

# Module 5

Inferences in FOL and Classical Planning

# Contents

## **1. Inference in First Order Logic :**

- Backward Chaining,
- Resolution

## **2. Classical Planning:**

- Definition of Classical Planning,
- Algorithms for Planning as State-Space Search,
- Planning Graphs Inference in First Order Logic:

## 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:

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

# 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,  $\theta$ ) yields a substitution  
for each rule (lhs  $\Rightarrow$  rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do  
  (lhs, rhs)  $\leftarrow$  STANDARDIZE-VARIABLES((lhs, rhs))  
  for each  $\theta'$  in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal,  $\theta$ )) do  
    yield  $\theta'$ 
```

---

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

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

# Overview of the Algorithm

**Goal:** The purpose of the algorithm is to determine whether a query (goal) can be derived from a given knowledge base (KB).

## 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.**

## 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.



# Key Terms Used

- **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.
- $\theta$ : The substitution carried into the current function call.
- $\theta'$ : A substitution produced by solving the first sub-goal in FOL-BC-AND.
- $\theta''$ : A substitution produced by solving the remaining sub-goals using the updated  $\theta'$ .

# Key Points

- Backward chaining focuses on **proving the goal by breaking** it into **smaller sub-goals** and matching them to rules in the KB.
- It uses **unification and substitutions** to ensure variable consistency.
- It **recursively checks all rules** until the **query is satisfied** or fails.

# Backward Chaining Algorithm : Example

## Knowledge Base:

1.  $\text{Parent}(x, y) \Rightarrow \text{Ancestor}(x, y)$  (Rule 1)
2.  $\text{Parent}(x, z) \wedge \text{Ancestor}(z, y) \Rightarrow \text{Ancestor}(x, y)$  (Rule 2)
3.  $\text{Parent}(\text{John}, \text{Mary})$  (Fact)
4.  $\text{Parent}(\text{Mary}, \text{Sam})$  (Fact)

**Query:**  $\text{Ancestor}(\text{John}, \text{Sam})$

---

# 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:  $\text{Ancestor}(\text{John}, \text{Sam})$
- Fetch rules for  $\text{Ancestor}$  :
  1. Rule 1:  $\text{Parent}(x, y) \Rightarrow \text{Ancestor}(x, y)$
  2. Rule 2:  $\text{Parent}(x, z) \wedge \text{Ancestor}(z, y) \Rightarrow \text{Ancestor}(x, y)$

### Case 1: Use Rule 1

- lhs =  $\text{Parent}(x, y)$ , rhs =  $\text{Ancestor}(x, y)$ .
- Unify  $\text{Ancestor}(\text{John}, \text{Sam})$  with  $\text{Ancestor}(x, y)$  :
  - Substitution:  $\theta = \{x=\text{John}, y=\text{Sam}\}$ .
- Sub-goal:  $\text{Parent}(\text{John}, \text{Sam})$ .

### 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.

## 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:  $[\text{Parent}(\text{John}, z), \text{Ancestor}(z, \text{Sam})]$ .

### 1. First sub-goal ( $\text{Parent}(\text{John}, z)$ ):

- Calls:  $\text{FOL-BC-OR}(\text{KB}, \text{Parent}(\text{John}, z), \theta)$ .
- Matches:  $\text{Parent}(\text{John}, \text{Mary})$ .
- Substitution:  $\{z=\text{Mary}\}$ .
- Update  $\theta'$ :  $\{x=\text{John}, y=\text{Sam}, z=\text{Mary}\}$ .

### 2. Second sub-goal ( $\text{Ancestor}(z, \text{Sam})$ ):

- Calls:  $\text{FOL-BC-OR}(\text{KB}, \text{Ancestor}(\text{Mary}, \text{Sam}), \theta')$ .
- Unify with Rule 1:  $\text{Parent}(x, y) \Rightarrow \text{Ancestor}(x, y)$ .
- Sub-goal:  $\text{Parent}(\text{Mary}, \text{Sam})$ .

