第 5 章

自 动 规 划

规划是智能行为的一个重要特征，目标是寻找从初始状态转移到最终状态的（最优）动作序列。在生产调度、航空航天、智能交通、城市建设等众多领域，规划都扮演着重要角色。随着社会发展，规划需要考虑的因素越来越多，状态空间也越来越大，因此机器自动规划成为了一个活跃的研究领域。自动规划可以看成是逻辑与搜索的结合，主要思想是在知识逻辑表示的基础上，通过搜索状态空间或规划空间取得一个可行解或最优解。本章将讲述自动规划的基本内容，包括规划问题的形式化表示和规划问题的求解算法。对于考虑时间约束的规划算法，本章也做了初步分析。

5.1 规划问题的形式化表示

一个规划问题通常由四个部分组成：①初始状态；②动作集合，即每一步可选择的动作；③结果集合，即每一个动作完成之后的状态；④目标状态。逻辑系统一章已经讲述了如何表示现实世界的事物和状态，因此还需要定义如何表示动作。我们选用一阶逻辑来达到此目的。这既保证了推理的完备性，又具备灵活性而不过于复杂。动作同样可以定义成“谓词”形式。例如，在路径搜索的例子中，定义“谓词”$Travel(a,b)$表示从 a 城市到相邻的 b 城市。这个动作是一个抽象表示，实际中可能有很多动作组成，实现方式也可以不同，但最终都能完成这样“一步”操作。明白这一点，我们就可以将 $Travel(a,b)$看成是一个不可再分的动作，称为基元动作（ground operator）。基元动作是针对当前规划层次而言的。在更低的层次上，上一层的基元动作可能变为下一层上的总任务。例如，实际执行 $Travel$（西安，成都）动作时，可能有[$RentCar(x$，西安）-$Drive(x$，西安，成都）-$ReturnCar(x$，成都）]、[$BuyTicket(y$，西安，成都）-$ByTrain(y$，西安，成都）]等多种选择。这本身又是一个规划（从分层规划的角度看，前者称为高层规划，后者称为低层规划）。虽然动作的符号表示可以看成是“谓词”，但是从逻辑推理一章可知，谓词逻辑本身是描述事物状态及其联系的，并不用来表示动作。例如陈述“小明从北京到天津”在逻辑推理中并无多大意义，其真值也不好确定。为区别这一点，我们采用动作模式这一术语。这也是前面将“谓词”加引号的原因。

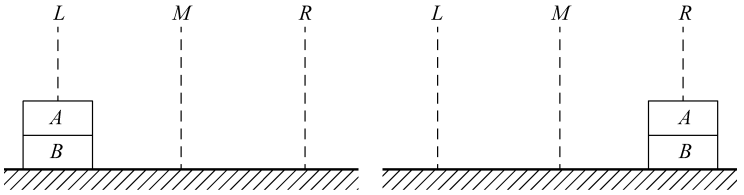
动作模式并不是任何情况都可以执行，其执行需要满足一定条件。按一阶逻辑的方法，定义状态谓词 $At(a)$ 表示处于 a 城市，$Connected(a,b)$ 表示城市 a 与 b 相邻。小明只有达到状态 $At(a) \wedge Connected(a,b)$，才能执行 $Travel(a,b)$ 动作。执行完 $Travel(a,b)$ 动作后，结果状态将转变为 $At(b) \wedge Connected(b,a)$。因此，以“西安”为起点“青岛”为终点的规划问题可形式化表示为

```
Init(At(西安) ∧ Connected(西安,郑州) ∧ …)
Goal(At(青岛) ∧ Connected(西安,郑州) ∧ …)
Action(Travel(a,b),PreCond: At(a) ∧ Connected(a,b),Result: At(b) ∧ Connected(b,a))
```

该规划问题的动作集中只有一个动作模式，初始状态和目标状态都由当前所处城市和城市间连接关系两部分构成。有两个问题需要说明。首先，初始状态和目标状态不包含变量。任何具体的规划都必须有确定的初始状态和目标状态。动作模式包含变量，且其结果状态中出现的变量必须出现在条件中。其次，对图的表示仅包含正文字，表示结点不相连的负文字（如 $\neg Connected$（太原，郑州））并未写出。这实质上是采用了“封闭世界”假设来避免表达式过于冗长（请回忆“封闭世界”假设的定义）。但是，如果动作模式的条件中含有负文字状态，那么应该将相关的负文字写出，以使得后续的搜索能够进行。

另一个稍复杂的例子来自简化版的积木世界。

例 5-1（积木世界[1]） 地上有 A、B 两个箱子，L、M、R 三个位置。开始时，A 和 B 位于位置 L 处，且 A 放置于 B 上，如图 5-1(a)初始态所示。当某个箱子上面没有其他箱子时，机器人可以将它移动到另外的空的位置。当两个箱子位于不同位置且上面均没有箱子时，机器人可以将一个箱子放置在另一个箱子之上。需要达到的终止状态如图 5-1(b)目标态所示。给出规划问题的形式化表示。



(a) 初始态　　(b) 目标态

图 5-1　简化版的积木世界

与路径规划的形式化表示一样，首先根据题意定义谓词。定义三个状态谓词符号：$At(x,a)$ 表示箱子 x 位于 a 处，$On(x,y)$ 表示箱子 x 位于箱子 y 上，$Vacant(a)$ 表示 a 处是空的。移动箱子的动作分三种情况，因此定义三个动作模式：$Move(x,a,b)$ 表示将箱子 x 从 a 处的地面移到空地 b，$Lay(x,y)$ 表示将箱子 x 放置在箱子 y 上，$Take(x,a,b)$ 表示将箱子 x 从位于 a 处的另外一个箱子上面取下并放在空地 b 处。再用一个常量符号 $Floor$ 表示地板。基于上述定义，积木世界的规划问题可以形式化表示为

$$Init(At(A,L) \wedge At(B,L) \wedge On(A,B) \wedge \neg On(B,A) \wedge On(B,Floor) \wedge Vacant(M) \wedge Vacant(R))$$

