AI_Module5

Syllabus :
Inference in First Order Logic: Backward Chaining, Resolution
Classical Planning: Definition of Classical Planning, Algorithms for Planning as State-Space Search, Planning Graphs
Chapter 9-9.4, 9.5
Chapter 10- 10.1,10.2,10.3

Topics:

1. **Inference in First Order Logic**
    1. **Backward Chaining,**
    2. **Resolution**
2. **Classical Planning**
    1. **Definition of Classical Planning**
    2. **Algorithms for Planning as State Space Search**
    3. **Planning Graphs**
    4. **Logic Programming**

5.1.1 Backward Chaining

Backward chaining is a reasoning method that starts with the goal and works backward through the inference rules to find out whether the goal can be satisfied by the known facts.

It's essentially **goal-driven reasoning,** where the system seeks to prove the hypothesis by breaking it down into subgoals and verifying if the premises support them.

Example : Consider the following knowledge base representing a simple diagnostic system:

1. *If a patient has a fever, it might be a cold.*
2. *If a patient has a sore throat, it might be strep throat.*
3. *If a patient has a fever and a sore throat, they should see a doctor.*

Given the facts:

- **The patient has a fever.**
- **The patient has a sore throat.**

Backward chaining would proceed as follows:

- **Start with the goal**: Should the patient see a doctor?
- **Check the third rule**: Does the patient have a cold and a sore throat? **Yes.**
- **Check the first and second rules**: Does the patient have a fever and sore throat? **Yes**.
- **The goal is satisfied**: The patient should see a doctor.

Backward chaining is useful when there is a specific goal to be achieved, and the system can efficiently backtrack through the inference rules to determine whether the goal can be satisfied.

5.1.1.1 Backward Chaining: Algorithm

These algorithms work backward from the goal, chaining through rules to find known facts that support the proof.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ'

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first, rest ← FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ'' in FOL-BC-AND(KB, rest, θ') do
        yield θ''
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

Source Book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson,2015

5.1.1.2 Overview of the Algorithm

1. **Goal**:
   - The purpose of the algorithm is to determine whether a query (goal) can be derived from a given knowledge base (KB).
2. **Process**:
   - It uses backward chaining, meaning it starts with the goal and works backward by looking for rules or facts in the knowledge base that could satisfy the goal.
   - The algorithm returns substitutions (values or variables) that make the query true.
3. **Key Components**:
   - **FOL-BC-ASK**: This is the main function that starts the backward-chaining process by calling `FOL-BC-OR`.

- **FOL-BC-OR**: This function checks whether the goal can be satisfied by any rule in the KB. It iterates over applicable rules and tries to unify the goal with the rule's conclusions.
- **FOL-BC-AND**: This function handles multiple sub-goals. It ensures that all sub-goals are satisfied for the main goal to be true.

4. **Key Terminology**:
   - **FOL-BC-ASK:** Entry point for the algorithm.
   - **FOL-BC-OR:** Handles rules and checks if the goal is satisfied by any rule.
   - **FOL-BC-AND:** Ensures all sub-goals are satisfied.
   - **FETCH-RULES-FOR-GOAL:** Retrieves applicable rules for a goal.
   - **UNIFY:** Matches terms by finding substitutions.
   - **Standardize Variables:** Ensures variable names are unique to avoid conflict.
   - **θ:** The substitution carried into the current function call**.**
   - **θ′:** A substitution produced by solving the first sub-goal in FOL-BC-AND.
   - **θ′′:** A substitution produced by solving the remaining sub-goals using the updated θ′.

Detailed Algorithm Steps

Step 1: FOL-BC-ASK(KB, query):

- Start with the query and call `FOL-BC-OR`.
- Example: For `Ancestor(John, Sam)`, call:
  - `FOL-BC-OR(KB, Ancestor(John, Sam), { }).`

Step 2: FOL-BC-OR(KB, goal, θ):

- Fetch all rules from the KB that could produce the `goal`.
- For each rule:
  1. **Standardize Variables**: Make rule variables unique to avoid conflicts.
  2. **Unify `rhs` and `goal`**: Match the conclusion of the rule (`rhs`) with the current `goal` using `Unify`. This updates `θ`.
  3. **Call FOL-BC-AND**: Recursively evaluate the conditions (`lhs`) of the rule with the updated `θ`.
- **Yield θ′**: Each substitution that satisfies the rule is yielded back to the caller.

Step 3: FOL-BC-AND(KB, goals, θ):

- Handles multiple sub-goals (`goals`) produced from the rule's conditions.

1. If `goals` is empty, yield `θ` because all sub-goals are satisfied.
2. Otherwise:
   - Split `goals` into `first` and `rest`.
   - Call `FOL-BC-OR` for the `first` goal.
   - For each result (`θ'`) from `FOL-BC-OR`:
     - Recursively solve `rest` using `FOL-BC-AND` with the updated `θ'`.
     - Yield `θ''`, the result of solving all sub-goals.

Step-by-Step Explanation with Example

Let's use the following **Knowledge Base (KB)** and query.

Knowledge Base:

1. `Parent(x, y) ⇒ Ancestor(x, y)` (Rule 1)
2. `Parent(x, z) ∧ Ancestor(z, y) ⇒ Ancestor(x, y)` (Rule 2)
3. `Parent(John, Mary)` (Fact)
4. `Parent(Mary, Sam)` (Fact)

Query: `Ancestor(John, Sam)`

Execution Steps

Step 1: FOL-BC-ASK

- Query: `Ancestor(John, Sam)`
- Calls: `FOL-BC-OR(KB, Ancestor(John, Sam), { }).`

Step 2: FOL-BC-OR

- Goal: `Ancestor(John, Sam)`
- Fetch rules for `Ancestor`:
  1. Rule 1: `Parent(x, y) ⇒ Ancestor(x, y)`
  2. Rule 2: `Parent(x, z) ∧ Ancestor(z, y) ⇒ Ancestor(x, y)`

Case 1: Use Rule 1

- `lhs = Parent(x, y),rhs = Ancestor(x, y).`
- Unify `Ancestor(John, Sam)` with `Ancestor(x, y)`:
  - Substitution: `θ = {x=John, y=Sam}.`
- Sub-goal: `Parent(John, Sam).`

Step 3: FOL-BC-AND

- Goals: `[Parent(John, Sam)]`
- Calls: `FOL-BC-OR(KB, Parent(John, Sam), {x=John, y=Sam}).`

Step 4: FOL-BC-OR

- Goal: `Parent(John, Sam)`
- Check the KB:
  - Facts: `Parent(John, Mary)` (no match for `Sam`).
  - Rule 1 fails.

Case 2: Use Rule 2

- `lhs = Parent(x, z) ∧ Ancestor(z, y),rhs = Ancestor(x, y).`
- Unify `Ancestor(John, Sam)` with `Ancestor(x, y)`:
  - Substitution: `θ = {x=John, y=Sam}.`
- Sub-goals:
  - `goals = [Parent(John, z), Ancestor(z, Sam)].`

Step 5: FOL-BC-AND

- Goals: `[Parent(John, z), Ancestor(z, Sam)].`

1. **First sub-goal (`Parent(John, z)`)**:
   - Calls: `FOL-BC-OR(KB, Parent(John, z), θ).`
   - Matches: `Parent(John, Mary).`
   - Substitution: `{z=Mary}.`
   - Update θ′: `{x=John, y=Sam, z=Mary}.`
2. **Second sub-goal (`Ancestor(z, Sam)`)**:
   - Calls: `FOL-BC-OR(KB, Ancestor(Mary, Sam), θ′).`
   - Unify with Rule 1: `Parent(x, y) ⇒ Ancestor(x, y).`
   - Sub-goal: `Parent(Mary, Sam).`

Step 6: FOL-BC-AND

- Goal: `[Parent(Mary, Sam)].`
- Matches fact: `Parent(Mary, Sam).`
- Substitution: `{x=Mary, y=Sam}.`
- Satisfies all sub-goals.

Final Result

- Combine all substitutions:
  - `{x=John, y=Sam, z=Mary}.`
- The query `Ancestor(John, Sam)` is **true**.

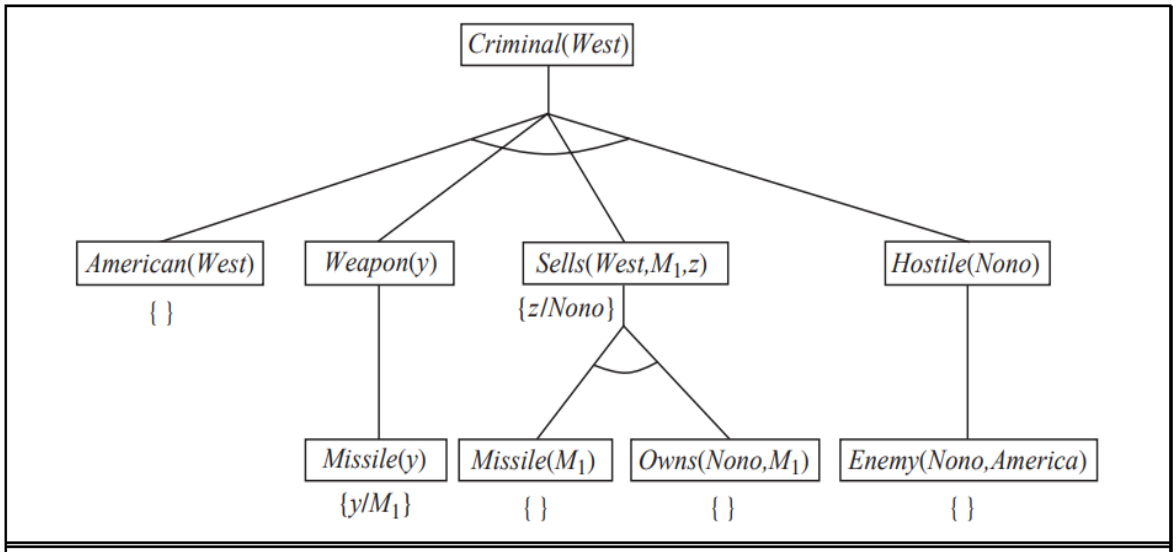


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

5.1.2 Resolution

Resolution is a fundamental inference rule used in automated theorem proving and logic programming. It is based on the **principle of proof by contradiction.**
Resolution **combines logical sentences in the form of clauses** to derive new sentences.
The resolution rule states that if there are **two clauses** that contain complementary literals (**one positive, one negative**) then these literals can be resolved, leading to a new clause that is inferred from the original clauses.

Example1:

Consider two logical statements:

1. PVQ
2. ¬PVR

Applying resolution: Resolve the statements by eliminating P:

- PVQ
- ¬PVR

Resolving P and ¬P: QVR
The resulting statement **QVR is a new clause** inferred from the original two.

Resolution is a key component of **logical reasoning in FOL**, especially in tasks like automated theorem proving and knowledge representation.

Example2:

Clause 1: (PVQVR)
Clause 2:(¬PV¬QVS)
To resolve these clauses, we look for complementary literals. In this case, **P and ¬P** are complementary.
So, we can resolve these two clauses by removing the complementary literals and combining the remaining literals:
Resolving P and ¬P gives**: (QVR)V(¬QVS)**
Resolving Q and ¬Q gives (RVS)

This is the resolvent.

Conjunctive Normal Form

A formula is in CNF if it is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals.

CNF Examples

1. $(A \vee B) \wedge (\neg A \vee C)$
   This expression has two clauses: $A \vee B$ and $\neg A \vee C$.
2. $(P \vee Q \vee R) \wedge (\neg P \vee \neg Q)$
   This expression also has two clauses: $P \vee Q \vee R$ and $\neg P \vee \neg Q$.
3. $(A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee D) \wedge (C \vee D)$
   This expression has three clauses: $A \vee B \vee \neg C$, $\neg A \vee \neg B \vee D$, and $C \vee D$.
4. $(P \vee Q)$
   This is already in CNF, with one clause: $P \vee Q$.
5. $(A \wedge B) \vee (\neg C \wedge D)$
   This expression is not in CNF because it's a disjunction of conjunctions. To convert it to CNF, you'd need to apply distribution, obtaining $(A \vee \neg C) \wedge (A \vee D) \wedge (B \vee \neg C) \wedge (B \vee D)$.

