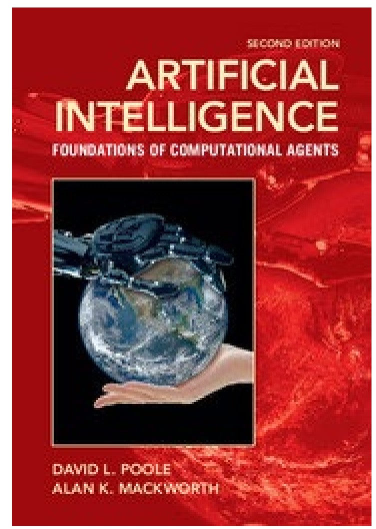
Chapter 5

Propositions and Inference

Textbook: *Artificial Intelligence Foundations of Computational Agents*, David L. Poole and Alan K Mackworth, Cambridge University Press.



Introduction

- This chapter presents several **reasoning formalisms** that use **propositions**.
- Here, a simple form of a **knowledge base (KB)** is presented, composed of **facts** and **rules** related to a problem world.
  - An **agent** can use such a **KB**, together with its **observations**, to determine the ***truth*** in the world.
- When **queried** about "*what must be **true** given a **KB,"*** it **answers** the **query** without enumerating all the possible worlds.

Propositions

- **Statements** about the world provide **constraints** about *what could be **true***
  - **Constraints** could be specified extensionally as tables of legal assignments to ***variables*** or ***formulas***.
- There are several reasons for using **<u>propositions</u>** for specifying **constraints** and **queries**:
  - It gives a more concise and readable logical statement about the **relationship** among **variables**.
  - The form of knowledge can be exploited to make **reasoning** more efficient.
    - They are **modular**, so small changes to the problem result in minor changes to the **KB**.

Syntax of Propositional Calculus

- A **proposition** is a ***sentence*** written in a language with a **truth value** (either ***true*** or ***false***) in a world.
  - A **proposition** is built from **atomic propositions** using **logical connectives**.
  - We use the convention that **propositions** consist of *letters*, *digits,* and the *underscore* (‘_’) and start with a **<u>lowercase letter</u>**.
  - For example, ***ai_is_fun***, ***live_outside***, and ***sunny*** are all propositional atoms.
- **Propositions** can be built from more straightforward propositions using **logical connectives**
  - A proposition or logical formula is either an **atomic proposition** or a **compound proposition**

Syntax of Propositional Calculus

- a **compound proposition** of the form

$\neg p$	"not p"	**negation** of p
$p \wedge q$	"p and q"	**conjunction** of p and q
$p \vee q$	"p or q"	**disjunction** of p and q
$p \rightarrow q$	"p implies q"	**implication** of q from p
$p \leftarrow q$	"p if q"	**implication** of p from q
$p \leftrightarrow q$	"p if and only if q"	**equivalence** of p and q

where p and q are propositions.

The operators $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$ and $\leftrightarrow$ are **logical connectives**.

Syntax of Propositional Calculus

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \leftarrow q$	$p \rightarrow q$	$p \leftrightarrow q$
true	*true*	*false*	*true*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*	*false*	*true*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*	*true*

Figure 5.1: Truth table defining $\neg$, $\wedge$, $\vee$, $\leftarrow$, $\rightarrow$, and $\leftrightarrow$

Semantics of the Propositional Calculus

- **Semantics** defines the **meaning** of the sentences of a language.
- When the **sentences** are about a world, **semantics** specifies ***how to put symbols of the language into correspondence with the world***.
- The **truth of atoms** gives the **truth** of other propositions in interpretations (see **Figure 5.1**).
- An interpretation consists of a **function** ***π*** that maps **atoms** to {***true***, ***false***}.
  - If ***π*(*a*)** = ***true***, atom ***a*** is ***true***
  - If ***π*(*a*) = *false***, atom ***a*** is ***false***
  - Suppose there are **three** atoms: ***ai_is_fun***, ***happy***, and ***light_on***; ***π*(*ai_is_fun*) = *true***, ***π*(*happy*) = *false***, ***π*(*light_on*) = *true***

Semantics of the Propositional Calculus

Example 5.20 Consider the knowledge base KB_2:

$$false \leftarrow a \wedge b.$$

$$a \leftarrow c.$$

$$b \leftarrow d.$$

$$b \leftarrow e.$$

Either c is false or d is false in every model of KB_2. If they were both true in some model I of KB_2, both a and b would be true in I, so the first clause would be false in I, a contradiction to I being a model of KB_2. Similarly, either c is false or e is false in every model of KB_2. Thus,

$$KB_2 \models \neg c \vee \neg d$$

$$KB_2 \models \neg c \vee \neg e.$$

Semantics of the Propositional Calculus

- A **KB** is a set of **true** propositions.
- An element of the **KB** is an **axiom**.
- A **model** of a **KB** is an interpretation in which all the propositions in **KB** are ***true***.
- If a knowledge base is **KB**, and ***g*** is a **proposition**, **KB |= *g*** is ***g*** logically follows from **KB**, or **KB entails *g*.**
- The logical entailment "**KB |= *g***" is a semantic relation between a set of propositions in **KB** and an external proposition, ***g***.
- Both **KB** and ***g*** are **symbolic** so that they can be represented in a computer.

