第3章 逻辑思维成为自动化

CHAPTER 3

3.1 逻辑思维意味着什么

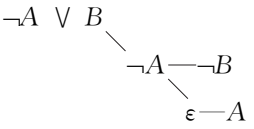
在人工智能研究的第一阶段，对一般问题解决方法的探索至少在形式逻辑上是成功的。我们指定了一个机械化的程序，用来确定公式的逻辑真实性。这个程序也可以由计算机执行，这种执行在计算机科学中引入了自动证明的方法。

自动证明的基本思想很容易理解。在代数中，加法(+)或减法(−)等算术运算中要使用 x，y，z 等字母。这些字母用作插入数字的空格(变量)。在形式逻辑中，命题由变量 A、B、C 等表示，这些变量通过逻辑联结词联结起来，比如：与($\wedge$)、或($\vee$)、如果-就($\rightarrow$)、非($\neg$)。这些命题变量充当空格，使用其值或真或假的陈述句。例如，当 A 为一个逻辑值为真的命题 $1+3=4$，B 为逻辑值为真的命题 $4=2+2$ 时，逻辑公式 $A \wedge B$ 就变成了真命题：$1+3=4 \wedge 4=2+2$。而在逻辑上，这就得到了正确的推论：$1+3=4 \wedge 4=2+2 \rightarrow 1+3=2+2$。但是通常来说，像 $A \wedge B \rightarrow C$ 这样的推论是不正确的。不过，像 $A \wedge B \rightarrow A$ 这样的推论在逻辑上是成立的，因为 $A \wedge B$ 与 A 的真值相同。

要证明一个逻辑推论的一般有效性在实践中可能非常复杂。因此，1965 年罗宾逊(J. A. Robinson)提出了一种所谓的解决方法，根据这种方法，可以通过逻辑上的证伪过程(反证法)来找到证明方法。从相反的假设(否定)开始，即假设这个逻辑结论不成立，下一步就会发现这个假设的所有可能的应用实例都会导向一个自相矛盾的结果。因此，否定的否定是肯定的，原逻辑结论被证明是成立的。罗宾逊的解决方法使用了逻辑简化方法，根据这种简化方法，任何逻辑公式都可以转换为所谓的合取范式。在命题逻辑中，一个合取范式由否定和非否定的命题变量(字母)组成，它们通过与($\wedge$)、或($\vee$)符号联结起来。

【举例】 对于合取范式$(\neg A \vee B) \wedge \neg B \wedge A$，该式由$\neg A \vee B$、$\neg B$、$A$ 三个子句组成，它们通过 $\wedge$ 联结起来。在这个例子里，字面意义上的$\neg A$ 逻辑上可以由$\neg A \vee B$ 和$\neg B$ 导出。原因很简单：$B \wedge \neg B$ 这样的联结对于 B 的每个应用实例而言总是假的，而$\neg A$ 逻辑上可以由$\neg A \wedge \neg B$ 导出。从$\neg A$ 和剩下的子句 A，在下一步中，紧接着的是总为假值的公式$\neg A \wedge A$

以及像这样的矛盾体 ε(“空词”)。



机械地看,这个过程包括从一个逻辑公式的联结元素中删除自相矛盾的部分命题(“解决方案”),然后用得到的“预解式”和该公式的另一个相应的联结元素重复这个过程,直到可以导出一个矛盾结果(“空词”)。

在相应的计算机程序中,因为是命题逻辑,这个过程是可终止的。因此,它在有限时间内判断了所提出的逻辑公式是否普遍有效。然而,根据先前已知的方法,这个计算时间随着一个公式中的字母数量呈指数级增长。考虑到“人工智能”,至少在命题逻辑中,原则上说,采用解析法的计算机程序至少可以自动判定逻辑结论的一般有效性。与计算机相比,要跟踪复杂和冗长的推论,人类的手工计算难度很大,并且速度要慢得多。随着计算能力的增加,机器可以更加有效地完成逻辑推演的任务。

在谓词逻辑中,一个命题被分解为属性(谓词),对象被分配给这个属性或被否定。因此,在命题 $P(a)$ 中,例如“安妮是个学生(Anne is a student)”,谓词“学生 student”(P)被指定给一个名为“安妮 Anne”(a)的个体。这个命题或者真,或者假。在状态 $P(x)$ 的谓词形式中,空格(单个变量)x、y、z 等用于假定应用领域的 a、b、c 等个体(例如,一所学校的学生)。除了谓词逻辑中的逻辑联结词,量词也可以被应用进来,例如 $\forall xP(x) \rightarrow \exists xP(x)$ 是谓词逻辑的一个普遍有效的推论。

对于谓词逻辑的公式,也可以设置一个通用的分解过程,以便从这个公式的一般无效性的假设中推导出一个矛盾。为此,谓词逻辑的公式必须转化为一种规范形式,从中可以机械地推导出一个矛盾。然而,由于在谓词逻辑(与命题逻辑相反)中,公式的一般有效性通常无法确定,因此可能出现解析过程不能结束的情况,然后导致计算机程序无限地运行。所以重要的是找到一些子类,在这些子类中计算过程不仅能有效地终止,而且能完全终止。机器智能确实可以提高决策过程的效率并加速它们;然而,它也与人类的智力类似,受到逻辑可判定性的原则性限制。

3.2　人工智能编程语言 PROLOG

要通过计算机解决一个问题,就必须将这个问题转换成一种编程语言。FORTRAN 语言就是一种最早的编程语言,它的一个程序由一系列发送给计算机的命令组成,如“跳转到程序中的 z 位置”“将值 a 写入变量 x”。这种编程语言的重点是变量,即存储和处理输入值的寄存器或存储单元。由于通过输入命令来执行,这种语言也被称为命令式编程语言。

另一方面,在谓词编程语言中,编程被理解为在由事实构成的系统中的证明过程。这种知识表示方法在逻辑学中普遍为人所知,相应的编程语言称为“逻辑编程”(即 PROLOG),