Steps to Convert a Formula to CNF:

1. **Eliminate Biconditionals (⇔):** Replace each biconditional (↔) with an equivalent expression in terms of conjunction (∧), disjunction (∨), and negation (¬).
   - Example: Replace A ⇔ B with (A⇒B) ∧(B⇒A)

2. **Eliminate Implications** (⇒): Replace each implication (⇒) with an equivalent expression using conjunction and negation.
   - Example: Replace A⇒B with ¬A ∨ B

3. **Move Negations Inward** (¬): Apply De Morgan's laws and distribute negations inward to literals.
   - ¬(¬A) ≡ A
   - ¬(A ∨ B) ≡(¬A ∧ ¬B)
   - ¬(A∧ B) ≡ (¬A ∨ ¬B)

4. **Distribute Disjunctions Over Conjunctions**: Apply the distributive law to ensure that disjunctions are only over literals or conjunctions of literals.
   Suppose we have the following formula: **H=(P∧Q)∨(R∧S∧T)** .
   Now, let's distribute disjunctions over conjunctions: **H=(P∨(R∧S∧T))∧(Q∨(R∧S∧T))**

Proof By Resolution Process includes the following steps in general

1. **Initial Set of Clauses (Knowledge Base)**
2. **Convert the Statement into Clausal Form**
3. **Skolemization**
4. **Standardize Variables**
5. **Unification**
6. **Resolution Rule**
7. **Iterative Application**

Resolution in First Order Logic (FOL) is a proof technique used in automated reasoning to determine the validity of a statement. It is based on the principle of refutation, where we attempt to derive a contradiction from a set of clauses.

Key Steps in Resolution in FOL:

1. **Convert the Statement into Clausal Form**:
   - The statement and its negation are converted into **conjunctive normal form (CNF)**.
   - CNF is a conjunction of disjunctions where each disjunction is called a clause.
   - Example: $(A \vee \neg B) \wedge (\neg A \vee C)$.
2. **Skolemization**:
   - Eliminate existential quantifiers by replacing them with Skolem functions or constants.
   - This step ensures the formula becomes purely universal.
3. **Standardize Variables**:
   - Rename variables so that no two clauses share the same variable names.
4. **Unification**:
   - **Unification** is the process of making two literals identical by finding a substitution for their variables.
   - **Example:** $P(x)$ and $P(a)$ can be unified with the substitution $x=a$.
5. **Resolution Rule**:
   - The **resolution rule** combines two clauses that contain complementary literals (e.g., $P(x)$ and $\neg P(x)$ to produce a new clause without those literals.
   - Example:
     - Clause 1: $P(x) \vee Q(x)$,
     - Clause 2: $\neg P(a) \vee R(x)$,
     - Resolvent: $Q(a) \vee R(x)$.

6. **Iterative Application**:
    - Apply the resolution rule repeatedly to derive new clauses.
    - If the empty clause □is derived, it indicates a contradiction, proving the original statement.

Example: Proving Validity

Suppose we have the following knowledge base:

1. $P(a) \vee Q(b)$
2. $\neg Q(b) \vee R(c)$
3. $\neg R(c)$

We want to prove $\neg P(a)$.

Steps:

1. Negate the statement to be proven and add it to the clauses:
    - $\neg P(a)$ becomes $P(a)$.
2. Apply the resolution rule:
    - Combine $P(a) \vee Q(b)$.
    - Combine $Q(b)$ and $\neg Q(b) \vee R(c)$.
    - Combine $R(c)$ and $\neg R(c)$ to get □ empty clause.

Since the empty clause □ is derived, the original statement $\neg P(a)$ is valid.

Applications of Resolution in FOL:

- Automated theorem proving.
- Reasoning in expert systems.
- Artificial intelligence, particularly for tasks involving logical inference.

Example 1:

Let's consider a simplified example of a knowledge base for the Wumpus World scenario and demonstrate proof by resolution to establish the unsatisfiability of a certain statement.

In Wumpus World**, an agent explores a grid containing** a Wumpus (a monster), pits, and gold. Apply the resolution to prove **P[1,2].**

- **Knowledge Base (KB)**

1. **W[1,1] ∨ P[1,2]**
2. **¬W[1,1]∨¬P[1,2]**
3. **B[1,2]⇒P[1,2]**
4. **¬B[1,2]⇒¬P[1,2]**

- **Convert the Knowledge Base (KB) into CNF**

1. W[1,1] ∨ P[1,2]	**1. W[1,1] ∨ P[1,2]**
2. ¬W[1,1]∨¬P[1,2]	**2. ¬W[1,1]∨¬P[1,2]**
3. B[1,2]⇒P[1,2]	**3. ¬ B[1,2] ∨ P[1,2]**
4. ¬B[1,2]⇒¬P[1,2]	**4. B[1,2] ∨ ¬P[1,2]**

- **Negated Conclusion**:

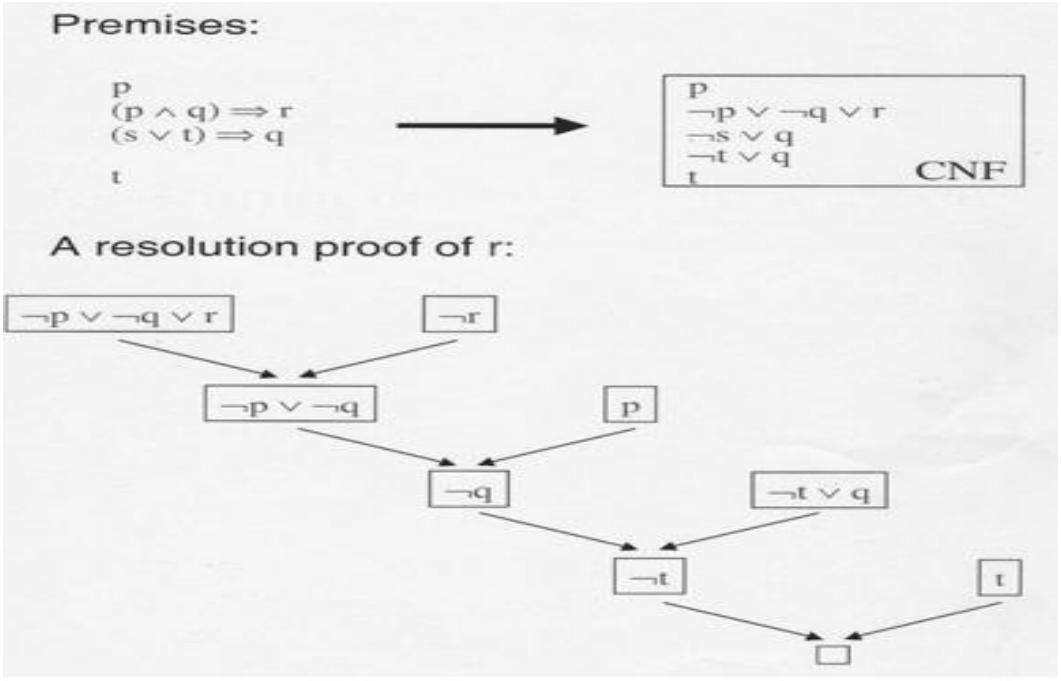
Let's say we want to prove the negation of the statement: **¬PitIn[1,2]**

- **Apply Resolution:**

1. **W[1,1] ∨ P[1,2] , ¬P[1,2] resolves into W[1,1]**
2. **¬W[1,1]∨¬P[1,2], W[1,1] resolves into ¬P[1,2]**
3. **¬B[1,2] ∨ P[1,2] , ¬P[1,2] resolves into ¬B[1,2]**
4. **B[1,2] ∨¬P[1,2], ¬B[1,2] resolves into ¬P[1,2]**

Applying resolution, we end up with: **¬P[1,2] ,** Which is not empty and also there is not further any clauses to continue. This gives conclusion that our negation conclusion is **True** and **P[1,2]** is **False** for the given knowledge base.

Example 2:



A grammar for conjunctive normal form

$$
\begin{aligned}
CNFSentence &\rightarrow Clause_1 \wedge \cdots \wedge Clause_n \\
Clause &\rightarrow Literal_1 \vee \cdots \vee Literal_m \\
Literal &\rightarrow Symbol \mid \neg Symbol \\
Symbol &\rightarrow P \mid Q \mid R \mid \ldots \\
HornClauseForm &\rightarrow DefiniteClauseForm \mid GoalClauseForm \\
DefiniteClauseForm &\rightarrow (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow Symbol \\
GoalClauseForm &\rightarrow (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow False
\end{aligned}
$$

Conjunctive normal form for first-order logic:

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form (CNF)—that is, a conjunction of clauses**, where each clause is a **disjunction of literals**.

Literals can contain **variables**, which are assumed to be **universally quantified**.

For example, the sentence

- **∀ x American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal(x)** becomes, in CNF,

¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x, y, z) ∨ ¬Hostile(z) ∨ Criminal(x)

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. The procedure for conversion to CNF is similar to the propositional case, The principal difference arises from the need to eliminate existential quantifiers.

We illustrate the procedure by translating the sentence

"Everyone who loves all animals is loved by someone,"

or

∀ x [∀ y Animal(y) ⇒ Loves(x, y)] ⇒ [∃ y Loves(y, x)] .

Steps

- **Eliminate implications**: ∀ x [¬∀ y ¬Animal(y) ∨ Loves(x, y)] ∨ [∃ y Loves(y, x)] .
- **Move ¬ inwards**: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have
    - ¬∀ x p becomes ∃ x ¬p
    - ¬∃ x p becomes ∀ x ¬p .
- **Our sentence goes through the following transformations**:
    - ∀ x [∃ y ¬(¬Animal(y) ∨ Loves(x, y))] ∨ [∃ y Loves(y, x)] .
    - ∀ x [∃ y ¬¬Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ y Loves(y, x)] .
    - ∀ x [∃ y Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ y Loves(y, x)] .
- **Standardize variables**: For sentences like (∃ x P(x))∨(∃ x Q(x)) which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have
    - ∀ x [∃ y Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ z Loves(z, x)] .

- **Skolemize**: Skolemization is the process of removing existential quantifiers by elimination. Translate ∃ x P(x) into P(A), where A is a new constant.
    - **Example :**
        - ∀ x [Animal(A) ∧ ¬Loves(x, A)] ∨ Loves(B, x) ,
        - ∀ x [Animal(F(x)) ∧ ¬Loves(x, F(x))] ∨ Loves(G(z), x) . Here F and G are Skolem functions.
- **Drop universal quantifiers**: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:
    - [Animal(F(x)) ∧ ¬Loves(x, F(x))] ∨ Loves(G(z), x) .

• **Distribute ∨ over ∧:**

[Animal(F(x)) ∨ Loves(G(z), x)] ∧ [¬Loves(x, F(x)) ∨ Loves(G(z), x)] .

The resolution inference rule

- Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain **complementary literals**.
- Propositional literals are complementary **if one is the negation of the other**;
- first-order literals are complementary if one unifies with the negation of the other.
- Thus We have

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

by eliminating the complementary literals $Loves(G(x), x)$ and $\neg Loves(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$

Example:

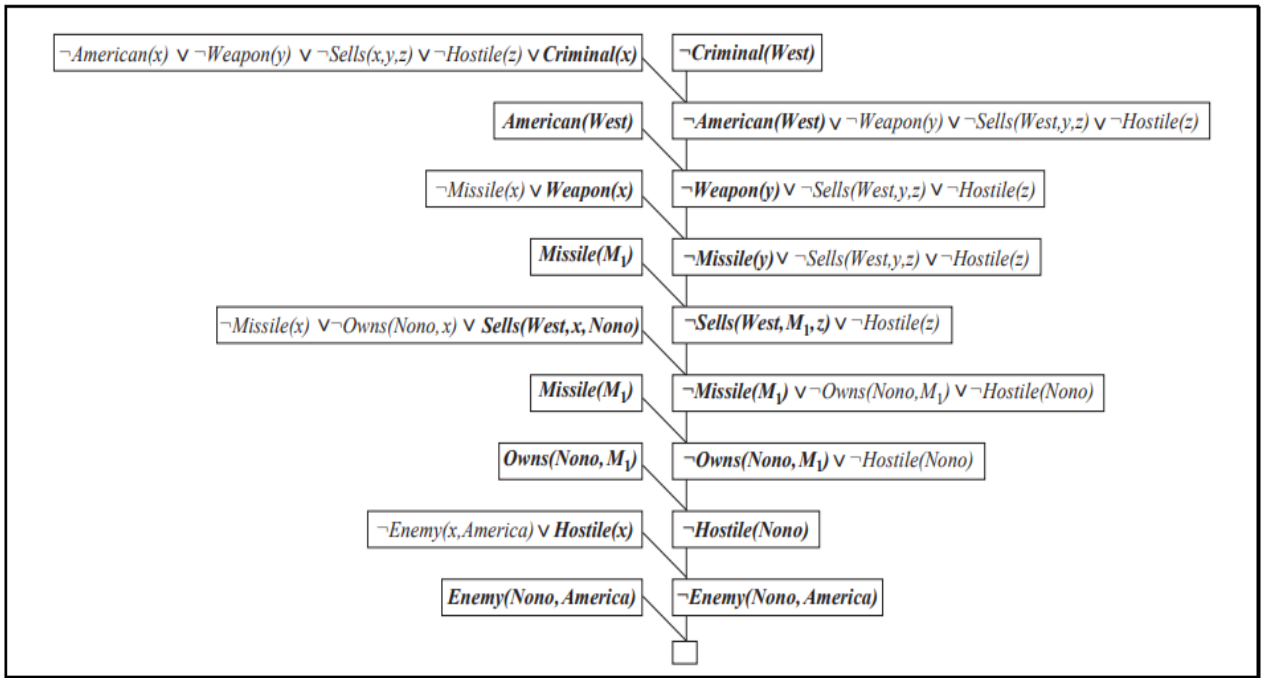


Figure 9.11 A resolution proof that West is a criminal. At each step, the literals that unify are in bold.

