计算机科学188：人工智能导论
2024年春季 Note 8

作者（所有其他注释）：尼基尔·夏尔马

作者（贝叶斯网络注释）：乔希·胡格和杰基·梁，由王蕾吉娜编辑

作者（逻辑注释）：亨利·朱，由考佩林编辑

学分（机器学习和逻辑注释）：部分章节改编自教科书《人工智能：一种现代方法》。

最后更新时间：2024年3月5日

命题逻辑

与其他语言一样，逻辑也有多种变体。我们将介绍两种：命题逻辑和一阶逻辑。命题逻辑由命题符号组成的句子书写而成，可能会通过逻辑连接词连接。命题符号通常用单个大写字母表示。每个命题符号代表关于世界的一个原子命题。一个模型是对所有命题符号赋予真或假的赋值，我们可以将其视为一个"可能世界"。例如，如果我们有命题 $A =$ "今天下雨了" 和 $B =$ "我忘了带伞"，那么可能的模型（或"世界"）如下：

1. { A=true,B=true } （"今天下雨了且我忘了带伞。"）
2. { A=true,B=false } （"今天下雨了但我没忘带伞。"）
3. { A=false,B=true } （"今天没下雨但我忘了带伞。"）
4. { A为假，B为假 } （"今天没下雨，而且我也没忘带伞。"）

一般来说，对于 N 个符号，存在 2^N 种可能的模型。如果一个句子在所有这些模型中都为真，我们就说它是有效的（例如句子True）；如果至少存在一个模型使其为真，那么它是可满足的；如果在任何模型中都不为真，那么它是不可满足的。例如，句子 $A \wedge B$ 是可满足的，因为它在模型1中为真，但不是有效的，因为它在模型2、3、4中为假。另一方面， $\neg A \wedge A$ 是不可满足的，因为对于 A 的任何选择都不会返回True。

以下是一些有用的逻辑等价关系，可用于将句子简化为更易于处理和推理的形式。

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha \quad \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) \quad \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) \quad \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) \quad \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

图7.11 标准逻辑等价式。符号 α, β 和 γ 代表命题逻辑中的任意句子。

命题逻辑中一种特别有用的语法是合取范式（CNF），它是子句的合取，每个子句是文字的析取。它具有一般形式 $(P_1 \vee \cdots \vee P_i) \wedge \cdots \wedge (P_j \vee \cdots \vee P_n)$ ，即它是 “或” 的 “与”。正如我们将看到的，这种形式的句子对某些分析很有用。重要的是，每个逻辑句子都有一个逻辑等价的合取范式。这意味着我们可以通过将这些CNF语句 “与” 在一起，将我们知识库中包含的所有信息（知识库只是不同句子的合取）表述为一个大的CNF语句。

合取范式（CNF）表示在命题逻辑中尤为重要。在此，我们将看到一个把句子转换为合取范式表示的示例。假设我们有句子 $A \Leftrightarrow (B \vee C)$ 并且我们想把它转换为合取范式。推导基于图7.11中的规则。

1. 消除 $\Leftrightarrow$： 使用双条件消除，表达式变为 $(A \Rightarrow (B \vee C)) \wedge ((B \vee C) \Rightarrow A)$ 。
2. 消除 $\Rightarrow$： 使用蕴含消除，表达式变为 $(\neg A \vee B \vee C) \wedge (\neg (B \vee C) \vee A)$ 。
3. 对于合取范式（CNF）表示，“非” $(\neg)$ 必须仅出现在文字上。使用德摩根定律，我们得到 $(\neg A \vee B \vee C) \wedge ((\neg B \wedge \neg C) \vee A)$ 。
4. 作为最后一步，我们应用分配律并得到 $(\neg A \vee B \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A)$ 。

最终表达式是三个或关系子句的合取，因此它是合取范式形式。

命题逻辑推理

逻辑之所以有用且强大，是因为它赋予了我们从已知信息得出新结论的能力。为了定义推理问题，我们首先需要定义一些术语。
我们说，如果在所有使 A 为真的模型中， B 也为真，那么句子 A 蕴含另一个句子 B ，我们将这种关系表示为 $A \models B$ 。注意，如果 $A \models B$ ，那么 A 的模型是 $B, (M(A) \subseteq M(B))$ 的模型的一个子集。推理问题可以表述为确定是否 $KB \models q$ ，其中 KB 是我们的逻辑句子知识库， q 是某个查询。例如，如果艾丽西亚发誓再也不踏入十字路口，那么我们可以推断，当我们寻找一起吃晚餐的朋友时，不会找到她。

