從位置 *x* 爬上一個箱子：*ClimbUp*(x, *b*)；從一個箱子爬下到位置 *x*：*ClimbDown*(*b*)。我們需要述詞 *On* 和常數 *Floor*。
- 將電燈開關作開啓或關閉：*TurnOn*(*s*, *b*)；*TurnOff*(*s*, *b*)。要打開或關閉電燈開關，Shakey 必須在電燈開關位置的一個箱子上。

從圖 10.14，寫出 Shakey 的六個行動的 PDDL 語句與初始狀態。建構一個讓 Shakey 把 $Box_2$ 帶到 $Room_2$ 裡的規劃。

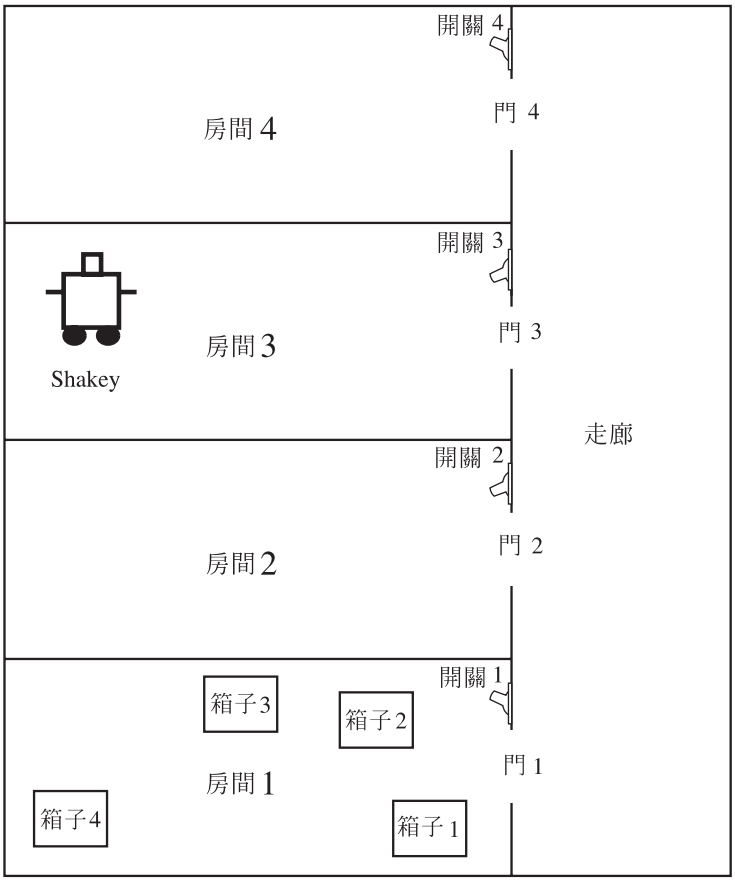


**圖 10.14** Shakey **世界**。Shakey **能夠在一個房間內的地標間移動，能夠穿過房間之間的門，能夠爬可爬的物件，也能夠推可推的物件，並且能按電燈開關**

**10.5** 一台有限的圖靈機有一個有限的一維格帶，每一格裝有數目有限的符號中的一個符號。一個格子上方有一個讀寫頭。該機器能身處的狀態是一個有限狀態的集合，其中一個是接受狀態。在每一個單位時間，根據位於讀寫頭下方之格子內存的符號以及機器目前的狀態，存在了一個我們能夠能夠從中選用的行動集。每個行動牽涉到寫符號到讀寫頭下的格子、轉換機器的狀態、以及將讀寫頭向左或向右移動。用於決定哪一個行動是被允許的對照是圖靈機的程式。你的目標控制這台機器進入接受狀態。將圖靈機的接受問題表示爲一個規劃問題。

如果你能夠做這件事情，它展示判斷一個規劃問題是否有解其難度最少也相當於圖靈機的接受問題者，它屬於 PSPACE-hard。

**10.6** 解譯爲什麼 STRIPS 問題中丟棄每個行動模式的負效果導致一個鬆弛問題。

**10.7** 圖 10.4 顯示了一個稱爲 **Sussman 不規則**的積木世界問題。這個問題之所以被認爲不規則是因爲 20 世紀 70 年代早期的非交叉規劃器不能解決它。寫下對問題的一個定義並求解之，可用手工方式也可用一規劃程式。一個非交叉規劃器是這樣一個規劃器：當給定兩個子目標 $G_1$ 和 $G_2$ 時，對 $G_1$ 產生一個和 $G_2$ 的規劃連接在一起的規劃，或者反過來。解譯爲什麼一個非交叉規劃器不能解決這個問題。

