《人工智能》课程系列

规划基础*

Preliminaries on Planning

武汉纺织大学数学与计算机学院

杜小勤

2020/08/25

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: September 7, 2020。

1 概述

经典的规划 (Classic Planning) 技术将人工智能领域中的搜索与逻辑这 2 种方法进行了结合，使得智能体除了具有应激行为或反应式行为 (Reactive Behavior) 之外，还具有某种程度的深思熟虑 (Deliberation) 能力。自规划技术诞生以来，半自动规划与自动规划的理论研究与实践开发一直是人工智能与控制领域的活跃与持续增长的热点。例如，在工业应用机器人、商业服务机器人、营救与探险机器人、自动空间工作站与卫星、无人驾驶汽车、视频游戏等应用与产品中，处处都有着规划技术的身影。

STRIPS 于 1971 年由 Richard Fikes 和 Nils Nilsson 共同开发，它是 SRI 的 Shakey 机器人项目中的规划系统，也是历史上第 1 个主流的规划系统[1]。

实际上，从理论研究的角度看，人工智能中的规划技术发端于对状态空间搜索、定理证明与控制理论的研究；从实践应用的角度看，规划技术受到机器人技术、调度 (Scheduling) 及其它相关应用领域的实际需求的驱动。

在“强化学习基础”一章中，我们曾经引入了环境与智能体的交互模型，即 EA(Environment-Agent) 模型，如图1-1所示。

在该模型中，左端的“State”与“Reward”相当于智能体感知到的环境反馈，它属于感知 (Percepting) 子模块；右端的“Action”相当于智能体对环境施加的动作或行为，它属于行动 (Acting) 子模块。对于只配备强化学习基础模块的智能体而言，智能体为了实现其自身目标，在与环境的交互过程中，会依据环境的状态生成一系列动作，这种行为方式被称为应激反应或反应式行为。与深思熟虑的能力相比，应激反应处于较低的层次[2]。在后面，为表达方便，将“深思熟虑”简称为“慎思”。

进一步地，为了体现智能体智能方式的多样性，我们对图1-1所展示的模型进行进一步的抽象，如图1-2所示。可以看出，智能体具有感知、反应式/慎思、行动这三个功能模块。其中，反应式与慎思模块相当于智能体的大脑 (Brain)。

[1]STRIPS 后来也被用来作为该规划系统使用的形式语言的名称，该语言也是至今仍在使用的大多数自动规划语言的基础。通常，类似 STRIPS 这样的语言也被称为动作语言 (Action Language)。

[2]当然，对于具有规划功能的强化学习智能体而言，它也具备深思熟虑的能力。另外，一个动作序列被称为策略。

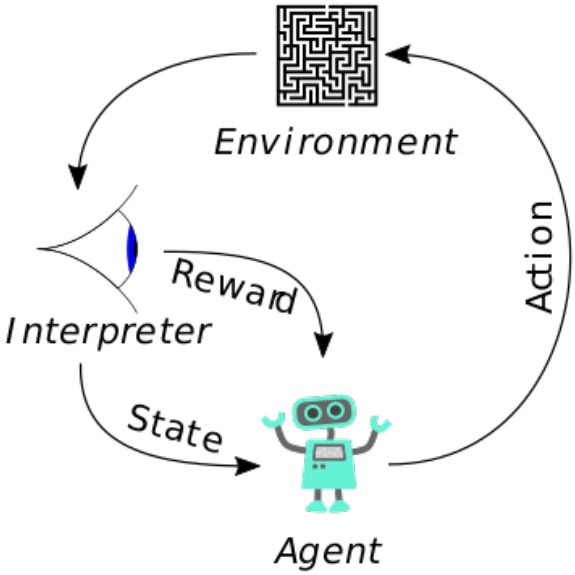


图 1-1: 环境与智能体的交互模型 (EA 模型)

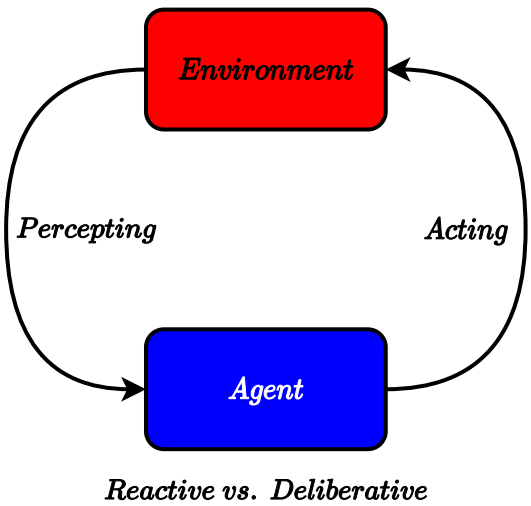


图 1-2: 环境与智能体交互的抽象模型

现在来考察一下鸟类的几种活动与行为方式。例如，一群大雁在飞行时，会有一只领头雁在前面飞行，而其它的大雁要以一定的队形跟随领头雁，且相互之间需要保持一定的距离，既不能太近，也不能太远[3]。一般而言，此类群体飞行模式，对于常年累月进行飞行与迁徙的鸟类而言，是一种几乎出自“本能”且容易实现的行为模式，具有应激行为或反应式行为的特征。而在亚马逊热带丛林中，有一种鸟，它能够巧妙地利用鸟喙将某种树叶，制作成带有“钩子”形状的工具，然后使用这种钩子工具，将食物从较深的地方勾出。显然，此类行为，即便从人类的角度而言，也是具有较高智能的行为。为了实现此类行为，鸟类需要具备某种程度的动作规划或慎思能力。在一些动物科学实验中，也观察到了类似的现象。例如，在一个实验中，将一只面包虫放在某个联动机关中，只有触动某个或几个开关，乌鸦才能吃到面包虫。乌鸦在尝试了几次之后，最后顺利地吃到了面包虫。此类行为，具有较高的智能属性。如果鸟类没有识别出其中的因果关系，仅依靠简单的应激模式，它们根本不足以实现相应的目标。

具体而言，我们将智能体的慎思能力归因于规划技术。规划技术指的是，智能体依据目标来制定出一个能够实现该目标的动作序列或策略。通常情况下，所制定的动作序列是非常复杂的，且涉及到多种粒度或多层次的动作及其交互；并且，在实际应用中，或者由于环境在动态地变化，或者由于动作和行为执行效果的不确定性，使得智能体在某个执行步，没有 100% 地实现既定子目标——在这种情况下，智能体需要重新进行规划——这意味着规划与行动是紧密交织在一起的。

例如，工厂车间中的自动巡逻与送货机器人，在没有任务时，可以执行巡逻任务；当接收到送货任务时，能够收集货物的需求信息，查询出存储位置，然后移动到仓库中的适当位置，抓取货物，最后将指定货物运送到目标位置。在此过程中，自动机器人需要随时监控当前的子任务是否得到有效的执行，如果没有实现预期目标，则需要重新进行规划。此外，在巡逻的过程中，除了巡逻任务本身需要不断地进行规划之外，当接收到新的送货任务时，也需要重新进行规划。在特定情况下，如果送货任务被取消，则自动机器人需要将货物送还原处，并再次进入巡逻状态。总之，自动机器人需要具备规划与再规划的能力。

从某种角度来看，规划与搜索比较相似。但是，规划更加复杂，它还包括逻辑推理、概率等各种组合问题。尤其在实际应用中，各种组合问题相互缠绕在一

[3]Reynolds 在 1986 年发表了关于模拟鸟类群体飞行的论文，提出了群体飞行所需要的三种行为：分离 (Separation)、队列 (Alignment)、聚集 (Cohesion)，成功地模拟了鸟类的飞行活动。

起，使得规划问题显得更加复杂化。

规划模块主要包括 2 项功能，即动作的选择与动作的组织。行动模块的主要功能是，负责将规划的动作进行细化，方便底层的执行机构进行执行。它们具有如下特点：

1. 半自动化或自动化

   不需要或较少依赖于人的控制。

2. 多样性

   多种多样的任务与环境。

3. 层次性与粒度

   规划的动作可以位于不同的粒度层次，有的需要进一步细化，直至低级动作为止。

4. 动态性

   在实际应用中，环境是动态变化的。通常情况下，智能体的规划、再规划与执行等要素交织在一起，且随时间演化。

5. 可观察性

   智能体对于环境的感知，或者从环境所获得的决策所需要的信息，可能是完全可观察的 (Fully Observable) 或部分可观察的 (Partially Observable)。