我们利用两个有用的定理来证明蕴含关系：

i.)（当且仅当 $A \Rightarrow B$ 有效时，$A \vDash B$ 成立）。

通过证明 $A \Rightarrow B$ 有效来证明蕴含关系被称为直接证明。

ii.) $(A \vDash B \text{iff} A \wedge \neg B \text{is unsatisfiable})$ 。

通过证明 $A \wedge \neg B$ 不可满足来证明蕴含关系被称为反证法。

模型检查

一种检查 $KB \vDash q$ 是否成立的简单算法是枚举所有可能的模型，并检查在所有 KB 为真的模型中，q 是否也为真。这种方法被称为模型检查。对于符号数量可行的句子，可以通过绘制真值表来进行枚举。
对于一个命题逻辑系统，如果有 N 个符号，就有 2^N 个模型需要检查，因此该算法的时间复杂度为 $O\left(2^N\right)$ ，而在一阶逻辑中，模型的数量是无限的。实际上，命题蕴含问题已知是co-NP完全问题。虽然最坏情况下的运行时间不可避免地是问题规模的指数函数，但有些算法在实际中可以更快地终止。我们将讨论两种用于命题逻辑的模型检查算法。

第一种方法由戴维斯（Davis）、普特南（Putnam）、洛根曼（Logemann）和洛夫兰德（Loveland）提出（我们将其称为DPLL算法），本质上是一种深度优先的回溯搜索，针对可能的模型运用了三种技巧来减少过度回溯。该算法旨在解决可满足性问题，即给定一个句子，找到对所有符号的有效赋值。如我们所述，蕴含问题可以简化为可满足性问题之一（证明 $A \wedge \neg B$ 不可满足），具体而言，DPLL处理的是合取范式（CNF）中的问题。可满足性可以如下表述为一个约束满足问题：将变量（节点）设为符号，将约束设为CNF所施加的逻辑约束。然后DPLL将继续为符号分配真值，直到找到一个满足的模型，或者在不违反逻辑约束的情况下无法为某个符号赋值，此时算法将回溯到上一个有效赋值。然而，DPLL相对于简单的回溯搜索有三点改进：

1. 提前终止：如果任何符号为真，则子句为真。因此，甚至在所有符号都被赋值之前，就可以知道该句子为真。此外，如果任何单个子句为假，则句子为假。在所有变量都被赋值之前，提前检查整个句子是否可以被判断为真或假，可以防止在子树中进行不必要的迂回。
2. 纯符号启发式：纯符号是在整个句子中仅以其肯定形式（或仅以其否定形式）出现的符号。纯符号可以立即被赋值为真或假。例如，在句子 $(A \vee B) \wedge(\neg B \vee C) \wedge(\neg C \vee A)$ 中，我们可以将 A 识别为唯一的纯符号，并可以立即将A赋值为真，将满足问题简化为仅找到 $(\neg B \vee C)$ 的一个满足赋值的问题。

3. 单元子句启发式：单元子句是仅包含一个文字的子句，或者是一个文字与多个假值的析取式。在单元子句中，我们可以立即为文字赋值，因为只有一种有效的赋值。例如，对于单元子句 $(B \vee \text{false} \vee \cdots \vee \text{false})$ 为真，B 必须为真。

函数DPLL - 可满足？(s) 返回真或假
  输入： s，一个命题逻辑中的句子

  子句 $\leftarrow$ s 的合取范式表示中的子句集
  符号 $\leftarrow$ s 中的命题符号列表
  返回DPLL(子句, 符号, {})

函数DPLL(子句, 符号, 模型) 返回真或假
  如果子句集中的每个子句在模型中都为真，则返回真
  如果子句集中的某个子句在模型中为假，则返回假
  P, 值 $\leftarrow$ FIND - 纯符号（符号集，子句集，模型）
  如果 P 不为空，则返回DPLL（子句集，符号集 - P，模型 $\cup\{P=$ 值 $\}$）
  P，值 $\leftarrow$ FIND - 单元子句（子句集，模型）
  如果 P 不为空，则返回DPLL（子句集，符号集 - P, 模型 $\cup\{P=$ 值 $\}$）
  P $\leftarrow$ 符号集的FIRST函数；剩余部分 $\leftarrow$ 符号集的REST函数
  返回DPLL(子句集, 剩余部分, 模型 $\cup\{P=$ 真 $\}$) 或者
      DPLL(子句集, 剩余部分, 模型 $\cup\{P=$ 假 $\}$))