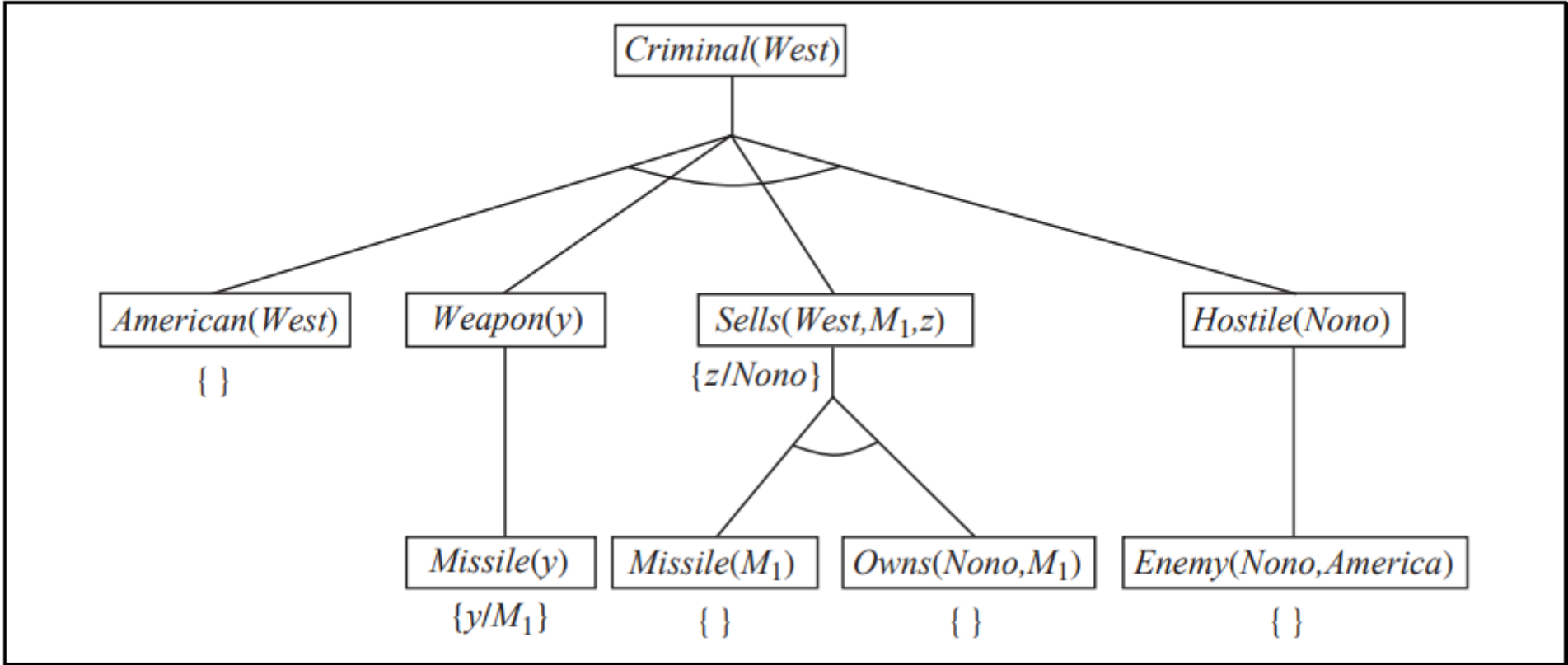


## Step 6: FOL-BC-AND

- 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  $Nono$ .

# Contents

## **1. Inference in First Order Logic :**

- Backward Chaining,
- Resolution

## **2. Classical Planning:**

- Definition of Classical Planning,
- Algorithms for Planning as State-Space Search,
- Planning Graphs Inference in First Order Logic:

# De Morgan's Laws in First-Order Logic (FOL)

De Morgan's Laws in **First-Order Logic (FOL)** are rules that describe how negation interacts with conjunction ( $\wedge$ ) and disjunction ( $\vee$ ) in logical expressions. They also extend to quantifiers ( $\forall$  and  $\exists$ ) in FOL.