6. 不确定性

   环境与动作的不确定性，使得智能体的感知以及动作执行的效果，都存在着某种程度的不确定性，并继而影响到规划。

7. 时间约束、在线或实时性

   在实际应用中，有些规划与行动系统需要具备实时性，例如，空间探险与救援机器人等。另外，动作的执行也是有期限的；不同的动作之间也需要满足时序约束关系。

8. 并发性

   在实际应用中，规划与行动可能需要并发执行。即便在一个规划的动作序列内部，不同的动作可能也需要并发执行。

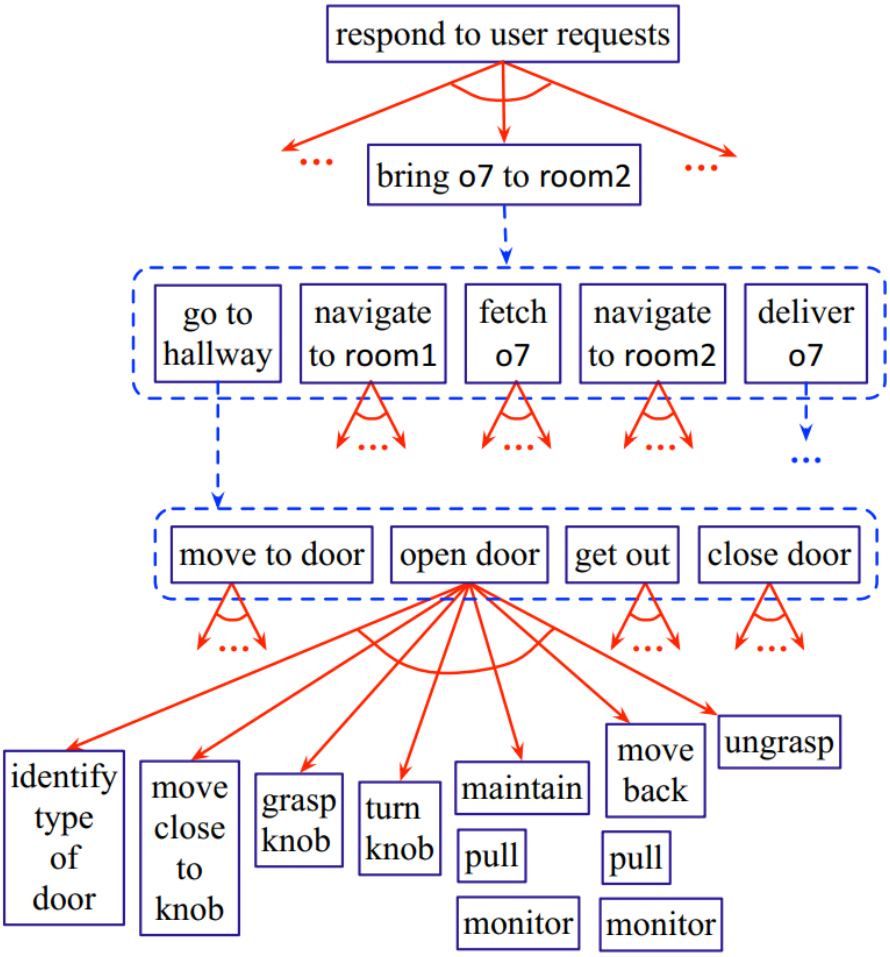


图 1-3: 规划与行动的层次结构示意图

9. 资源约束

   对于一些实际系统，在所规划出的动作序列中，一些动作之间存在资源合作与竞争的约束关系，因而它们之间需要进行同步。

10. 多智能体

    在现实世界中，存在着许多的多智能体系统，例如网球与乒乓球等球类运动中的双打比赛、工厂中的多机器人协作装配车间等。与资源约束情形类似，多智能体之间也存在着合作与竞争的约束关系。

图1-3展示了规划与行动的层次结构示意图。图中动作的抽象程度，从下往上，越来越高：在最底层，都是一些能够直接驱动底层机构加以执行的具体动作、命令或指令；而在较高层，则是一些智能体赖以进行高效规划的动作或行为。在该

图中，红色带箭头的实线，表示一个抽象动作的具体细化；蓝色虚线框及带箭头的虚线，则表示将一个抽象动作、行为或子任务展开为一个动作序列。

显然，上述层次化的组织结构模式，往往也会出现在许多其它具有智能的复杂系统上，俨然已经成为一种具有普适性的结构模式。对于规划与行动系统而言，层次化结构的引入，不仅仅可以提高动作规划的效率，也能够为动作表示与执行的异构性机制，提供有力的技术保障。

例如，在动作的规划阶段，可以采用描述性模型 (Descriptive Model) 来描述动作及智能体的目标。代表模型包括命题逻辑、一阶逻辑及情景演算等。其优点是，抽象程度较高，能够很好地表达动作的执行效果，允许动作规划系统高效地生成动作序列或策略的层次结构。另一个优点是，在动作规划阶段，智能体不需要考虑动作或行为的具体执行内容，起到了屏蔽或隔绝作用。而在动作的执行阶段，可以采用过程式模型 (Procedural Model) 来驱动具体动作的执行——智能体依据动作规划的层次结构，向底层的执行机构发出具体的命令或指令，将抽象的高层动作转换为具体的底层动作。这一部分功能，需要程序员以硬编码的方式，实现状态-动作映射系统或有限状态机。显然，上述层次结构，有助于规划与行动中所呈现的异构动作功能的实现。当然，描述性模型与过程式模型应该是一种相互取长补短的关系。例如，描述性模型可以将常规流程逻辑化，同时为了避免逻辑系统的复杂性，不必过多地考虑实际应用中出现的各种事件及异常情况的处理。相反地，过程式模型可以发挥自身的优势，对描述性模型不能处理的事件与异常情况加以处理。

总之，在智能体的规划与行动系统的设计过程中，可以引入数据抽象与过程抽象、层次性等特征，能够有效地提高动作规划与行动执行的效率，且使得智能体的感知、规划与行动这三个模块之间能够流畅地运行。

在本章，我们只考虑智能体在完全可观察的、确定的、离散的、静态的环境中的规划[4]。我们把满足上述条件的环境，称为经典的规划环境 (Classical Planning Environment)。与之相对的环境，被称为非经典规划环境 (Nonclassical Planning Environment)，例如环境可能是部分可观察的 (Partially Observable) 或随机的 (Stochastic) 等。

[4]离散指的是智能体按时间步与环境进行交互。静态并不意味着环境不发生变化，而是指环境仅因智能体执行动作而发生变化。

2 规划的标准化语言

我们在约束满足问题与一阶逻辑中，已经体会到了状态的结构化表示方法的优点。一方面，它有利于通用策略求解技术的使用。另一方面，它也有利于利用结构中存在的约束消解大量的搜索空间。

从特征或要素的角度看，这种结构化的表示方法实际上属于特征表示 (Feature Representation) 或因式化表示 (Factored Representation)——用一组变量或特征表示环境中的一个状态[5]。

在自动规划 (Automated Planning) 领域中，动作及其表达是描述规划任务的核心要素，而动作的应用前提条件与应用效果，则是规划得以自动进行的关键。因此，为了推动与促进自动规划的研究，研究者们纷纷提出了若干动作语言 (Action Language)，并进而基于这些标准化语言提出各种自动规划算法。

下面，介绍几种常见的规划或动作标准化语言。

2.1 STRIPS

STRIPS(STanford Research Institute Problem Solver) 是自动规划器 (Automated Planner)STRIPS 的动作规划语言，于 1971 年由 Richard Fikes 和 Nils Nilsson 联合开发。该语言开创了规划形式化语言的先河，并成为现今大多数自动规划语言的基础。