自 20 世纪 70 年代初开始被使用，它的基础是谓词逻辑，这已经在第 3.1 节中介绍了。知识在谓词逻辑中表示为一组值为真的陈述句，知识处理是人工智能研究的核心，因此 PROLOG 是一种重要的人工智能编程语言。

这里将介绍 PROLOG 的一些模块，以阐明它与知识处理的联系。逻辑语句“对象 O_1，…，O_n 之间存在一种关系 R”对应于一个事实，在谓词逻辑中，它被赋予了一般形式 $R(O_1,\cdots,O_n)$。在 PROLOG 中写作：

```
NAME(O1, …, On),
```

其中，*NAME* 是任何一个关于关系的名称，以事实的语法形式表示的字符串被称为变量。

关于事实或变量的一个例子是：

```
married (Socrates, xantippe),
married (abélard, eloise),
is a teacher (Socrates, Plato),
is a teacher (abélard, eloise).
```

现在关于给定事实的陈述和证明可以被引入问答系统。问题用问号标记，其答案用星号标记：

```
? married (Socrates, xantippe),
 * yes,
? is a teacher (Socrates, xantippe),
 * no.
```

在这种情况下，问题还可以专门指代使用变量的对象。编程语言为此使用描述性名称，例如 Man 表示任何人，Teacher 表示任何教师：

```
? married (Man, xantippe),
 * Man = Socrates,
? is a teacher (Teacher, xantippe),
 * Teacher = Socrates.
```

一般来说，PROLOG 中的一个问题是“L_1，L_2…，L_n 都成立吗?”，或者简言之：

```
?L1, L2, …, Ln
```

其中，L_1，L_2，…，L_n 是变量。对于逻辑推论规则，如直接推论(modus ponens)就是：“如果 L_1，L_2，…，L_n 是真的，那么 L 也是真的”，或者简言之：

```
L: - L1, L2,…, Ln
```

【举例】 一个规则的引入方式如下：

```
is a pupil (pupil, teacher): - is a teacher (teacher, pupil)
```

然后，根据给定的事实，求出：

```
? is a pupil (Student, Socrates),
 * Student = platon
```

PROLOG 基于一个给定的变量形式的知识库，使用解析方法可以找到一个问题的解决方案。

3.3　人工智能编程语言 LISP

除了用陈述性的谓词及其相互关系表示以外，知识也可以用函数和分类表示，就像数学中使用的函数和分类。因此，函数式程序设计语言不把程序看作事实和推理的系统（如PROLOG），而是把程序看作输出值集上的输入值集的函数。谓词编程语言涉及谓词逻辑，而函数式编程语言基于阿隆佐·邱奇（A. Church）在 1932/1933 年定义的用于计算规则的函数形式化的 λ-calculus 公式。LISP 是函数式编程语言的典例，早在 20 世纪 50 年代末由麦卡锡（J. McCarthy）在人工智能的第一个发展阶段开发出来。因此，它是历史最悠久的编程语言之一，诞生之初就与人工智能的目标相联系，把人类的知识处理方法应用到计算机上。知识以数据结构的方式表示，知识处理以作为有效函数的算法表示。

符号的线性列表在“列表处理语言”（LISP）中被用作数据结构。LISP 最小的（不可分割的）构建模块被称为原子，它可以是数字、数字序列或名称。在算术中，自然数是通过计数产生的，从作为“原子”的“1”开始，然后在前一个数 n 的基础上加 1 生成后续的 $n+1$。因此，所有自然数的算术属性都是递归性地定义的：首先为“1”定义一个属性；递归过程中，在为任意一个数 n 定义某种性质的前提下，也为后续的 $n+1$ 定义了这种属性。递归定义可以推广到任何一个有限的符号序列。因此，s 表达式（“s”代表“符号”）是由作为 LISP 对象的原子递归形成的：

（1）一个原子是一个 s 表达式。

（2）如果 x 和 y 是 s 表达式，那么 $(x.y)$ 也是。

s 表达式的例子：211，$(A.B)$，$(TIMES.(2.2))$，其中 211、A、B 和 TIMES 都被认为是原子。列表现在也是递归定义的：

（1）NIL（“空符号序列”）是一个列表。

（2）如果 x 是一个 s 表达式，y 是一个列表，那么 s 表达式 $(x.y)$ 也是一个列表。

作为一种简化的表示法，空列表 NIL 被记为()，并且许多括号是以这种通用形式来表达的：

$$(S1.(S2.(\cdots(SN.NIL)\cdots)))$$

简化了的 $(S1,S2,\cdots,SN)$，列表也可以包含新的列表，作为其元素，这能构造非常复杂的数据结构。

LISP 编程意味着 s 表达式和列表的算法处理。一个函数应用程序称为一个列表，其中

第一个列表元素是函数的名称，其余元素是这个函数的参数。这需要以下的基本函数：应用于 s 表达式的 CAR 函数返回左边的部分，

```
(CAR(x.y)) = x
```

CDR 函数给出右边的部分，

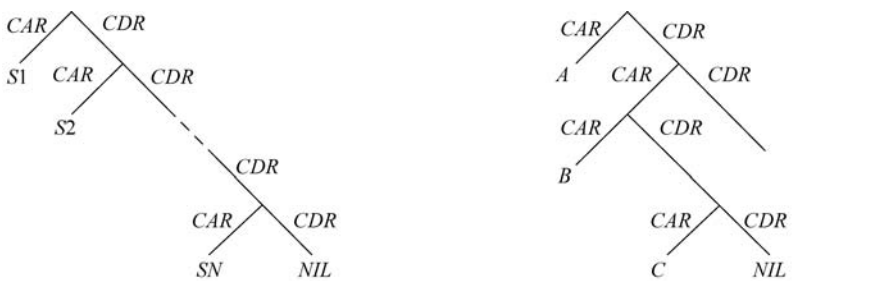
```
(CDR(x.y)) = y
```

$CONS$ 函数将两个 s 表达式组合成一个 s 表达式，

```
(CONS x y) = (x.y)
```

如果对列表使用这些函数，CAR 返回第一个元素；CDR 返回列表的其余部分而不返回第一个元素；$CONS$ 返回一个列表，第一个参数作为第一个元素，第二个参数是其余部分。