## 1. For Logical Connectives:

- $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$

*(The negation of a conjunction is equivalent to the disjunction of the negations.)*

- $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$

*(The negation of a disjunction is equivalent to the conjunction of the negations.)*

## 2. For Quantifiers:

De Morgan's Laws apply to quantifiers ( $\forall$  and  $\exists$ ) as well:

- $\neg(\forall x P(x)) \equiv \exists x (\neg P(x))$

*(The negation of "for all  $x$ ,  $P(x)$ " is equivalent to "there exists an  $x$  such that  $P(x)$  is not true.")*

- $\neg(\exists x P(x)) \equiv \forall x (\neg P(x))$

*(The negation of "there exists an  $x$  such that  $P(x)$ " is equivalent to "for all  $x$ ,  $P(x)$  is not true.")*

## Example 1 (Connectives):

If  $P(x) = x > 5$  and  $Q(x) = x < 10$ , then:

- $\neg(P(x) \wedge Q(x)) \equiv (\neg P(x) \vee \neg Q(x))$
- $\neg(x > 5 \wedge x < 10) \equiv (x \leq 5 \vee x \geq 10)$

## Example 2 (Quantifiers):

If  $P(x) = x > 5$ , then:

- $\neg(\forall x P(x)) \equiv \exists x (\neg P(x))$

*"Not all  $x$  are greater than 5" is equivalent to "there exists some  $x$  that is not greater than 5."*

- $\neg(\exists x P(x)) \equiv \forall x (\neg P(x))$

*"There does not exist an  $x$  greater than 5" is equivalent to "all  $x$  are not greater than 5."*

# Distributive law in First Order Logic

## 1. Distributive Law of $\vee$ over $\wedge$ :

$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$

*(Disjunction distributes over conjunction.)*

## 2. Distributive Law of $\wedge$ over $\vee$ :

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$

*(Conjunction distributes over disjunction.)*



### 3. Distributive Law with $\forall$ :

- $\forall x (P(x) \wedge Q(x)) \equiv (\forall x P(x)) \wedge (\forall x Q(x))$   
*(Universal quantifier distributes over conjunction.)*
- $\forall x (P(x) \vee Q(x)) \not\equiv (\forall x P(x)) \vee (\forall x Q(x))$   
*(Universal quantifier does **not** distribute over disjunction in general.)*

### 4. Distributive Law with $\exists$ :

- $\exists x (P(x) \vee Q(x)) \equiv (\exists x P(x)) \vee (\exists x Q(x))$   
*(Existential quantifier distributes over disjunction.)*
- $\exists x (P(x) \wedge Q(x)) \not\equiv (\exists x P(x)) \wedge (\exists x Q(x))$   
*(Existential quantifier does **not** distribute over conjunction in general.)*

# What is Tautology ?

- A **tautology** is a statement or logical formula that is always **true**, regardless of the truth values of its individual components.
- **Example:  $P \vee \neg P$**   
This means "P or not P." No matter whether P is true or false, one of them will always be true. **Hence, it's a tautology**

$P$	$\neg P$	$P \vee \neg P$
True	False	True
False	True	True

The result is always **true**, so  $P \vee \neg P$  is a tautology.

# Common Tautologies in Logic:

1.  $P \vee \neg P$  ("Law of excluded middle")
  2.  $P \implies P$  ("Self-implication")
  3.  $(P \wedge Q) \implies P$  ("Conjunction implies one operand")
- In essence, tautologies are statements that cannot be false and are always true in classical logic.

# 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 ( $\Leftrightarrow$ ):** Replace each biconditional ( $\Leftrightarrow$ ) with an equivalent expression in terms of conjunction ( $\wedge$ ), disjunction ( $\vee$ ), and negation ( $\neg$ ).
  - Example: Replace  $A \Leftrightarrow B$  with  $(A \Rightarrow B) \wedge (B \Rightarrow A)$
- 2. Eliminate Implications ( $\Rightarrow$ ):** Replace each implication ( $\Rightarrow$ ) with an equivalent expression using conjunction and negation.
  - Example: Replace  $A \Rightarrow B$  with  $\neg A \vee B$
- 3. Move Negations Inward ( $\neg$ ):** Apply De Morgan's laws and distribute negations inward to literals.
  - $\neg(\neg A) \equiv A$
  - $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$
  - $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$
- 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:  $H = (P \wedge Q) \vee (R \wedge S \wedge T)$ . Now, let's distribute disjunctions over conjunctions:  
 $H = (P \vee (R \wedge S \wedge T)) \wedge (Q \vee (R \wedge S \wedge T))$

# Example: CNF Sentence

- $\forall x \text{ American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$  becomes, in CNF,
- $\neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x, y, z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x)$  .

# Example

**We illustrate the procedure by translating the sentence**

- “Everyone who loves all animals is loved by someone,” or
- $\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{ Loves}(y, x)]$  .



# Steps

- **Eliminate implications:**  $\forall x [\neg\forall y \neg\text{Animal}(y) \vee \text{Loves}(x, y)] \vee [\exists y \text{Loves}(y, x)]$  .
- **Move  $\neg$  inwards:** In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have
  - $\neg\forall x p$  becomes  $\exists x \neg p$
  - $\neg\exists x p$  becomes  $\forall x \neg p$  .