Propositional Constraints

- The class of **propositional satisfiability problems** has:
  - **Boolean variables:** If ***X*** is a Boolean variable, ***X*** = ***false*** as **¬*x***. Thus, given a Boolean variable ***Happy***, the proposition ***happy*** means ***Happy*** = ***true***, and **¬*happy*** means ***Happy*** = ***false***.
  - **Clausal constraints:** a **clause** is **disjoints of atoms** and is expressed as ($l_1 \vee l_2 \vee \ldots \vee l_k$), where each l_i is **literal**.
    - A **literal** is an **atom** or the **negation** of an **atom**
    - A **clause** is satisfied in a possible world if at least one of the **literals** that make up the clause is ***true*** in that possible world.

Propositional Constraints

Example 5.4 The clause $\boxed{happy \lor sad \lor \neg living}$ is a constraint among the variables *Happy*, *Sad*, and *Living*, which is true if *Happy* has value *true*, *Sad* has value *true*, or *Living* has value *false*. The atoms *happy* and *sad* appear positively in the clause, and *living* appears negatively in the clause.

The assignment $\boxed{\neg happy, \neg sad, living}$ violates the constraint of clause $happy \lor sad \lor \neg living$. It is the only assignment of these three variables that violates this clause.

Propositional Constraints

- It is possible to convert any finite **CSP** into a **Propositional Satisfiable Problem** (**PSP**):
  - A **CSP** variable ***Y*** with domain $\{v_1, \ldots, v_k\}$ can be converted into ***k*** **Boolean variables** $\{Y_1, \ldots, Y_k\}$, where Y_i is ***true*** when ***Y*** has value v_i and is ***false*** otherwise. Each Y_i is called an **indicator variable**. Thus ***k*** atoms, $y_1, \ldots, yk$, are used to represent the **CSP variable**.
  - There are **constraints** that specify that y_i and y_j cannot both be ***true*** when ($i \neq j$). There is a **constraint** that one of the y_i must be ***true***. Thus, the **KB** contains the **clauses:** $\neg y_i \vee \neg y_j$ for $i < j$ and $y_1 \vee \cdot \cdot \cdot \vee y_k$.
  - There is a **clause** for each ***false*** assignment in each **constraint**; for example, the clauses ($a \vee b \vee c$) and ($a \vee b \vee \neg c$) can be combined with ($a \vee b$).

Clausal Form for Consistency Algorithms

- **Consistency algorithms** can be made more efficient for **propositional satisfiability problems** (**PSP**)
  - When there are only *two values*, **pruning** a value from the domain is equivalent to assigning the opposite value
  - Thus, if ***X*** has domain {***true***, ***false***}, pruning ***true*** from the domain of ***X*** is the same as assigning ***X*** to have the value ***false***
- **Arc consistency** can be used to **prune** the set of values and the set of constraints.

Clausal Form for Consistency Algorithms

- Assigning a value to a **Boolean variable** can simplify the **set of constraints:**
  - If ***X*** is assigned ***true***, all of the **clauses** with ***X*** = ***true*** become redundant; they are automatically satisfied. These clauses can be removed. Similarly, if ***X*** is assigned ***false***, clauses containing ***X*** = ***false*** can be removed.
  - If ***X*** is assigned ***true***, any **clause** with ***X*** = ***false*** can be simplified by removing ***X*** = ***false*** from the **clause**. Similarly, if ***X*** is assigned ***false***, then ***X*** = ***true*** can be removed from any **clause** it appears in. This step is called **unit resolution**.
  - After **pruning the clauses**, there is a **clause** that contains just one assignment, ***Y*** = ***v***, the other value can be removed from the domain of ***Y.*** This is a form of **arc consistency**
    - ***If all of the assignments are removed from a clause, the constraints are unsatisfiable.***

Clausal Form for Consistency Algorithms

- If a **variable** has the same value in all remaining **clauses**, and the algorithm must only find **one model**, it can assign that value to that variable
  – For example, if variable ***Y*** only appears as ***Y*** = ***true*** (i.e., **¬*y*** is not in any clause), then ***Y*** can be assigned the value ***true***.
  – That assignment does not remove all of the models;
  – A variable that has only one value in all of the clauses is called a **pure literal**.