列表和 s 表达式也可以表示为二进制有序树。图 3.1(a)为通用列表($S1$,$S2$,…,SN)的树状表示，以及基本函数 CAR 和 CDR 的各自应用，图 3.1(b)为 s 表达式(A.((B.(C.NIL))))的树状表示。诸如(CAR.(CDR.x))的函数组合表示两个函数应用的连续执行，其中首先计算内层函数。对于多参数的函数，首先计算所有参数的值，然后计算该函数的值。列表通常被视为一个函数的应用，那么($ABCDEF$)意味着函数 A 将应用于 B、C、D、E 和 F。



(a) 列表($S1$,$S2$,…,SN)的树状表示　　(b) s表达式(A.((B.(C.NIL))))的树状表示

图　3.1

当然，将列表解释为(有序的)符号集通常是有意义的。因此，当涉及数字排序任务时，(14235)可以理解为将函数 1 应用于参数 4、2、3、5 是没有意义的。在 LISP 中引入了符号 $QUOTE$，根据该符号，以下列表不应理解为函数指令，而是符号的枚举，如 $QUOTE$(14235)或简化为 '(14235)。根据定义，B 是 CAR'(123)＝1、CDR'(123)＝'(23)和 $CONS$1'(23)＝'(123)。虽然变量是用字母原子记录的，但非数字常量可以通过引用变量来区分，例如变量 x 和 $LISTE$ 以及常数 'x 和 '$LISTE$。

根据这些约定，可以使用 LISP 中的基本函数来定义新函数。函数定义的一般形式如下：

```
(DE NAME (P1,P2,,…,PN) s - expression)
```

其中，$P1$，$P2$，…，PN 是函数的形式参数，NAME 是函数的名称。s 表达式是函数的主体，用形式参数描述函数的应用。如果在一个程序中，函数 NAME 以 NAME($A1$，$A2$，…，AN)的形式出现，则形式参数 $P1$，$P2$，…，PN 必须替换为函数体中相应的当前参数 $A1$、$A2$，…，AN 并且计算以这种方式更改的函数体。

【举例】 定义一个函数 THREE，它计算列表的第 3 个元素：

```
(DE THREE (LISTE)(CAR(CDR(CDR(LISTE)))))
```

其中，函数语句 THREE'(415)用(CAR(CDR(CDR'(415))))替换函数 THREE 主体中的形式参数，然后计算并返回值 5，即所提交列表中的第 3 个元素的值。

为了能够使产生函数的条件和案例区分，设计了新的原子，如 NIL 表示“假”，T 表示“真”，以及用来比较两个对象的新的基本函数(如 EQUAL)：

```
(EQUAL 12) = NIL,
(EQUAL 11) = T.
```

LISP 中条件表达式的一般形式如下：

```
(condition 1 s - expression 1)
(condition 2 s - expression 2)
⋮
(condition N s - expression N)
```

如果第 i 个条件($1 \leqslant i \leqslant N$)提供逻辑值 T，并且所有先前条件提供值 NIL，则这个条件表达式的结果为第 i 个 s 表达式。如果所有条件都有值 NIL，则条件表达式获得值 NIL。

【举例】 定义一个可以计算列表长度的函数：

```
(DE LENGTH (LISTE)
(COND
((EQUAL LISTE NIL)0)
(T(PLUS(LENGTH
(CDR LISTE))1))))
```

第一个条件确定列表是否为空。在本例中，它的长度为 0。第二个条件假定列表不是空的。在本例中，通过将数字 1 加到由第一个元素(LENGTH(CDR LISTE))缩短的列表长度上，来计算列表的长度。

现在定义在 LISP 程序下通常所知道的内容。

【定义】 LISP 程序本身就是一个函数定义列表和要用这些函数定义求值的表达式：

```
((DE Funct 1 …)
(DE Funct 1 …)
⋮
(DE Funct N …)
```

```
s- expression)
```

所有先前定义的函数都可以在用点表示的函数体中使用。由于 LISP 程序本身又是一个 s 表达式，LISP 中的程序和数据具有相同的形式，因此 LISP 也可以作为程序的元语言，即 LISP 可以用来讨论 LISP 程序。由于符号和结构的灵活处理，使 LISP 对于人工智能中知识处理问题的进一步适用成为可能。数值计算只是特殊情况。

人工智能试图从算法层面构造解决问题的策略，然后将其转换成人工智能编程语言，如 LISP。搜索问题是人工智能的一个核心应用领域。例如，如果一个对象被大量搜索，而没有关于问题解决方案的知识，那么人类也会选择一种启发式解决方案，这就是大英博物馆算法。

大英博物馆算法的例子有：在图书馆里搜索一本书、一个保险箱中的数字组合，或在给定条件下有限可能性的化学公式。如果最终检查了所有的可能性或情况，并且满足以下条件，那么在本程序之后一定会找到解决方案：

（1）有一组包含解决方案的形式对象。

（2）有一个生成器，即该集合的完整枚举过程。

（3）有一个测试，即谓词，它决定一个被创建的元素是否属于问题解决集。

因此，搜索算法也被称为“生成_测试(集合)”，首先从内容方面进行描述：

```
Function GENERATE_AND_TEST (SET)
If the SET quantity to be examined is empty,
    then Failure,
in other respects
    ELEM is the next element from SET;
If ELEM target element,
    then deliver it as a solution,
otherwise repeat this function
with the SET quantity reduced by ELEM.
```