图7.17 用于检查命题逻辑中句子可满足性的DPLL算法。文本中描述了FIND - PURE - SYMBOL和FIND - UNIT - CLAUSE背后的思想；每个函数返回一个符号（或空值）以及要赋给该符号的真值。与TT - ENTAILS?一样，DPLL在部分模型上运行。

DPLL：示例

假设我们有以下合取范式（CNF）的句子：

$$(\neg N \vee \neg S) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (S)$$

我们想用DPLL算法来确定它是否可满足。假设我们使用固定的变量排序（字母顺序）和固定的值排序（真先于假）。
在每次对DPLL函数的递归调用中，我们跟踪三件事：

- model是我们到目前为止已赋值的符号及其值的列表。例如，$\{A:T, B:F\}$ 告诉我们到目前为止已赋值的两个符号的值。
- symbols是仍需赋值的未赋值符号的列表。
- clauses是合取范式（CNF）中的子句（析取式）列表，在此调用或对DPLL的未来递归调用中仍需考虑。

换句话说，每次对DPLL的调用都在解决一个较小的可满足性问题，通常子句更少、符号更少，并且有一个已经为一些符号赋值的模型。
我们首先使用一个空模型（尚未分配任何符号）、包含原句中所有符号的符号集以及包含原句中所有子句的子句集来调用DPLL。我们最初的DPLL调用如下所示：

- 模型: $\{\}$
- 符号: $[L, M, N, P, Q, R, S]$

- *clauses*: $(\neg N \vee \neg S) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (S)$

首先，我们应用提前终止：我们检查给定当前模型时，每个子句是否为真，或者至少有一个子句为假。由于模型尚未为任何符号赋值，我们还不知道哪些子句为真或为假。

接下来，我们检查纯文字。没有仅以非否定形式出现的符号，也没有仅以否定形式出现的符号，所以没有我们可以简化的纯文字。例如， N 不是纯文字，因为第一个子句使用了否定的 $\neg N$ ，而第二个子句使用了非否定的 N 。
接下来，我们检查单元子句（只有一个符号的子句）。有一个单元子句 S 。为了使整个句子为真，我们知道 S 必须为真（没有其他方法可以满足该子句）。因此，我们可以再次调用DPLL，在我们的模型中将 S 赋值为真，并从仍需要赋值的符号列表中删除 S 。

我们的第二次DPLL调用如下所示：

•模型：$\{S : T\}$

•符号：$[L, M, N, P, Q, R]$

- *clauses*: $(\neg N \vee \neg S) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (S)$

首先，我们可以通过代入来自我们模型的新赋值（ S 为真， $\neg S$ 为假）来简化子句：

$$(\neg N \vee F) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (T)$$

$$(\neg N) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L)$$

对于我们新简化的子句，我们可以检查是否提前终止。我们仍然没有足够的信息来得出所有句子都为真，或者至少有一个句子为假的结论。
接下来，我们检查纯文字。和之前一样，没有只以非否定形式出现的符号，也没有以否定形式出现的符号。
接下来，我们检查单元子句。有一个单元子句 $(\neg N)$ 。为了使整个句子为真， $(\neg N)$ 必须为真，所以 N 必须为假。
因此，我们可以在模型中将 N 赋值为假，并从仍需赋值的符号列表中移除 N ，然后再次调用DPLL。我们还可以使用在DPLL此次调用中计算出的简化子句（我们从子句中简化掉了 S ）。

我们的第三次DPLL调用如下所示：

•模型：$\{S : T, N : F\}$

•符号：$[L, M, P, Q, R]$

- *clauses*: $(\neg N) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L)$

在此次调用中，我们首先要做的是通过代入模型中的新赋值（ $(N$ 为假， $\neg N$ 为真）来简化子句：

$$(T) \wedge (M \vee Q \vee F) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee F) \wedge (\neg R \vee \neg L)$$

$$(M \vee Q) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

使用我们新简化的子句，我们检查是否提前终止，然后检查纯文字。和之前一样，我们没有找到这两者中的任何一个。
接下来，我们检查单元子句。我们没有找到只剩下一个符号的子句。