Propositional Definite Clauses

- The syntax of **definite propositional clauses** is defined as follows:
  - An atomic proposition or atom is the same as in propositional calculus.
  - A **definite clause** is of the form: $\boldsymbol{a \leftarrow b_1 \wedge \ldots \wedge b_m}$, where $\boldsymbol{a}$ is **the head of the clause and is always positive**, and $\boldsymbol{a}$ and each $\boldsymbol{b_i}$ are called **atoms**. It can be read as "$\boldsymbol{a}$ if $\boldsymbol{b_1}$ and $\boldsymbol{b_2}$. . . and $\boldsymbol{b_m}$"
    - $\boldsymbol{(a \leftarrow (b_1 \wedge b_2 \wedge \ldots\ldots \wedge b_m)) = (a \vee \neg b_1 \vee \neg b_2 \ldots\ldots\ldots \vee \neg b_m)}$
      - If $\boldsymbol{m > 0}$, the clause is called a **rule**, where $\boldsymbol{b_1 \wedge \ldots \wedge b_m}$ is the **body** of the clause ("**if**" part of the rule).
      - If $\boldsymbol{m = 0}$, the **arrow can be omitted,** and the clause is called an **atomic clause** or a **fact** (it is the **clause with an empty body)**.
  - A **knowledge base (KB)** is a set of **definite clauses**.

Propositional Definite Clauses

Example 5.6 The elements of the knowledge base in Example 5.20 are all definite clauses.

The following are *not* definite clauses:

$\neg apple\_is\_eaten.$

$apple\_is\_eaten \wedge bird\_eats\_apple.$

$sam\_is\_in\_room \wedge night\_time \leftarrow switch\_1\_is\_up.$

$Apple\_is\_eaten \leftarrow Bird\_eats\_apple.$

$happy \vee sad \vee \neg alive.$

The fourth statement is not a definite clause because an atom must start with a lower-case letter.

Propositional Definite Clauses

- Note that a **definite clause** is a restricted form of a **clause.** For example, the **definite clause**

  ($\boldsymbol{a} \leftarrow \boldsymbol{b} \wedge \boldsymbol{c} \wedge \boldsymbol{d}$) is **equivalent** to the **clause** ($\boldsymbol{a} \vee \neg\boldsymbol{b} \vee \neg\boldsymbol{c} \vee \neg\boldsymbol{d}$). That is, ($\boldsymbol{a} \leftarrow \boldsymbol{b} \wedge \boldsymbol{c} \wedge \boldsymbol{d}$) **=** ($\boldsymbol{a} \vee \neg\boldsymbol{b} \vee \neg\boldsymbol{c} \vee \neg\boldsymbol{d}$)

- In general, <u>a **definite clause is equivalent to a clause with precisely one positive literal**</u>.

Propositional Definite Clauses

- **Example 5.7** Consider how to axiomatize the electrical environment of **Figure 5.2** following the methodology for the user's view of semantics.