要在 LISP 中表达此函数，需要辅助函数，其具体含义为：GENERATE 创建给定集合的一个元素。GOALP 是一个谓词函数，如果其参数属于该解集，则返回 T，否则返回 NIL。SOLUTION 处理要输出的解集元素。REMOVE 返回给定元素减少的数量。

LISP 中的形式如下：

```
(DE GENERATE_AND_TEST(SET)
(COND ((EQUAL SET NIL)'FAIL)
(T(LET(ELEM(GENERATE SET))
(COND((GOALP ELEM)(SOLUTION ELEM))
(T(GENERATE_AND_TEST
(REMOVE ELEM SET)))))))
```

这个例子清楚地表明，人类的思维不一定能够成为有效解决机械化问题的模型，目标是用富有表现力的人工智能编程语言，来优化人机界面。这些语言及其数据结构是否也模拟

或描绘了人类思维的认知结构，属于认知心理学的一个课题。人工智能编程语言主要被用作计算机辅助工具，来优化解决问题的过程。

3.4　自动证明

如果要用计算机实现智能，那么就必须追溯回计算能力。但是，算术是一个机械化的过程，可以分解为基本的算术步骤，即小学生就可以完成基本的算术计算步骤。以波斯数学家阿尔·奇瓦里斯米(al-Chwarismi)的名字来谈论“算法”，他在公元 800 年左右发现了简单的代数方程的求解方法。1936 年，图灵展示了如何将计算过程分解为一系列最小且最简单的步骤，因此他第一次在逻辑和数学上成功地发展了一种有效方法(算法)的一般术语。只有这样，才有可能回答一个问题在原则上是否可以计算，而不管相应的计算机技术如何。

图灵把他的机器想象成一个打字机，就像一个可移动的书写头，一个处理器可以将一个有限字母表中的区分良好的字符逐个打印到一个带状磁带上，如图 3.2 所示。带状磁带可以划分为一个个单独的字段，基本不受左右长度的限制。一个图灵机的程序由简单的基本指令组成，这些指令按顺序执行。机器可以打印或删除字母表中的符号，将读/写头向左或向右移动一个字段，然后经过许多步骤后停止。与打字机不同，图灵机可以逐个读取磁带中各个字段的内容，并执行后面的步骤。

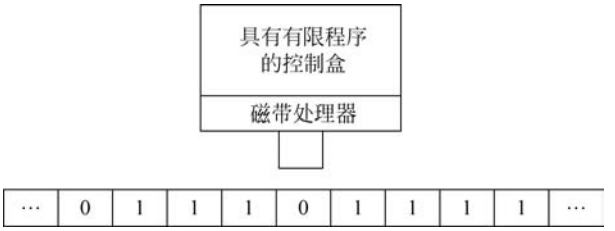


图 3.2　具有打孔磁带的图灵机

一个有限字母表的例子由 0 和 1 两个字符组成，用它们可以表示所有自然数 1，2，…。与计数一样，每个自然数 1，2，3，4，…都可以通过加 1 产生，即 1，1＋1，1＋1＋1，1＋1＋1＋1，…。因此，一个自然数就由图灵机的打字机条状磁带上由逐个打印的 1 的链表示。每个数字都以开头和结尾的数字 0 为界限。在图 3.2 中，数字 3 和 4 在带状胶带上被打印出来。

【举例】 一个计算 3＋4 的加法程序包括删除两个 1 构成的链之间的 0，并将左边的 1 链向右移动 1 字段的位置。因此，创建了一个包含 7 个 1 的链，即数字 7 的表示，然后程序停止。

不是图灵机的每个程序都像加法一样简单，然而原则上，使用自然数进行计算可以追溯到使用图灵机的基本命令来操作 0 和 1。一般来说，算术上研究具有参数 $x1,\cdots,xn$ 的 n 元函数 f，例如 $f(x1,x2)=x1+x2$。每个参数都是图灵磁带上由数字 1 构成的链所表示的一个数字，其余字段为空，即打印为 0。现在，一个函数的图灵可计算性可以被通用地定

义为：

在一条图灵带上的计算开始时，只有 1 组成的链，它们被零隔开，即…$0x_1 0x_2 \cdots 0x_n \cdots$。一个具有参数 $x_1, \cdots, x_n$ 的 n 元函数 f 称为图灵可计算的，当且仅当有一个带标签…$0x_1 0x_2 \cdots 0x_n \cdots$的图灵机，在经过许多步骤后，停止在标记为…$0f(x_1, \cdots, x_n)0 \cdots$的磁带处。函数值 $f(x_1, \cdots, x_n)$由相应的 1 组成的链表示。

每个图灵机可以由它的指令列表唯一地定义。这个图灵程序由有限多个指令和有限集字母表中的字符组成，指令和字符可以用数字编码。因此，一个图灵机可以唯一地用一个数字代码（机器号）来表征，它用有限的字符数和排列来对相应的机器程序进行加密。与任何数字一样，这个机器号可以被记录为图灵磁带上的一系列 0 和 1。因此，图灵用一个特定的具有一个给定图灵磁带的图灵机可以模拟任何类型磁带上的任何图灵机的行为。图灵称这种机器是通用的，它将所模拟机器的每一条指令（其机器代码记录在其磁带上）转换成任何给定磁带标记的相应处理步骤。

从逻辑的角度来看，任何一台通用的程序控制计算机（如约翰·冯·诺依曼或祖泽（Zuse）发明的计算机）都不过是这种通用图灵机的一种技术实现，它可以执行任何可能的图灵程序。今天的计算机是一种多用途的仪器，可以把它用作打字机、计算机、书籍、图书馆、视频设备、打印机或多媒体播放器，这取决于如何设置程序和运行何种程序。不仅如此，智能手机和汽车里也充满了计算机程序。原则上，这些程序中的每一个都可以追溯到一个图灵程序。由于它们有许多任务，这些图灵程序肯定会比今天所安装的程序更复杂、更庞大、更缓慢。但是，从逻辑的角度来看，这些技术问题是无关紧要的。原则上，每台计算机都可以计算同一类函数，这些函数也可以由图灵机通过任意增加内存容量和延长计算时间计算出来。