此时，我们需要尝试给一个变量赋值。根据我们固定的变量排序，我们将首先给 M 赋值，并且根据我们固定的值排序，我们将首先尝试使 M 为真。如果将 M 赋值为真会导致一个不可满足的句子，那么我们需要回溯并再次尝试将 M 赋值为假。如果将 M 赋值为假也会导致一个不可满足的句子，那么我们就会知道整个句子是不可满足的。换句话说，我们现在将对DPLL进行两次递归调用，一次将 M 赋值为真，一次将 M 赋值为假，并检查是否有一个能产生可满足的赋值。

在分支上对DPLL的第一次调用中，当 M 为真时，我们将把 M 为真添加到我们的模型中，并使用上一次调用中的简化子句：

- 模型：$\{S: T, N: F, M: T\}$
- 符号：$[L, P, Q, R]$
- 子句：$(M \vee Q) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

首先，我们通过代入模型中的新赋值（M为真）来简化子句：

$$(T \vee Q) \wedge (L \vee F) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

$$(L) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

对于新简化的子句，我们检查是否提前终止；和之前一样，没有找到。然而，我们确实找到了一个纯文字，$\neg Q$（回想一下，由于没有 Q 的实例且只有 $\neg Q$ 的实例，这算作一个纯文字）。我们将 Q 设为假，以便 $\neg Q$ 可以为真并继续。
在我们对 M 为真的分支进行的第二次DPLL调用中：

- 模型：$\{S: T, N: F, M: T, Q: F\}$
- 符号：$[L, P, R]$
- 子句：$(L) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

我们相应地简化我们的子句：

$$(L) \wedge (L \vee T) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

$$(L) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

检查是否存在早期终止和纯文字，我们未发现这两者。但我们确实找到了单位子句(L)，然后我们可以将其设为真。

在同一分支的下一次调用中，当 M 为真时，我们现在有：

- •模型：$\{S:T, N:F, M:T, Q:F, L:T\}$
- •符号：$[P, R]$
- •子句：$(L) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

让我们简化我们的子句：

$$(T) \wedge (F \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee F)$$

$$(\neg P) \wedge (R \vee P) \wedge (\neg R)$$

检查早期终止和纯文字，我们未发现任何情况。在检查单元子句时，我们发现了 $(\neg P)$ 。为了下一次DPLL调用，我们将整个表达式设为真，即将 P 设为假。

我们的下一次调用过程如下：

- •模型：$\{S:T, N:F, M:T, Q:F, L:T, P:F\}$
- •符号：$[R]$
- •子句：$(\neg P) \wedge (R \vee P) \wedge (\neg R)$

我们将 P 设为假进行简化，得到子句：

$$(T) \wedge (R \vee F) \wedge (\neg R)$$
$$(R) \wedge (\neg R)$$

我们检查是否提前终止。我们注意到这个句子同时包含 R 和 $\neg R$ ，这两者不可能同时得到满足。在这一点上，我们可以说这个句子是不可满足的。

由于 M 为真的分支以一个不可满足的句子结束，我们回溯到将 M 赋值为真之前的点，然后改为对 M 为假进行DPLL调用。我们在 M 为假的分支上的第一次DPLL调用：

- •模型：$\{S:T, N:F, M:F\}$
- •符号：$[L, P, Q, R]$
- •子句：$(M \vee Q) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

我们通过代入模型中的新赋值（ M 假）来简化子句：

$$(F \vee Q) \wedge (L \vee T) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

$$(Q) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

我们无法提前终止，并且我们没有找到任何纯文字。我们找到一个单元子句 Q ，所以我们使用 Q 为真（并从我们的符号列表中移除）再次调用DPLL。
我们在 M 为假的分支上进行的第二次DPLL调用：

- 模型：$\{S:T,N:F,M:F,Q:T\}$
- 符号：$[L,P,R]$
- 子句：$(Q)\wedge(L\vee\neg Q)\wedge(\neg L\vee\neg P)\wedge(R\vee P)\wedge(\neg R\vee\neg L)$

将新赋值（Q为真）代入我们的子句中：

$$(T)\wedge(L\vee F)\wedge(\neg L\vee\neg P)\wedge(R\vee P)\wedge(\neg R\vee\neg L)$$

$$(L)\wedge(\neg L\vee\neg P)\wedge(R\vee P)\wedge(\neg R\vee\neg L)$$

我们无法提前终止，并且没有找到任何纯文字。我们找到一个单元子句 L ，所以我们进行另一次DPLL调用，设 L 为真（并从我们的符号列表中移除）。
我们在 M 为假的分支上进行的第三次DPLL调用：