$$Goal(At(A,R) \wedge At(B,R) \wedge On(A,B) \wedge \neg On(B,A) \wedge On(B,Floor) \wedge Vacant(L) \wedge Vacant(M))$$

$$Action(Move(x,a,b),$$
$$Precond: At(x,a) \wedge \neg On(y,x) \wedge On(x,Floor) \wedge Vacant(b),$$
$$Effect: At(x,b) \wedge \neg On(y,x) \wedge On(x,Floor) \wedge Vacant(a));$$
$$Action(Lay(x,y,a,b),$$
$$Precond: At(x,a) \wedge At(y,b) \wedge (a \neq b) \wedge \neg On(x,y) \wedge \neg On(y,x) \wedge On(y,Floor),$$
$$Effect: At(x,b) \wedge At(y,b) \wedge On(x,y) \wedge \neg On(y,x) \wedge On(y,Floor) \wedge Vacant(a))$$
$$Action(Take(x,a,b),$$
$$Precond: At(x,a) \wedge At(y,a) \wedge On(x,y) \wedge \neg On(y,x) \wedge On(y,Floor) \wedge Vacant(b),$$
$$Effect: At(x,b) \wedge At(y,a) \wedge \neg On(x,y) \wedge \neg On(y,x) \wedge On(x,Floor) \wedge On(y,Floor)$$

这个例子的动作模式条件包含了负文字，因此初始状态和目标状态也相应地写出了负文字。

5.2　状态空间规划

完成规划问题的表示后，本节进入如何设计求解算法。对于一个规划问题，所有规划对象的所有可能取值构成了一个状态空间。规划的目标就是在规划域上寻找一个动作序列使得所有对象由初始状态转移到目标状态。最简单的方法是采用前向搜索，由初始状态开始搜索状态可能的转移路径[2-4]。从前面的章节我们了解到，前向搜索实际上是遍历了所有可能的状态转移路径。因此，当规划问题有解时，前向搜索在有限的状态空间内一定能够找到解。当规划问题无解时，前向搜索将返回失败。前向搜索是一种迭代计算模式，在每一轮迭代中，程序将完成：

(1) 检查当前状态是否是目标状态。若是，则搜索停止，得到解序列。

(2) 寻找匹配当前状态的所有动作模式。若没有动作模式与当前状态匹配，则表明问题无解，计算停止。

(3) 应用并保存匹配成功的动作模式，将动作结果作为新的当前状态，转步骤 1。

在搜索尚未到达目标状态时，生成的动作模式序列称为规划的部分解。前向搜索的伪代码如表 5-1 所示。注意算法第 9 行从匹配成功的动作模式中选择一个执行。该操作实质上是只尝试了一条可能的路径。因此无法保证搜索的完备性。这一步也可以执行 *applicable* 集合中的所有动作，将搜索变为宽度优先，从而保证完备性。动作模式的选择也可以结合启发策略，有利于搜索更快速地到达目标状态。这里给出的前向搜索算法是循环形式的，读者也可以稍作改动，将其与深度优先搜索结合，写为递归式。

表 5-1　前向搜索规划伪代码

```
ForwardSearching(KB, InitState, GoalState)
Input: KB, state space and action modes;
       InitState, start state;
```

续表

```
      GoalState, objective;
Output: an action sequence or failure;
1. currState <-- { InitState };
2. plan <-- { };
3. Repeat
4.    If currState satisfies GoalState
5.      return plan;
6.    applicable <-- {a | a is a ground operator in KB that Precond(a) is true in currState};
7.    If applicable = ∅
8.      return failure;
9.    a = ChooseAction(applicable);
10.   currState <-- Effect(a);
11.   plan <-- plan ∪ a;
12. Until Maximum Iteration.
```

与前向搜索对应的是反向搜索。此时，搜索从目标状态出发逐渐向后扩展，直至到达初始状态。反向搜索的步骤与前向搜索类似，区别仅在于每一轮迭代采用动作模式的结果来匹配当前状态。若匹配成功则将动作的条件作为新的状态继续搜索。反向搜索的伪代码如表 5-2 所示。其中第 6 行计算所有结果状态为当前态的动作模式。作为示例，考虑上一节中形式化表示的积木问题。将目标状态

$$Goal(At(A,R) \wedge At(B,R) \wedge On(A,B) \wedge \neg On(B,A) \wedge On(B,Floor) \wedge Vacant(L) \wedge Vacant(M))$$

表 5-2　反向搜索规划伪代码

```
BackwardSearching(KB, InitState, GoalState)
Input: KB, state space and action modes;
       InitState, start state;
       GoalState, objective;
Output: an action sequence or failure;
1. currState <-- { GoalState };
2. plan <-- { };
3. Repeat
4.    If currState satisfies InitState
5.      return plan;
6.    relevant <-- {a | a is a ground operator in KB that Effect(a) is true in currState};
7.    If relevant = ∅
8.      return failure;
9.    a = ChooseAction(relevant);
10.   currState <-- Precond(a);
11.   plan <-- a ∪ plan;
12. Until Maximum Iteration.
```

匹配动作模式得到候选动作 $Lay(A,B,L,R)$ 和 $Lay(A,B,L,M)$。算法选取其中之一作为关联动作，将当前状态更新为

$$At(A,M) \wedge At(B,R) \wedge (A \neq B) \wedge \neg On(A,B) \wedge \neg On(B,A) \wedge On(B,Floor)$$

此时算法选取一个匹配 $Move(B,L,R)$，得到新状态

$$At(B,L) \wedge \neg On(A,B) \wedge On(B,Floor) \wedge Vacant(R)$$

按此继续搜索，算法最终将得到一个解序列。

需要明确，前向或反向搜索指的是搜索方向，即从问题的初态开始还是从目标态开始。而宽度和深度优先搜索则指的是具体搜索方式。二者并无直接联系。前向搜索可以采用宽度搜索，也可以采用深度搜索。反向搜索类似。然而，正如搜索一章提到的一样，搜索需要保存已经访问过的状态结点，以避免在后续计算中生成相同的状态。否则，搜索将进入无限循环，无法得到结果。