除了图灵机之外，人们还提出了定义可计算函数的各种其他方法，这些方法被证明与图灵机的可计算性在数学上是相等的。

丘奇（A. Church）在一个以他名字命名的论题（丘奇论题）中指出，可计算性的直觉概念完全可以被图灵可计算性这样的定义所涵盖。丘奇论题当然不能被证明，因为它将较为精确的术语（如图灵可计算性）和计算程序的直觉概念进行比较。然而，所有先前关于可计算性定义的建议在数学上都等同于图灵可计算性，这一点支持了丘奇论题。

如果想确定一个问题的解决方案有多聪明，必须首先弄清楚一个问题有多困难和复杂。在可以追溯到图灵的可计算性理论中，一个问题的复杂性是由解决它所需的计算消耗衡量的。根据丘奇论题，图灵机是对可计算性的精确度量。一个问题能否得到有效的判定，与它的可计算性直接相关。例如，一个自然数是否为偶数的问题，可以通过在有限步骤后检查所给定的自然数是否可被 2 整除来确定，这可以用一个图灵机的程序计算出来。

然而，对于待解问题来说，只运用一个特定的决策程序是不够的。这往往是一个寻找各种解决办法的问题。设想一个机器程序，它系统地列出了所有解决问题或满足某个属性的数字。

如果一个算术属性的实现数可以用有效的可计算方法（算法）枚举出来（找到），那么它就是有效可枚举的。

为了判断一个任意呈现的数字是不是偶数，有效地逐一列举所有偶数来确定要查找的数字是否包括在内是不够的，还必须能够有效地枚举所有非偶数(奇数)，以便确定要查找的数字是否属于不符合所需属性的数字集。

一般来说，如果一个集合及其补集(其元素不属于该集合)是有效可枚举的，那么它是有效可判定的。因此，每个有效可判定的量也是有效可枚举的。然而，有一些有效可枚举集是不可判定的。这就引出了一个关键问题，即是否也存在非计算(非算法)思维。

一个无法有效解决的问题的例子就是图灵机本身。

图灵机的停止问题。原则上，对于任意图灵机在任意输入下是否在有限步骤后停止的问题，没有通用的判定程序。

图灵从是否所有实数都可计算的问题开始，证明停止问题的不可判定性。类似 $\pi=3.1415926$ 的实数，小数点后由无限个数字组成，这些数字似乎是随机分布的。然而，可以设计一个有限的子程序或程序来逐步计算每个数字，使得 π 的精度逐步提高，因此 π 是一个可计算的实数。图灵在第一步中就定义了一个可证明的不可计算的实数。

【背景资料】 一个图灵程序由有限的符号列表和操作指令组成，可以用数字代码对它们进行加密。事实上，这也发生在计算机的机器程序中。这样，每个机器程序都可以用一个数字代码唯一地描述，称这个数字为机器程序的代码或程序号。现在设想一个所有可能的程序编号的列表，它们按照 p1，p2，p3，…的顺序排列，并且其尺度越来越大。如果一个程序计算一个在小数点后有无限位数的实数(如 π)，那么这会在相应的程序号之后的列表中被注明；否则该程序编号后面的行是空的，例如：

$$
\begin{aligned}
&\mathrm{p1}\ -.\ z_{\underline{11}}\ z_{12}\ z_{13}\ z_{14}\ z_{15}\ z_{16}\ z_{17}\cdots\\
&\mathrm{p2}\ -.\ z_{21}\ z_{\underline{22}}\ z_{23}\ z_{24}\ z_{25}\ z_{26}\ z_{27}\cdots\\
&\mathrm{p3}\ -.\ z_{31}\ z_{32}\ z_{\underline{33}}\ z_{34}\ z_{35}\ z_{36}\ z_{37}\cdots\\
&\mathrm{p4}\ -.\ z_{41}\ z_{42}\ z_{43}\ z_{\underline{44}}\ z_{45}\ z_{46}\ z_{47}\cdots\\
&\mathrm{p5}\ -.\ z_{51}\ z_{52}\ z_{53}\ z_{54}\ z_{\underline{55}}\ z_{56}\ z_{57}\cdots\\
&\vdots
\end{aligned}
$$

为了定义这个不可计算数，图灵选择列表对角线上带下画线的值，对其进行修改(例如通过加 1)，并将这些修改后的值(用 * 表示)在新实数的开头用一个小数点组合起来：

$$-.\ z^{*}_{11}\ z^{*}_{22}\ z^{*}_{33}\ z^{*}_{44}\ z^{*}_{55}\cdots$$

这个新数字不能出现在列表中，因为它与在 p1 之后第一个数字的第一位数，p2 之后第二个数字的第二位数……小数点后的所有数字都不同。因此，用这种方式定义的实数是不可计算的。

通过这个例子，图灵在下一步中证明了停止问题的不可判定性。如果停止问题可以确定，那么就可以决定在有限的步骤之后，是否由第 n 个计算机程序($n=1,2,\cdots$)计算、停止和打印小数点后面的第 n 个十进制数。因此，可以计算一个实数，根据它的定义，它不能出现在所有可计算实数的列表中。

逻辑演算的形式推导(证明)可以理解为枚举方法，用它可以枚举逻辑真理的代码数字。

在这个意义上，一阶谓词逻辑（PL1）的逻辑真理集是有效可枚举的。然而，由于对于任何一个数字，无论它是否是 PL1 的可证公式（即逻辑真理）的代码数字，都没有一个通用的计算过程。应该强调的是，这种算法没有通用的决策程序。