- **Our sentence goes through the following transformations:**
  - $\forall x [\exists y \neg(\neg\text{Animal}(y) \vee \text{Loves}(x, y))] \vee [\exists y \text{Loves}(y, x)]$  .
  - $\forall x [\exists y \neg\neg\text{Animal}(y) \wedge \neg\text{Loves}(x, y)] \vee [\exists y \text{Loves}(y, x)]$  .
  - $\forall x [\exists y \text{Animal}(y) \wedge \neg\text{Loves}(x, y)] \vee [\exists y \text{Loves}(y, x)]$  .
- **Standardize variables:** For sentences like  $(\exists x P(x))\vee(\exists x Q(x))$  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
  - $\forall x [\exists y \text{Animal}(y) \wedge \neg\text{Loves}(x, y)] \vee [\exists z \text{Loves}(z, x)]$  .

- **Skolemize:** Skolemization is the process of **removing existential quantifiers by elimination**. Translate  $\exists x P(x)$  into  $P(A)$ , where  $A$  is a new constant.
  - **Example :**
    - $\forall x [\text{Animal}(A) \wedge \neg \text{Loves}(x, A)] \vee \text{Loves}(B, x)$  ,
    - $\forall x [\text{Animal}(F(x)) \wedge \neg \text{Loves}(x, F(x))] \vee \text{Loves}(G(z), x)$  . 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:
  - $[\text{Animal}(F(x)) \wedge \neg \text{Loves}(x, F(x))] \vee \text{Loves}(G(z), x)$  .
- **Distribute  $\vee$  over  $\wedge$ :**

$$[\text{Animal}(F(x)) \vee \text{Loves}(G(z), x)] \wedge [\neg \text{Loves}(x, F(x)) \vee \text{Loves}(G(z), x)] .$$

## 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.
- Resolution is a method in logic that can prove whether a set of statements is **unsatisfiable**.
- **Unsatisfiable set of statements**: Means the statements can't all be true together.

# Resolution Rule

- 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**.

# Example 1:

Consider two logical statements:

1.  $P \vee Q$
2.  $\neg P \vee R$

**Applying resolution:** Resolve the statements by eliminating P:

- $P \vee Q$
- $\neg P \vee R$
- Resolving P and  $\neg P$ :  **$Q \vee R$**

The resulting statement  **$Q \vee R$**  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.

# Example 2

**Clause 1:  $(P \vee Q \vee R)$**

**Clause 2:  $(\neg P \vee \neg Q \vee S)$**

**Apply Resolution**

- Resolving P and  $\neg P$ :  $(Q \vee R) \vee (\neg Q \vee S)$
- Resolving Q and  $\neg Q$  gives  $(R \vee S)$

**$(R \vee S)$  is the resolvent.**

# Example 3

Premises:

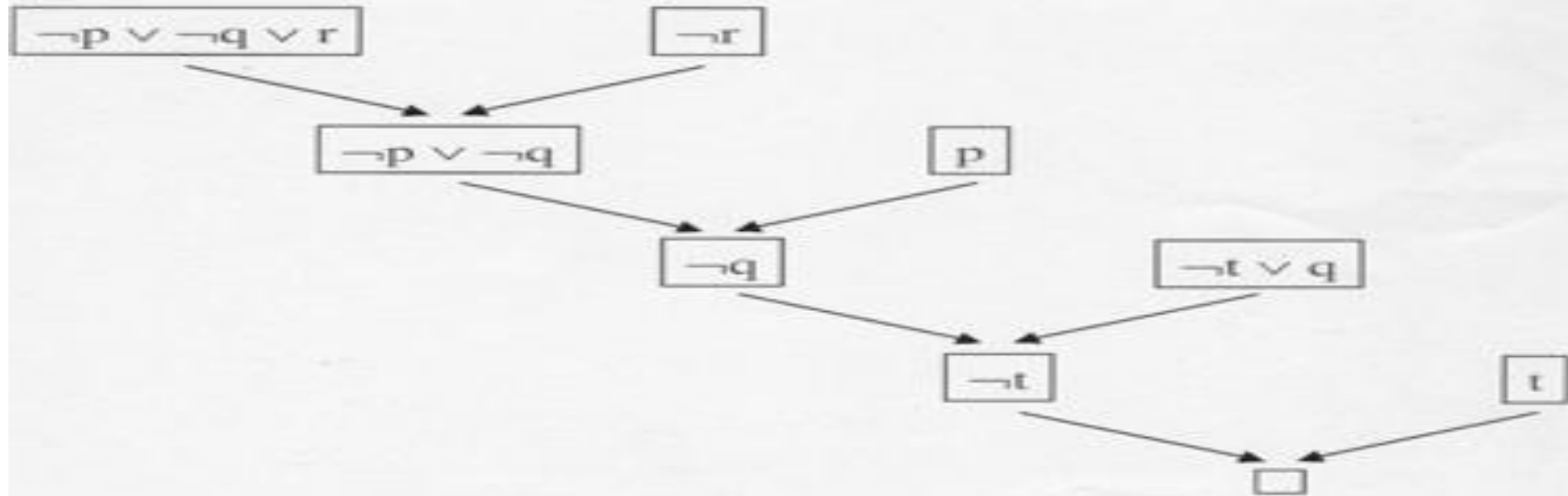
$p$   
 $(p \wedge q) \Rightarrow r$   
 $(s \vee t) \Rightarrow q$   
 $t$



$p$
$\neg p \vee \neg q \vee r$
$\neg s \vee q$
$\neg t \vee q$
$t$

CNF

A resolution proof of  $r$ :



# Note

- The empty clause derived always implies the assumed **negation(contradiction)** is false.