一个基本的 STRIPS 实例包括以下三项：

- 初始状态；
- 目标状态；
- 动作集合。每个动作包括以下 2 项：
  1. 前置条件 (Precondition)：动作执行的先决条件；
  2. 后置条件 (Postcondition)：动作执行后产生的效果；

[5]在机器学习领域，这种表示方法得到了普遍的使用——可以使用各种建模方法来表达状态及其关系的各种定量属性，例如朴素贝叶斯、HMM 等。人工智能领域中的约束满足问题、一阶逻辑与规划等，则从其它的角度来研究状态及其关系。

形式上，STRIPS 是一个四元组 $\langle V, A, I, G\rangle$。其中，$V$ 表示所有条件或命题变量 (Propositional Variable) 组成的集合[6]；A 表示算子或动作集合，而每个动作又是一个四元组 $\langle \alpha, \beta, \gamma, \delta\rangle$，该四元组的每个元素都是条件的集合：$\alpha$ 指定了动作执行之前真值必须为 $True$ 的条件、β 指定了动作执行之前真值必须为 $False$ 的条件、γ 指定了动作执行之后真值必须为 $True$ 的条件、δ 指定了动作执行之后真值必须为 $False$ 的条件；I 表示初始状态，其中所有条件的真值为 $True$，而不在初始状态中的条件，其真值缺省为 $False$；G 表示目标状态，以二元组的形式 $\langle N, M\rangle$，分别指定了目标实现时哪些条件的真值为 $True$，哪些条件的真值为 $False$。

于是，我们可以将规划 (Planning) 定义为一个动作序列，该动作序列从初始状态出发，依次按顺序执行序列上的动作，并最终到达目标状态。

有了上面的形式化定义，我们可以将状态转移函数 (Transition Function) 定义为 $succ : 2^V \times A \to 2^V$，其中 2^V 表示 V 的所有子集组成的集合，即幂集 $2^V = \{s | s \subseteq V\}$，其中 s 即为状态，它为条件的集合。通常，出现在 s 中的条件或变量，其取值均为 $True$。设 $s_2 = succ(s_1, a_1) = succ(s_1, \langle \alpha_1, \beta_1, \gamma_1, \delta_1\rangle)$，则它所表达的含义为：在状态 s_1 下，执行动作 $a_1 = \langle \alpha_1, \beta_1, \gamma_1, \delta_1\rangle$ 之后，环境将转移到状态 s_2。

进一步地，根据动作 $a = \langle \alpha, \beta, \gamma, \delta\rangle$ 的定义，可以将状态转移函数写为如下的具体形式：

$$succ(s, \langle \alpha, \beta, \gamma, \delta\rangle) = \begin{cases} (s \backslash \delta) \cup \gamma & if\ \alpha \subseteq s\ and\ \beta \cap s = \emptyset \\ s & otherwise \end{cases} \tag{1}$$

上式中，第 1 行表示允许动作 a 执行的所有取值为 $True$ 的前提条件均出现在当前状态 s 中 $(\alpha \subseteq s)$，且所有取值为 $False$ 的前提条件均未出现在当前状态 s 中 $(\beta \cap s = \emptyset)$；第 2 行表示动作 a 执行的前提条件没有得到满足时，动作 a 不允许执行，状态 s 维持现状，即状态 s 并未被改变。由此，可以给出状态转移函数 $succ$ 的递归定义：

$$\begin{cases} succ(s, [\,]) = s \\ succ(s, [a_1, a_2, \cdots, a_n]) = succ\left(succ\left(s, a_1\right), [a_2, a_3, \cdots, a_n]\right) \end{cases} \tag{2}$$

进而，设任务的初始状态为 I，目标状态为 $G = \langle N, M\rangle$，则 $P = [a_1, a_2, \cdots, a_n]$

[6]在 PDDL 动作规划语言中，类似条件或命题变量的对象被称为流 (Fluent)。

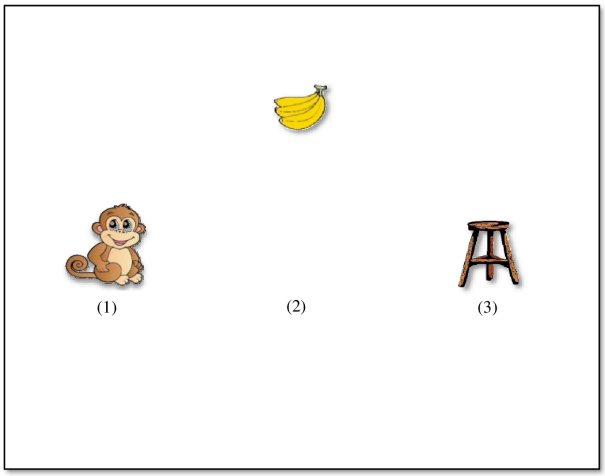


图 2-4: “猴子摘香蕉”问题

是该任务的一个规划，如果 $F = succ(I, [a_1, a_2, \cdots, a_n])$ 满足如下条件：

$$\begin{cases} N \subseteq F \\ M \cap F = \emptyset \end{cases} \tag{3}$$

公式 (3) 的解释与公式 (1) 类似。

一般情况下，需要对基本的 STRIPS 语言进行扩展，以反应实际应用问题中复杂的对象及对象之间的关系。例如，可以使用一阶逻辑取代基本 STRIPS 中使用的命题逻辑。此外，基本的 STRIPS 假定初始状态 I 是完全可观察的。实际上，对于实际问题而言，这一限制过于严厉，并不符合大多数应用实情。因而，也有必要将部分可观察的初始状态引入到基本的 STRIPS 中。

下面给出一个使用 STRIPS 语言描述的规划实例：猴子摘香蕉问题，如图2-4所示。猴子位于位置 (1) 处，香蕉位于位置 (2) 处，凳子位于位置 (3) 处。为了吃到香蕉，猴子需要移动到位置 (3) 处，将凳子搬到位置 (2) 处，然后爬上去，摘下香蕉。猴子“规划”的过程效果示意图如图2-5所示。

下面，给出该问题的 STRIPS 描述。其中 A、B 与 C 分别表示位置 (1)、位置 (2) 与位置 (3)。

```
Initial state: At(A), Level(low), BoxAt(C), BananasAt(B)
Goal state:    Have(bananas)
Actions:
               // move from X to Y
               _Move(X, Y)_
               Preconditions:  At(X), Level(low)
               Postconditions: not At(X), At(Y)

```

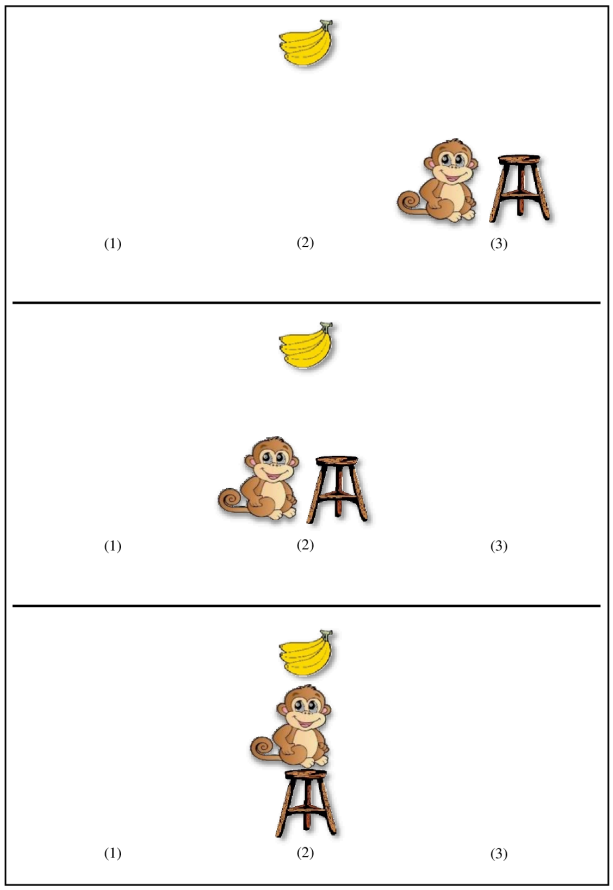


图 2-5: 猴子“规划”的过程效果示意图

```
9                 // climb up on the box
10                _ClimbUp(Location)_
11                Preconditions:  At(Location), BoxAt(Location),
                  Level(low)
12                Postconditions: Level(high), not Level(low)
13
14                // climb down from the box
15                _ClimbDown(Location)_
16                Preconditions:  At(Location), BoxAt(Location),
                  Level(high)
17                Postconditions: Level(low), not Level(high)
18
19                // move monkey and box from X to Y
20                _MoveBox(X, Y)_
21                Preconditions:  At(X), BoxAt(X), Level(low)
22                Postconditions: BoxAt(Y), not BoxAt(X), At(Y), not
                  At(X)
23
24                // take the bananas
25                _TakeBananas(Location)_
26                Preconditions:  At(Location), BananasAt(Location),
                  Level(high)
27                Postconditions: Have(bananas)
```