【重点】 PL1 的形式逻辑演算是完整的，因为可以用它形式化地导出一阶谓词逻辑的所有逻辑真理。

相比之下，算术的形式主义及其基本算术运算是不完整的[1931 年哥德尔（K. Gödel）的第一个不完整性定理]。

与哥德尔的广泛证明相反，算术的不完全性直接来自图灵停止问题。

【背景资料】 如果有一个完整的形式公理系统，从中可以导出所有的数学真理，那么也将有一个关于计算机程序是否会在某个点停止的决策过程。

只需要简单地检查各类证明，直到找到程序停止的证据，或者找到它从未停止的证据。

因此，如果有一组有限的公理可以导出所有的数学证据，就可以判定一个计算机程序是否在有限的步骤后停止，这与停止问题的不可判定性相矛盾。

哥德尔第二不完整性定理表明，一个形式系统的一致性不能用系统本身的有限方法来证明，此处的有限证据是指模拟计数过程 1，2，3，…的程序。

如果将证明方法扩展到这类有限证明方法之外，形式数论的一致性就可以用更强的方法证明。这是逻辑学家和数学家根策恩（G. Gentzen，1909—1945）的基本思想，他使用该思想介绍了现代证明理论，并对后来计算机科学中的计算机程序产生了重要的推动作用。1936 年，图灵写了一篇关于决策问题的著名文章："我们也可以用这样一种方式来表达它：对于数论来说，不可能一劳永逸地指出充分的推论体系，而是可以一次又一次地找到命题，而命题的证明需要新颖的推论。"

应用到计算机上，不可能有这样的"超级计算机"，即它可以为任意输入决定所有可能的（数学）问题。然而，可以不断地补全已知的数学公式，以获得更丰富、更强大的程序。

【背景资料】 公式的复杂性导致要考虑可判定性的程度："对于所有自然数，都存在 n、m 以及自然数 p，使得 $m+n=p$（形式上：$\wedge m \wedge n \vee p\ m+n=p$）"。这个公式包括一个等式 $m+n=p$ 和变量 m、n、p，并通过一个存在量词和两个全称量词得到扩展。两个数的加法是有效可计算的，因此方程中声明的性质是有效可判定的。

一般来说，一个算术公式由一个有效可判定的属性组成，这个属性被逻辑量词扩展。根据这些量词的数量、类型和顺序，可以区分不同的复杂公式类别，它们对应于可判定性程度。

可判定属性对应于在经过许多步骤后停止的图灵机。如果加上量词，图灵机的概念必须扩展，因为计算过程可能要无限地运行几次（即沿所有自然数运行），这些过程有时被称为"超级计算"。然而，这些只是计算机器的形式化模型，超出了物理计算机的技术物理实现。

值得注意的是，图灵在他的论文中已经讨论了超级可计算性的主题，并质疑了机器在有效算法之外的行为。

对于人工智能，可计算性和可判定性的类别和程度是基于逻辑和数学证明的，因此它们与物理计算机的技术性能无关。即使是未来的超级计算机也无法克服逻辑和数学定律！

就智力水平而言，原则上能否决定一个问题，不仅是解决问题的唯一有趣的方面，而且是如何以及用什么样的努力才能做出决定的问题。

【举例】 关注一个众所周知的问题：一个旅行商人不得不用尽可能短的路线，逐个把客户的订单送到不同的城市。例如，对于 3 个客户，有 3 种可能的路线：先到达第一个客户的城市；对于第二个客户，有 2＝3－1 种可能的路线；对于第三个客户，只有 1＝3－2 种路线到达那里，然后开车回家。因此，路线数包括 3 * 2 * 1＝6 种可能性。数学家不说可能性而是说“阶乘”，然后写成 3!＝6。随着客户数量的增长，可能性快速增长，从 4!＝24、5!＝120 到 10!＝3,628,800。

一个实际应用是一台机器如何以最短路径在电路板上钻 442 个孔的问题。这种印刷电路板在家用电器、电视接收器或计算机中也有类似的数量。442! 的数值是一个一千位以上的十进制数字，是无法一一尝试的。如何有效地解决这样的问题？

使用确定性图灵机的计算时间很长，是由于这样一个事实造成的：一个问题的所有局部问题和实例区别都必须逐个进行系统的检查和计算。因此，有时通过随机决策从有限的可能性中选择一个解决方案似乎更为明智。这就是非确定性图灵机的工作原理：它随机选择问题的一个可能的解决方案，然后证明所选的可能性。例如，为了决定一个自然数是否是一个合数，非确定性机器选择一个除数，去除所给定的自然数，会有余数，然后检查余数是否为 0；如果余数为 0，机器确认该数字是合数。另一方面，确定性图灵机必须系统地搜索所有除数，通过枚举每个小于给定自然数的数，将其作为除数去除这个自然数，看余数是否为 0。

由一个确定性机器在多项式时间内决定的问题称为 P 问题。如果问题是由多项式时间的一个非确定性机器决定的，称为 NP 问题。根据此定义，所有的 P 问题也是 NP 问题。然而，是否所有的 NP 问题都是 P 问题，即非确定性机器是否能被多项式计算时间的确定性机器代替，仍然是一个悬而未决的问题。

今天还有一些尚未决定的问题，但可以确切地确定它们有多困难。从命题逻辑中，知道如何找出哪些基本命题是真是假，从而使得由这些基本命题组成的命题是真的（见 3.1 节）。

因此，当且仅当基本命题 A 或 B 都为真时，由逻辑联结词“与(and)”联结的基本命题 A 和 B 得到一个真正的复合命题。当且仅当至少一个基本命题为真时，由“或”联结的基本命题才产生真正的复合命题。在这些情况下，复合命题是“可满足的”。另一方面，由“与 and”联结的基本命题 A 及其否定并不构成一个可满足命题：两个基本命题不可能同时为真。