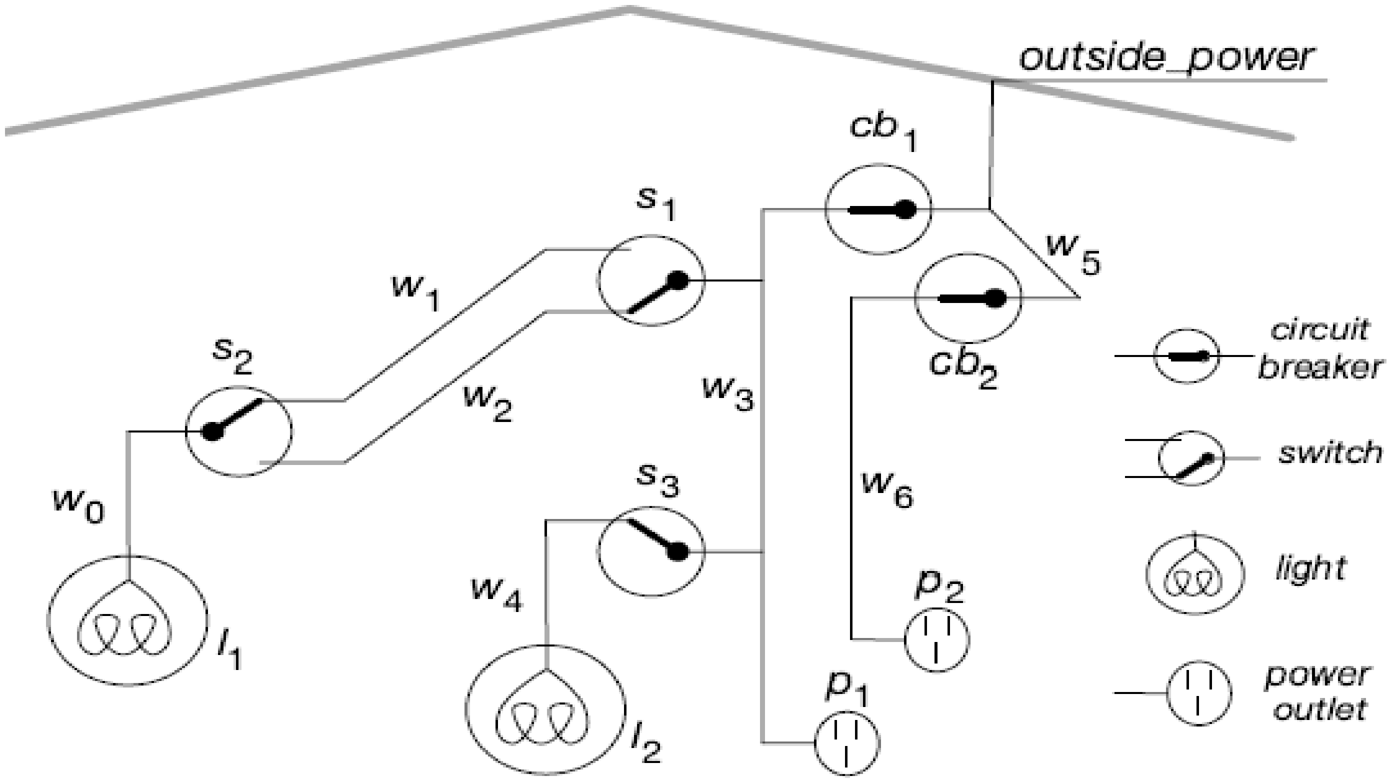


Figure 5.2: An electrical environment with components named

Propositional Definite Clauses

- The designer may look at part of the domain and know that **light l_1** is **live** if **wire w_0** is **live** because they are connected but may not know whether w_0 is **live**. Such knowledge is expressible in terms of rules.
- The **KB** consists of all of the **definite clauses**, whether specified as background knowledge or as observations.

$$live\_l_1 \leftarrow live\_w_0.$$
$$live\_w_0 \leftarrow live\_w_1 \wedge up\_s_2.$$
$$live\_w_0 \leftarrow live\_w_2 \wedge down\_s_2.$$
$$live\_w_1 \leftarrow live\_w_3 \wedge up\_s_1.$$
$$live\_w_2 \leftarrow live\_w_3 \wedge down\_s_1.$$
$$live\_l_2 \leftarrow live\_w_4.$$
$$live\_w_4 \leftarrow live\_w_3 \wedge up\_s_3.$$
$$live\_p_1 \leftarrow live\_w_3.$$
$$live\_w_3 \leftarrow live\_w_5 \wedge ok\_cb_1.$$
$$live\_p_2 \leftarrow live\_w_6.$$
$$live\_w_6 \leftarrow live\_w_5 \wedge ok\_cb_2.$$
$$live\_w_5 \leftarrow live\_outside.$$
$$lit\_l_1 \leftarrow light\_l_1 \wedge live\_l_1 \wedge ok\_l_1.$$
$$lit\_l_2 \leftarrow light\_l_2 \wedge live\_l_2 \wedge ok\_l_2.$$

Questions and Answers

- A **query** is a way of asking whether a proposition is a **logical consequence** of a **KB**.
- A **query** is a question with the answer "***yes***" if the body is a logical **consequence** of the **KB** or
- the answer "***no***" if the body is **not a consequence** of the **KB**.

Questions and Answers

Example 5.8 Once the computer has been told the knowledge base of Example 5.7 (page 183), it can answer queries such as

ask *light_l*$_1$.

for which the answer is *yes*. The query

ask *light_l*$_6$.

has answer *no*. The computer does not have enough information to know whether or not l_6 is a light. The query

ask *lit_l*$_2$.

has answer *yes*. This atom is true in all models.

The user can interpret this answer with respect to the intended interpretation.

Proofs

- A **theorem** is a **provable proposition**.
  - A **proof** is a mechanically derivable demonstration that a proposition logically follows from a **KB**.
- A **proof procedure** is a – possibly **non-deterministic algorithm** for deriving consequences of a **KB**
  - A **proof procedure** is **sound** concerning semantics if everything that can be derived from a **KB** is a **logical consequence** of the **KB**.
  - A **proof procedure** is **complete** concerning semantics if there is **proof of each logical consequence** of the **KB**.
  - Two ways to **construct proofs** for definite propositional clauses:
    - a **bottom-up procedure** and
    - a **top-down procedure**.

Bottom-Up Proof Procedure

- A **bottom-up proof procedure** can be used to derive **all logical consequences of a KB**
  - The **bottom-up proof procedure** builds on atoms that have already been established.
  - We say that a **bottom-up procedure** is **forward chaining** on the **definite clauses**, in the sense of g***oing forward from what is known*** rather than going backward from the query.
  - The general idea is based on **one rule of derivation** is a generalized form of the rule of **inference** called **modus ponens**:
    - If "$h \leftarrow a_1 \wedge . . . \wedge a_m$" is a **definite clause** in the **KB**, and each a_i has been derived, then h can be derived.
- **Figure 5.3** gives a procedure for computing the **consequence set C** of a set KB of **definite clauses**.