5.3 规划空间规划

使用状态空间搜索求解规划问题是直观的。对应的搜索树中，结点表示系统中间状态，边表示相邻状态转移的动作。本节将讨论另外一种直接在规划空间搜索的求解方法[5,6]。该方法的搜索树上，结点对应着规划解的一个部分动作序列，称为部分规划(Partially Specified Plan)或部分解。边代表规划改进操作(Plan Refinement Operator)，将父结点的部分解向规划完成的方向做改进。原则上，改进操作应避免向部分解规划中添加任何达到目标状态不需要的约束，这被称为最少限制原则(Least Commitment Principle)。在搜索解的过程中，算法不断扩展搜索树，直至到达某个叶结点。叶结点代表的部分解能够实现目标态。

与状态空间规划相比，规划空间规划不仅在搜索空间上存在差异，对解的定义方式也不相同。直观上，规划问题的解是一个动作序列。在确定解的动作序列时，涉及两种规划操作算子：动作模式的选择和所选动作顺序的指定。在此意义下，只需要根据变量绑定约束选取动作集合，并且确定合适的顺序。进一步考虑，对一个部分规划，将其向完成规划的方向改进需要执行四步操作：添加动作、添加动作顺序约束、添加动作之前或之后的状态表示(称为临时联系，Casual Link[7])、添加变量绑定约束。我们仍然以积木世界的例子来说明此过程。

假设部分规划包含一个动作 $Lay(A,B,L,R)$。将动作 $Lay(A,B,L,R)$ 的条件 $At(A,L) \wedge At(B,R) \wedge (A \neq B) \wedge \neg On(A,B) \wedge \neg On(B,A) \wedge On(B,Floor)$ 作为子目标。为达成该子目标，添加动作 $Move(A,a,L)$ 和 $Move(B,b,R)$。它们的执行顺序应该在 $Lay(A,B,L,R)$ 之前，记为 $Move(A,a,L) \prec Lay(A,B,L,R)$，$Move(B,b,R) \prec Lay(A,B,L,R)$。对于这两个动作之间的顺序，当前的子目标并没有限制。按照最小限制原则，我们不指定它们之间的顺序。至此，部分规划变成 $\{Move(A,a,L), Move(B,b,R) \prec Lay(A,B,L,R)\}$。一个潜在的问题是，符号 $\prec$ 仅代表 $Move(A,a,L)$ 的执行时间先于 $Lay(A,B,L,R)$，尚无法准确表达二者在时间轴上的相邻关系。在后续的规划中，规划器可能在二者之间插入其他操作。为避免这一点，需要进一步添加二者的临时联系。临时联系用符号

$Move(A,a,L)\xrightarrow{At(A,L)}Lay(A,B,L,R)$表示。箭头上的是 $Move$ 动作的结果状态，也是 Lay 动作的执行条件。最后，我们考察变量绑定约束。显然，动作模式 $Move(x,a,b)$的变量绑定$\{x/A,b/L\}$和$\{x/B,b/R\}$是产生临时联系 $At(A,L)$所必须的。而依据最小限制原则，变量 a 在当前的部分规划中不需要指定。从以上过程可以看到，部分规划给出了一个动作模式集合，以及该集合中动作的顺序、变量绑定约束和临时联系。称没有临时联系的动作条件为子目标(Subgoal)[8,9]。子目标实质上是待改进的状态。为使部分解经过改进、扩展最终能连接初始状态和目标状态，引入两个哑动作模式 a_0 和 a_∞。初态定义为 a_0 的结果状态，并且 a_0 不包含条件。终态定义成 a_∞的条件，并且 a_∞没有结果状态。这样，规划问题就转化为对部分解$\{a_0\prec a_\infty\}$的改进。

前面提到，规划空间规划是采用规划算子寻找一个动作序列。然而，规划器每一次改进当前的部分解时，并不可能尝试所有的动作模式。我们需要进一步分析规划解的特征，从而减小搜索空间。一个显然的启发式是临时联系。这跟状态空间规划类似。

定义 5-1(威胁)　设 π 是一个部分规划，对于临时联系 $a_i\xrightarrow{p}a_j$，动作 a_k 是 π 的一个威胁(Threat)当且仅当以下三项同时成立：

(1) a_k 产生结果 $\neg q$，且 $\neg q$ 与 p 不一致。例如，当 p 和 q 可合一时，$\neg q$ 与 p 就不一致。

(2) 任意顺序约束$(a_i\prec a_k)$和$(a_k\prec a_j)$是一致的，即不出现循环。

(3) p 和 q 合一的变量绑定约束符合 π 的变量绑定。

简言之，部分规划不存在威胁是指动作序列不存在循环、不会产生矛盾的临时联系状态并且符合变量绑定约束。

定义 5-2(缺陷)　设 $\pi=(A,\prec,B,L)$是一个部分规划，其中 A 代表动作集合，$\prec$代表顺序约束，B 代表变量绑定约束，L 代表临时联系。定义 π 的缺陷为：

(1) π 的子目标(A 中一个无临时联系的动作条件)；

(2) π 的威胁(例如一个可能与某临时关系发生冲突的动作)。

定理 5-1　部分规划 $\pi=(A,\prec,B,L)$是规划问题 $P=(\Sigma,s_0,g)$(Σ 是规划空间，s_0 是初始状态，g 为目标状态)的一个解，如果 π 不包含缺陷并且顺序约束$\prec$和绑定约束 B 都一致。

证明：归纳步，假设定理对任意包含$(n-1)$个动作的规划成立。考虑包含 n 个动作且无缺陷的规划。令 $A_i=\{a_{i1},a_{i2},\cdots,a_{ik}\}$是一个动作集合，其中每一个动作在$\prec$顺序定义下的直接前序动作是 a_0。那么 π 中满足$\prec$顺序约束的任意完全排序动作序列，都必然由 A_i 中的某个动作 a_i 开始。因为 π 中不包含缺陷，所以 a_i 的所有条件都被与 a_0 的临时关系满足(初始状态)，即 $a_0\xrightarrow{s_0}a_i$。