需要注意的是，在一些文献中，将 STRIPS 动作描述中的后置条件 *Postconditions* 作进一步的区分，表示为 *Add List* 和 *Delete List*。例如，可将上述 $\_Move(X,Y)\_$ 动作描述表示为等价的形式：

```
1  _Move(X, Y)_
2  Preconditions:  At(X), Level(low)
3  Postconditions:
4                  Add List: At(Y)
5                  Delete List: At(X)
```

它表达的含义是，为了生成动作 $\_Move(X,Y)\_$ 执行后的状态描述，首先需要从动作执行前的状态描述中删除掉 *Delete List* 中的所有条件，然后加入 *Add List* 中的所有条件，而在 *Delete List* 中没有提到的所有条件，将延续至动作执行后的状态描述中。这种延续叫做 STRIPS 假设，它是解决框架问题 (Frame Problem) 的一种方法[7]。

在下文，为讨论方便，将会混用上述 2 种表示方法。实际上，只要将 *Postconditions* 中带前缀“*not*”的条件放入 *Delete List* 中，而其余条件放入 *Add List* 中，即可将前者转换为后者。

[7]框架问题指的是，对于逻辑智能体而言，使用传统的一阶逻辑来表示智能体的状态，需要许多额外的公理来表示那些在与环境的交互过程中，不因某个动作执行而发生改变的条件。例如，在下文的积木块世界中，假设桌面上有 A、B 与 C 三个积木块，将 A 放置到 B 上，并不会改变 C 的状态。换句话说，动作的执行只有“局部的”影响，有一些条件或流本身并没有因动作的执行而改变。为了对这些不变的状态执行正确的推理，可以为每个动作和每个条件或流，提供一对所谓的框架公理 (Frame Axiom)，这是比较繁琐的。因此，许多解决框架问题的方法就应运而生了。

2.2 ADL

ADL(Action Description Language) 是一种专为机器人设计的规划语言，由数据抽象与建模专家 Edwin Pednault 受 STRIPS 启发而于 1987 年提出。

与 STRIPS 不同的是，ADL 遵循开放环境 (Open World) 原则，即 ADL 认为未出现的条件其取值应该是未知的，而不是像 STRIPS 那样，将所有未出现的条件缺省处理为 $False$。此外，STRIPS 仅允许条件以正文字 (Positive Literal) 及合取的形式出现，例如 $A \wedge B \wedge C$ 是合法的，而 $\neg(\neg A \vee \neg B \vee \neg C)$ 是不允许出现的。与之不同，ADL 也允许条件以负文字 (Negative Literal) 及析取式的形式出现。

表2-1给出了 STRIPS 与 ADL 这 2 种语言之间的比较。关于 ADL 语言的详细内容，请阅读相关文献。

2.3 PDDL

PDDL(Planning Domain Definition Language) 由 Drew McDermott 及其同事受 STRIPS 和 ADL 语言启发于 1998 年提出，并成为 1998/2000 国际规划竞赛 (International Planning Competition, IPC) 的标准语言，且随着竞赛的举办而不断地得到修订与改进。

PDDL 的目标是,促进规划领域的研究,方便研究者构建基准问题 (Benchmark Problem) 并比较算法的执行效果，试图成为规划领域的标准语言。PDDL 的正式版本包括 1.2、2.1、2.2、3.0 及 3.1，其中 PDDL 3.1 为最新版本。

PDDL 1.2 是 1998 年举办的第 1 届 IPC 竞赛与 2000 年举办的第 2 届 IPC 竞赛的正式语言。它将规划问题分解为 2 个主要模块:领域描述 (Domain Description) 与相关问题描述 (Related Problem Description)，即将规划问题中的共性问题与特性问题进行分离并建模，然后将其提交给领域无关 (Domain-Independent AI Planner)AI 规划器进行处理，最后得到全序或偏序规划 (Totally/Partially ordered Plan)，即动作序列[8]。总体而言，PDDL 1.2 支持确定的、离散的、完全可观察的单智能体规划。

PDDL 2.1 是 2002 年举办的第 3 届 IPC 竞赛的正式语言：引入了数值流 (Numeric Fluent)，允许对非二值型资源建模，例如油量、时间、能量与距离等；引入了规划度量准则 (Plan-Metric)，允许对规划进行定量的评估，即将单一的目标

[8]注意，规划器可以允许动作序列中没有耦合关系且没有顺序约束的动作并行执行。

表 2-1: STRIPS 与 ADL 的比较

STRIPS	ADL
Only positive literals in states: $Poor \wedge Unknown$	Positive and negative literals in states: $\neg Rich \wedge Famous$
Closed World Assumption: Unmentioned literals are false.	Open World Assumption: Unmentioned literals are unknown.
Effect $P \wedge \neg Q$ means add P and delete Q.	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q.
Only ground literals in goals: $Rich \wedge Famous$	Quantified variables in goals: $\exists x At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals are conjunctions: $Rich \wedge Famous$	Goals allow conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart)$
Effects are conjunctions.	Conditional effects allowed: $when\ P : E$ means E is an effect only if P is satisfied.
No support for equality.	Equality predicate $(x = y)$ is built in.
No support for types.	Variables can have types, as in $(p : Plane)$.

驱动规划升级为效用驱动的规划，允许优化规划；引入了连续动作。因此，PDDL 2.1 适用于许多实际规划问题。

PDDL 2.2 是 2004 年举办的第 4 届 IPC Deterministic Track 竞赛的正式语言：引入了派生谓词 (Derived Predicate)，例如它可以建模传递性——$A = ReachFrom(B)$，$B = ReachFrom(C)$，则 $A = ReachFrom(C)$；引入了定时初始化文字 (Timed Initial Literal)，例如建模独立于规划执行的外部事件。与 PDDL 2.1 的升级相比，PDDl 2.2 并没有取得重大进展。

PDDL 3.0 是 2006 年举办的第 5 届 IPC Deterministic Track 竞赛的正式语言：引入了状态轨迹约束 (State-Trajectory Constraint) 与偏好 (Preference)，前者属于必须遵守的硬约束 (Hard Constraint)，后者属于非必须遵守的软约束 (Soft Constraint)。此外，PDDL 3.0 融入了规划领域的最新重要的进展，提升了语言的表达能力。

PDDL 3.1 是 2008 年与 2011 年举办的第 6 届与第 7 届 IPC Deterministic Track 竞赛的正式语言：引入了对象流 (Object Fluent)，使得 PDDL 能够处理任意的对象类型，而不仅仅是数值流，极大地提升了语义表达能力。

下面，通过一个实例，对 PDDL 的主要语法进行介绍。首先，介绍 PDDL 的构成成分或组件：

1. 对象 (Object)

   环境中智能体所感兴趣的个体。

2. 谓词 (Predicate)

   用于描述对象的属性及对象之间的某种关系。

3. 初始状态 (Initial State)

   环境的起始状态。

4. 目标 (Goal)

   环境的终止状态。

5. 动作或算子 (Action/Operator)