Another Example :

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$

B. $\forall x\ [\exists z\ Animal(z) \wedge Kills(x,z)] \Rightarrow [\forall y\ \neg Loves(y,x)]$

C. $\forall x\ Animal(x) \Rightarrow Loves(Jack,x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\forall x\ Cat(x) \Rightarrow Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

Now we apply the conversion procedure to convert each sentence to CNF:

A1. $Animal(F(x)) \vee Loves(G(x),x)$

A2. $\neg Loves(x,F(x)) \vee Loves(G(x),x)$

B. $\neg Loves(y,x) \vee \neg Animal(z) \vee \neg Kills(x,z)$

C. $\neg Animal(x) \vee Loves(Jack,x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\neg Cat(x) \vee Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

Suppose Curiosity did not **kill Tuna**. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.

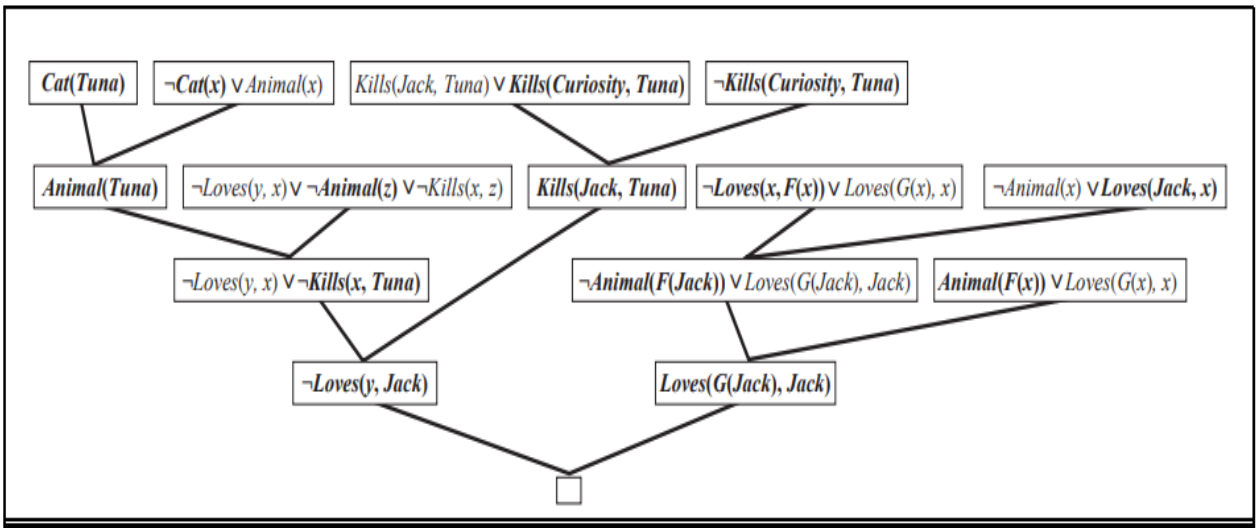


Figure 9.12 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $Loves(G(Jack), Jack)$. Notice also in the upper right, the unification of $Loves(x, F(x))$ and $Loves(Jack, x)$ can only succeed after the variables have been standardized apart.

Summary

1. F**orward chaining starts** with known facts and moves forward to reach conclusions,
2. **Backward chaining** starts with the goal and moves backward to verify if the goal can be satisfied, and
3. **Resolution** is an inference rule used to derive new clauses by combining existing ones.

These ***techniques are essential for reasoning and inference in First-Order Logic systems***.

Note : The empty clause derived always implies the assumed **negation(contradiction)** is false.

Completeness of resolution

Resolution is a method in logic that can prove whether a set of statements is unsatisfiable. If the statements are unsatisfiable (i.e., there's no way they can all be true at once), resolution will eventually find a contradiction.

- **Unsatisfiable set of statements**: Means the statements can't all be true together.
- **Contradiction**: A clear proof that the statements conflict with each other.

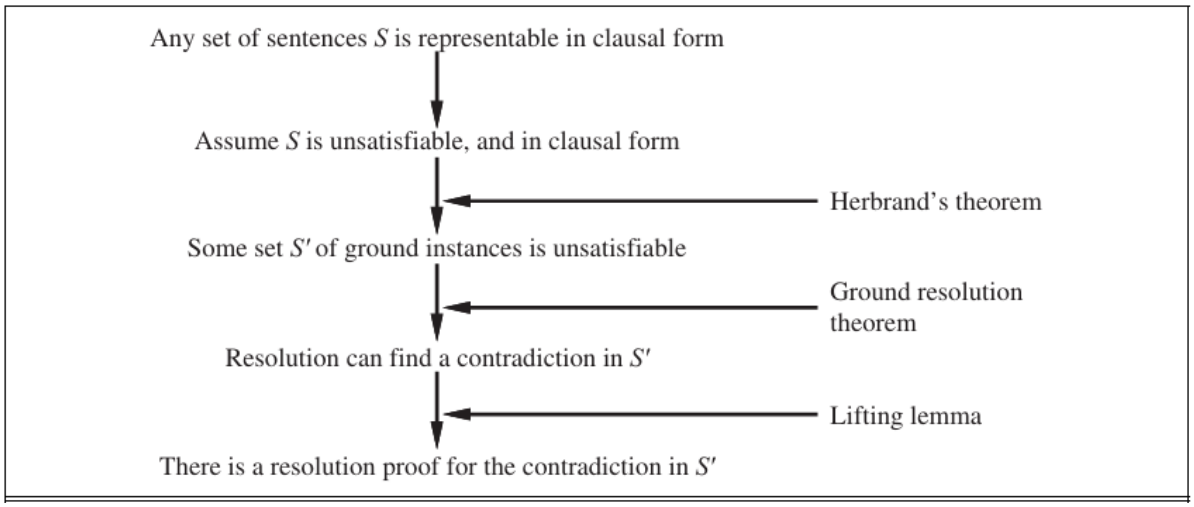
Key Idea

If a set of statements is unsatisfiable, resolution can always derive a contradiction, proving unsatisfiability. This doesn't mean resolution finds all logical consequences—it's focused on checking contradictions.

Steps in Proving Completeness

1. **Transforming to Clausal Form**:
   Any logical statement can be converted into a standard form called Conjunctive Normal Form (CNF). This is the foundation for using resolution.
2. **Using Herbrand's Theorem**:
   Herbrand's theorem says if the set of statements is unsatisfiable, there's a specific subset of ground instances (statements without variables) that's also unsatisfiable.
3. **Applying Ground Resolution**:
   For these ground instances, propositional resolution (which works with statements without variables) can find the contradiction.
4. **Lifting to First-Order Logic**:
   A "lifting lemma" proves that if there's a resolution proof for the ground instances, there's also one for the original statements with variables. This ensures resolution works for first-order logic, not just simple ground statements.

Structure of a completeness proof for resolution is illustrated in the figure below:



What Are Ground Terms and Herbrand's Universe?

- **Ground terms**: Statements with no variables, created by substituting constants or functions.
- **Herbrand Universe**: A collection of all possible ground terms that can be built from the constants and functions in the given statements.
- **Saturation**: Generating all possible combinations of ground terms in the statements.

Herbrand's theorem ensures we only need to check a finite subset of these terms to find a contradiction.

Why is the Lifting Lemma Important?

The lifting lemma connects proofs for ground terms to proofs for first-order logic. It "lifts" results from simpler cases (propositional logic) to more general cases (with variables). This step is essential to show resolution's power in first-order logic.

The Conclusion

If a set of statements is unsatisfiable:

- Resolution finds a contradiction using a finite number of steps.
- This proof works for both simple ground statements and complex first-order logic.

This makes resolution a powerful tool in automated theorem proving!

The Lifting Lemma Explained

The **lifting lemma** is a principle that allows us to "lift" a resolution proof from specific ground instances (statements without variables) to general first-order logic (statements with variables). Here's how it works:

- **C_1 and C_2**: Two clauses that do not share variables.
- **C'_1 and C'_2**: Ground instances of C_1 and C_2 (created by substituting variables with constants or terms).
- **C'**: A resolvent (a result of applying the resolution rule) of C'_1 and C'_2.

The lemma states:
There exists a clause **C** such that:

1. **C** is a resolvent of C_1 and C_2 (it works at the variable level).
2. **C′** is a ground instance of **C**.

In simpler terms, if resolution works for specific ground instances, we can always find a corresponding proof for the original first-order clauses.

Example

Let's illustrate with an example:

1. **Original clauses with variables**:
   - $C1=\neg P(x,F(x,A))\vee\neg Q(x,A)\vee R(x,B)$
   - $C2=\neg N(G(y),z)\vee P(H(y),z)$
2. **Ground instances (after substituting variables with specific terms)**:
   - $C'1=\neg P(H(B),F(H(B),A))\vee\neg Q(H(B),A)\vee R(H(B),B)$
     $C'2=\neg N(G(B),F(H(B),A))\vee P(H(B),F(H(B),A))$
3. **Resolvent of the ground instances**:
   - $C'=\neg N(G(B),F(H(B),A))\vee\neg Q(H(B),A)\vee R(H(B),B)$
4. **Lifted clause (with variables)**:
   - $C=\neg N(G(y),F(H(y),A))\vee\neg Q(H(y),A)\vee R(H(y),B)$

Here, C' is a ground instance of C, showing how the lifting lemma bridges ground-level proofs to general first-order logic.

This lemma is crucial because it ensures that resolution proofs for specific cases (ground terms) can be generalized to more complex first-order logic, making the method powerful and versatile.

Handling Equality in Inference Systems

So far, inference methods don't naturally handle statements like x=y. To deal with equality, we can take one of three approaches.

---

1. Axiomatizing Equality

We write rules (axioms) in the knowledge base that define how equality works. These rules must express:

- **Reflexivity**: $x=x$
- **Symmetry**: $x=y \Rightarrow y=x$
- **Transitivity**: $x=y \wedge y=z \Rightarrow x=z$

Additionally, we add rules to allow substitution of equal terms in predicates and functions. For example:

- $x=y \Rightarrow (P(x) \Leftrightarrow P(y))$ (for predicates P)
- $w=y \wedge x=z \Rightarrow F(w,x)=F(y,z)$(for functions F)

Using these axioms, standard inference methods like resolution can handle equality reasoning (e.g., solving equations). However, this approach can generate many unnecessary conclusions, making it inefficient.

---

2. Demodulation: Adding Inference Rules

Instead of axioms, we can add specific inference rules like **demodulation** to handle equality.

How it works:

- If x=y (a unit clause) and a clause α contains x, we replace x with y in α.
- Demodulation simplifies expressions in one direction (e.g., x+0=x allows x+0 to simplify to x, but not vice versa).

Example:
Given:

- Father(Father(x))=PaternalGrandfather(x)
- Birthdate(Father(Father(Bella)), 1926)

We use demodulation to derive:

- Birthdate(PaternalGrandfather(Bella), 1926)

---

3. Paramodulation

A more general rule, **paramodulation**, extends demodulation to handle cases where equalities are part of more complex clauses.

How it works:

- If x=y appears as part of a clause and a term z in another clause unifies with x, substitute y for x in z.

Formal Rule:

- For any terms x, y, and z:
    - If z appears in a clause mmm and x unifies with z,
    - Replace x with y in m, while preserving other parts of the clause.

Summary

- **Axiomatization** defines equality with explicit rules but can be inefficient.
- **Demodulation** simplifies terms by replacing variables with their equal counterparts in one direction.
- **Paramodulation** generalizes equality handling for complex clauses.

These methods provide efficient ways to incorporate equality reasoning into inference systems.

More formally we have

- **Demodulation**: For any terms x, y, and z, where z appears somewhere in literal m_i and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{x = y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), m_1 \vee \cdots \vee m_n)} .$$

  where SUBST is the usual substitution of a binding list, and $\text{SUB}(x, y, m)$ means to replace x with y everywhere that x occurs within m.

- **Paramodulation**: For any terms x, y, and z, where z appears somewhere in literal m_i, and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), \text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_n))} .$$

Resolution Strategies

Resolution inference is guaranteed to find a proof if one exists, but some strategies can make the process more efficient. Below are key strategies and their applications.