Bottom-Up Proof Procedure

- **Modus Ponens**: Create a conclusion from a rule and a fact. For example, consider the following **rule** and a **fact**:

  Rule: **(a ← b)**

  Fact: **b**

  Conclusion: **a**

Bottom-Up Proof Procedure

procedure *Prove_DC_BU*(KB)
Inputs
KB: a set of definite clauses
Output
Set of all atoms that are logical consequences of KB
Local
C is a set of atoms
$C := \{\}$
repeat
select "$h \leftarrow a_1 \wedge \ldots \wedge a_m$" in KB where $a_i \in C$ for all i, and $h \notin C$
$C := C \cup \{h\}$
until no more definite clauses can be selected
return C

Figure 5.3: Bottom-up proof procedure for computing consequences of KB

Bottom-Up Proof Procedure

Example 5.9 Suppose the system is given the knowledge base *KB*: $a \leftarrow b \wedge c.$ $b \leftarrow d \wedge e.$ $b \leftarrow g \wedge e.$ $c \leftarrow e.$ $d.$ $e.$ $f \leftarrow a \wedge g.$	One trace of the value assigned to C in the bottom-up procedure is $\{\}$ $\{d\}$ $\{e,d\}$ $\{c,e,d\}$ $\{b,c,e,d\}$ $\{a,b,c,e,d\}.$ The algorithm terminates with $C = \{a,b,c,e,d\}$. Thus, $KB \vdash a$, $KB \vdash b$, and so on.

The last rule in KB is never used. **The bottom-up proof procedure** cannot derive ***f*** or ***g***. This is as it should be because there is a model of the KB in which ***f*** and ***g*** are both ***false***.

Top-Down Proof Procedure

- An alternative **proof method** is to search **backward** or **top-down** from a **query** to determine whether it is a logical consequence of the given **definite clauses**.
- **Top-down procedure** is also called **backward chaining**
- The **top-down proof procedure** can be understood in the **answer clause** (it is a definite close with the head "**yes**").
  - An **answer clause** is of the form: $\boldsymbol{yes \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_m}$ where ***yes*** is a special atom.
  - Intuitively, ***yes*** is going to be ***true*** exactly when the answer to the query is "***yes***."
- If the **query** is ($\boldsymbol{ask\ q_1 \wedge \ldots \wedge q_m}$), then the initial **answer clause** is $\boldsymbol{yes \leftarrow q_1 \wedge \ldots \wedge q_m}$

Top-Down Proof Procedure

- Given an **answer clause**, the **top-down algorithm** selects an atom in the body of the **answer clause**.
- Suppose it selects $\boldsymbol{a_1}$**.** The atom selected is called a **subgoal.**
- The algorithm proceeds by doing steps of **resolution**
- In one step of **resolution**, it chooses a **definite clause** in **KB** with $\boldsymbol{a_1}$ as the **head**. If there is no such **clause**, the algorithm fails.
- The **resolvent** of the above **answer clause** on the selection $\boldsymbol{a_1}$ with the **definite clause** ($\boldsymbol{a_1 \leftarrow b_1 \wedge \ldots \wedge b_p}$) is the **answer clause**: ($\boldsymbol{yes \leftarrow b_1 \wedge \ldots \wedge b_p \wedge a_2 \wedge \ldots \wedge a_m}$)
- That is, the **subgoal** in the **answer clause** is replaced by the body of the chosen **definite clause**.
- <u>An **answer** is an **answer clause** with an **empty body** (***m* = 0**), it is the **answer clause** (***yes*** ←) ***.***</u>

Top-Down Proof Procedure

- **Figure 5.4** specifies a **non-deterministic procedure** for **solving a query**. It follows the definition of a derivation.
- In this procedure, ***G*** is the set of **atoms** in the body of the **answer clause**.
- The procedure is **nondeterministic: line 12** has to choose a **definite clause** to resolve against.
- If there are choices that result in ***G*** being the ***empty set***, the algorithm returns ***yes***; otherwise, it ***fails***, and the answer is ***no***.
- This algorithm treats the body of a clause as a set of atoms, and ***G*** is also a set of atoms.
- An alternative is to have ***G*** as an ordered list of atoms, perhaps with an atom appearing more than once.

Top-Down Proof Procedure

```
non-deterministic procedure Prove_DC_TD(KB, Query)
    Inputs
        KB: a set of definite clauses
        Query: a set of atoms to prove
    Output
        yes if KB ⊨ Query and the procedure fails otherwise
    Local
        G is a set of atoms
    G := Query
    repeat
        select an atom a in G
        choose definite clause "a ← B" in KB with a as head
        G := B ∪ (G \ {a})
    until G = {}
    return yes
```

Figure 5.4: Top-down definite clause proof procedure

Top-Down Proof Procedure

Example 5.10 Suppose the system is given the knowledge base:

$$a \leftarrow b \wedge c.$$
$$b \leftarrow d \wedge e.$$
$$b \leftarrow g \wedge e.$$
$$c \leftarrow e.$$
$$d.$$
$$e.$$
$$f \leftarrow a \wedge g.$$

It is asked the query: ask a.