令$[a_0,a_i]$表示由初始状态 s_0 执行动作 a_i 后的第一个状态，并令 $\pi'=(A',\prec',B',L')$为剩余的规划。那么 π'可以通过在 π 中用一个单一的动作取代 a_0 和 a_i 得到。新的动作没有条件，结果为状态$[a_0,a_i]$。绑定约束也能根据动作 a_i 添加到 B 中。下面证明 π'也是一个解：

(1) $\prec'$是一致的。因为从 π 到 π'并没有添加新的顺序约束；

(2) B'是一致的。因为动作 a_i 的绑定约束原本就在 B 中，所以 B'与 B 是一致的；

(3) π'不包含威胁。因为 π 本身不包含威胁,并且也没有添加新动作;

(4) π'不包含子目标。因为 π 中每一个被动作 $a \neq a_0$ 的临时关系所满足的条件在 π' 中仍然满足(临时关系不变)。再考虑临时关系($a_0 \xrightarrow{p} a_j$)满足的条件 p。由于 a_i 在 π 中并不是一个威胁,任何 a_i 作变量绑定后的执行结果都不会与 p 产生不一致。所以 p 仍然被第一个状态$[a_0, a_i]$满足。临时关系变为($[a_0, a_i] \xrightarrow{p} a_j$)。

基于以上证明,再由归纳假设知 π'是一个包含$(n-1)$步动作且无缺陷的解,因此定理得证。

定理 5-1 表明,规划器在求解过程中并不需要将所有的动作进行排序,也不必查询动作对所有的常量绑定。这大大缩减了搜索空间。基于此,我们有规划空间的 PSP(Plan-Space Planning)算法。PSP 算法的基本思想是在保证顺序和变量绑定约束的前提下,对一个部分解 π 作改进,直至不包含缺陷为止。PSP 算法的伪代码如表 5-3 所示。flaws 代表当前部分解的缺陷,*resolvers* 表示对 π 中缺陷 f 的所有改进策略,π'是 π 改进缺陷 f 后的部分解。规划算法以初始部分解 π_0 为输入,每一轮成功的递归都在当前解上改进了一个缺陷。*OpenGoals* 函数用于寻找所有的未被临时关系支持的动作前提。为提高执行效率,该函数通常采用 *agenda* 数据结构。对 π 中任意一个新的动作 a,首先将其全部条件加入 *agenda* 中,然后检查新的临时关系,如果新的临时关系与原有关系不矛盾,则移除相应的条件。函数 *Threats* 查询所有可能对某些临时关系构成威胁的动作。该查询可通过测试所有动作对($a_i \xrightarrow{p} a_j$)完成,复杂度为 $O(n^3)$(n 为 π 中的动作数)。一般采用增量查询:对于 π 中的新动作,测试所有不涉及该动作的临时关系($O(n^2)$复杂度),对于 π 中新的临时关系,测试所有剩余的动作($O(n)$复杂度)。函数 *Resolve* 寻找所有解决缺陷 f 的方法。如果 f 是某个动作 a_j 的条件 p,那么改进策略有两种可能:

(1) 若 π 中包含结果为 p 的动作 a_i,则改进策略是一条临时关系($a_i \xrightarrow{p} a_j$),以及相应的顺序和变量绑定约束。

(2) 改进策略还可能是一个产生状态 p 的新动作 a,包含 a 对应的临时关系、顺序和变量绑定约束。

表 5-3　PSP 算法伪代码

```
PSP(π)
1. flaws <-- OpenGoals(π) ∪ Threats(π);
2. If flaws = ∅
3.    Return π
4. f = ChooseFlaw(flaws);
5. resolvers←Resolve(f,π);
6. If resolvers = ∅
7.    Return failure;
8. ρ = ChooseResolver(resolvers);
9. π'←Refine(ρ,π);
10. Return PSP(π').
```

如果 f 是对某条临时关系($a_i \xrightarrow{p} a_j$)的威胁。该威胁由动作 a_k 产生结果 $\neg q$,且 q 与 p 可合一,那么改进策略有三种可能:

(1) 若 $a_k \prec a_i$ 满足顺序约束,改进策略是将 a_k 置于 a_i 之前的顺序约束;

(2) 若 $a_j \prec a_k$ 满足顺序约束,改进策略是将 a_k 置于 a_j 之后的顺序约束;

(3) 使得 q 与 p 无法合一的变量绑定约束。

Refine 函数用选定的改进策略改进部分解。这一步的操作是容易的,无须其他的测试,直接向 π 中添加一条顺序约束、一条或多条绑定约束、一条临时关系和/或一个新的动作。

规划空间规划虽然搜索空间不同,但其本质上与状态空间规划是等价的。在规划空间下,状态被表示成了临时关系,状态空间中的搜索路径被转换成了缺陷改进和顺序、绑定约束。

5.4 规划图

前两节介绍了状态空间规划和规划空间规划,它们的技术基础都是搜索。本节将搜索与图结合,介绍一种经典的规划方法,称为规划图方法[10,11]。实质上,规划图本身就是一个多项式时间的可采纳启发函数,它能回答规划问题是否有解,并在有解的情况下引导搜索快速找到解。已经证明,规划图是求解困难规划问题的有效手段。但是,这种方法只能求解不含变量的逻辑规划。

规划图是一个分层有向图。状态层(S)和动作层(A)交替出现。处于第 i 个动作层的动作 A_i 以它前面的 S_i 层中的状态为条件,其结果状态位于紧随其后的 S_{i+1} 层。随着迭代次数增加,i 逐渐增大,规划图的层数也逐渐增多,直到达到特定的终止条件为止。下面我们用规划图继续求解积木世界的例子。绘制规划图的第一步是将所有的初始状态文字排列在 S_0 层,满足条件的动作模式放在 A_0 层,动作结果生成 S_1 层。如图 5-2 所示。在动作层引入一个“空操作”(用方框画出),表示不进行任何动作。相应的状态也保持不变。A_0 层中,动作 $Take(A,L,M)$ 和空操作不能同时发生,称为互斥动作模式,用虚线连接。因为它们产生了互斥的结果 $On(A,B)$ 和 $\neg On(A,B)$。

通常,判断两个动作模式是否互斥遵循以下三个条件:

(1) 不一致效果,即两个动作的某个结果状态互为相反文字。

(2) 冲突,即一个动作的效果之一是另一个动作前提的否定。

(3) 竞争,即两个动作的某个前提互为相反文字。

注意第二轮规划图中并未画出空操作的互斥关系。积木问题第三轮迭代的规划图更加复杂。为清楚表示,图 5-3 中只给出了得到解的动作。执行完图中动作后,目标状态已经出现。接下来需要进行反向搜索求出解。具体过程是:

(1) 设定目标状态为当前状态集。

(2) 选取结果覆盖当前状态集合的动作模式集合。所选取的动作模式中不能包含互斥关系。

(3) 将动作模式的前提状态作为新的当前状态集合。

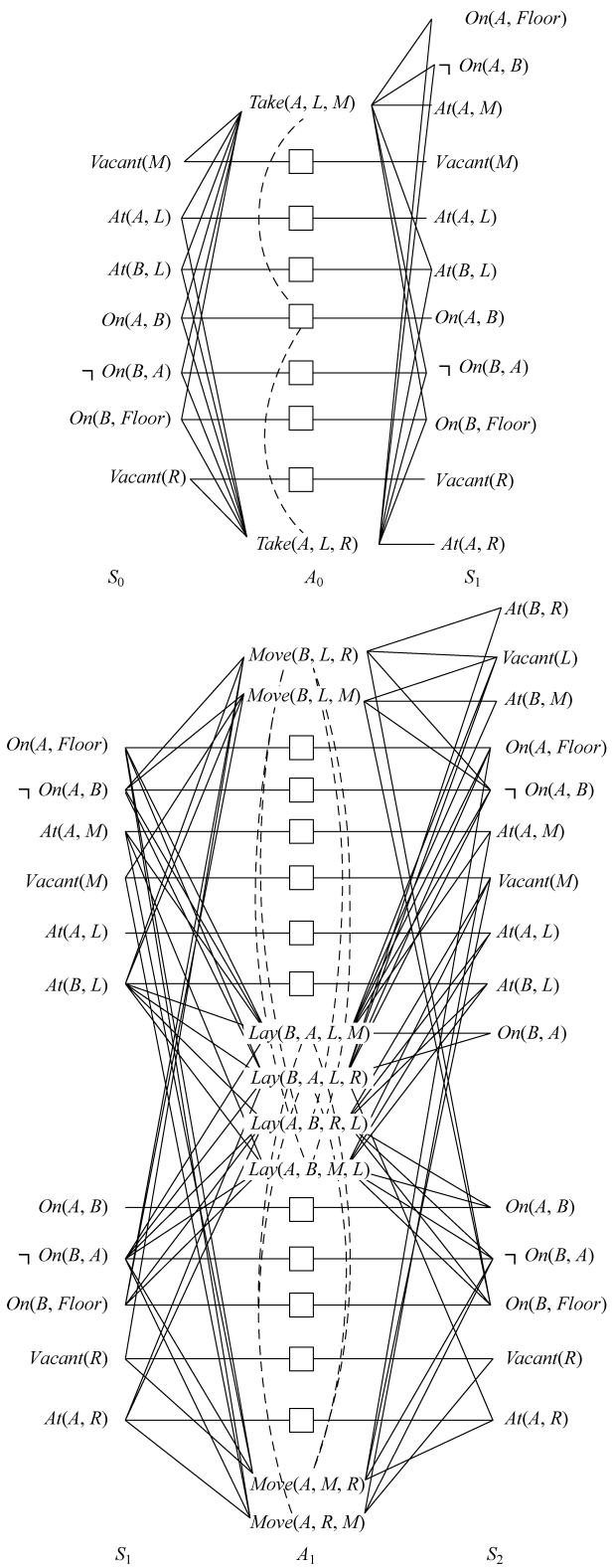


图 5-2　积木世界的前两轮规划图

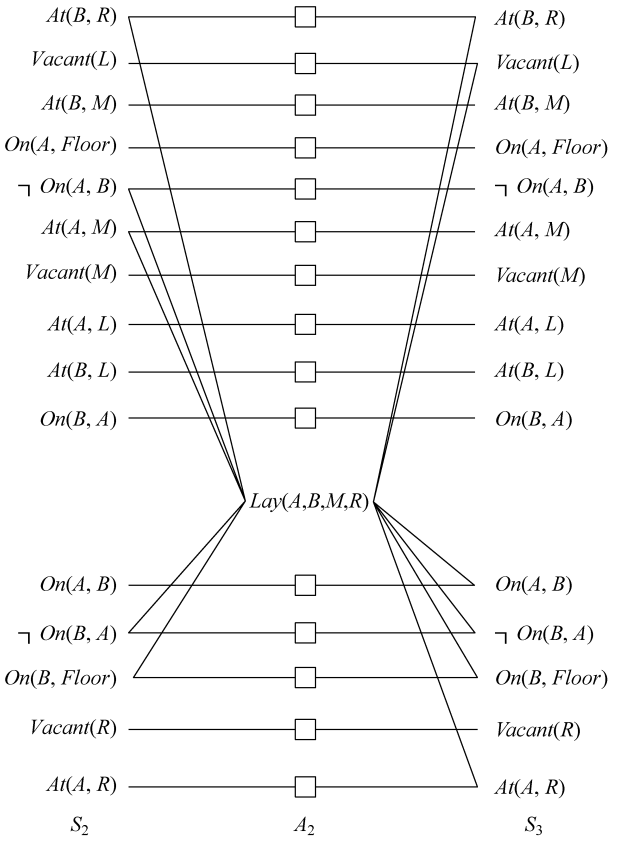


图 5-3　积木问题第三轮得到解的规划图

(4) 重复第 2 和第 3 步直到当前状态集合属于 S_0 层。

对我们的例子而言，首先将目标态作为当前状态集。结果覆盖该状态集的动作模式只有 $Lay(A,B,M,R)$。将 $Lay(A,B,M,R)$的前提状态作为当前状态集为

$$At(B,R) \wedge \neg On(A,B) \wedge At(A,M) \wedge \neg On(B,A) \wedge On(B,Floor)$$