Unit Preference

- Focuses on resolving clauses where one is a **unit clause** (a single literal).
- Resolving a unit clause (e.g., P) with a longer clause (e.g., ¬P∨¬Q∨R) results in a shorter clause (¬Q∨R).
- This strategy, first applied in 1964, dramatically improved the efficiency of propositional inference.
- **Unit Resolution**: A restricted form of this strategy, requiring every resolution step to involve a unit clause.
  - **Complete for Horn Clauses**: Proofs resemble forward chaining.
  - **Incomplete in General**: Not suitable for all forms of knowledge bases.
- **Example:** The **OTTER theorem prover** employs a best-first search with a heuristic that assigns “weights” to clauses, favoring shorter ones (e.g., unit clauses).

Set of Support

- Restricts resolutions to involve at least one clause from a predefined **set of support**.
- The **set of support** typically includes the negated query or clauses likely to lead to a proof.
- Resolutions add their results to this set, significantly reducing the search space if the set is small.
- This strategy is **complete** if the remaining sentences in the knowledge base are satisfiable.
- **Advantages**:
    - Generates **goal-directed proof trees**, which are easier for humans to interpret.

Input Resolution

- In this strategy, resolutions always involve one of the **original input clauses** (from the knowledge base or the query).
- Example: **Modus Ponens** in Horn knowledge bases is an input resolution strategy, as it combines an implication from the KB with other sentences.
- **Linear Resolution**: A generalization where PPP and QQQ can be resolved if PPP is either an input clause or an ancestor of QQQ in the proof tree.
    - **Complete for Linear Resolution**: Particularly useful in structured proofs.

Subsumption

- **Eliminates redundant sentences** in the knowledge base that are subsumed by more general sentences.
- Example: If P(x) is in the KB, there's no need to add P(A) or P(A)∨Q(B).
- **Benefits**:
    - Reduces the size of the knowledge base.
    - Keeps the search space manageable.

Applications of Resolution Theorem Provers

Resolution theorem provers are widely used in the synthesis and verification of both hardware and software systems.

1. Hardware Design and Verification

- Axioms describe the interactions between signals and circuit components.
- Example: Logical reasoners have verified entire CPUs, including timing properties.
- **AURA Theorem Prover**: Used to design highly compact circuits.

2. Software Verification and Synthesis

- Similar to reasoning about actions, axioms define the preconditions and effects of program statements.
- **Algorithm Synthesis**:
  - Deductive synthesis constructs programs to meet specific criteria.
  - Although fully automated synthesis is not yet practical for general-purpose programming, hand-guided synthesis has successfully created sophisticated algorithms.
- **Verification Tools**: Systems like the **SPIN model checker** are used to verify programs such as:
  - Remote spacecraft control systems.
  - Algorithms like RSA encryption and Boyer–Moore string matching.

Summary

Resolution strategies like unit preference, set of support, input resolution, and subsumption improve proof efficiency by focusing on relevance, reducing redundancy, and constraining the search space. Applications in hardware and software demonstrate their importance in real-world problem-solving.

5.2 Classical Planning:

Syllabus: Definition of Classical Planning, Algorithms for Planning as State-Space Search, Planning Graphs.

5.2.1 The Definition of Classical Planning

Classical planning focuses on solving problems by identifying sequences of actions that transition from an initial state to a goal state. In this approach factored **representations is adopted**, where a state is expressed as a collection of variables. This approach uses the Planning Domain Definition Language (PDDL), which enables concise representation of actions through schemas, reducing redundancy. For instance, instead of defining individual actions for all possible combinations, a single action schema in PDDL can represent multiple actions by using variables.

5.2.1.1 Representing States in Classical Planning

States are represented as **conjunctions of fluents**—**ground, functionless atomic facts**. For example:

- **Poor ∧ Unknown**: Represents the state of a struggling agent.
- **At(Truck1, Melbourne) ∧ At(Truck2, Sydney)**: Represents locations of trucks in a delivery problem.

The representation follows:

1. **Closed-world assumption**: Any fluent not explicitly mentioned is considered false.
2. **Unique names assumption**: Different symbols (e.g., Truck1 and Truck2) represent distinct entities.

Certain constructs are **disallowed** in states, such as:

- Non-ground fluents: e.g., **At(x, y)**.
- Negations: e.g., **¬Poor**.
- Function symbols: e.g., **At(Father(Fred), Sydney)**.

States can be manipulated as either conjunctions of fluents (using logical inference) or sets of fluents (using set operations).

5.2.1.2 Defining Actions with Schemas

Actions are defined using **schemas**, which specify:

- The action name and variables.
- **Precondition:** The required state for the action to execute.
- **Effect:** The state resulting from the action.

For example, an action schema for flying a plane is:

```
Action(Fly(p, from, to),
   PRECOND: At (p, from)∧Plane(p) ∧ Airport(from) ∧
Airport(to),
    EFFECT: ¬At (p, from) ∧ At (p, to))
```

From this schema, specific actions can be instantiated by substituting variable values.
For instance:

```
Action (Fly (P1, SFO, JFK),
    PRECOND: At (P1, SFO) ∧ Plane(P1) ∧ Airport(SFO)
∧ Airport(JFK),
    EFFECT: ¬At (P1, SFO) ∧ At(P1, JFK))
```

An action is **applicable** in a state if its preconditions are satisfied. When executed, the resulting state is determined by:

- Removing fluents in the **delete list** (negative effects).
- Adding fluents in the **add list** (positive effects).

For example, executing **Fly(P1, SFO, JFK)** in a state would remove **At(P1, SFO)** and add **At(P1, JFK)**.

5.2.1.3 Planning Domains and Problems

A planning domain is defined by a set of action schemas. A specific problem within the domain includes:

1. **Initial state**: A conjunction of ground fluents.
2. **Goal**: A conjunction of literals, possibly containing variables treated as existentially quantified.

The planning problem is solved when a sequence of actions leads to a state that satisfies the goal.
For example:

- The state **Plane(Plane1) ∧ At(Plane1, SFO)** satisfies the goal **At(p, SFO) ∧ Plane(p)**.

5.2.1.4 Limitations in Early Approaches:

1. **Atomic State Representations (Chapter 3 Problem-Solving Agent):**
   - States are treated as indivisible entities, leading to a lack of structure in representations.
   - This approach relies heavily on **domain-specific heuristics** for effective problem-solving, limiting its generalizability and requiring significant manual tuning for each domain.
2. **Ground Propositional Inference (Chapter 7 Hybrid Propositional Logic Agent):**
   - Uses **domain-independent heuristics**, reducing the need for manual tuning.
   - However, it relies on ground (variable-free) propositional inference, which becomes computationally **infeasible with large state spaces** or a high number of actions.
   - Example: In the Wumpus World, a simple move action must account for all possible orientations, time steps, and grid locations, causing a combinatorial explosion in the number of actions.

5.2.1.5 Overcoming Limitations with Factored Representations:

1. **Structured State Representation:**
   - States are expressed as **collections of variables**, enabling a more structured and compact representation.
   - This approach captures relationships and dependencies between state components, improving efficiency and scalability.
2. **Use of PDDL (Planning Domain Definition Language):**
   - **Action Schemas:** Introduced to reduce redundancy by representing actions with variables rather than enumerating all possible instances.
     Example: Instead of defining actions for every plane and airport combination, a single schema can describe the action of flying a plane between airports.
   - **Domain Independence:** PDDL allows concise and reusable descriptions of actions and states, supporting various domains without customization.
3. **Improved Computational Efficiency:**

- By focusing on **factored representations** and logical reasoning over variables, the new approach avoids the combinatorial explosion seen in propositional logic.
- This scalability makes classical planning applicable to complex domains with numerous states and actions.

Summary of Improvement: The transition from atomic and ground representations to **factored representations with PDDL** enables classical planning to handle larger, more complex problems with greater efficiency and flexibility. This advancement overcomes the scalability challenges of earlier approaches while reducing dependency on domain-specific heuristics.

5.2.1.6 Example : Air cargo transport

$$
\begin{aligned}
&Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \\
&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \\
&\quad \land Airport(JFK) \land Airport(SFO)) \\
&Goal(At(C_1, JFK) \land At(C_2, SFO)) \\
&Action(Load(c, p, a), \\
&\quad \text{PRECOND: } At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } \neg At(c, a) \land In(c, p)) \\
&Action(Unload(c, p, a), \\
&\quad \text{PRECOND: } In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } At(c, a) \land \neg In(c, p)) \\
&Action(Fly(p, from, to), \\
&\quad \text{PRECOND: } At(p, from) \land Plane(p) \land Airport(from) \land Airport(to) \\
&\quad \text{EFFECT: } \neg At(p, from) \land At(p, to))
\end{aligned}
$$

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

The air cargo transport problem, illustrated in Figure 10.1, involves transporting cargo between airports by loading, unloading, and flying planes. This problem uses three main actions: **Load**, **Unload**, and **Fly**, which operate on two primary predicates:

1. **In(c, p):** Indicates that cargo `c` is inside plane `p`.
2. **At(x, a):** Specifies that an object `x` (plane or cargo) is located at airport `a`.

To ensure the correct maintenance of the **At** predicates, special care is required. When a plane flies from one airport to another, all cargo inside the plane must

also move with it. While first-order logic can easily quantify over all objects within the plane, basic PDDL lacks universal quantifiers. Therefore, a different solution is adopted:

- Cargo ceases to be **At** any location once it is loaded into a plane (it is considered **In** the plane).
- The cargo becomes **At** the destination airport only when it is unloaded from the plane.

Thus, **At(x, a)** effectively signifies "available for use at a given location."

Example Solution Plan : A valid solution plan for transporting cargo `C1` and `C2` is as follows:

1. **Load(C1, P1, SFO):** Load cargo `C1` onto plane `P1` at airport SFO.
2. **Fly(P1, SFO, JFK):** Fly plane `P1` from SFO to JFK.
3. **Unload(C1, P1, JFK):** Unload cargo `C1` from plane `P1` at JFK.
4. **Load(C2, P2, JFK):** Load cargo `C2` onto plane `P2` at JFK.
5. **Fly(P2, JFK, SFO):** Fly plane `P2` from JFK to SFO.
6. **Unload(C2, P2, SFO):** Unload cargo `C2` from plane `P2` at SFO.

Handling Spurious Actions : The problem can also involve spurious actions**, such as** Fly(P1, JFK, JFK)**, which would be a no-op but can produce contradictory effects (e.g., both `At(P1, JFK)` and ¬`At(P1, JFK)`). While such issues are often ignored in practice because they rarely lead to incorrect plans, the proper way to prevent them is by adding** inequality preconditions**, ensuring that the departure (`from`) and arrival (`to`) airports are different.**

In the context of the air cargo transport problem, **SFO** and **JFK** refer to airport codes:

- **SFO**: San Francisco International Airport
- **JFK**: John F. Kennedy International Airport (located in New York City)

These are commonly used IATA airport codes to represent specific locations in transportation and logistics scenarios. In this problem, they are used as example locations for cargo and planes.

5.2.1.7 Example: The Spare Tire Problem

Imagine the task of changing a flat tire, as shown in Figure 10.2. The goal is to replace the flat tire on the car's axle with a good spare tire. Initially, the flat tire is mounted on the axle, and the spare tire is in the trunk. For simplicity, this

problem is abstracted—there are no challenges like stubborn lug nuts or other real-world complications.

In this scenario, there are only four actions available:

1. Removing the spare tire from the trunk.
2. Removing the flat tire from the axle.
3. Mounting the spare tire onto the axle.
4. Leaving the car unattended overnight.

It is assumed that leaving the car unattended in a dangerous neighborhood results in all the tires disappearing. A valid solution to this problem would be the sequence:

```
[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)].
```

$$
\begin{array}{l}
Init(Tire(Flat) \wedge Tire(Spare) \wedge At(Flat, Axle) \wedge At(Spare, Trunk)) \\
Goal(At(Spare, Axle)) \\
Action(Remove(obj, loc), \\
\quad \text{PRECOND: } At(obj, loc) \\
\quad \text{EFFECT: } \neg At(obj, loc) \wedge At(obj, Ground)) \\
Action(PutOn(t, Axle), \\
\quad \text{PRECOND: } Tire(t) \wedge At(t, Ground) \wedge \neg At(Flat, Axle) \\
\quad \text{EFFECT: } \neg At(t, Ground) \wedge At(t, Axle)) \\
Action(LeaveOvernight, \\
\quad \text{PRECOND:} \\
\quad \text{EFFECT: } \neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk) \\
\qquad \wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk))
\end{array}
$$

Figure 10.2 The simple spare tire problem.

5.2.1.8 **Example:** The Blocks World

The **blocks world** is a classic planning domain often used to study problem-solving and AI planning. It involves manipulating cube-shaped blocks on a table to achieve a specified configuration.

Key Concepts:

1. **Setup:**
   - Blocks can be placed on the table or stacked on top of one another.
   - Only one block can fit directly on top of another block.

- A robot arm is used to move the blocks:
  - It can pick up only **one block at a time**.
  - It cannot pick up a block that has another block on top of it.