The following shows a derivation that corresponds to a sequence of assignments to G in the repeat loop of Figure 5.4. Here we have written G in the form of an answer clause, and always selected the leftmost atom in the body:

Top-Down Proof Procedure

$$yes \leftarrow a$$
$$yes \leftarrow b \wedge c$$
$$yes \leftarrow d \wedge e \wedge c$$
$$yes \leftarrow e \wedge c$$
$$yes \leftarrow c$$
$$yes \leftarrow e$$
$$yes \leftarrow$$

The following shows a sequence of choices, where the second definite clause for b was chosen. This choice does not lead to a proof.

$$yes \leftarrow a$$
$$yes \leftarrow b \wedge c$$
$$yes \leftarrow g \wedge e \wedge c$$

If g is selected, there are no rules that can be chosen. This proof attempt is said to *fail*.

Top-Down Proof Procedure

- The **non-deterministic top-down** algorithm of **Figure 5.4,** together with a selection strategy, induces a **search graph**.
- Each **node** in the search graph represents an **answer clause**.
- The neighbors of a **node** ($\boldsymbol{yes} \leftarrow \boldsymbol{a_1} \land \ldots \land \boldsymbol{a_m}$), where $\boldsymbol{a_1}$ is the **selected atom**, represent all of the possible answer clauses obtained by resolving on $\boldsymbol{a_1}$.
- There is a neighbor for each **definite clause** whose head is $\boldsymbol{a_1}$.
- The **goal nodes** of the search are of the form $\boldsymbol{yes} \leftarrow$.

Top-Down Proof Procedure

Example 5.11 Given the knowledge base

$a \leftarrow b \land c.$	$a \leftarrow g.$	$a \leftarrow h.$
$b \leftarrow j.$	$b \leftarrow k.$	$d \leftarrow m.$
$d \leftarrow p.$	$f \leftarrow m.$	$f \leftarrow p.$
$g \leftarrow m.$	$g \leftarrow f.$	$k \leftarrow m.$
$h \leftarrow m.$	$p.$	

and the query

ask $a \land d.$

the search graph for an SLD derivation, assuming the leftmost atom is selected in each answer clause, is shown in Figure 5.5.