- In the example, the derivation of the empty clause **proves that "r"** is indeed a **logical consequence** of the premises.



# Proof By Resolution Process includes the following steps in general

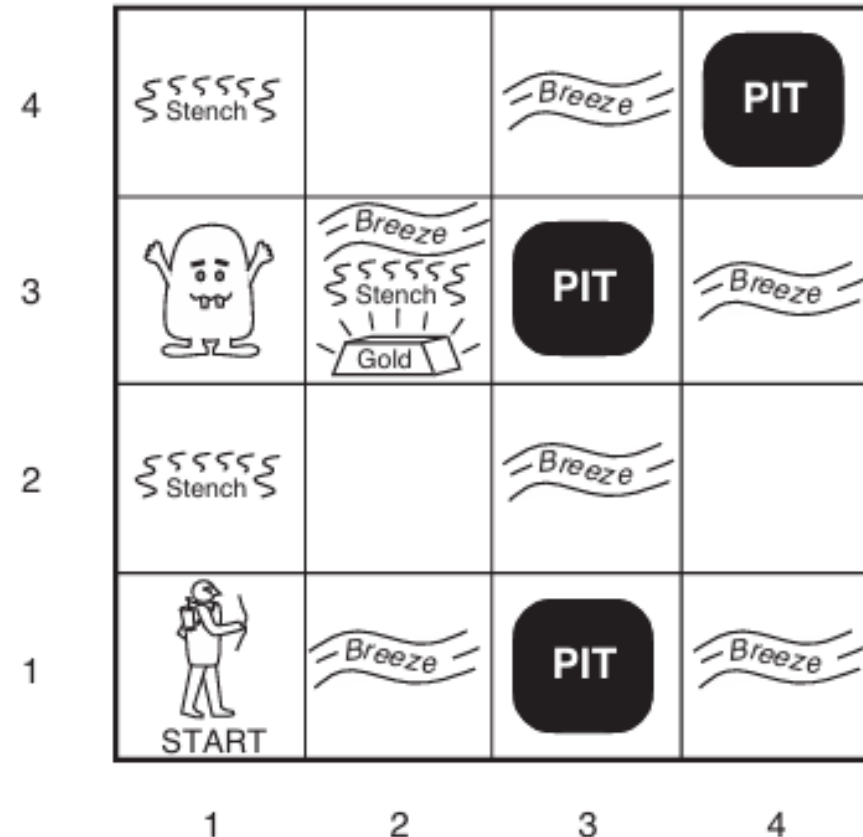
- 1. Initial Set of Clauses (Knowledge Base)**
- 2. Convert the Statement into Clausal Form**
- 3. Skolemization**
- 4. Standardize Variables**
- 5. Unification**
- 6. Resolution Rule**
- 7. Iterative Application**

# Example

- 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] is true or false**.