- 模型：$\{S:T,N:F,M:F,Q:T,L:T\}$
- *symbols*: $[P,R]$
- *clauses*: $(L)\wedge(\neg L\vee\neg P)\wedge(R\vee P)\wedge(\neg R\vee\neg L)$

将新赋值 (L 为真)代入我们的子句中：

$$(T)\wedge(F\vee\neg P)\wedge(R\vee P)\wedge(\neg R\vee F)$$

$$(\neg P)\wedge(R\vee P)\wedge(\neg R)$$

我们无法提前终止，并且没有找到任何纯文字。我们找到两个单元子句 $(\neg P)$ 和 $(\neg R)$ 。根据我们的变量排序，我们先选择 P ，所以我们进行另一次DPLL调用，设 P 为假（并从我们的符号列表中移除）。
我们在 M 为假的分支上进行的第三次DPLL调用：

- 模型: $\{S:T,N:F,M:F,Q:T,L:T,P:F\}$
- 符号: $[R]$
- 子句: $(\neg P)\wedge(R\vee P)\wedge(\neg R)$

将新赋值（P为假）代入我们的子句中:

$$(T)\wedge(R\vee F)\wedge(\neg R)$$

$$(R) \wedge (\neg R)$$

我们检查是否提前终止。我们注意到这个句子同时包含 R 和 $\neg R$，而这两者不能同时得到满足。在这一点上，我们可以说这个句子是不可满足的。

因为 M 的真值赋值导致句子不可满足，而 M 的假值赋值也导致句子不可满足，所以我们可以得出结论，整个句子是不可满足的，这样我们就完成了。

定理证明

另一种方法是对 KB 应用推理规则来证明 $KB \models q$。例如，如果我们的知识库包含 A 和 $A \Rightarrow B$，那么我们可以推断出 B （这条规则称为假言推理)。前面提到的两种算法利用了事实ii.)，即将 $A \wedge \neg B$ 写成合取范式并表明它是否可满足。

我们也可以使用三条推理规则来证明蕴含关系：

1. 如果我们的知识库包含 A 和 $A \Rightarrow B$，那么我们可以推断出 B （假言推理)。
2. 如果我们的知识库包含 $A \wedge B$，那么我们可以推断出 A。我们也可以推断出B。(与消除)。
3. 如果我们的知识库包含 A 和 B，那么我们可以推断出 $A \wedge B$ （归结)。

最后一条规则构成了归结算法的基础，该算法将其迭代地应用于知识库和新推断出的句子，直到推断出 q，在这种情况下我们证明了 $KB \models q$，或者没有剩余可推断的内容，在这种情况下 $KB \vDash \not{}\ q$。尽管此算法既可靠（答案将是正确的）又完备（将找到答案)，但它在最坏情况下的运行时间与知识库的大小成指数关系。

然而，在我们的知识库仅包含文字（单独的符号）和蕴含关系：$(P_1 \wedge \cdots \wedge P_n \Rightarrow Q) \equiv (\neg P_1 \vee \cdots \vee \neg P_2 \vee Q)$ 的特殊情况下，我们可以在与知识库大小成线性关系的时间内证明蕴含关系。一种算法，即前向链接，会遍历前提（左手边）已知为真的每一个蕴含语句，将结论（右手边）添加到已知事实列表中。重复此过程，直到 q 被添加到已知事实列表中，或者无法再推断出更多内容。