   用于将环境从一种状态，改变为另一种状态。

从文件的角度看，PDDL 规划任务被分成 2 个文件：

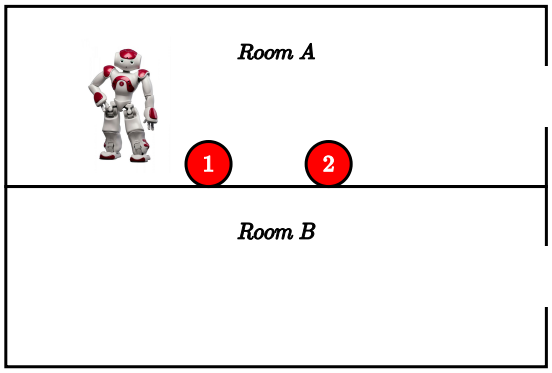


图 2-6: 双臂机器人抓取的规划问题

- domain 文件

  用于定义谓词与动作；

- problem 文件

  用于定义对象、初始状态与目标；

现在，假设有 2 个房间 A 与 B，在 A 房间中有 2 个球，其编号分别为 1 和 2。同时，在 A 房间有一个双臂抓取机器人 Robby，要求 Robby 将球 1 抓取，并运送至 B 房间，如图2-6所示。

这是一个典型的规划任务，可以使用 PDDL 定义双臂抓取机器人的规划任务。下面的 domain 文件与 problem 文件分别给出了该规划任务的完整描述。

文件“gripper2_domain.pddl”，定义了自动规划所需要的要素：

```
(define (domain gripper-strips)
  (:predicates (room ?r) (ball ?b) (gripper ?g) (at-robby ?r)
               (at ?b ?r) (free ?g) (carry ?o ?g))
  (:action move
   :parameters (?from ?to)
   :precondition (and (room ?from)
                      (room ?to)
                      (at-robby ?from))
   :effect (and (at-robby ?to)
                (not (at-robby ?from))))
  (:action pick
   :parameters (?obj ?room ?gripper)
   :precondition (and (ball ?obj)
                      (room ?room)
                      (gripper ?gripper)
                      (at ?obj ?room)
                      (at-robby ?room)
                      (free ?gripper))
   :effect (and (carry ?obj ?gripper)
                (not (at ?obj ?room))
```

```
                (not (free ?gripper))))
  (:action drop
   :parameters (?obj ?room ?gripper)
   :precondition (and (ball ?obj)
                      (room ?room)
                      (gripper ?gripper)
                      (carry ?obj ?gripper)
                      (at-robby ?room))
   :effect (and (at ?obj ?room)
                (free ?gripper)
                (not (carry ?obj ?gripper)))))
```

文件“gripper2_problem.pddl”，给出了环境的具体配置：

```
(define (problem strips-gripper2)
    (:domain gripper-strips)
    (:objects rooma roomb ball1 ball2 left right)
    (:init (room rooma)
           (room roomb)
           (ball ball1)
           (ball ball2)
           (gripper left)
           (gripper right)
           (at-robby rooma)
           (free left)
           (free right)
           (at ball1 rooma)
           (at ball2 rooma))
    (:goal (at ball1 roomb)))
```

使用上述 2 个文件，利用 Fast Downward 规划器，将得到如下的规划结果[9]：

```
(pick ball1 rooma left)
(move rooma roomb)
(drop ball1 roomb left)
; cost = 3 (unit cost)
```

该规划问题的动作序列示意图分别如图2-7、2-8、2-9与2-10所示。

下面，结合上述 2 个文件，给出 PDDL 语法的相关解释：

1. *domain* 文件的第 1 行，定义了 *domain* 的名字“$gripper-strips$”；

2. *problem* 文件的第 1 行，定义了 *problem* 的名字“$strips-gripper2$”。第 2 行，指定了 *domain* 的名字“$gripper-strips$”，必须与 *domain* 文件中定义的 *domain* 名字相匹配；

3. *problem* 文件的第 3 行，定义了环境中的对象：房间 *rooma* 和 *roomb*、球 *ball*1 和 *ball*2、机器人的 2 个抓取手臂 $left$ 和 *right*；

[9]关于 Fast Downward 规划器的安装与使用方法，请阅读附录 [5.1]。

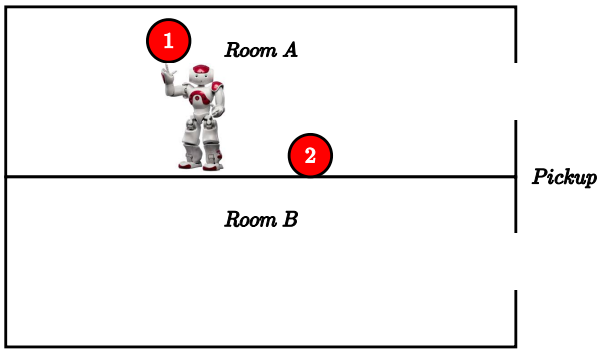


图 2-7: Robot 执行动作 (pick ball1 rooma left)

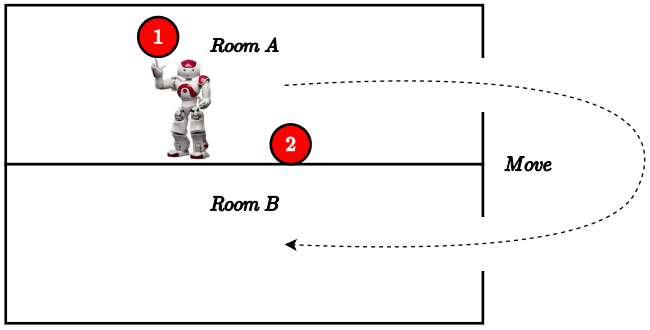


图 2-8: Robot 执行动作 (move rooma roomb)

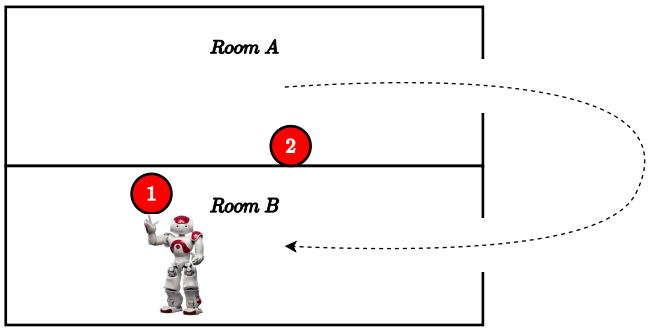


图 2-9: Robot 执行动作 (move rooma roomb)

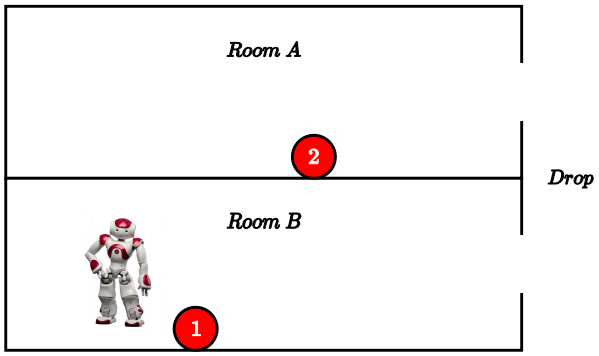


图 2-10: Robot 执行动作 (drop ball1 roomb left)

4. *domain* 文件的第 2-3 行，定义了环境中对象的属性及其关系算子。例如，“$room\ ?r$”表示对象 r 是否为房间，返回 $True$ 或 $False$；“$at-robby\ ?r$”表示机器人 $Robby$ 是否在房间 r 中；“$free\ ?g$”表示机器人手臂 g 是否空闲；其余谓词的解释也是较为直观的；

5. *domain* 文件的剩余部分，定义了动作 *move*、*pick*、*drop*，分别指定了各自所需的参数、动作应用前提条件与执行效果；

6. *problem* 文件的第 4-14 行，定义了环境的初始状态，没有出现的状态缺省为 $False$；

7. *problem* 文件的第 15 行，定义了环境的目标状态，没有出现的状态其取值可为任意值，即对智能体而言，它们的取值并不重要；

3 规划算法

3.1 算法分类

总体上而言，规划器既能被视为解求解的搜索程序，也能被视为解存在的证明程序。常见的规划方法包括：

- 基于状态空间搜索的规划

  它包括前向搜索、后向搜索以及结合启发式信息的搜索等。FF(FastForward) 就是一个著名的前向启发式搜索算法。

- 规划图 (Planning Graph)