# Knowledge Base (KB)

1.  $W[1,1] \vee P[1,2]$
2.  $\neg W[1,1] \vee \neg P[1,2]$
3.  $B[1,2] \Rightarrow P[1,2]$
4.  $\neg B[1,2] \Rightarrow \neg P[1,2]$



# Convert the Knowledge Base (KB) into CNF

1.  $W[1,1] \vee P[1,2]$

2.  $\neg W[1,1] \vee \neg P[1,2]$

3.  $B[1,2] \Rightarrow P[1,2]$

4.  $\neg B[1,2] \Rightarrow \neg P[1,2]$

1.  $W[1,1] \vee P[1,2]$

2.  $\neg W[1,1] \vee \neg P[1,2]$

3.  $\neg B[1,2] \vee P[1,2]$

4.  $B[1,2] \vee \neg P[1,2]$

## Negated Conclusion:

Let's say we want to prove the negation of the statement:

**$\neg \text{PitIn}[1,2]$**

# Apply Resolution:

1.  $W[1,1] \vee P[1,2]$  ,  $\neg P[1,2]$  resolves into  $W[1,1]$
2.  $\neg W[1,1] \vee \neg P[1,2]$  ,  $W[1,1]$  resolves into  $\neg P[1,2]$
3.  $\neg B[1,2] \vee P[1,2]$  ,  $\neg P[1,2]$  resolves into  $\neg B[1,2]$