在第二轮规划图中，寻找无互斥的动作集合得到

$$\{Move(B,L,R), At(A,M) \text{ 的空操作}\}$$

继续迭代，新的当前状态集为

$$At(B,L) \wedge \neg On(A,B) \wedge On(B,Floor) \wedge Vacant(R) \wedge At(A,M)$$

在第一轮规划图中寻找动作集合得到

$$\{Take(A,L,M), Vacant(R) \text{ 的空操作}\}$$

更新当前状态为 S_0 态，搜索结束，得到一个解的动作序列：$[Take(A,L,M) \rightarrow Move(B,L,R) \rightarrow Lay(A,B,M,R)]$。反向搜索并不总是能得到解，也可能返回失败，表明问题无解。反向搜索也可以指定每一步的代价，从而得到最优规划。

积木世界的例子展示了如何构造规划图。一般情况下，规划图随迭代进行逐层增加。扩展终止的条件稍显复杂[12,13]。首先，当出现完全相同的两个状态层时，说明此时规划图达到了稳定，扩展暂时停止，采用反向搜索计算解。若反向搜索没有得到解，则记录规划图的层数和不满足的目标状态(称为 No-Good)。在扩展规划图至下一层。若 No-Good 的数

量减少，则继续扩展，否则表明规划无解。最后，表 5-4 给出了规划图算法伪代码。

表 5-4　规划图算法伪代码

```
GraphPlan(KB, InitState, GoalState)
Input: KB, state space and action modes;
       InitState, start state;
       GoalState, objective;
Output: an action sequence or failure;
Graph <-- { InitState }, a graph plan initialized by InitState;
NoGoods <-- { }, a set that keeps the unsatisfied states;
1. Repeat
2.   If all goal states are contained in S_t and are not conflicted
3.     ActSequence <-- BackSearch(Graph, GoalState, NoGoods);
4.     If ActSequence is not Failure
5.       return ActSequence;
7.   If Graph and NoGoods are leveled off
8.     return Failure;
9.   Expand Graph;
10. Until Maximum Iteration.
```

5.5　时序规划

到目前为止，本章讲述的规划问题只考虑了有限状态空间上的动作先后顺序。这一问题被称为经典规划。然而，实际生产和管理中还有很多问题需要明确考虑动作的持续时间、资源的配置约束等。这类问题被称为时序规划和资源调度[14,15]。其问题表示和求解方法与经典规划相比都存在区别。本节的剩余部分将分别讨论。

时序规划可被定义为由常量符号、变量符号、关系符号和约束四部分组成。常量符号是规划域上不同对象的指代名称，如 Box、Floor 以及前面例子中的 A、B 等。变量符号分为对象变量和时间变量，前者表示规划域上某类型的对象，取值范围是离散的常量符号，后者专指时间，取值为实数（本节中时间以实数表示）。关系符号有两类。一类是严格关系符号，表示不随时间变化的关系，如 $Adjacent(L,M)$；另一类是可变关系符号，表示在规划过程中并不总是成立的关系，如 $At(A,M)$。约束也分为两类。一类是时间约束，此处的时间约束并不采用真实数值实例化，而是以约束变量的一致集合表示；另一类是绑定约束，指明了对象变量能够被实例化的范围，该范围实质是对象变量的定义域。由于规划域上的常量符号是有限的，因此绑定约束给出的对象变量定义域也是有限的。约束在描述上可分为时不变描述和时变描述。前者是严格关系和绑定约束（统称为对象约束），后者是可变关系和时间约束。

形式化地，我们定义时间限定表达式（Temporally Qualified Expression，TQE）为如下形式

$$p(\zeta_i,\cdots,\zeta_k)@[t_s,t_e)$$

其中，p 是可变关系，$\zeta_i,\cdots,\zeta_k$ 是约束或对象变量，t_s,t_e 是时间变量，满足 $t_s<t_e$。该时间限定表达式声明了对任意满足 $t_s\leqslant t<t_e$ 的时刻 t，$p(\zeta_i,\cdots,\zeta_k)$ 成立。此处为了数学上表示方便，采用左闭右开的时间区间。进一步定义时间数据库(Temporal Database)为

$$\Phi=(\mathcal{F},\mathcal{C})$$

其中，$\mathcal{F}$是一个有限的时间限定表达式集合，$\mathcal{C}$是一个时间和对象约束的有限集合。另外，从规划解存在的角度看，以$\mathcal{C}$为约束的约束满足问题必须存在可行解(即存在覆盖所有变量的实例满足$\mathcal{C}$的全部约束)，否则规划问题无解。时间数据库给出了世界状态随动作执行的变化情况。图 5-4 给出了积木世界例子的部分时间数据库，用时间限定表达式集合可表示为

$$\begin{aligned}\Phi=(&\{At(A,L)@[\tau_0,\tau_1),At(A,Route)@[\tau_1,\tau_2),At(A,M)@[\tau_2,\tau_3),\\&Vacant(L)@[\tau_1,\tau_3),Vacant(M)@[\tau_0,\tau_2)\},\\&\{Adjacent(L,M),\tau_0<\tau_1<\tau_2<\tau_3\})\end{aligned}$$

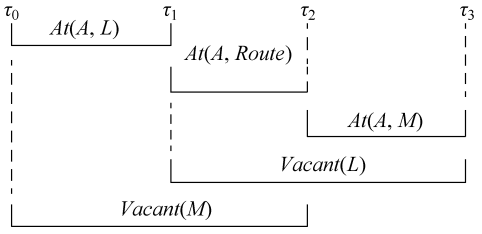


图 5-4　积木世界的时间数据库(部分)

前三条时间限定表达式指出了积木 A 分别在$[\tau_0,\tau_1)$、$[\tau_1,\tau_2)$、$[\tau_2,\tau_3)$时间段内位于 L 处、移动路径上和 M 处。后两条表达式指出了 L 和 M 的空置时间。约束部分给出了 L 和 M 的相邻关系，以及对时间点的约束。此例中，时间限定表达式都是原子谓词，并不包含逻辑连接符，也不包含谓词的否定。这遵循了封闭世界假设，即时间表达式只在限定时间内成立，严格关系在数据库内恒成立。