检查复合命题的所有真值组合的算法的计算时间取决于其基本命题的个数。到目前为止，还没有一种算法可以在短时间内解决多项式问题，也不确定这种算法是否存在。然而，美国数学家库克(A. Cook)在 1971 年证明了可满足性问题至少和其他 NP 问题一样困难。

如果一个问题的一个解决方案也能解决另一个问题，那么这两个问题的难度是相等的。在这个意义上等价于某一类问题的问题可以称为相对于该类问题的完全问题。在满足性问题之后，其他经典问题（如旅行商问题）也可以证明为 NP 完全问题。

NP 完全问题被认为是极其困难的。因此，实践者不是在寻找精确的解决方案，而是在切实可行的限制下寻求几乎最优的解决方案，这里需要想象力和创造力。在实际的网络环

境下，网络的规划问题和规划问题都在发生变化。所需的计算量、时间和存储容量越少，实际问题的解决方案就越便宜、越经济。因此，复杂性理论为实际的、智能的问题解决方案提供了框架条件。

在人工智能中，算法通过从数据结构中派生出更多的字符序列来实现知识处理，这符合自古以来数学证明的理想。欧几里得(Euclid)已经证明了数学定理是如何从只有通过逻辑推理才能假定为真的公理中推导和证明的。在人工智能中，出现的问题是数学证明是否可以转换为算法和“自动化”。在这背后，是一个基本的人工智能问题，即思维是否可以自动化以及在何种程度上可以由计算机执行。

关注这方面的一个经典证明：大约公元前 300 年，欧几里得证明了无穷多个质数的存在，他避免了“无限”一词，并声称：“总有更多的素数比任何已有的素数都多”。素数是一个有且仅有两个自然数作为其除数的自然数，因此质数是大于 1 的自然数，它只能被自身和整数 1 除，例如 2,3,5,7,…。

欧几里得用互为矛盾的证据来论证。他假设了互为相反的论断，在这个假设下有逻辑地得出了两个互相矛盾的结论，因此这个假设是错误的。如果现在要确定这个陈述是真是假，那么与该假设相反的是：这个陈述是真的。

【举例】 现在矛盾的证明：假设最后只有许多质数 $p_1,\cdots,p_n$。用 m 表示可以除以所有这些质数的最小数，乘积 $m=p1\cdot pn$。$m+1$ 的继任者有两种可能：

第一种情况：$m+1$ 是质数。根据定义，它大于 $p_1,\cdots,p_n$，因此是一个额外的质数，与假设相矛盾。

第二种情况：$m+1$ 不是质数。那么它必须有除数 q。假设 q 必须是质数 $p_1,\cdots,p_n$ 中的一个，这也使它成为 m 的除数，因此质数 q 将 m 和后继的 $m+1$ 分开。然后再除以 m 和 $m+1$ 的差，即 1。但是这并不适用，因为根据定义，1 作为除数时不能称为质数。

这个证明的缺点是没有建设性地证明质数的存在性，而只表明了假设的反面会导致矛盾。为了证明一个对象的存在，需要一个算法来创建一个对象并证明这个例子中的语句是正确的。在形式上，存在性论断称作 $A\equiv\exists xB(x)$(见第 3.1 节)。在简化形式下，可以用一个最终的多个数字组成的列表 $t_1,\cdots,t_n$ 来构造语句 B 的候选者，使或语句适用，即 $B(t_1),\cdots,B(t_n)$语句适用于至少一个所构造的数字 $t_1,\cdots,t_n$。如果，对于所有的 x 和 y，$B(x,y)$都存在，即形式上 $A\equiv\forall x\exists yB(x,y)$是有效的，那么需要一个算法 p，它为每个 x 值构造一个值 $y=p(x)$，以便对所有 x 的 $B(x,p(x))$是有效的，即形式上是 $\forall xB(x,p(x))$。在一个较弱的形式中，如果对于给定 x 值前提下，y 值的搜索过程至少可以计算一个上限 $b(x)$，即形式上 $\forall x\exists y\leqslant b(x)B(x,y)$，将满足这个结果。这样可以精确估计搜索过程。

因此，美国逻辑学家克雷塞尔(G. Kreisel)要求的证明不仅仅是验证。在某种程度上，它们是“冻结”的算法，只需在证明中发现它们，然后把它们“卷起”(展开证明)。它们也可以接管机器的计算。

【举例】 事实上，在欧几里得的证明中，一个建构程序是“隐藏”的。对于一个质数 p_r

的任何位置r(在枚举$p_1=2,p_2=3,p_3=5,\cdots$中),可以计算一个上限$b(r)$,也就是说,可以为每个提交的质数指定一个进一步的质数,但是,这些质数位于可计算的上限之下。在欧几里得证明中,最终假定了许多质数,这些质数小于或等于一个上限x,而$p\leqslant x$,由此可以构造出一个$1+\Pi p\leqslant x_p$的数,并由此导出矛盾。($\Pi p\leqslant x_p$是所有小于或等于x的质数的乘积。)因此,首先构造上限:

$$g(x):=1+x!\geqslant 1+\Pi p\leqslant xp$$

阶乘函数$1\cdot 2\cdot\cdots\cdot x=x!$可以用所谓的斯特林公式来估计。然而,针对的是第$r+1$个质数$p_{r+1}$的一个上界,它只依赖于$r$在质数计数中的位置,而不是未知的阈值$x\geqslant p_r$。欧几里得证明表明$p_{r+1}\leqslant p_1\cdot\cdots\cdot p_{r+1}$。由此可以证明,对所有$r\geqslant 1$(通过$r$的完全归纳),$p_r<2^{2^r}$。因此,要寻找的可计算势垒是$b(r)=2^{2^r}$。

在逻辑学和数学中,公式(即一系列符号)是一步一步推导出来的,直到一个结论的证明完成为止。计算机程序的工作基本上和证明一样,也是一步一步地根据定义的规则派生字符串,直到找到一个表示问题解决方案的形式表达式。例如,想象在装配线上制造一辆汽车,相应的计算机程序描述了汽车是如何根据规则,从预先设定的各个零件开始,一步一步地制造出来的。