2. **Goal:**
   - The goal is defined by a specific arrangement of blocks, e.g., block **A on B** and block **B on C**.
3. **Predicates:**
   - **On(b, x):** Block `b` is on `x` (where `x` is another block or the table).
   - **Clear(x):** Block `x` is clear, meaning no other block is on it.
4. **Actions:**
   - **Move(b, x, y):** Moves block `b` from `x` to `y` (either another block or the table).
     - **Preconditions:**
       `On(b, x) ∧ Clear(b) ∧ Clear(y)`
       (Block `b` is on `x`, block `b` is clear, and the destination `y` is clear.)
     - **Effects:**
       `On(b, y) ∧ Clear(x) ∧ ¬On(b, x) ∧ ¬Clear(y)`
       (Block `b` is on `y`, `x` becomes clear, `b` is no longer on `x`, and `y` is no longer clear.)
5. **Issues and Solutions:**
   - **Problem:** The initial action schema does not handle the **table** correctly:
     - When moving a block from or to the table, the **Clear(Table)** predicate is mishandled.
     - For example:
       - `Clear(Table)` should always be true, as the table always has space.
       - However, the original schema treats the table like a block, leading to incorrect interpretations.
   - **Fixes:**
     - Introduce a new action, **MoveToTable(b, x):**
       - **Preconditions:**
         `On(b, x) ∧ Clear(b)`
         (Block `b` is on `x` and is clear.)
       - **Effects:**
         `On(b, Table) ∧ Clear(x) ∧ ¬On(b, x)`
         (Block `b` is now on the table, `x` is clear, and `b` is no longer on `x`.)

- Reinterpret **Clear(x):**
  "There is space on x to hold a block."
  (Under this interpretation, **Clear(Table)** is always true.)

6. **Optional Optimization:**
   - To prevent redundant use of **Move(b, x, Table)** instead of **MoveToTable(b, x):**
     - Add the predicate **Block(y)** to the **Move** action's precondition.
     - This ensures **Move** is only used for moving blocks between other blocks, not the table.

By making these adjustments, the blocks world planner becomes more accurate and efficient, avoiding unnecessary computational overhead.

$$
\begin{aligned}
&Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A) \\
&\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)) \\
&Goal(On(A, B) \wedge On(B, C)) \\
&Action(Move(b, x, y), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge \\
&\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y), \\
&\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \\
&Action(MoveToTable(b, x), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x), \\
&\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))
\end{aligned}
$$

Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

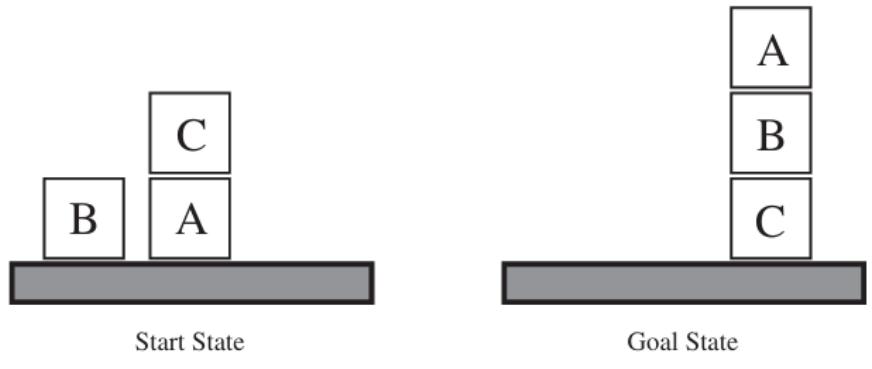


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

5.2.1.9 **The Complexity of Classical Planning**

In this subsection we consider the theoretical complexity of planning and distinguish two decision problems. **PlanSAT** is the question of whether there exists any plan that solves a planning problem. Bounded **PlanSAT** asks whether there is a solution of length k or less; this can be used to find an optimal plan.

1. **Key Decision Problems:**
   - **PlanSAT:** Determines whether a solution (plan) exists for a given planning problem.
   - **Bounded PlanSAT:** Checks if a solution of length ≤ *k* exists, often used to find optimal plans.
2. **Decidability:**
   - Both PlanSAT and Bounded PlanSAT are **decidable** for classical planning because the state space is finite.
   - When **function symbols** are added (creating an infinite state space):
     - PlanSAT becomes **semidecidable**: it terminates for solvable problems but may not terminate for unsolvable ones.
     - Bounded PlanSAT remains **decidable** even with function symbols.
3. **Complexity Classes:**
   - Both problems are in **PSPACE**, a complexity class harder than NP, requiring polynomial space to solve.
   - Even with restrictions:
     - Without **negative effects**, both problems are **NP-hard**.
     - Without **negative preconditions**, PlanSAT reduces to the easier class **P**.
4. **Practical Implications:**
   - Although the worst-case scenarios are complex, real-world problems in specific domains (e.g., blocks world, air cargo) are often simpler.
     - For many domains:
       - **Bounded PlanSAT** is **NP-complete** (hard for optimal planning).
       - **PlanSAT** is in **P** (easier for suboptimal solutions).
5. **Role of Heuristics:**
   - Classical planning's advantage lies in the development of **domain-independent heuristics**, which perform well on practical problems.
   - This contrasts with systems based on first-order logic, which struggle to create effective heuristics.

In summary, while planning problems can be theoretically hard, domain-specific scenarios and effective heuristics often simplify practical implementations.

PlanSAT and Bounded PlanSAT
This subsection explores the theoretical complexity of classical planning by distinguishing two decision problems:

1. **PlanSAT**: Determines whether there exists any plan that solves a given planning problem.
2. **Bounded PlanSAT**: Asks if there is a solution of length kkk or less, which can help find an optimal plan.

Decidability

Both problems are decidable for classical planning due to the finiteness of states. However, introducing function symbols to the language makes the number of states infinite. In this case:

- **PlanSAT** becomes **semidecidable**: an algorithm can terminate with the correct answer for solvable problems but might not terminate for unsolvable ones.
- **Bounded PlanSAT** remains decidable even with function symbols.

For detailed proofs, refer to Ghallab et al. (2004).

Complexity Class

Both PlanSAT and Bounded PlanSAT belong to the complexity class **PSPACE**, which includes problems solvable by a deterministic Turing machine using polynomial space. PSPACE is broader and more challenging than NP. Even with severe restrictions, these problems remain complex:

- Disallowing negative effects keeps them NP-hard.
- Disallowing negative preconditions reduces **PlanSAT** to **P**.

Practical Implications

These theoretical results might seem daunting, but practical planning rarely involves worst-case scenarios. For instance:

- In specific domains like the blocks-world or air-cargo problems, **Bounded PlanSAT** is NP-complete, while **PlanSAT** is in P.
- This implies optimal planning is often challenging, but suboptimal planning can be relatively easier.

To handle such cases effectively, good search heuristics are essential. Classical planning has advanced significantly by enabling highly accurate domain-independent heuristics. In contrast, systems relying on successor state axioms in first-order logic have struggled to develop strong heuristics.

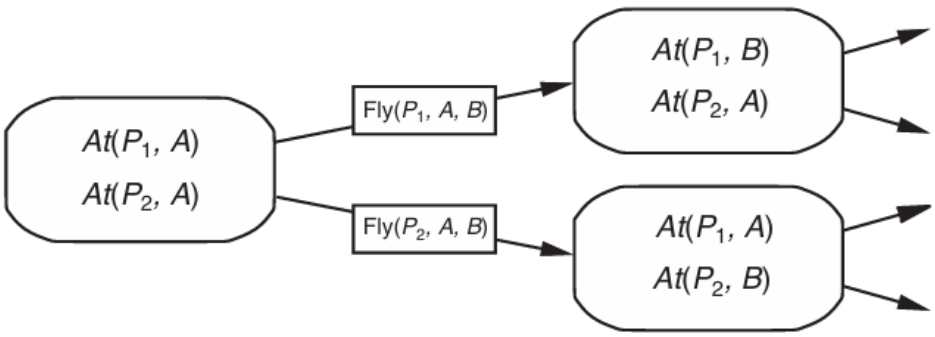
5.2.2 Algorithms for Planning as State-Space Search

Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

Forward (Progression) State-Space Search

- **Description:** Starts from the initial state and applies actions to reach the goal.
    - It explores all possible actions from the current state, leading to a large branching factor and potential inefficiency without heuristics.
- **Challenges:**
    1. Explores irrelevant actions.
    2. Handles large state spaces with numerous possible states and actions.
- **Example:**
  In an air cargo problem with 10 airports, 5 planes, and 20 cargo items:
    - At each step, the search needs to evaluate thousands of possible actions like flying planes, loading cargo, or unloading it.
    - Without a heuristic, this leads to a massive search space.

(a)

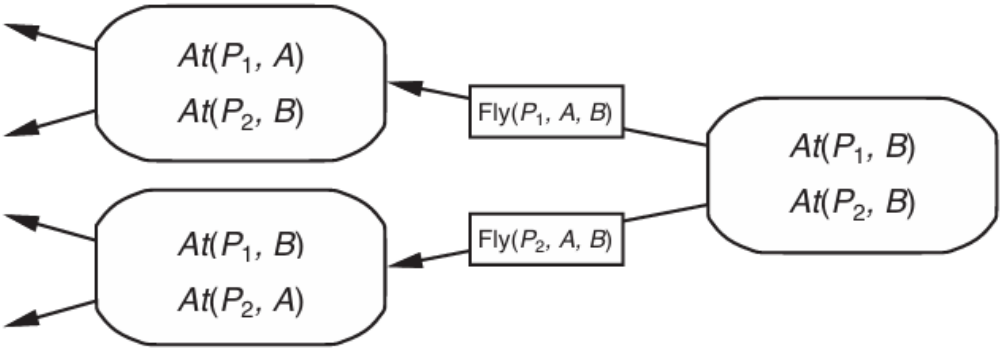


Backward (Regression) Relevant-States Search

- **Description:** Starts from the goal and works backward by identifying actions that can lead to the goal state.
- **Advantages:** Focuses only on **relevant actions** and avoids irrelevant branches.

- **Example:**
  If the goal is `At(C2, SFO)`, the algorithm considers the action **Unload(C2, p, SFO)**:
  - Precondition: `In(C2, p) ∧ At(p, SFO).`
  - Effect: `At(C2, SFO).`
    It regresses to find the predecessor state where these preconditions are true.

(b)



Heuristics for Planning

- **Purpose:** Estimate the cost of reaching the goal from the current state to guide search algorithms like A*.
- **Types of Heuristics:**
  1. **Ignore Preconditions:**
     - Drops preconditions, making every action applicable.
     - Example: Simplifies the 8-puzzle by ignoring adjacency requirements for moves.
  2. **Ignore Delete Lists:**
     - Assumes actions cannot undo progress, making the problem monotonic.
     - Example: In a transportation problem, unloading an item is never undone.
  3. **State Abstraction:**
     - Groups states by ignoring irrelevant fluents to reduce the state space.
     - Example: In air cargo, consider only packages and destinations while abstracting plane details.

Figure 10.6 diagrams part of the state space for two planning problems using the ignore-delete-lists heuristic. The dots represent states and the edges actions, and the height of each dot above the bottom plane represents the

heuristic value. States on the bottom plane are solutions. In both these problems, there is a wide path to the goal. There are no dead ends, so no need for backtracking; a simple hill climbing search will easily find a solution to these problems (although it may not be an optimal solution).

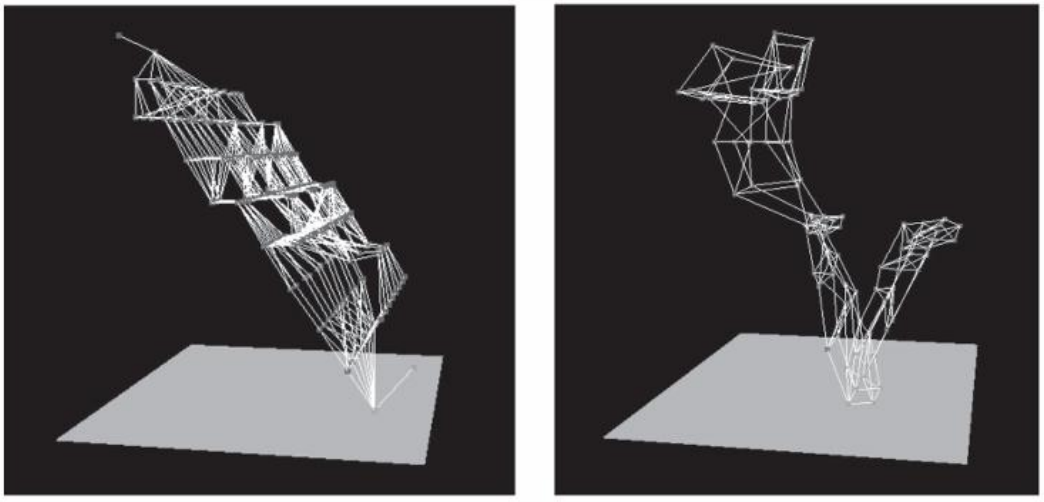


Figure 10.6 Two state spaces from planning problems with the ignore-delete-lists heuristic. The height above the bottom plane is the heuristic score of a state; states on the bottom plane are goals. There are no local minima, so search for the goal is straightforward. From Hoffmann (2005).