假设给定一条一般形式的时间限定表达式 $T=Vacant(a)@[t_1,t_2)$，其中 a,t_1,t_2 均为变量。如果时间数据库 Φ 中存在所有变量的一个置换，使得变量绑定后 T 成立，那么就称 Φ 支持该表达式 T。例如在积木世界的例子中，声明 $Vacant(L)@[\tau_1,\tau_3)$在置换$\{a/L,t_1/\tau_1,t_2/\tau_3\}$下能够成功匹配表达式，所以 Φ 支持 T。注意，Φ 支持 T 并不需要时间变量的精确置换。任何满足$\{\tau_1\leqslant t_1,t_2\leqslant\tau_3\}$的置换都能保证 T 成立。使时间表达式 T 成立的变量绑定约束以及相应的时间约束(如 $\tau_1\leqslant t_1,t_2\leqslant\tau_3$)共同称为 T 的可行条件(Enabling Condition)。

形式化表示时序规划问题后，我们定义时序规划的规划算子，它也由四部分组成，记为

$$o=(name(o),precond(o),effects(o),const(o))$$

$name(o)$是形如 $o(x_1,\cdots,x_k,t_s,t_e)$的表达式。括号外是动作符号，括号内是对象变量和时间变量。$precond(o)$和 $effects(o)$是时间限定表达式。$const(o)$是时间约束和对象约束(对象约束包括严格关系和变量绑定约束)。为简洁表示，规划算子也写成

$$\begin{aligned}&Move(x,a,b)@[t_s,t_e),\\&precond:At(x,a)@[t_1,t_s)\wedge Vacant(b)@[t_1,t_e)\\&effects:At(x,Routes)@[t_s,t_e)\wedge At(x,b)@[t_e,t_2)\end{aligned}$$

$$\wedge\ Vacant(a)@[t_s,t_2)$$

$$const:\ t_1<t_s<t_e<t_2,Adjacent(a,b)$$

其中 t_1 和 t_2 是自由变量。时序规划的动作模式定义为规划算子经过部分变量置换后的结果。给定时间数据库 $\Phi=(\mathcal{F},\mathcal{C})$，若 $\mathcal{F}$ 支持动作模式 a 的所有 $precond$，并且存在使得 $\mathcal{C}\cup const(a)\cup c$ 满足约束的可执行条件 c 时，那么称动作模式 a 对 Φ 是可执行的。动作持续时间从 t_s 到 t_e，执行后的状态由 $effects$ 部分给出。需要指出，在经典规划中，一个动作的执行结果状态只影响到该动作涉及的变量，而不会更改其他对象的状态。但在时序规划中，一个动作的执行可能改变其他对象状态。而其他对象的这些更改并没有反映在 $effects$ 中。此外，经典规划中一个动作执行后可能导致对原状态事实集合增加新事实和删去不再满足的事实陈述。而在时序规划中，动作的执行结果只会对原事实集合进行添加而不会删除。这是因为不同时间段下的状态被视为不同的事实，因此随着时间的流逝，新的事实会不断被加入。

请再次考虑积木世界的时间数据库。一个基本事实是，同一块积木不能同时放置在两个不同的位置。这条常识被称为公理(axiom)，可用表达式表示为

$$\{At(x,a)@[t_s,t_e),At(y,b)@[t'_s,t'_e)\}\rightarrow(x\neq y)\vee(a=b)\vee(t_e\leqslant t'_s)\vee(t'_e\leqslant t_s)$$

显然，如果箭头之前的两个时间限定表达式成立，那么要么是不同的积木($x\neq y$)，要么是同一个积木位于相同的地点($a=b$)，要么是同一个积木在先后不同的时间段内位于不同的地点($(t_e\leqslant t_s')\vee(t_e'\leqslant t_s)$)。因此箭头之后的析取式总是成立。一般地，公理被定义为如下形式

$$\rho=cond(\rho)\rightarrow disj(\rho)$$

其中，$cond(\rho)$是时间限制表达式集合，$disj(\rho)$是时间和对象约束的析取式。

现在，我们已经具备了定义时序规划问题的条件。先定义时序规划的规划域为三元组

$$\mathcal{D}=(\Lambda_\Phi,\mathcal{O},X)$$

其中，Λ_Φ 是通过常量、变量、关系和约束定义的所有时间数据库集合；$\mathcal{O}$是时序规划算子集合；X 是规划域公理集合。

一个时序规划问题被定义为三元组

$$\mathcal{P}=(\mathcal{D},\Phi_0,\Phi_g)$$

其中，$\mathcal{D}$是时序规划域。$\Phi_0=(\mathcal{F},\mathcal{C})$是 Λ_Φ 中满足公理集合 X 的数据库。实际上，Φ_0 给出了一个初始场景，该场景不仅描述了规划域的初始状态，也给出了规划动作下独立发生的演化预测。$\Phi_g=(\mathcal{G},\mathcal{C}_g)$是表示目标状态的数据库，$\mathcal{G}$是时间限定表达式集合，$\mathcal{C}_g$是$\mathcal{G}$中变量的对象和时间约束。

在规划域明确的情况下，时序规划问题也常常被写为 $P=(\mathcal{O},X,\Phi_0,\Phi_g)$。规划解定义为一个动作集合 $\pi=\{a_1,\cdots,a_k\}$。这里没有给出动作顺序是因为执行时间已经被包含在动作中了。

在给出了规划问题及其解的表示形式之后，接下来我们关注如何求得解集合 π，使得集合中的动作是规划算子$\mathcal{O}$的实例化，并且存在能导出目标状态$\Phi_g=(\mathcal{G},\mathcal{C}_g)$的时间数据库 $\Phi=(\mathcal{F},\mathcal{C})$。实际应用中，规划器常常需要在求出解集合的基础上，再给出 Φ。典型的时序规划算法有 TPS 算法(Temporal Planning Search)，它与 PSP 算法类似。TPS 算法的伪代码如表 5-5。Ω 是一个给出了当前部分解的数据结构。在未取得规划解时，Ω 始终包含缺