**10.8** 證明逆向搜尋 PDDL 問題是完整的。

**10.9** 爲圖 10.1 中所示問題構造階段 0、階段 1 和階段 2 的規劃圖。

**10.10** 證明下列關於規劃圖的斷言：

**a.** 在圖的最後階段沒有出現的文字是無法獲得的。

**b.** 串列圖中的一個文字的階段成本不會比獲得它的最優規劃的實際成本更大。

**10.11** 集合-階段啓發式(見 10.3.1 節)使用規劃圖估算從目前的狀態達到一個連言目標的代價。集合-階段啓發式解答適用於什麼樣的鬆弛問題？

**10.12** 考察第三章中**雙向搜尋**的定義。

**a.** 雙向狀態空間搜尋對於規劃問題是一個好想法嗎？

**b.** 偏序規劃空間的雙向搜尋如何？

**c.** 設計一個偏序規劃的版本，其中如果行動前提能被規劃中已經存在的行動效果獲得，那麼行動應該能夠添加到規劃中。解譯如何處理衝突和定序限制。這個演算法本質上與前向狀態空間搜尋一樣嗎？

**10.13** 將前向和後向狀態空間搜尋規劃器與偏序規劃器進行對比，假定後者是一個規劃空間搜尋器。解譯前向和後向狀態空間搜尋也可以被視爲規劃空間搜尋器，並說明規劃器的改進運算元是什麼。

**10.14** 迄今爲止我們已經假設我們製作的規劃必然會確認行動的前提是被滿足的。讓我們現在探究命題的後繼狀態公理如 $HaveArrow^{t+1} \Leftrightarrow (HaveArrow^{t} \wedge \neg Shoot^{t})$對於前提並沒有被滿足的行動該說些什麼。

**a.** 說明公理預測當一個行動在一個前提不滿足的狀態中執行時，沒有事情會發生。

**b.** 考慮一個包含獲得目標所需的行動，但同時也包括非法行動的規劃 $p$。是否爲如下的情況：

初始狀態 $\wedge$ 後繼狀態公理 $\wedge p \models$ 目標？

**c.** 利用情景演算中的一階後繼狀態公理，是否可能證明一包含非法行動的規劃將獲得目標？

**10.15** 考慮如何轉譯一組行動模式成爲情景演算的後繼-狀態公理。

**a.** 考慮 $Fly(p, from, to)$所用的模式。針對述詞 $Poss(Fly(p, from, to), s)$寫出一個邏輯定義，如果 $Fly(p, from, to)$之前提於情景 $s$ 是滿足的則其爲眞。

**b.** 接下來，假設 $Fly(p, from, to)$是代理人能得到的唯一行動模式，寫下 $At(p, x, s)$的後繼狀態公理，捕捉與行動模式同樣的資訊。

**c.** 現在假設有旅行的附加方法：$Teleport(p, from, to)$。它有附加前提$\neg Warped(p)$和附加效果 $Warped(p)$。解譯情景演算知識庫該怎樣修改。

**d.** 最後，開發一個通用而又準確的用來執行從一組 STRIPS 模式轉換到一組後繼狀態公理的特定過程。

**10.16** 在圖 7.22 所示的 SATPLAN 演算法中，對可滿足性演算法的每次呼叫斷言一個目標 $g^T$，其中 $T$ 的範圍是從 0 到 $T_{\max}$。換假設可滿足性演算法只呼叫一次，且目標是 $g^0 \vee g^1 \vee ... \vee g^{T_{\max}}$ 。

**a.** 這個演算法是否總能返回一個長度小於等於 $T_{\max}$ 的規劃，如果這樣的規劃存在的話？

**b.** 這個方法會引進任何新的虛假「解」嗎？

**c.** 討論可能如何修改諸如 WALKSAT 的可滿足性演算法，從而當給定一個這種形式的目標選言時，它能找到較短的解(如果存在的話)。

## 本 章 註 腳

[1] PDDL 推導自原始的 STRIPS 規劃語言(Fikes 及 Nilsson，1971)。它較 PDDL 略為受限：STRIPS 的前提與目標不能含有負文字。

[2] 規劃搜索中使用的積木世界比 SHRDLU 的版本簡單得多，如 1.3.3 節所示。

[3] 許多的問題都是根據這個慣例而寫出來。對於問題並非如是者，將目標中的每個負文字 $\neg P$ 或前提替換為正文字，$P'$。