The image illustrates two state spaces derived from planning problems where the **ignore-delete-lists heuristic** is applied. Here's a breakdown of the key details:

1. **State Space Representation**:
   - The white lines represent the connections between different states (nodes) in the planning problem.
   - Each state corresponds to a possible configuration of the problem, and transitions represent actions that lead from one state to another.
2. **Heuristic Scores**:
   - The height of a state above the base plane indicates its **heuristic score**. The heuristic score reflects an estimate of the distance to the goal state (lower is better).
   - The visualization shows how the heuristic value changes as states progress toward the goal.
3. **Goal States**:
   - States that lie directly on the bottom plane are **goal states** (i.e., configurations that satisfy the problem's requirements).
4. **Ignore-Delete-Lists Heuristic**:
   - This heuristic simplifies the problem by ignoring negative effects of actions (delete lists) during the planning process.

- It creates a state space without **local minima**, meaning there are no dead ends or misleading paths that can trap the search. This makes the search for the goal straightforward.

5. **Two Examples**:
   - The two diagrams represent different problem instances:
     - The left diagram shows a relatively "steep" and orderly descent to the goal, indicating a direct and simple path.
     - The right diagram has a more "twisted" state space, with complex paths leading to the goal, but still without any local minima.

These visualizations help explain why the ignore-delete-lists heuristic is effective: the absence of local minima ensures the search algorithm does not get stuck, and the heuristic guides the planner efficiently toward the goal.

5.2.3 Planning Graphs

Definition:

A **planning graph** is a directed, leveled graph that represents actions and literals in alternating layers, capturing all possible states and actions up to a certain time step.

Construction of a Planning Graph:

1. **Levels:**
   - **S_0:** Represents the initial state.
   - **A_0:** Represents actions applicable in **S_0**.
   - Alternates between states (S_1, S_2, ...) and actions (A_1, A_2, ...).
2. **Termination:**
   - Stops when two consecutive levels are identical (levelled off).

Example: For the problem "**Have Cake and Eat Cake Too**":

- **S_0:** `{Have(Cake)}`
- **A_0:** `{Eat(Cake), Bake(Cake)}`
- **S_1:** `{Have(Cake), Eaten(Cake)}`
- **Mutex Links:** Highlight conflicts, e.g., eating and having the cake.

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
  PRECOND: $Have(Cake)$
  EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
  PRECOND: $\neg\, Have(Cake)$
  EFFECT: $Have(Cake))$

Figure 10.7 The "have cake and eat cake too" problem.

Figure 10.8 **shows t**he planning graph for the "have cake and eat cake too" problem up to level S2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at Si, then the

persistence actions for those literals will be mutex at Ai and we need not draw that mutex link.

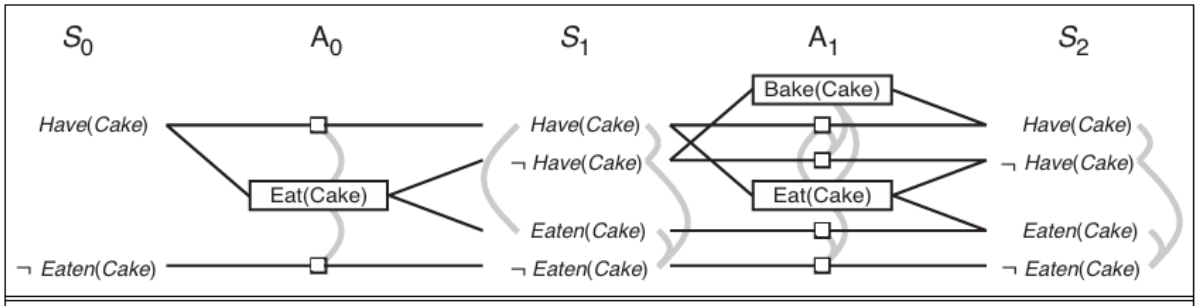


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i and we need not draw that mutex link.

5.2.3.1 Planning Graphs for Heuristic Estimation

A **planning graph** provides valuable insights into a problem once constructed. Here's how it aids heuristic estimation:

1. **Unsolvability Check**:
   If a goal literal does not appear in the final level of the graph, the problem is unsolvable.
2. **Estimating Goal Costs**:
   The cost of achieving a goal literal gi from an initial state s is estimated as the level at which gi first appears in the planning graph constructed from s. This is termed the **level cost** of gi. For example, in Figure 10.8, `Have(Cake)` has a level cost of 0, and `Eaten(Cake)` has a level cost of 1.
3. **Accuracy and Serial Graphs**:
   The level cost may not always align with reality because planning graphs allow multiple actions per level, while the heuristic only considers levels, not actions. To improve accuracy, **serial planning graphs** are often used. These enforce only one action per time step by adding mutual exclusion (mutex) links between non-persistence actions. Costs derived from serial graphs are more realistic.
4. **Estimating Conjunction Costs**:
   Estimating costs for a conjunction of goals involves three approaches:

- **Max-Level Heuristic**: Uses the maximum level cost of any goal. This heuristic is admissible but may lack precision.
- **Level-Sum Heuristic**: Assumes subgoal independence and sums the level costs of the goals. While inadmissible, it performs well for decomposable problems. For instance, for `Have(Cake)` ∧ `Eaten(Cake)`, this heuristic estimates 0+1=10 + 1 = 10+1=1, though the correct answer is 2, achieved by the plan `[Eat(Cake), Bake(Cake)]`. However, it may underestimate when certain actions, like `Bake(Cake)`, are missing.
- **Set-Level Heuristic**: Determines the level where all literals in the conjunction appear without mutual exclusivity. This heuristic is admissible, dominates the max-level heuristic, and performs effectively when subplans interact significantly. For the above example, it gives the correct value of 2 and identifies infeasibility (infinity) when `Bake(Cake)` is absent.

5. **Planning Graphs as Relaxed Problems**:
   Planning graphs model a relaxed problem by ensuring:
   - If a literal g does not appear at a level Si, no plan exists to achieve g within i steps.
   - If g does appear, it implies a plan exists with no **obvious flaws** (e.g., mutex violations between two actions or literals). However, it does not guarantee the absence of more subtle flaws involving three or more actions.
6. **Limits of Planning Graphs**:
   Planning graphs cannot detect some unsolvable problems. For instance, in a blocks-world scenario with the goal of stacking `A` on `B`, `B` on `C`, and `C` on `A` (an impossible circular tower), no mutex exists between any two subgoals. The impossibility emerges only when considering all three together. Detecting such cases would require searching the graph.

In summary, planning graphs are powerful tools for generating heuristics by efficiently solving a relaxed version of the problem. While they provide useful approximations, certain complexities may remain undetected without deeper search efforts.

5.2.3.2 The GRAPHPLAN Algorithm

This subsection explains how to extract a plan directly from the planning graph instead of using it solely for heuristic estimation. The **GRAPHPLAN algorithm** (illustrated in Figure 10.9) iteratively builds the planning graph using the **EXPAND-GRAPH** function. When all goal literals appear in the graph without mutual exclusions (mutex), the algorithm invokes **EXTRACT-SOLUTION** to search for a valid plan. If the search fails, GRAPHPLAN adds another level to the graph and retries. The process ends with failure if further expansion becomes futile.

The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
      if goals all non-mutex in S_t of graph then
          solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
          if solution ≠ failure then return solution
      if graph and nogoods have both leveled off then return failure
      graph ← EXPAND-GRAPH(graph, problem)
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Tracing GRAPHPLAN on the Spare Tire Problem

This section demonstrates how the **GRAPHPLAN** algorithm operates using the spare tire problem , with its planning graph depicted in Figure 10.10.

1. **Initialization**:
   GRAPHPLAN begins by initializing the planning graph with a single level (S0) that represents the initial state. The positive and negative fluents from the initial state are included, while unchanging positive literals (e.g., `Tire(Spare)`) and irrelevant negative literals are omitted.
   Since the goal `At(Spare, Axle)` does not appear in S0, EXTRACT-SOLUTION is not called. Instead, EXPAND-GRAPH adds actions to A0 whose preconditions are met in S0 (all actions except `PutOn(Spare,`

`Axle)`), along with persistence actions for S0's literals. The effects of these actions form S1, and mutex relations are identified and added.

2. **Expanding the Graph**:
   In S1, the goal `At(Spare, Axle)` is still absent, so EXPAND-GRAPH is called again to produce A1 and S2, resulting in the full planning graph (Figure 10.10). The process highlights examples of mutex relationships:
   - **Inconsistent Effects**: `Remove(Spare, Trunk)` is mutex with `LeaveOvernight` due to conflicting effects (`At(Spare, Ground)` vs. its negation).
   - **Interference**: `Remove(Flat, Axle)` is mutex with `LeaveOvernight` because one requires `At(Flat, Axle)` as a precondition, while the other negates it.
   - **Competing Needs**: `PutOn(Spare, Axle)` is mutex with `Remove(Flat, Axle)` as they depend on conflicting preconditions (`At(Flat, Axle)` vs. its negation).
   - **Inconsistent Support**: `At(Spare, Axle)` and `At(Flat, Axle)` in S2 are mutex because achieving `At(Spare, Axle)` requires `PutOn(Spare, Axle)`, which conflicts with the persistence of `At(Flat, Axle)`.
3. **Solution Extraction**:
   When all goal literals appear in S2 without mutex, EXTRACT-SOLUTION is invoked. This process is framed as a **Boolean constraint satisfaction problem (CSP)**, with variables representing actions, values being inclusion in or exclusion from the plan, and constraints being mutexes and goal/precondition requirements. Alternatively, it can be defined as a **backward search problem**, where:
   - The initial state includes the last graph level (Sn) and the unsatisfied goals.
   - Actions at Si are conflict-free subsets of Ai−1 that satisfy the current goals.
   - The goal is to reach S0 with all goals satisfied, with each action incurring a cost of 1.
4. **Example Execution**:
   Starting at S2 with the goal `At(Spare, Axle)`, the only relevant action is `PutOn(Spare, Axle)`, leading to S1 with goals `At(Spare, Ground)` and `¬At(Flat, Axle)`.
   - `At(Spare, Ground)` is achieved by `Remove(Spare, Trunk)`.
   - `¬At(Flat, Axle)` is achieved by either `Remove(Flat, Axle)` or `LeaveOvernight`, but the latter is mutex with `Remove(Spare, Trunk)`.

Thus, the chosen actions are `Remove(Spare, Trunk)` and `Remove(Flat, Axle)`, leading to S0 with goals `At(Spare, Trunk)` and `At(Flat, Axle)`, both satisfied. The solution consists of `Remove(Spare, Trunk)` and `Remove(Flat, Axle)` at A0, followed by `PutOn(Spare, Axle)` at A1.

5. **Handling Failure**:
   If EXTRACT-SOLUTION fails for a given level and set of goals, the pair is recorded as a "no-good," preventing redundant searches in subsequent iterations.
6. **Heuristic Guidance**:
   Extracting solutions is computationally intractable in the worst case, so heuristic guidance is essential. A greedy approach involves:
   - Prioritizing literals with the highest level cost.
   - Selecting actions with easier preconditions, where "easier" minimizes the sum or maximum of the preconditions' level costs.

This strategy balances efficiency with practicality for solving complex planning problems.