  既可以利用规划图给出更好的启发式估计信息，从而将其应用于任何启发式搜索算法之中，也能够在规划图自身形成的空间中，使用 GraphPlan 算法直接搜索解。此外，规划图能够生成层次的有向图。

  规划图已被证明是求解困难规划问题的有效方法。

- 转换为布尔满足性 SAT 的规划

  SAT 既能够证明命题逻辑中的命题，也能够求解 PDDL 等语言描述的 (使用命题逻辑表达的) 规划问题。

- 转换为一阶逻辑推理的规划：情景演算 (Situation Calculus)

  PDDL 是一种在语言表达能力与算法复杂度之间取得某种平衡的规划语言。对于一些难以使用 PDDL 描述的规划问题，可以利用一阶逻辑中的全称量词来抽象简洁地表达。例如，对于“将所有货物从 A 移动到 B”这一命题，就非常适合一阶逻辑来表达。此外，一阶逻辑也能够精确地表达全局约束关系。例如，一阶逻辑能够很好地表达“不超过四个机器人可以同时出现在该地点”这一命题，而 PDDL 只能在每个可能涉及移动的动作中，需要使用重复的前置条件才能表达该命题。

  基于一阶逻辑的情景演算可以绕开这些缺陷。在该领域，当前的大量工作主要集中于规划的形式语义及开辟新的研究领域。但是，到目前为止，并没有出现使用情景演算的大规模的实际规划系统。部分原因是，一阶逻辑的高效推理存在困难，且没有为规划算法开发出有效的基于情景演算的启发式。

- 转换为约束满足问题的规划

  可以将限界规划问题转换为约束满足问题，而约束满足问题本身与布尔满足 SAT 问题之间也存在着紧密的联系。因此，对三者之间关系的研究，也将是十分自然的。

- 转换为改良的偏序规划的规划

  一般的规划方法，将求解出一个规划，即动作序列，而该动作序列是一个严格的线性序列。然而，在一个规划问题中，一些动作及其子目标或子任务之

间是相互独立的，即这些子任务之间并不存在严格的时序约束关系——它们可以并行地执行。

于是，可以将规划表示为偏序结构 (Partial Order)：为动作引入约束集合。该约束用来表示动作发生的时序关系。此外，偏序规划在规划空间上进行解的搜索，而不是在状态空间上进行搜索。

从某种角度而言，偏序规划利用规划问题本身具有的子任务独立性进行有效的规划，是分而治之算法的一种典型应用。

- 规划-调度的结合

  在许多实际应用问题中，规划与调度是 2 个紧密关联的阶段，一般遵循“先规划，后调度”的原则。例如，经典规划的任务是确定“以何种顺序执行何种任务 (动作)”，此阶段并不会与具体的时间相关联，例如动作何时发生以及持续多久等。实际上，与具体时间及资源相结合，是调度的主要任务。具体而言，在规划阶段，选择动作，并考虑时序约束，以满足问题的目标；在调度阶段，将具体的时间信息 (例如子任务的起点及持续时间等) 及资源同规划结合，以满足时间与资源约束。规划与调度的结合，也衍生出了许多算法。

- 层次规划 (Hierarchical Planning)

  层次化分解是处理复杂性系统的一种强有力的普适方法。在规划领域，层次任务网络 (Hierarchical Task Network, HTN) 就是这一思维方法的具体体现，并且它经常与偏序规划结合在一起。此外，HTN 也允许多个不同的高层动作 (High-Level Action) 共享并细化较低层的动作，以最大限度地获得层次结构所带来的好处。

- 其它非经典环境下的规划

  许多实际问题，需要将经典规划环境扩展为部分可观察的、非确定性的、未知的环境，并由此发展出了许多规划算法。

- 多智能体规划

  在多智能体环境下，多智能体之间可能同时存在合作/竞争的关系，它们之间以及它们与环境之间都存在一定程度的交互作用，情形异常复杂。大体上，

多智能体的规划可以粗略地分为 2 类：多个同步动作的表示和规划；多个智能体的合作与协调规划。

3.2 基于状态空间搜索的规划

最直接的规划算法是使用状态空间搜索方法，即直接依据规划问题中动作的描述——前置条件与后置条件，或者从初始状态开始，执行前向搜索 (Forward Search)，或者从目标状态开始，执行后向搜索 (Backward Search)。此外，我们也能够从显示的动作与目标表示中，自动地获取有效的启发式信息，以提高规划效率。

3.2.1 前向搜索

为方便，下面将以积木块世界 (Blocks World) 为例，讨论算法。在积木块世界中，对象包括若干方块、机器人抓取臂及一张桌子。这些方块可以垂直堆叠，但一个方块的上面只能放一个方块。机器人抓取臂每次只能抓取一个方块，且可以移动到指定位置，并将方块放置在桌子上或另一个方块上。

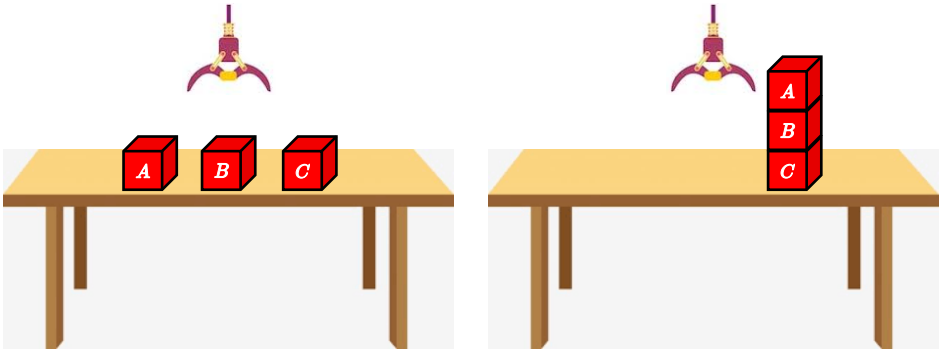
该问题的一个 STRIPS 描述如下：

```
Initial state: On(A, Table), On(B, Table), On(C, Table), Block(A),
               Block(B), Block(C), Clear(A), Clear(B), Clear(C)
Goal state:    On(A, B), On(B, C)
Actions:
               Move(b, x, y)
               Preconditions: On(b, x), Clear(b), Clear(y), Block(b),
                              (b != x), (b != y), (x != y)
               Postconditions: On(b, y), Clear(x), not On(b, x),
                               not Clear(y)

               MoveToTable(b, x)
               Preconditions: On(b, x), Clear(b), Block(b), (b != x)
               Postconditions: On(b, Table), Clear(x), not On(b, x)
```

其中，谓词 $On(b,x)$ 表示积木块 b 在 x 的上面；谓词 $Block(x)$ 表示 x 是否为积木块，$True$ 表示“是”，$False$ 表示“否”；谓词 $Clear(x)$ 表示 x 的上面是否可以放置方块，$True$ 表示“可以”，$False$ 表示“不可以”[10]；动作 $Move(b,x,y)$ 表示

[10] $Clear(x)$ 的这种语义解释，既适用于 x 为方块的情形，也适用于 x 为桌面的情形，不会引起逻辑上的矛盾；否则，如果将 $Clear(x)$ 解释为“x 的上面为空”，则对于 x 为桌面的情形，就会出现矛盾——将一个方块从桌面移到另一个方块的上面后，桌面上肯定不为空。因此，在本例中，$Clear(Table)$ 的取值始终为 $True$。



(a) 初始状态 (b) 目标状态

图 3-11: 一个积木块世界实例

将 x 上面的积木块 b 放到 y 的上面；动作 $MoveToTable(b,x)$ 表示将 x 上面的积木块 b 放到桌面上。

上述 STRIPS 所描述的场景示意图如图3-11所示。

下面，以动作 $Move(b,x,y)$ 为例，研究一下动作执行前后，状态的变化情况。为方便，以 *Add List* 与 *Delete List* 表示方式，重新列出动作 $Move(b,x,y)$ 的 STRIPS 描述：

```
Move(b, x, y)
Preconditions: On(b, x), Clear(b), Clear(y), Block(b), (b != x), (b
!= y), (x != y)
Postconditions:
                Add List: On(b, y), Clear(x)
                Delete List: On(b, x), Clear(y)
```