Top-Down Proof Procedure

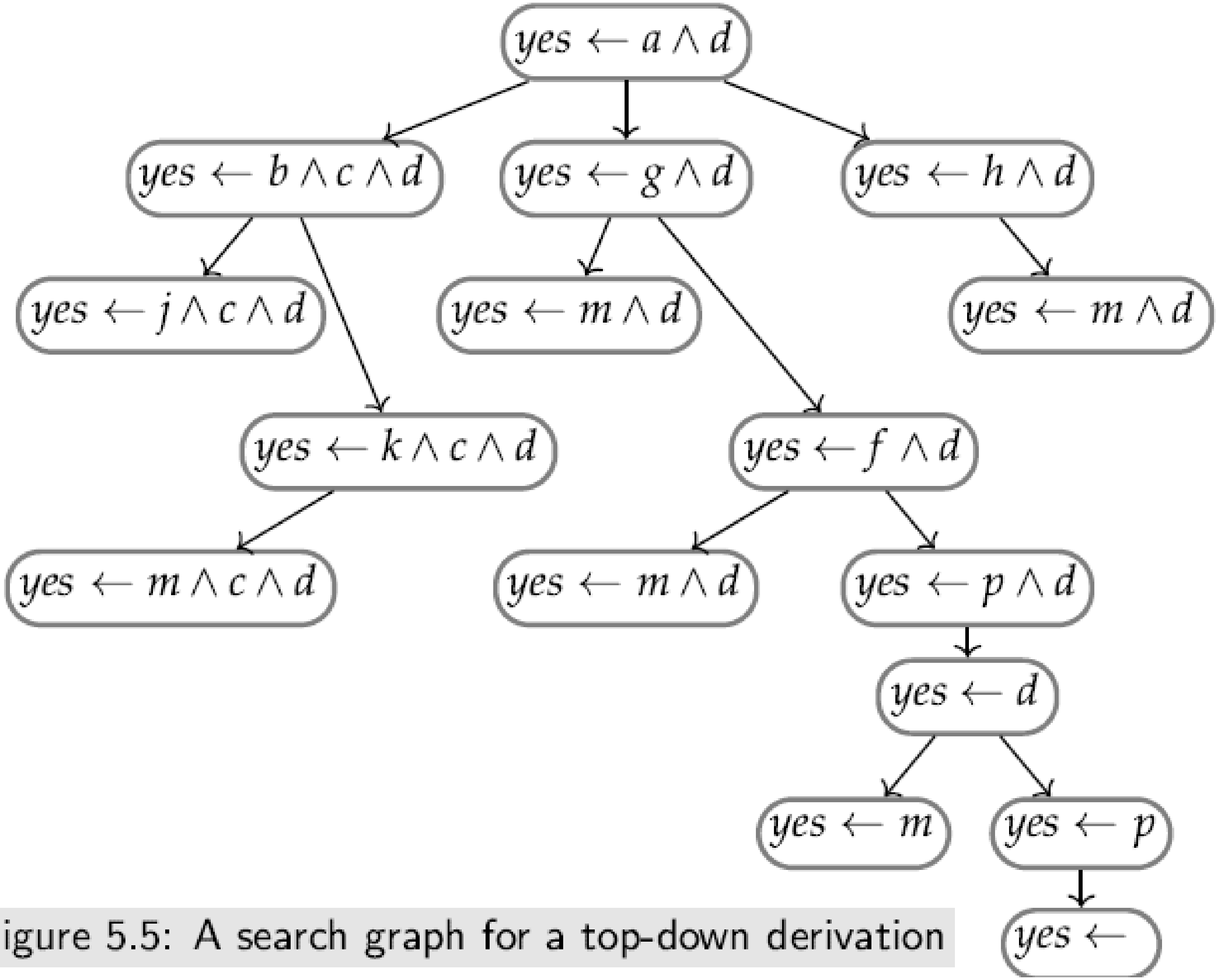


Figure 5.5: A search graph for a top-down derivation

Top-Down Proof Procedure

- It is possible that the proof procedure can get into an **infinite loop**, as in the following example (without cycle pruning).

Example 5.12 Consider the knowledge base and query:

$g \leftarrow a.$

$a \leftarrow b.$

$b \leftarrow a.$

$g \leftarrow c.$

$c.$

ask g.

Atoms g and c are the only atomic logical consequences of this knowledge base, and the bottom-up proof procedure will halt with fixed point $\{c, g\}$. However, the top-down proof procedure with a depth-first search will go on indefinitely, and not halt if the first clause for g is chosen, and there is no cycle pruning.

Background Knowledge and Observations

- Assume an <u>**observation**</u> is a set of **atomic propositions** which are **implicitly conjoined**.
- <u>**Observations do not provide rules directly**</u>.
- **Observation** is like **background knowledge** in a **KB,** allowing the agent to do something useful.
- At **design time** or **offline**, the **information arrives online** as **observations** from *users*, *sensors*, and *external knowledge sources*
  - For example, a **medical diagnosis program** may have knowledge represented as **definite clauses** about the possible diseases and symptoms, but *it would not know the actual symptoms manifested by a particular patient*
  - A simple way to acquire information from a user is to incorporate an **ask-the-user** mechanism into the **top-down proof procedure**.

Knowledge-Level Explanation

- The explicit use of semantics allows explanation and debugging at the **knowledge level**
- To make a system usable by people, the system cannot just give an answer and expect the user to believe it
  - Consider the case of a system **advising doctors** who are legally responsible for the treatment they carry out based on the diagnosis.
    - The doctors must be convinced that the diagnosis is appropriate.
    - The system must be able to justify that its answer is correct.
    - The exact mechanism can explain how the system found a result and debugged the **KB**.

Knowledge-Level Explanation

- **Three complementary** means of interrogation are used to explain the **relevant knowledge in a KB:**
  - ***(1) how the question*** is used to describe ***how an answer was proved***,
  - ***(2) why the question*** is used to ask the system ***why it is asking the user a question***, and
  - ***(3) why not question*** is used to ask ***why an atom was not proven***.
- To explain how an answer was proved, a "**how**" question can be asked by a user when the system has returned the answer
  - The system provides the **definite clause** used to deduce the answer.
  - The user can ask "**why**" when asked a question.
  - The system **replies** by giving the rule that produced the question.
  - The user can then ask **why** the head of that rule was being proved.

Knowledge-Level Explanation

- **How a System can Prove an Atom?**
- The first explanation procedure allows the user to ask "**how**" an atom was proved.
- If there is a proof for $\boldsymbol{g}$, either $\boldsymbol{g}$ must be an **atomic clause** or there must be a rule:

  $\boldsymbol{g} \leftarrow (\boldsymbol{a_1} \wedge \ldots \wedge \boldsymbol{a_k})$ such that each $\boldsymbol{a_i}$ has been proved.

Proving by Contradiction

- **Definite clauses** can be used in a proof by **contradiction** by allowing rules that give **contradictions**.
  - The **definite clause language** does not allow a contradiction to be stated.
- An **integrity constraint** is a clause of the form: $\boldsymbol{false} \leftarrow \boldsymbol{a_1} \land \ldots \land \boldsymbol{a_k}$, where the $\boldsymbol{a_i}$ are atoms and ***false*** is a particular atom that is ***false*** in all interpretations.
- **A Horn clause** is either a **definite clause** or an **integrity constraint**.
- That is, a **Horn clause** has either ***false*** or a **normal atom** as its **head**.