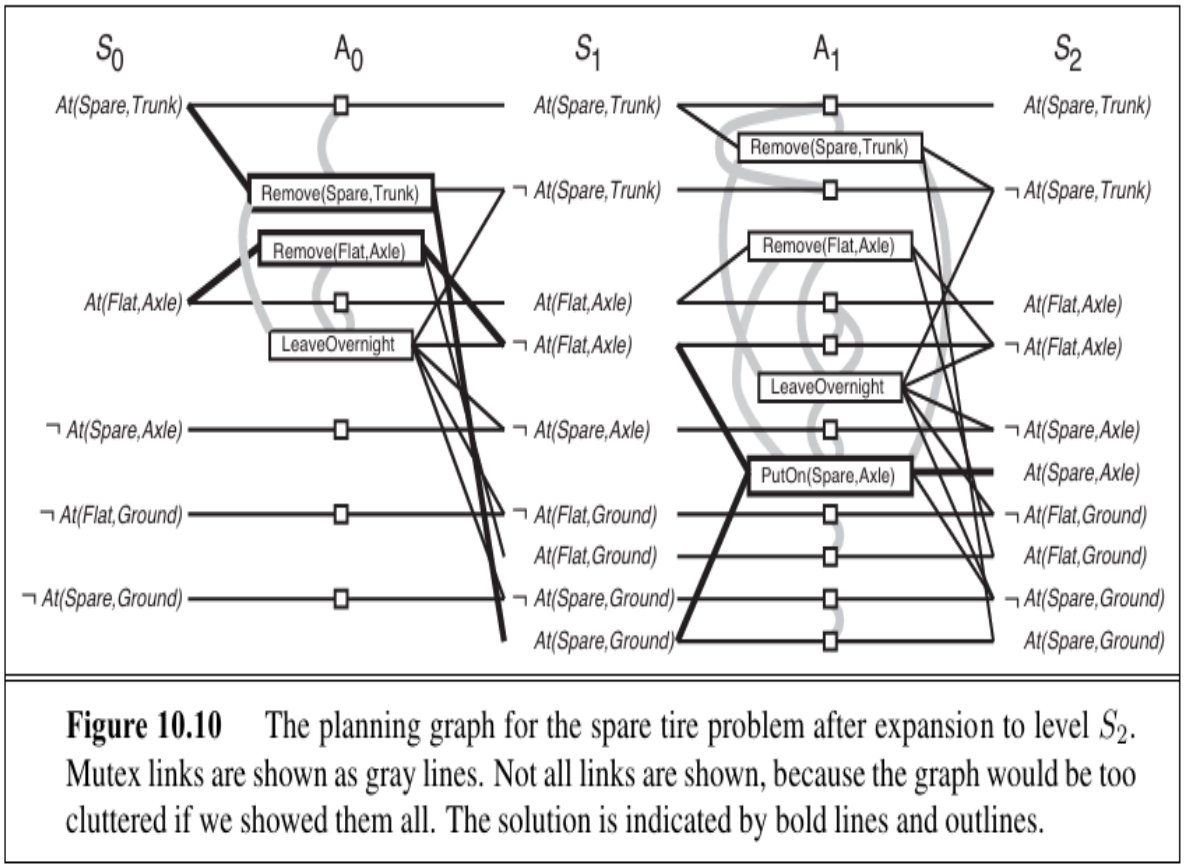


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

5.2.3.3 Termination of GRAPHPLAN

GRAPHPLAN ensures termination and returns failure when no solution exists. Here's an explanation of how it achieves this:

Why Not Stop at Level-Off?

The graph levels off when no new literals, actions, or mutex relations are added. However, this does not guarantee a solution. Consider an **air cargo problem** where one plane must transport nnn pieces of cargo from airport A to airport B, but only one piece fits in the plane at a time.

- The graph levels off at level 4, reflecting the steps required to load, fly, and unload a single piece of cargo.
- However, solving the problem requires 4n−1 steps, including return trips for additional cargo. Thus, leveling off does not necessarily mean a solution exists at that point.

When to Terminate?

The algorithm continues expanding the graph as long as new possibilities arise:

1. **No-Goods**: If EXTRACT-SOLUTION fails, it indicates that some goals are unachievable and are marked as no-goods.
2. **Leveling-Off**: Termination occurs when both the graph and the no-goods stabilize, i.e., no new literals, actions, or mutexes are added, and no further reduction in no-goods is possible. At this point, if no solution is found, GRAPHPLAN terminates with failure.

Proof of Leveling-Off

The key to proving that the graph and no-goods stabilize lies in the **monotonic properties** of planning graphs:

- **Literals Increase Monotonically**: Once a literal appears at a level, it persists at all subsequent levels due to persistence actions.
- **Actions Increase Monotonically**: If an action appears at a level, it remains in all subsequent levels, as its preconditions (which are literals) persist.
- **Mutexes Decrease Monotonically**: Mutex relations never reappear once removed.
    - **Inconsistent Effects** and **Interference** mutexes depend on inherent properties of actions and persist across levels.

    - **Competing Needs** mutexes depend on level-specific preconditions, which become achievable as actions increase monotonically.
- **No-Goods Decrease Monotonically**: If a set of goals is unachievable at one level, it remains unachievable at all previous levels, as persistence actions cannot retroactively make them achievable.

Finite Nature of Planning Graphs

Since actions and literals increase monotonically and are finite in number, the graph eventually stabilizes at a level where no new actions or literals are introduced. Similarly, mutexes and no-goods, which decrease monotonically and cannot fall below zero, also stabilize.

Termination Condition

When the graph stabilizes and either:

1. A goal is missing, or
2. Any goal is mutex with another,

the algorithm terminates, returning failure. This guarantees that further expansion would not yield a solution.

For a detailed formal proof, refer to **Ghallab et al. (2004)**.

5.2.4 Logic Programming

Logic programming is a method of building systems by writing rules and facts in a formal language. Problems are solved by reasoning based on this knowledge. This concept is summed up by Robert Kowalski's principle:
Algorithm = Logic + Control
This means that logic specifies *what* the system should do, while control defines *how* it should execute.

PROLOG

Prolog is the most popular logic programming language. It's used for quick prototyping and tasks like:

- Writing compilers
- Parsing natural language
- Creating expert systems in fields like law, medicine, and finance

Prolog Programs

Prolog programs consist of rules and facts (called definite clauses) written in a special syntax. Here's what makes Prolog different:

1. **Variables and Constants**: Variables are uppercase (e.g., `X`), and constants are lowercase (e.g., `john`).
2. **Clause Structure**: Instead of `A ∧ B ⇒ C`, Prolog writes it as `C :- A, B`. For example:

   ```
   criminal(X) :- american(X), weapon(Y),
   sells(X,Y,Z), hostile(Z).
   ```

   This means: "X is a criminal if X is American, Y is a weapon, X sells Y to Z, and Z is hostile."

3. **Lists**: `[E|L]` represents a list where `E` is the first item, and `L` is the rest.

Example: Appending Lists

Here's a Prolog program to join two lists, `X` and `Y`, into `Z`:

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

This means:

1. Appending an empty list to `Y` gives `Y`.
2. To append `[A|X]` to `Y`, the result is `[A|Z]` if appending `X` to `Y` gives `Z`.

You can also use it in reverse! For example, asking:

```
append(X,Y,[1,2]).
```

This query finds pairs of lists `X` and `Y` that combine to `[1,2]`. The answers are:

- `X=[]` and `Y=[1,2]`
- `X=[1]` and `Y=[2]`
- `X=[1,2]` and `Y=[]`

How Prolog Executes

Prolog works using **depth-first backward chaining**:

- It tries rules one by one, in the order written.
- It stops as soon as a solution is found.
- Some features make it faster but can cause issues:
    - **Arithmetic Built-ins**: It calculates results directly. For example:
        - `X is 4+3` → Prolog sets `X = 7`.
        - `5 is X+Y` → Fails because Prolog doesn't solve general equations.
    - **Side Effects**: Predicates like `assert` (add facts) and `retract` (remove facts) can behave unpredictably.
    - **Infinite Recursion**: Prolog doesn't check for infinite loops, so wrong rules might cause it to hang.

Design Philosophy : Prolog balances **declarative logic** (what should happen) with **execution efficiency** (how it runs). While not perfect, it's a powerful tool for certain types of tasks!

5.2.4.1 Efficient Implementation of Logic Programs

Prolog programs can be executed in two modes: **interpreted** and **compiled**. Each mode has distinct characteristics and optimizations:

Interpreted Mode
In this mode, Prolog functions like the **FOL-BC-ASK algorithm** (Figure 9.6), treating the program as a knowledge base. However, Prolog interpreters include optimizations for better efficiency. Two key improvements are:

1. **Global Stack of Choice Points**:
    - Instead of explicitly managing iterations over possible results, Prolog uses a **global stack of choice points** to track alternatives considered in the **FOL-BC-OR** step.
    - This approach is not only more efficient but also simplifies debugging, as the debugger can traverse up and down the stack to inspect states.
2. **Logic Variables and Trails**:
    - Prolog uses **logic variables** that dynamically remember their current bindings. At any time, a variable is either unbound or bound to a specific value. These bindings implicitly define the substitution for the current proof branch.
    - New variable bindings extend the path, but attempts to rebind an already bound variable fail due to unification constraints.
    - When backtracking occurs after a goal fails, variables are **unbound** in reverse order using a **trail stack**. Each variable bound by `UNIFY-VAR` is pushed onto the trail, and during backtracking, variables are unbound as they are popped from the trail.

Despite these optimizations, Prolog interpreters still require thousands of machine instructions per inference step due to the overhead of operations like index lookups, unification, and recursive call stack management. In essence, the interpreter processes each query as if encountering the program for the first time, repeatedly finding clauses to match the goal.

Compiled Mode
Compiled Prolog programs offer significant performance improvements by tailoring inference procedures to specific sets of clauses. This eliminates much of the interpretation overhead:

1. **Optimized Clause Matching**:
    - Unlike interpreters, compiled Prolog "remembers" which clauses match a given goal, streamlining the process.

- It generates a dedicated inference procedure (essentially a small theorem prover) for each predicate, avoiding the need for repetitive clause searching.

2. **Open-Coded Unification**:
   - Compilers can create **open-coded unification** routines for specific calls, bypassing the need for general-purpose term structure analysis. This improves execution speed.
3. **Intermediate Language Compilation**:
   - Direct compilation of Prolog to machine code is inefficient due to the mismatch between Prolog semantics and modern processor architectures.
   - Instead, Prolog is typically compiled into an **intermediate language** like the **Warren Abstract Machine (WAM)**, which abstracts Prolog's operations for efficient execution.
   - The WAM, developed by David H. D. Warren, is a widely-used intermediate instruction set designed for Prolog, which can either be interpreted or further compiled into machine code.
4. **High-Level Language Translation**:
   - Some Prolog compilers translate Prolog code into high-level languages like **Lisp** or **C**, leveraging their compilers to generate machine code.

For example, the Prolog `append` predicate can be compiled into efficient low-level code as shown in Figure 9.8, using strategies like choice points and term structure analysis to enhance performance.

```
procedure APPEND(ax, y, az, continuation)

  trail ← GLOBAL-TRAIL-POINTER()
  if ax = [] and UNIFY(y, az) then CALL(continuation)
  RESET-TRAIL(trail)
  a, x, z ← NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
  if UNIFY(ax, [a | x]) and UNIFY(az, [a | z]) then APPEND(x, y, z, continuation)
```

Figure 9.8 Pseudocode representing the result of compiling the `Append` predicate. The function NEW-VARIABLE returns a new variable, distinct from all other variables used so far. The procedure CALL(*continuation*) continues execution with the specified continuation.

The `append` predicate in Prolog can be compiled into an optimized form, such as the example shown in Figure 9.8. Several noteworthy points highlight how this compilation enhances efficiency:

Transforming Clauses into Procedures

- Instead of searching the knowledge base for `append` clauses during execution, the clauses are compiled into a **procedure**.
- Inferences are then performed by directly calling the procedure, significantly improving execution speed and reducing overhead.

Trail Management for Variable Bindings

- Prolog maintains **variable bindings** on a **trail**, which records their current state.
- The procedure begins by saving the trail's state. If the first clause fails, `RESET-TRAIL` restores the original state, undoing any bindings created by the initial call to `UNIFY`.
- This ensures correctness and prepares Prolog to explore alternative paths during backtracking.

Use of Continuations for Choice Points

- **Continuations** are critical for handling **choice points** during execution. A continuation packages a procedure and its arguments, specifying what should happen next when a goal succeeds.
- This approach prevents premature termination of a procedure like `append` when multiple solutions exist. Each success triggers the continuation, enabling all possibilities to be explored.
- For example, if `append` is invoked at the top level:
    - If the first list is empty and the second unifies with the third, the predicate succeeds.
    - The continuation is then called with the current bindings on the trail to perform the next steps (e.g., printing variable bindings).

Impact of Prolog Compilation

Before the advancements by Warren and others, Prolog's performance was too slow for practical use. However, compilers like the **Warren Abstract Machine**

(WAM) enabled Prolog to achieve execution speeds competitive with C on standard benchmarks (Van Roy, 1990).

Prolog's ability to express complex logic, such as planners or natural language parsers, in just a few lines makes it ideal for prototyping small-scale AI research projects, often surpassing C in ease of use.

Parallelism in Prolog :

Prolog also leverages **parallelism** to achieve substantial speedups by exploiting the independence of branches in logic programming:

1. **OR-Parallelism**:
    - Occurs when a goal can unify with multiple clauses in the knowledge base.
    - Each unification forms an independent branch of the search space, which can be solved in parallel.
2. **AND-Parallelism**:
    - Arises from solving multiple conjuncts in the body of a rule simultaneously.
    - Unlike OR-parallelism, AND-parallelism is more challenging, as it requires **consistent bindings** across all conjunctive branches. Communication between branches ensures a globally valid solution.