图3-12展示了执行动作 $Move(B,Table,C)$ 前后，状态的变化情况。在该图中，使用虚线框标出了动作执行后需要从状态描述中删除与添加的条件，那些没有标出的条件，维持“Unchanged”状态，继续保留在状态描述中。可以看出，环境处于状态 s_0 时，执行动作 $Move(B,Table,C)$ 之后，状态将转变为 s_1。

因此，只要我们从初始状态出发，执行合适的动作序列，最终将到达我们所希望的目标状态。于是，这就给出了前向搜索算法：从初始状态出发，以当前状态下所有能够执行的动作作为执行分支，生成新状态；然后，检测新状态是否为目标状态：是，表明找到一条从根节点到目标节点的路径，路径上的动作构成了动作序列，即得到了一个规划；否，则重复上述过程，直至到达目标，得到一个规划，或者搜索完毕，无法找到合适的动作序列。

图3-13展示了上述积木块世界前向搜索的一个示意图。

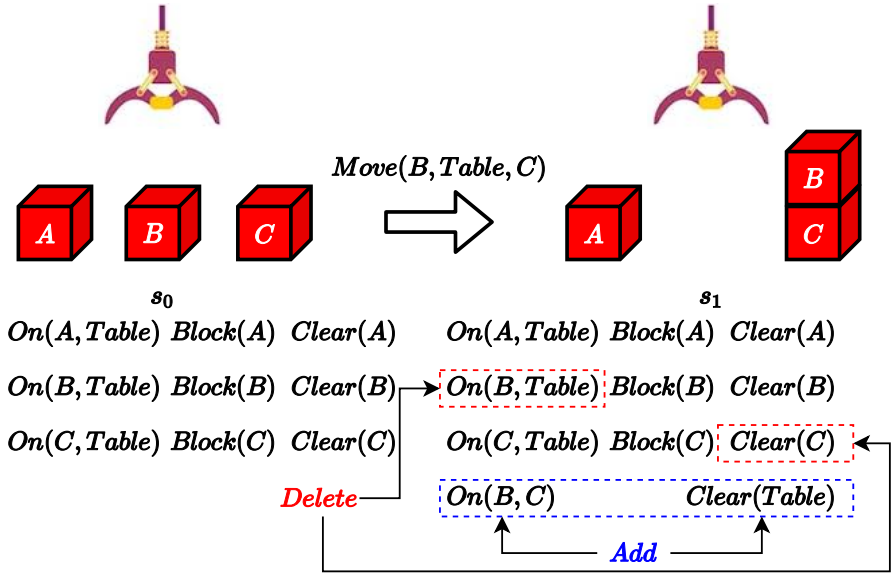


图 3-12: Robot 执行动作 $Move(B, Table, C)$

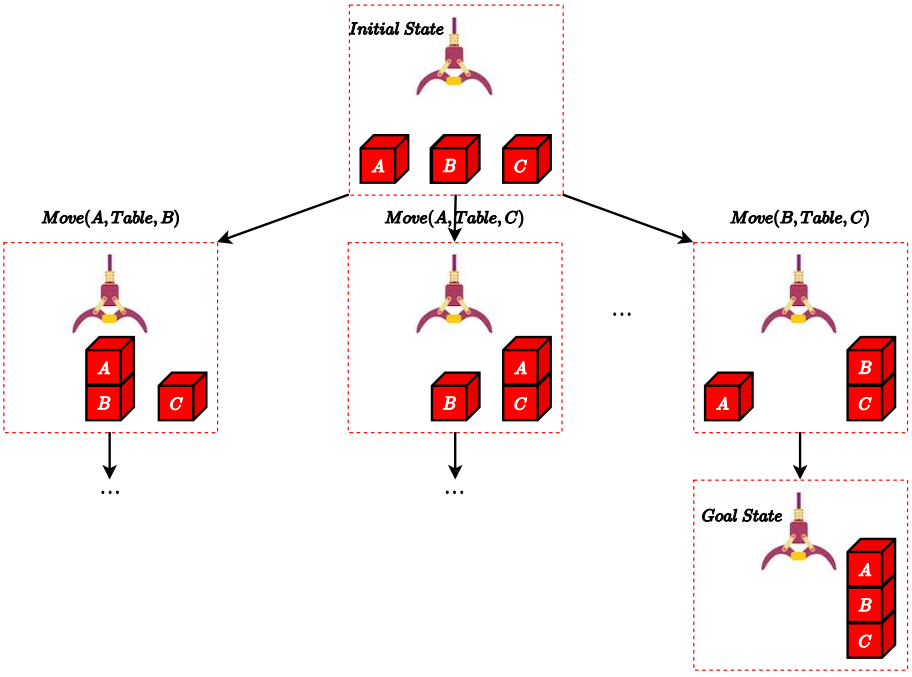


图 3-13: 积木块世界的前向搜索示意图

显然，我们已经将一个规划问题转换为一个状态空间上的搜索问题，或者我们基于状态空间上的搜索来执行规划任务。因此，可以将前面章节中的搜索技术应用于前向搜索算法中。例如，利用规划问题的结构化表示的特点，获取启发式信息，并执行 A* 搜索算法，从而快速地得到一个规划。

下面，给出前向搜索算法。

算法 **3.1**（*Forward Search* 算法）

Input: set of operators O, initial state s, goal state g
Output: Plan P
def ForwardSearch(O, s, g):

1. let $P = [\,]$
2. while $g \not\subseteq s$:
3. let $E = \{a | a$ is ground instance of an operator in O, and $Preconditions(a)$ hold in $s\}$
4. if $E == \{\}$:
5. return $Failure$
6. nondeterministically choose an action $a \in E$
7. let $s = s \backslash DeleteList(a) \cup AddList(a)$
8. $P = P@[a]$
9. return P

注意，在上述伪代码中，“ground instance”指的是，在动作执行前，动作描述中的所有参数变量必须实例化为具体的对象，即动作能够针对具体的对象执行某种操作；“nondeterministically choose an action $a \in E$”指的是，以非确定性的方式从 E 中选择合适的动作 a 来执行，一般包括如下情况[11]：

- 当前的动作不能到达目标状态时，能够执行回溯 (Backtracking) 操作，以选取潜在的可能导向目标状态的动作来执行；
- 能够使用启发式信息选取当前最适合的动作来执行[12]。

对前向搜索算法稍加分析，易知，该算法是健全的或可靠的 (Sound)，即对任一由非确定性的动作选取轨迹而返回的规划，一定是问题的解；该算法也是完备的

[11]还有一种理解方式，即“Try all actions in parallel”。当然，这种理解方式，并不有助于代码的实现。一种可行的方法是，使用深度优先搜索的方式，以确定性的方式逐一地从 E 中选取动作并执行，且在必要时，进行回溯。或者，以 A* 搜索算法作为基本框架，将启发式信息作为动作的优先级——优先级高的动作，优先得到执行，以提高算法的搜索效率。

[12]例如，在 A* 算法中，使用启发式信息寻找快速导向目标状态的路径，以提高搜索效率。这些启发式信息能够对 E 中的动作进行优先级排序。那些较有希望的动作，优先级较高，较先得到执行。

(Complete)，只要问题的解存在，则由非确定性的动作选取轨迹而形成的路径，一定能返回一个可行解。

3.2.2 后向搜索

在前向搜索中，设当前状态为 s，如果动作 a 的前置条件得到满足，则动作 a 可以得到执行，那么新状态 s' 可表示为 $s\backslash DeleteList(a) \cup AddList(a)$。