4.  $B[1,2] \vee \neg P[1,2]$  ,  $\neg B[1,2]$  resolves into  $\neg P[1,2]$

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

# The resolution inference rule

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

# The resolution inference rule

- Thus We have

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

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

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

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

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

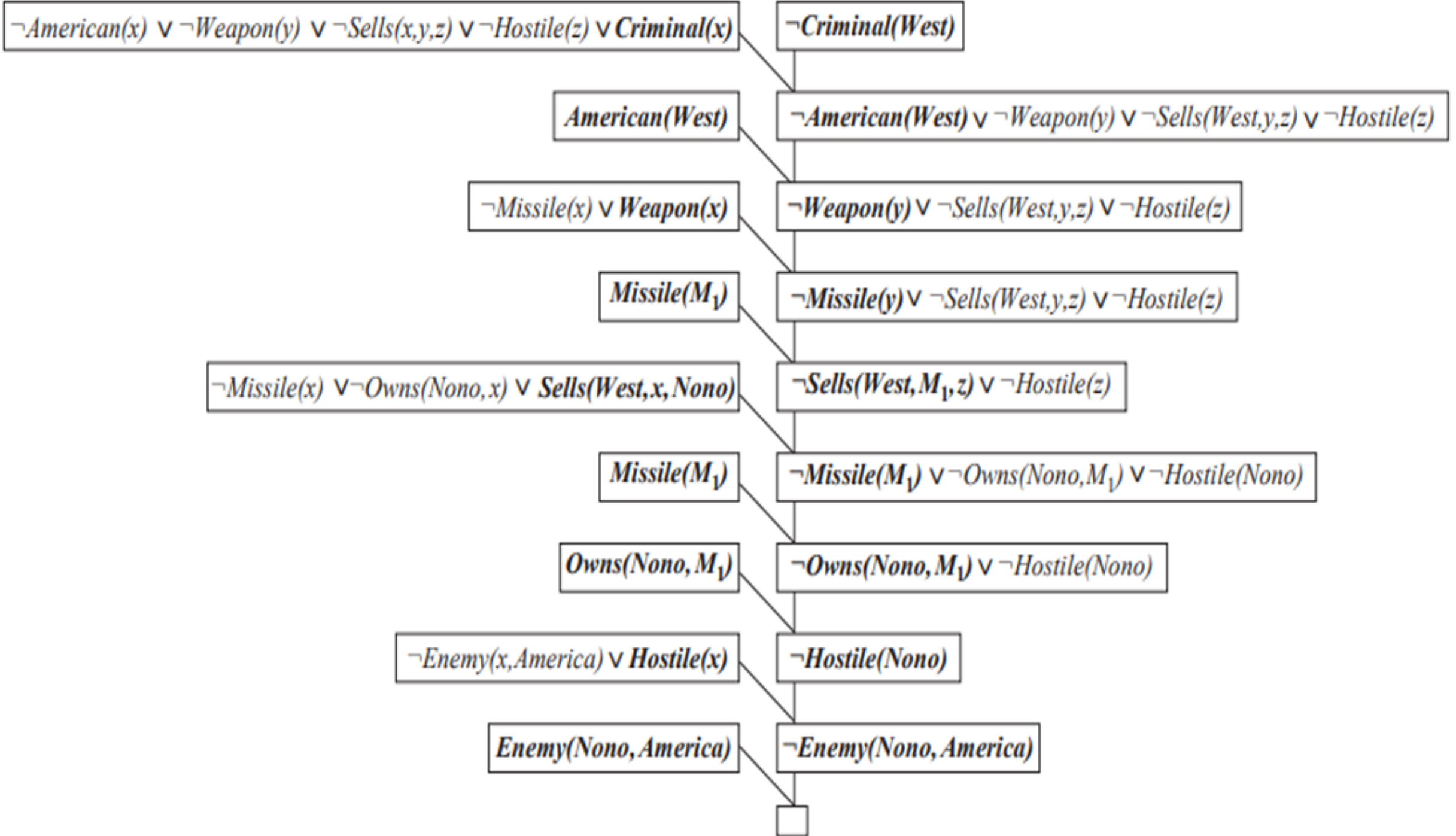


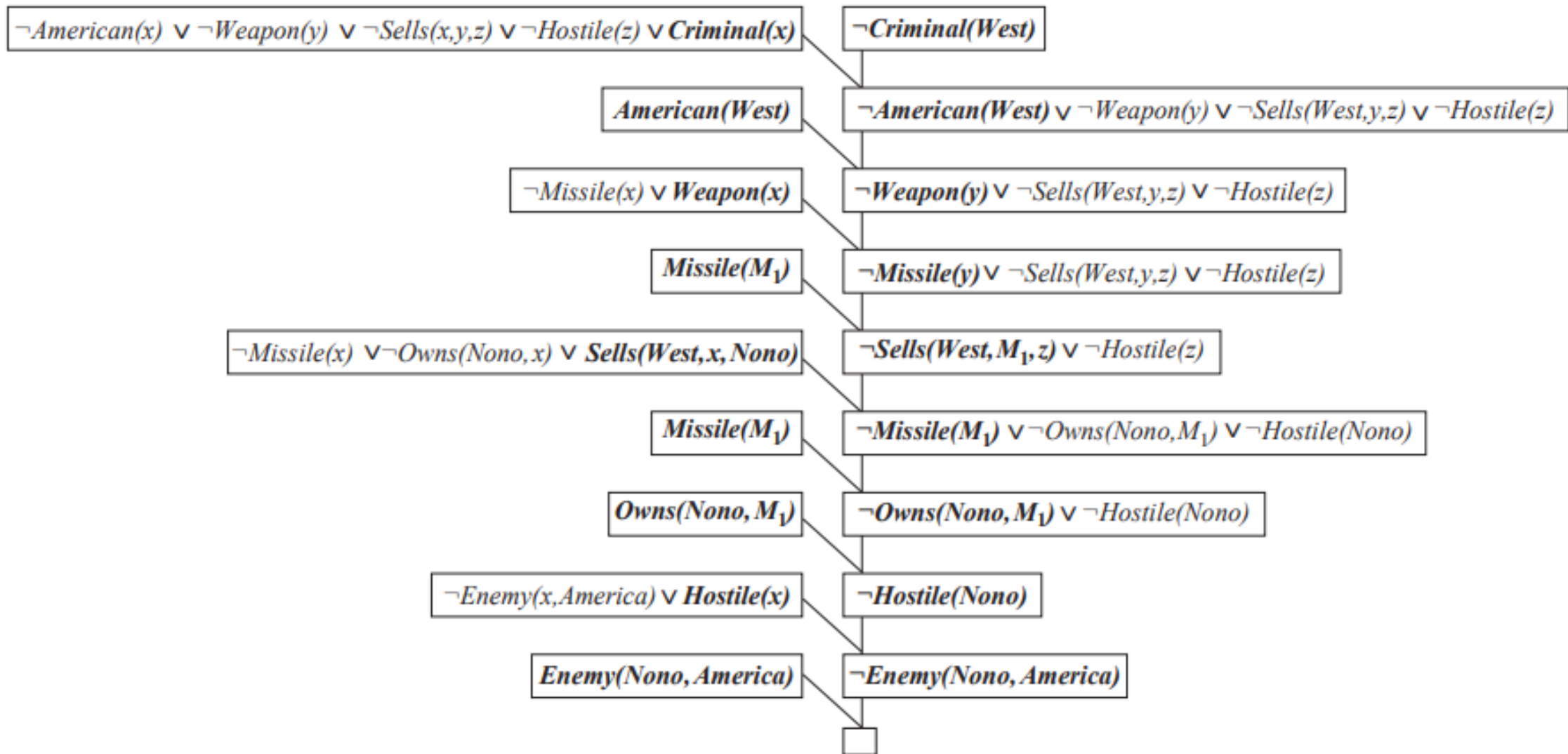
# Example1

- A Resolution proof that West is Criminal

# Knowledge Base

1.  $\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$  .