陷。算法仍然先选择一个缺陷并搜索所有的缺陷消除方法，然后选取其中一种消除方式对部分解改进。缺陷消除方式的选取将被保存下来，若后续搜索中无法最终消除所有缺陷以致求解失败，算法将回溯到保存点上选取另外的消除方式进行搜索（实质上是深度优先）。部分解缺陷可能以开放目标、不满足公理和威胁三种形式存在。若 $G=OpenGoals(\Omega)$ 不为空，则 G 中的每一个目标表达式 e 均不被 Φ 支持。这类缺陷的改进策略可能是：

（1）若 $\mathcal{F}$ 中 e 的可执行条件不为空，且至少包含一个与 $\mathcal{C}$ 一致的可执行条件，那么改进策略是一条支持 e 的时间限定表达式。对 Ω 的相应改进是 $\mathcal{K}\leftarrow\mathcal{K}\cup\{\theta(e/\mathcal{F})\}$，$G\leftarrow G-\{e\}$。其中，$\mathcal{K}$ 是搜索动作的可执行条件和公理的一致性条件集合，$\theta(e/\mathcal{F})$ 表示 $\mathcal{F}$ 中与 $\mathcal{C}$ 一致的可执行条件。

（2）某个规划算子的实例化动作 a。该动作的结果 $effects(a)$ 支持 e 并且 $const(a)$ 与 $\mathcal{C}$ 一致。相应地，Ω 的改进策略是

$$\pi\leftarrow\pi\cup\{a\},\mathcal{F}\leftarrow\mathcal{F}\cup effects(a),\mathcal{C}\leftarrow\mathcal{C}\cup const(a),$$
$$G\leftarrow(G-\{e\})\cup precond(a),\mathcal{K}\leftarrow\mathcal{K}\cup\{\theta(a/\Phi)\}$$

缺陷的第二种形式是不满足的公理，即 Φ 中与 X 不一致的实例。这类缺陷的改进策略就是 $\theta(X/\Phi)$ 中的任意一个一致性条件，Ω 的更新为 $\mathcal{K}\leftarrow\mathcal{K}\cup\{\theta(X/\Phi)\}$。第三类缺陷是威胁，即 $\mathcal{K}$ 中每一条一致性条件和动作可执行条件 $\mathcal{C}_i$ 都无法从当前的 Φ 中导出。其改进策略是向 $\mathcal{C}$ 中添加一个约束 c，使得它与 $\mathcal{C}_i$ 一致：$\mathcal{C}\leftarrow\mathcal{C}\cup c$，$\mathcal{K}\leftarrow\mathcal{K}-\{\mathcal{C}_i\}$。

表 5-5　TPS 算法伪代码

```
TPS(Ω)
1. flaws <-- OpenGoals(Ω) ∪ UnisatisfiedAxioms(Ω) ∪ Threats(Ω);
2. If flaws = ∅
3.   Return Ω
4. f = ChooseFlaw(flaws);
5. resolvers←Resolve(f,Ω);
6. If resolvers = ∅
7.   Return failure;
8. ρ = ChooseResolver(resolvers);
9. Ω´←Refine(ρ,Ω);
10. Return PSP(Ω´).
```

TPS 算法是一个一般的算法流程。采用不同的具体启发式和搜索策略实例化其中的子函数就能得到不同的规划算法。例如，我们可以在选取缺陷时加入启发式，从而更快地得到解，也可以在析取的关系下，并行地同时执行多个规划器，提高规划效率。另外，如何高效地处理 TPS 中多种不同的约束也是需要仔细设计的一个方面。

5.6　本章小结

机器自动规划一直是活跃的研究领域之一。在日常生产生活中，自动规划带来了不可估计的效益。建立在逻辑表示和搜索之上的经典规划具备严格的数学支撑，其解的存在性是可判定的，其规划结果也是确定而可信的。然而一般情况下规划问题的求解往往是复杂

的。本章给出了状态空间搜索、规划空间搜索和规划图三种经典规划方法，并在此基础上引入时间变量，进一步讲述了时序规划的表示和求解。针对具体应用问题，特别是规划解的动作序列较长的情况，一方面我们需要设计好的搜索策略和启发式来平衡空间和时间的计算复杂度，另一方面可以借助成本越来越低的硬件优势，采用并行化的方法提高搜索效率。

参考文献

[1] T. Winograd. Understanding Natural Language[M]. Academic Press，1972.

[2] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning [J]. Artificial Intelligence，2000，116：123-191.

[3] J. Hoffmann. FF：The Fast-Forward planning system[J]. AI Magazine，2001，22(3)：57-62.

[4] D. S. Nau，Y. Cao，A. Lotem，et al. SHOP：Simple Hierarchical Ordered Planner. In Proceedings of the International Joint Conferenceon Artificial Intelligence (IJCAI)[C]，1999：968-973.

[5] E. Sacerdoti. Planning in a hierarchy of abstraction spaces[J]. Artificial Intelligence，1974，5：115-135.

[6] E. Sacerdoti. The nonlinear nature of plans. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)[C]，1975：206-214.

[7] A. Tate. Generating project networks. In Proceedings of the International Joint Conference on Artificial Intelligence (IICAI)[C]，1977：888-893.

[8] R. Korf. Planning as search：A quantitative approach[J]. Artificial Intelligence，1987，33：65-88.

[9] A. Barrett and D. S. Weld. Partial order planning：Evaluating possible efficiency gains[J]. Artificial Intelligence，1994，67(1)：71-112.

[10] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)[C]，1995：1636-1642.

[11] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis[J]. Artificial Intelligence，1997，90：281-300.

[12] S. Kambhampati，E. Parker and E. Lambrecht. Understanding and extending Graphplan. In Proceedings of the European Conference on Planning (ECP)[C]，1997：260-272.

[13] S. Kambhampati and X. Yang. On the role of disjunctive representations and constraint propagation in refinement planning. In Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)[C]，1996.

[14] K. Baker. Introduction to Sequencing and Scheduling[M]. Wiley，1974.

[15] S. French. Sequencing and Scheduling：An Introduction to the Mathematics of the Job Shop[M]. Horwood，1982.