在后向搜索中，需要从目标状态 g 出发进行反向搜索，直至初始状态为止。在这种情况下，如何表示状态的演化关系呢？

设目标状态 g 的前一个状态为 g'，且假设在 g' 下执行了动作 a，便得到了目标 g，则这 2 个状态之间存在如下关系：$g = g'\backslash DeleteList(a) \cup AddList(a)$。显然，$AddList(a) \subseteq g$ 且 $DeleteList(a) \cap g = \emptyset$ 是动作 a 的必要条件。因此，可将上述条件作为反向搜索时选取动作的依据。

另一方面，从 g' 的角度看，$Preconditions(a) \subseteq g'$ 是动作 a 得以执行的必要条件。因此，为了从 g 得到 g'，可以利用公式 $g' = g\backslash AddList(a) \cup Preconditions(a)$，该公式可以看作公式 $g = g'\backslash DeleteList(a) \cup AddList(a)$ 的“逆操作”，其中 $DeleteList(a) \subseteq Preconditions(a)$。实际上，我们可以将 g' 看作新的目标状态。因而，可以利用上述递推公式，建立后向搜索算法。

下面，给出后向搜索算法。

算法 **3.2**（*Backward Search* 算法）

Input: set of operators O, initial state s, goal state g
Output: Plan P
def BackwardSearch(O, s, g):
1. let $P = [\,]$
2. while $s \not\subseteq g$:
3. let $E = \{a | a$ is ground instance of an operator in O, $AddList(a) \subseteq g$, and $DeleteList(a) \cap g = \emptyset\}$
4. if $E == \{\}$:
5. return $Failure$
6. nondeterministically choose an action $a \in E$
7. let $g = g\backslash AddList(a) \cup Preconditions(a)$
8. $P = a :: P$
9. return P

对图3-12应用逆操作，可以很容易地验证反向操作是成立的。与图3-13类似，我们也能够从目标状态出发，得到一棵后向搜索树。

前向搜索算法的分支因子可能非常高，即当前状态下存在大量的、可应用的动作，但许多动作并不导向目标状态。因此，在执行前向搜索时，大量的时间将耗费在许多不相关的动作分支上。一般认为，后向搜索算法直接从目标状态出发，有助于减少分支因子。但是，实际上，后向搜索算法也存在分支因子过高的问题。例如，假设一个动作算子 o 满足当前目标状态下的可应用条件，但是算子 o 有许多的基元实例 (Ground Instance)a_1、a_2、$\cdots$、a_n，但是这些基元实例所在的状态与初始状态之间并没有可行的路径。因此，在执行后向搜索时，大量的时间也将耗费在许多不相关的动作分支上。

在实际应用中，研究者们发现，能够有效利用启发式信息的前向搜索算法，其表现往往要优于后向搜索算法。一个使用有效启发式的前向搜索规划器是 FF，即 FastForward，由 Hoffmann 于 2005 年提出。该规划器利用松弛技术获取启发式信息——通过忽略 $DeleteList(a)$，并在规划图 (Planning Graph) 的帮助下获得有用的启发式信息。此外，FF 还巧妙地使用了多种搜索技术，例如爬山法、迭代加深搜索等，提高搜索效率。

4 练习

1. 使用 Fast Downward 规划器完成讲义中的 Gripper2 规划实验；

2. 查找相关文献，使用 Fast Downward 规划器完成 8 数码规划实验；

5 附录

5.1 Fast Downward 规划器

Fast Downward 是一个基于启发式搜索的经典规划器，支持 PDDL 2.2 语言，可以解决常见的确定性规划问题。与其它流行的规划器 (例如 HSP 和 FF) 一样，Fast Downward 属于前向规划器，即以前向 (Forward Direction) 的方式搜索状态空间。然而，与其它 PDDL 规划器不一样，Fast Downward 并不直接使用命题的 PDDL 表示，而是先将表示转换为所谓的多值规划任务 (Multivalued Planning Task) 表示，这使得命题规划任务中隐含的约束表示能够显式化；然后，Fast Downward 再使用规划任务的层次分解 (Hierarchical Decomposition) 方法，计算

启发式函数[13]。

在 Windows 下，需要安装“Visual Studio>=2017”、“Python >=3.6”、“Git”以及“CMake”软件。然后，依次执行如下步骤：

1. 打开 CMD 窗口,获取源代码:git clone https://github.com/aibasel/downward.git downward；
2. 执行命令“C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/VC/Auxiliary/Build/vcvarsall.bat x64”，配置 build 所需环境；
3. cd downward；
4. python build.py

   注意，需要确保 Python3.6 在系统路径“PATH”上。此步骤将编译链接生成 Fast Downward 的 release 版本，生成的 downward.exe 在目录“downward/builds/release /bin”下；
5. 运行“python fast-downward.py gripper2_domain.pddl gripper2_problem.pddl –heuristic ”h=ff()” –search ”astar(lmcut())””

   需要按第2.3节的内容,准备文件“gripper2_domain.pddl”与文件“gripper2_problem.pddl”，放置于“fast-downward.py”所在的目录；
6. 如果配置顺利，将在当前目录下，生成一个规划求解文件“sas_plan”，它的每一行展示了执行动作，所有行就构成了一个动作序列，即一个规划 (Plan)。内容如下：

```
(pick ball1 rooma left)
(move rooma roomb)
(drop ball1 roomb left)
; cost = 3 (unit cost)
```

6 参考文献

1. Stuart Russell, Peter Norvig. Chapter11 Planning. http://aima.cs.berkeley.edu/newchap11.pdf. Artificial Intelligence: A Modern Approach, Fourth edition,

[13]这被称为因果图启发式 (Causal Graph Heuristic)。这种方式与传统的 HSP 启发式完全不同，因为传统方式通过忽略算子的负交互 (Negative Interactions of Operators) 来计算启发式信息。

2020. http://aima.cs.berkeley.edu/index.html.

2. Stuart J. Russell, Peter Norvig, 殷建平等译。《人工智能：一种现代的方法》，第 3 版，清华大学出版社。

3. Nils J. Nilsson. Artificial Intelligence: A New Synthesis. Morgan Kaufmann, 1997.

4. Thomas Dean, James Allen, Yiannis Aloimonos. 顾国昌，刘海波，仲宇等译。《人工智能——理论与实践》，电子工业出版社，2004 年 6 月。

5. 王洪源，陈慕羿，华宇宁，石征锦。《Unity3D 人工智能编程精粹》，清华大学出版社，2014 年 11 月第 1 版。

6. Malik Ghallab, Dana Nau, Paolo Traverso. Automated Planning: Theory and Practice, Morgann Kaufmann, 2004.

7. Malik Ghallab, Dana Nau, Paolo Traverso. Automated Planning and Acting, Cambridge University Press, 2016.

8. STRIPS. https://en.wikipedia.org/wiki/Stanford_Research_Institute_Problem_Solver.

9. Automated planning and scheduling. https://en.wikipedia.org/wiki/Automated_planning_and_scheduling.

10. Planning Domain Definition Language. https://en.wikipedia.org/wiki/Planning_Domain_Definition_Language.

11. Action description language. https://en.wikipedia.org/wiki/Action_description_language.

12. An Introduction to PDDL. https://www.cs.toronto.edu/ sheila/2542/s14/A1/introtopddl2.pdf.

13. The Fast Downward Planning System. https://planning.wiki/ref/planners/fd.

14. Obtaining and running Fast Downward. http://www.fast-downward.org/ObtainingAndRunningFastDownward.

15. Archive: Fall Semester 2016 for Planning and Optimization. https://ai.dmi.unibas.ch/archive/hs2016/planning-and-optimization/index.html.

16. Planning: STRIPS and POP planners. https://people.cs.pitt.edu/ milos/courses/cs1571-Fall2010/Lectures/Class18.pdf.

17. Frame Problem. https://en.wikipedia.org/wiki/Frame_problem.

18. STRIPS Planning. https://www.cs.bham.ac.uk/~vxs/teaching/ai/slides/lecture6-slides.pdf.

19. Chapter 4 State-Space Planning(PPT). https://www.cs.umd.edu/ nau/planning/slides/chapter04.pdf. Lecture Slides for Automated Planning: Theory and Practice. https://www.cs.umd.edu/ nau/planning/slides/.