Dynamic Programming for Efficiency

By combining compilation techniques with parallelization and dynamic programming, Prolog minimizes redundant computations and optimizes search processes. These advancements make Prolog an efficient and desirable choice for logic programming in AI and other computational fields.

5.2.4.2 Redundant Inference and Infinite Loops in Prolog

Prolog's reliance on **depth-first search (DFS)** introduces challenges when dealing with search trees containing **repeated states** and **infinite paths**, creating significant inefficiencies.

Infinite Loops in Depth-First Search

Consider the following Prolog program, which checks if a path exists between two points in a directed graph:

```
path(X, Z) :- link(X, Z).
path(X, Z) :- path(X, Y), link(Y, Z).
```

Given a simple three-node graph with facts `link(a, b)` and `link(b, c)` (Figure 9.9(a)):

1. If queried as `path(a, c)`, the proof tree (Figure 9.10(a)) demonstrates a successful result.
2. However, changing the order of the clauses:

   ```
   path(X, Z) :- path(X, Y), link(Y, Z).
   path(X, Z) :- link(X, Z).
   ```

   causes Prolog to follow an **infinite loop** (Figure 9.10(b)).

This happens because Prolog's DFS prioritizes depth over breadth and fails to detect cyclic dependencies. Consequently, Prolog is **incomplete** as a theorem prover for definite clauses—even for simple **Datalog programs** like this one.

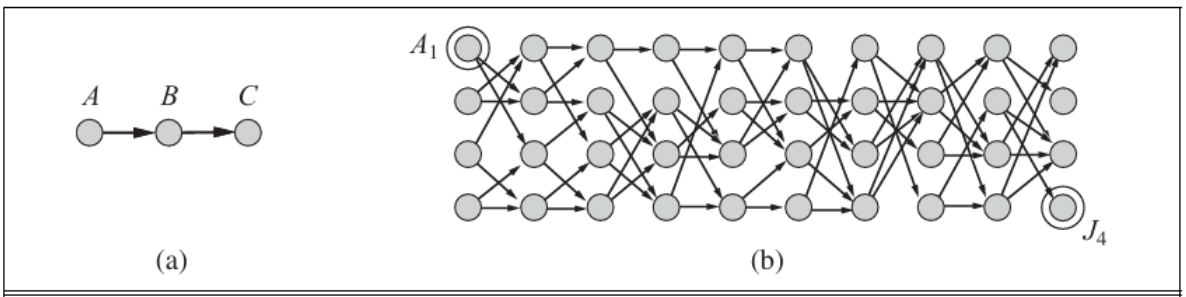


Figure 9.9 (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A_1 to J_4 requires 877 inferences.

Forward chaining, in contrast, avoids this issue by systematically generating facts. Once `path(a, b)`, `path(b, c)`, and `path(a, c)` are inferred, the process halts.

Redundant Computations

Another major drawback of DFS in Prolog is **redundant computations**, particularly in graph problems.

For example:

- When finding a path from `A1` to `J4` in a more complex graph (Figure 9.9(b)), Prolog performs **877 inferences**, exploring multiple unnecessary paths, including those leading to unreachable nodes.
- This redundancy resembles the **repeated-state problem** in Chapter 3.

Forward chaining, applied to the same problem, is far more efficient:

- It generates at most n^2 `path(X, Y)` facts for n nodes, avoiding repeated calculations.
- For the problem in Figure 9.9(b), forward chaining requires only **62 inferences**, compared to the exponential growth of DFS.

Dynamic Programming in Forward Chaining

Forward chaining for graph search exemplifies **dynamic programming**, where solutions to smaller subproblems are incrementally combined to solve larger ones. This approach eliminates the inefficiencies of redundant inferences and infinite loops inherent in Prolog's DFS, making it a more effective strategy for certain problem domains.

Subproblems are cached to prevent recomputation. A similar effect can be achieved in a backward chaining system through **memorization**—caching solutions to subgoals as they are found and reusing these solutions when the subgoal is encountered again, instead of repeating the computation. This is the approach used by **tabled logic programming systems**, which implement efficient storage and retrieval mechanisms for memoization. Tabled logic

programming combines the goal-directed nature of backward chaining with the dynamic programming efficiency of forward chaining. It is also **complete for Datalog knowledge bases**, meaning the programmer has less concern about infinite loops. However, infinite loops can still occur with predicates like `father(X,Y)`, which may refer to a potentially unbounded number of objects.

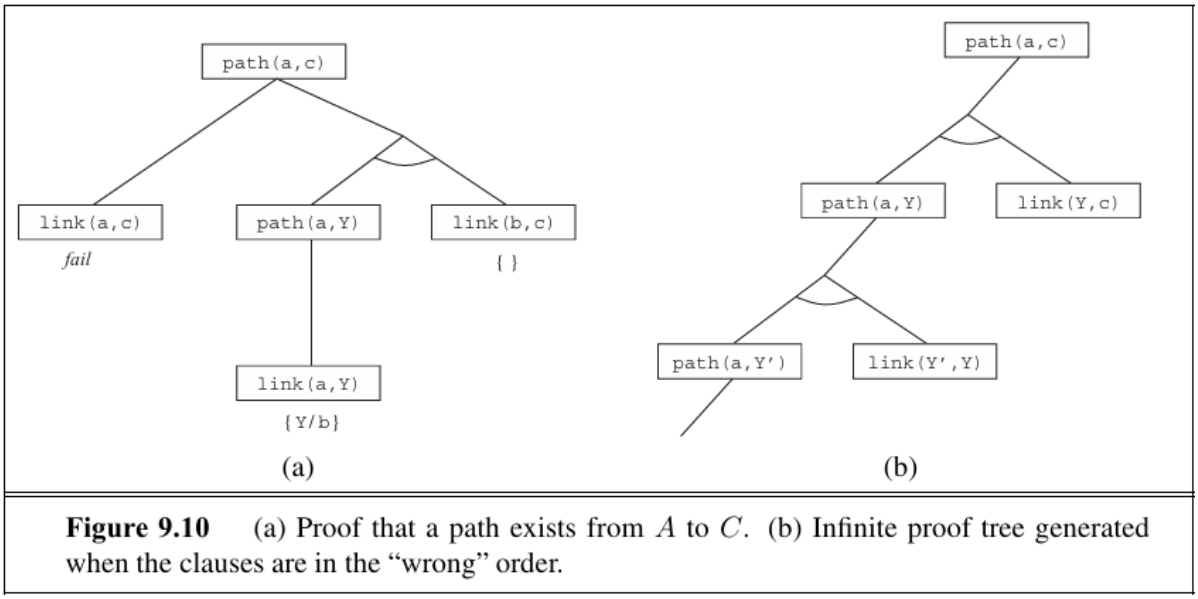


Figure 9.10 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

Source Book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson,2015

5.2.4.3 Database Semantics of Prolog

Prolog employs **database semantics**. The **unique names assumption** states that each Prolog constant and ground term refers to a distinct object, while the **closed world assumption** asserts that only the sentences entailed by the knowledge base are considered true. In Prolog, there is no way to assert that a sentence is false, which makes Prolog less expressive than first-order logic (FOL). However, this limitation contributes to Prolog's efficiency and conciseness.

Consider the following Prolog assertions about course offerings:

```
Course(CS, 101), Course(CS, 102), Course(CS, 106),
Course(EE, 101).
```

Under the **unique names assumption**, `CS` and `EE` are distinct, as are the course numbers `101`, `102`, and `106`. This means that there are exactly four distinct courses. According to the **closed-world assumption**, there are no other courses, so the total number of courses is exactly four.

In contrast, if these were assertions in **FOL**, they would only imply that there are at least one and at most infinity courses. This is because FOL does not deny the existence of other unmentioned courses, nor does it specify that the mentioned courses are distinct. In FOL, the assertions would be expressed as:

```
Course(d, n) ⇔ (d=CS ∧ n=101) ∨ (d=CS ∧ n=102) ∨ (d=CS
∧ n=106) ∨ (d=EE ∧ n=101).
```

This is the **completion** of the Prolog assertions, which expresses the idea that there are at most four courses in FOL. To express that there are at least four courses in FOL, we would need to expand the equality predicate as follows:

```
x = y ⇔ (x=CS ∧ y=CS) ∨ (x=EE ∧ y=EE) ∨ (x=101 ∧ y=101)
∨ (x=102 ∧ y=102) ∨ (x=106 ∧ y=106).
```

While the completion is useful for understanding database semantics, it is often more efficient to work directly with Prolog or another database semantics system for practical problems. Translating the problem into FOL and reasoning with a full FOL theorem prover can be more cumbersome and less efficient.

5.2.4.4 Constraint Logic Programming

Standard Prolog solves Constraint Satisfaction problems (CSP) using a backtracking algorithm, However, backtracking works by enumerating the domains of the variables, making it suitable only for **finite-domain CSPs**. This means there must be a finite number of possible solutions for any goal with unbound variables.

For **infinite-domain CSPs**, such as those involving integer or real-valued variables, different algorithms are needed, like bounds propagation or linear programming.

Example: Triangle Inequality

Consider the following example, where `triangle(X, Y, Z)` is a predicate that holds if the three arguments satisfy the triangle inequality:

```
triangle(X, Y, Z) :- X > 0, Y > 0, Z > 0, X + Y >= Z,
Y + Z >= X, X + Z >= Y.
```

- When we query `triangle(3, 4, 5)`, Prolog successfully returns true.
- However, when we query `triangle(3, 4, Z)`, no solution is found because the subgoal `Z >= 0` cannot be handled by Prolog—Prolog cannot compare an unbound value to 0.

Constraint Logic Programming (CLP) extends traditional logic programming by allowing variables to be **constrained** rather than bound. In CLP, the solution

is the most specific set of constraints that can be derived from the knowledge base for the query variables.
For example, the solution to the query `triangle(3, 4, Z)` would be the constraint `7 >= Z >= 1`, which means that `Z` must be between 1 and 7. In contrast, standard logic programs are a special case of CLP, where the solution constraints are always equality constraints (bindings).
CLP systems include various **constraint-solving algorithms** tailored to the types of constraints allowed in the system. For example, a CLP system that supports **linear inequalities** on real-valued variables might employ a **linear programming algorithm** to solve such constraints.

Flexible Query Solving in CLP :
CLP systems offer more flexibility than traditional logic programming. Rather than relying solely on **depth-first, left-to-right backtracking**, CLP systems can use more efficient algorithms, including:

- **Heuristic conjunct ordering**
- **Backjumping**
- **Cutset conditioning**

These approaches improve the efficiency of solving logic programming queries and integrate techniques from **constraint satisfaction algorithms**, **logic programming**, and **deductive databases**.

Control Over Search Order:

Several CLP systems allow programmers to control the order in which inferences are made. For example, the **MRS language** (Genesereth and Smith, 1981; Russell, 1985) lets the programmer define **metarules** to determine the order of conjunct evaluations. A programmer could write a rule to prioritize goals with fewer variables or define domain-specific rules for particular predicates.
In summary, CLP systems enhance traditional logic programming by incorporating constraint-solving techniques, providing greater flexibility and efficiency for handling a wide range of problems, from finite-domain CSPs to complex real-valued constraints.

Examples of Prolog Programs

1. To Find the Sum of Two Numbers

```
% Rule to find the sum of two numbers
sum(X, Y, Z) :- Z is X + Y.

% Example Query:
% ?- sum(5, 3, Result).
% Result = 8.
```

---

2. To Swap Two Numbers

```
% Rule to swap two numbers
swap(X, Y, SwappedX, SwappedY) :- SwappedX = Y,
SwappedY = X.

% Example Query:
% ?- swap(5, 3, A, B).
% A = 3,
% B = 5.
```

---

3. To Add Two Lists

```
% Rule to add corresponding elements of two lists
add_lists([], [], []). % Base case: Adding two empty
lists gives an empty list.
add_lists([A|X], [B|Y], [C|Z]) :- C is A + B,
add_lists(X, Y, Z).

% Example Query:
% ?- add_lists([1, 2, 3], [4, 5, 6], Result).
% Result = [5, 7, 9].
```

---

4. To Find the Factorial of a Number

```
% Base case: Factorial of 0 is 1
factorial(0, 1).

% Recursive case: N! = N * (N-1)!
```

```
factorial(N, Result) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, SubResult),
    Result is N * SubResult.

% Example Query:
% ?- factorial(5, Result).
% Result = 120.
```

Example Usage

1. **Sum of Two Numbers**

   ```
   ?- sum(10, 15, Result).
   Result = 25.
   ```

2. **Swap Two Numbers**

   ```
   ?- swap(7, 9, A, B).
   A = 9,
   B = 7.
   ```

3. **Add Two Lists**

   ```
   ?- add_lists([2, 4, 6], [1, 3, 5], Result).
   Result = [3, 7, 11].
   ```

4. **Find Factorial**

   ```
   ?- factorial(4, Result).
   Result = 24.
   ```