Proving by Contradiction

- **Integrity constraints** allow the system to prove that some **conjunction of atoms is *false*** in all models of a **KB**.
- **¬*p*** is the **negation of *p***, which is ***true*** in an interpretation when ***p*** is ***false*** in that interpretation,
- and ***p*** ∨ ***q*** is the **disjunction** of ***p*** and ***q***, which is ***true*** in an interpretation if ***p*** is ***true*** or ***q*** is ***true,*** or both are ***true*** in the interpretation.
- The **integrity constraint: *false*** $\leftarrow a_1 \land \ldots \land a_k$ is logically equivalent to $\neg a_1 \lor \ldots \lor \neg a_k$.
- A **Horn clause KB** can imply **negations of atoms**, as shown in **Example 5.19**.

Proving by Contradiction

Example 5.19 Consider the knowledge base KB_1:

$$false \leftarrow a \wedge b.$$

$$a \leftarrow c.$$

$$b \leftarrow c.$$

The atom c is false in all models of KB_1. To see this, suppose instead that c is true in model I of KB_1. Then a and b would both be true in I (otherwise I would not be a model of KB_1). Because *false* is false in I and a and b are true in I, the first clause is false in I, a contradiction to I being a model of KB_1. Thus $\neg c$ is true in all models of KB_1, which can be written as

$$KB_1 \models \neg c$$

Proving by Contradiction

- Although the language of **Horn clauses** does not allow **disjunctions** and **negations** to be input, disjunctions of negations of atoms can be derived, as the following example shows.

Example 5.20 Consider the knowledge base KB_2:

$$false \leftarrow a \wedge b.$$

$$a \leftarrow c.$$

$$b \leftarrow d.$$

$$b \leftarrow e.$$

Either c is false or d is false in every model of KB_2. If they were both true in some model I of KB_2, both a and b would be true in I, so the first clause would be false in I, a contradiction to I being a model of KB_2. Similarly, either c is false or e is false in every model of KB_2. Thus,

$$KB_2 \models \neg c \vee \neg d$$

$$KB_2 \models \neg c \vee \neg e.$$

Proving by Contradiction

- A set of clauses is **unsatisfiable** if it has **no models**.
- A set of clauses is provably **inconsistent** concerning a **proof procedure**.
- If a proof procedure is **sound** and **complete**, a set of clauses is provably **inconsistent** if and only if it is **unsatisfiable**.
- It is always possible to find **a model** for a set of **definite clauses**.
- The interpretation with all atoms ***true*** is a model of any set of **definite clauses**.
- Thus, **<u>a definite-clause KB is always satisfiable</u>**.
- However, a set of **<u>Horn clauses can be unsatisfiable.</u>**

Abduction

- **Abduction** is a form of reasoning where ***assumptions are made to explain observations.***
  - For example, *if an agent observes that some light is not working, it hypothesizes what is happening in the world to explain why the light is not working.*
- In **abduction**, an agent **hypothesizes** what may be ***true*** about an observed case.
- An agent determines what implies its observations – what could be ***true*** to make them ***true***.
- To formalize **abduction**, we use **Horn clauses** and **assumable**. The system is given:
  - a **KB**, a set of **Horn clauses**, and a **set A of atoms**, called the **assumable**, are the building blocks of **hypotheses** (see **Example 5.33**).

Abduction

Example 5.33 Consider the following simplistic knowledge base and assumables for a diagnostic assistant:

$$bronchitis \leftarrow influenza.$$

$$bronchitis \leftarrow smokes.$$

$$coughing \leftarrow bronchitis.$$

$$wheezing \leftarrow bronchitis.$$

$$fever \leftarrow influenza.$$

$$fever \leftarrow infection.$$

$$soreThroat \leftarrow influenza.$$

$$false \leftarrow smokes \wedge nonsmoker.$$

assumable $smokes, nonsmoker, influenza, infection.$

Abduction

- If the agent observes ***wheezing***, there are two minimal explanations: {***influenza***} and {***smokes***}
- These explanations imply ***bronchitis*** and ***coughing***.
- If (***wheezing*** ∧ ***fever***) is observed, the **minimal explanations** are

  {***influenza***} and {***smokes***, ***infection***}.
- If (***wheezing*** ∧ ***nonsmoker***) was observed, there is one **minimal explanation:** {***influenza***, ***nonsmoker***}.
- The other explanation of ***wheezing*** is inconsistent with being a **non-smoker**.

Abduction

Example 5.34 Consider the knowledge base:

$alarm \leftarrow tampering.$

$alarm \leftarrow fire.$

$smoke \leftarrow fire.$

If *alarm* is observed, there are two minimal explanations:

$\{tampering\}$ and $\{fire\}$.

If $alarm \wedge smoke$ is observed, there is one minimal explanation:

$\{fire\}$.

Notice how, when *smoke* is observed, there is no need to hypothesize *tampering* to explain *alarm*; it has been **explained away** by *fire*.