一个客户想要一个计算机科学家的计算机程序来解决类似的问题。在一个非常复杂和令人困惑的生产过程中,他肯定希望事先证明这个程序能够运行正常。可能的错误会是危险的,或者会造成相当大的额外成本。有一种能自动从问题的形式属性中提取证据的软件,就像“数据挖掘”中的软件用于搜索数据或数据相关性一样,合适的软件也可以用于自动搜索证据以证明。这被称为“证据挖掘”,对应于乔治·克雷塞尔(Georg Kreisel)从证明中过滤算法的方法(展开证明),但现在由计算机程序自动完成。

然而,这就提出了一个问题:用于提取证据的软件本身是否可靠。这种底层软件可靠性的证明可以在一个精确定义的逻辑框架内提供,这样客户就一定能确信计算机程序在正确地解决这个问题。因此,这种“自动证明”不仅对现代软件技术具有相当重要的意义,它也导致哲学上的深层问题,即(数学)思维在多大程度上可以自动化:证明的寻找是自动化的。然而,用于此目的的软件正确性证明是由一个数学家提供的。假设再次自动化这个证明,一个基本的认识论问题出现了:这难道不会导致人类退化,使得最终总是使用这类工具(必须使用)吗?

一个例子是交互式证明系统 MINLOG,它自动从正式证据中提取计算机程序,并使用计算机语言 LISP(见 3.3 节)。一个简单的例子:对于 LISP 中的每个符号列表v,总有一个反向列表w与v中符号的顺序相反。这是另一种形式的$A\equiv vwb(v,w)$的断言。通过对列表结构v的归纳,可以非正式地提供证明,MINLOG 自动从中提取一个合适的计算机程序。但是这个软件也可以用于数学证明的要求。可靠性的一般证明保证这种软件提供正确的程序。

1969 年,逻辑学家霍华德(W. A. Howard)观察到,根策恩(Gentzen)的自然演绎证明系统可以在其直观的版本中直接解释为计算模式的一种类型化变体λ演算(lambda 计算)。

这种对数学证明理论的基本见解为新一代交互式和自动化的证明助手开辟了道路。

根据丘奇(A.Church)论题,$\lambda a.b$ 是指应用$\lambda a.b[a]=b$,将元素 a 映射到函数值b 上的函数。在下面,证明用术语 $a,b,c,\cdots$表示;命题用 $A,B,C,\cdots$表示。蕴涵 $A\rightarrow B$ 的证明(即如果 A,那么 B)从命题 A 的假设[A](用括号标记的假设)和导出的命题 B 开始。这个证明被理解为函数 $a.b$,它将命题 a 的假定证明 A 映射到命题 b 的证明 B 上,后者用根策恩风格写成:

$$\begin{array}{l}[A]\\ \lambda a.b \quad \vdots\\ \underline{B}\\ A\rightarrow B\end{array}$$

根据丘奇论题,lambda 术语 $\lambda a.b$ 代表一个计算机程序。按照库里-霍华德(Curry-Howard)对应,一个证明被认为是一个程序,它证明的公式是程序的类型。在本例中,$A\rightarrow B$ 是程序 $\lambda a.b$ 的类型。

构造演算 CoC 是蒂埃里-科克安(Thierry-Coquand)等的一种类型理论,它既可作为类型化编程语言,又可作为数学的建设性基础。它将库里-霍华德对应关系推广到全直觉谓词演算中的证明。CoC 只需要很少的几句话来解释规则。CoC 的对象包括证明(以命题为类型的术语)、命题、谓词(返回命题的函数)和大类型(谓词类型)。

归纳结构演算 CiC 是以 CoC 为基础,丰富了构造术语的归纳定义。归纳类型是由一定数量的构造函数自由生成的。一个例子是自然数的类型,它由构造器 0 和 succ(继承者)来递归地定义。A 型元素有限列表的类型是用构造函数 NIL 和 CONS 递归地定义的(参见 3.3 节)。

证明助手 Coq 实现了一个基于 CiC 的程序,该程序将高阶逻辑和丰富类型的函数语言结合到一起。Coq 的命令允许下列操作:

(1) 定义(可有效评估的)函数或谓词。

(2) 陈述数学定理和软件规范。

(3) 交互式地开发这些定理的形式证明。

(4) 通过相对较小的认证对这些证明进行机器检查。

(5) 将认证程序提取为计算机语言(如 Objective Caml、Haskell、Scheme)。

Coq 提供了交互式证明方法、决策和半决策算法,与外部定理证明器的联结是可用的。Coq 是一个用于验证数学证明和 CiC 中计算机程序验证的平台。

一个硬件或软件程序如果可以被验证遵循 CIC 中给定的规范,则它是正确的("由 Coq 认证")。

【举例】 Coq 已应用于巴黎的 14 号线全自动控制。

电路的结构和行为可以用互连的有限自动机进行数学建模。在电路中,人们必须处理无限长的时间序列的数据(流)。如果在一定条件下结构自动机的输出流与行为自动机的输出流是相同的,那么电路是正确的。因此,为了证明电路的正确性,必须在 CiC 中实现自动

机理论。

在软件和硬件验证方面，Coq 证明辅助工具有哪些优点？

(1) 在 Coq 中，计算机程序的验证与构造形式主义中的数学证明一样强大。

(2) Coq 类型的使用提供了精确可靠的规范。

(3) 分层和模块化方法允许在与预焙炉组件相关的复杂验证过程中获得正确性结果。

还有更多的证明助手，例如阿格达和伊莎贝尔(Agda 和 Isabelle)。从实际角度来看，人工智能程序的复杂性日益增加，而这些程序往往不能由具有数学证明准确性的证明助手来处理。在本书的后面，将面对现代机器学习中的安全性挑战。

参考文献