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
     p ← POP(agenda)
     if p = q then return true
     if inferred[p] = false then
        inferred[p] ← true
        for each clause c in KB where p is in c.PREMISE do
           decrement count[c]
           if count[c] = 0 then add c.CONCLUSION to agenda
  return false
```

图7.15 命题逻辑的前向链接算法。议程跟踪已知为真但尚未“处理”的符号。计数表跟踪每个蕴含式的前提中尚有多少未知。每当处理议程中的一个新符号 p 时，对于前提中出现 p 的每个蕴含式，计数减一（通过适当的索引可以在常量时间内轻松识别）。如果计数达到零，则蕴含式的所有前提都已知，因此其结论可以添加到议程中。最后，我们需要跟踪哪些符号已经被处理；已经在推理符号集中的符号不必再次添加到议程中。这避免了冗余工作并防止了由诸如 $P \Rightarrow Q$ 和 $Q \Rightarrow P$ 之类的蕴含式引起的循环。

前向链接：示例

假设我们有以下知识库：

1. $A \to B$
2. $A \to C$
3. $B \wedge C \to D$
4. $D \wedge E \to Q$
5. $A \wedge D \to Q$
6. A

我们想用前向链接来确定 Q 是否为真。

为了初始化算法，我们将初始化一个数字列表count。列表中的第i个数字告诉我们第 i 个子句的前提中有多少个符号。例如，第三个子句 $B \wedge C \to D$ 的前提中有2个符号 (B 和 C，所以我们列表中的第三个数字应该是2。注意，第六个子句 A 的前提中有0个符号，因为它等价于真 $\to A$。

然后，我们将初始化inferred，它是一个将每个符号映射为真/假的映射。这告诉我们哪些符号已被证明为真。最初，所有符号都将为假，因为我们尚未证明任何符号为真。

最后，我们将初始化一个符号列表agenda，它是一个我们可以证明为真但尚未传播其影响的符号列表。

例如，如果 D 在议程中，这将表明我们准备好证明 D 为真，但我们仍需检查这对其他子句有何影响。最初，议程仅包含我们直接知道为真的符号，在此处即为 A 。（换句话说，议程从前提中没有符号的任何子句开始。）
我们的起始状态如下：

- 计数： $[1,1,2,2,2,0]$
- 已推断： $\{A:F,B:F,C:F,D:F,E:F,Q:F\}$
- 议程： $[A]$

在每次迭代中，我们会从议程中弹出一个元素。在此处，我们只能弹出一个元素： A 。我们弹出的符号不是我们要分析的符号（Q），所以我们还没有完成算法。

根据推断表， A 为假。然而，由于我们刚刚将 A 从议程中移除，所以我们可以将其设置为真。

接下来，我们需要传播 A 为真的结果。对于每个前提中包含 A 的子句，我们将其相应的计数减1，以表明前提中需要检查的符号少了一个。在这个例子中，子句1、2和5的前提中包含 A ，所以我们将计数中的元素1、2和5减1。

最后，我们检查是否有任何子句的计数达到了0。我们注意到子句1和2出现了这种情况。这表明子句1和子句2中的每个前提都已得到满足，所以子句1和2中的结论可以被推断出来。例如，在子句1中，所有前提（这里只有 A ）都已得到满足，所以结论 B 可以被推断出来。我们将把子句1和2中的结论添加到议程中。
在第0次迭代之后，我们的算法如下所示：

- 计数: $[0,0,2,2,1,0]$
- 推断值: $\{A:T,B:F,C:F,D:F,E:F,Q:F\}$
- 议程: $[B,C]$

在下一次迭代中，我们将从议程中弹出一个元素。这里我们选择弹出 B 。我们弹出的符号不是我们想要分析的符号(Q)，所以我们的算法还没有完成。
根据推断表， B 为假。然而，由于我们刚刚从议程中弹出了 B ，我们能够将其设置为真。

接下来，我们需要传播 B 为真的结果。 B 在前提中的唯一子句是子句3。我们必须减少其相应的计数。
最后，我们检查是否有任何子句的计数达到了0。没有子句新达到计数为0的情况，所以我们无法得出任何新结论，也不能在议程中添加任何新内容。
在第1次迭代之后，我们的算法如下所示：

- 计数： $[0,0,1,2,1,0]$
- 推断： $\{A:T,B:T,C:F,D:F,E:F,Q:F\}$
- 议程： $[C]$

接下来，我们将从议程中移除 C （由于它不是 Q ，所以算法尚未完成）。我们可以在推断列表中将 C 设置为真。
为了传播 C 为真的结果，我们将子句3的计数减1（子句3是前提中唯一包含 C 的子句）。

子句3的计数新达到了0，所以我们可以将它的结论 D 添加到议程中。
在第2次迭代之后，我们的算法如下所示：

- 计数：$[0, 0, 0, 2, 1, 0]$
- 推断：$\{A : T, B : T, C : T, D : F, E : F, Q : F\}$
- 议程：$[D]$

接下来，我们将从议程中移除 D （不是 Q ，所以算法尚未完成）。我们可以在推断列表上将 D 设置为真。
为了传播 D 为真的结果，我们将子句4和5的计数减1（前提中包含 D ）。

子句5的计数新达到了0，所以我们将其结论 Q 添加到议程中。
在第3次迭代之后，我们的算法如下所示：

- 计数：$[0, 0, 0, 1, 0, 0]$
- 推断：$\{A : T, B : T, C : T, D : T, E : F, Q : F\}$
- 议程：$[Q]$

接下来，我们将从议程中移除 Q 。这是我们想要评估的符号，将其从议程中移除表明它已被证明为真。我们得出结论， Q 为真并结束算法。