2.  $\text{Owns}(\text{Nono}, \text{M1})$
3.  $\text{Missile}(\text{M1})$
4.  $\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$  .
5.  $\text{Missile}(x) \Rightarrow \text{Weapon}(x)$
6.  $\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$  .
7.  $\text{American}(\text{West})$  .
8.  $\text{Enemy}(\text{Nono}, \text{America})$  .





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

Example2

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

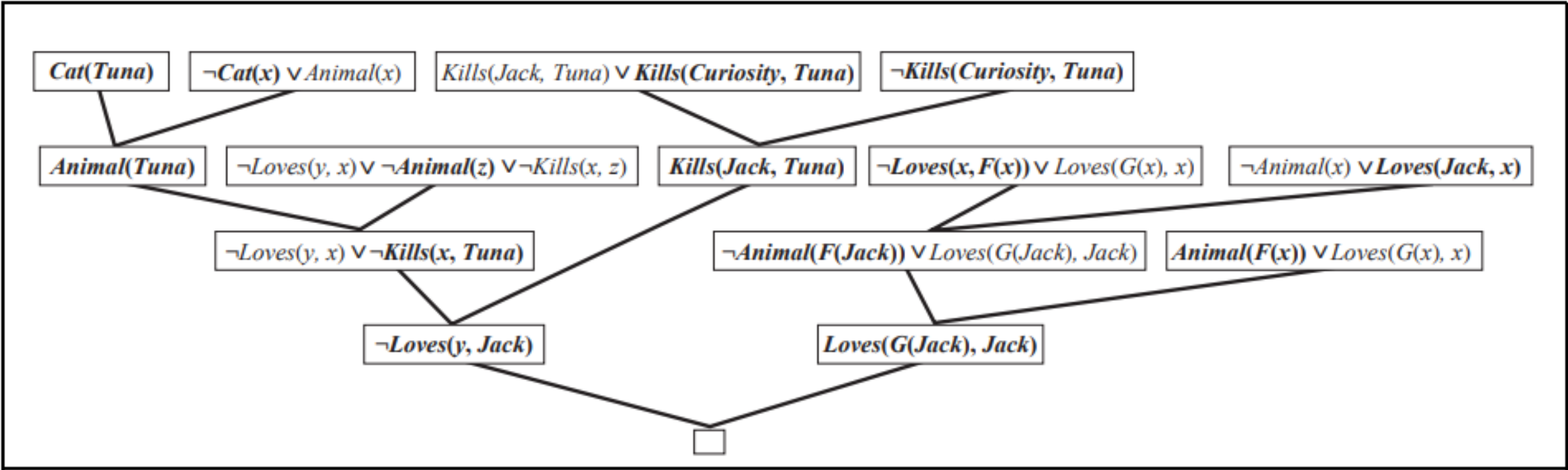
- A.  $\forall x [\forall y \textit{Animal}(y) \Rightarrow \textit{Loves}(x, y)] \Rightarrow [\exists y \textit{Loves}(y, x)]$
- B.  $\forall x [\exists z \textit{Animal}(z) \wedge \textit{Kills}(x, z)] \Rightarrow [\forall y \neg \textit{Loves}(y, x)]$
- C.  $\forall x \textit{Animal}(x) \Rightarrow \textit{Loves}(\textit{Jack}, x)$
- D.  $\textit{Kills}(\textit{Jack}, \textit{Tuna}) \vee \textit{Kills}(\textit{Curiosity}, \textit{Tuna})$
- E.  $\textit{Cat}(\textit{Tuna})$
- F.  $\forall x \textit{Cat}(x) \Rightarrow \textit{Animal}(x)$
- ¬G.  $\neg \textit{Kills}(\textit{Curiosity}, \textit{Tuna})$



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



# 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**

# Note

- Resolution is a method in logic that can prove whether a set of statements is **unsatisfiable**.
- **Unsatisfiable set of statements:** Means the statements can't all be true together.

# Steps in Proving Completeness

- 1. Transforming to Clausal Form:**
- 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**
- 4. Lifting to First-Order Logic**

Any set of sentences  $S$  is representable in clausal form



Assume  $S$  is unsatisfiable, and in clausal form



Herbrand's theorem



Some set  $S'$  of ground instances is unsatisfiable



Ground resolution theorem



Resolution can find a contradiction in  $S'$



Lifting lemma



There is a resolution proof for the contradiction in  $S'$

# The lifting lemma

- There exists a clause **C** such that:
  - **C** is a resolvent of  $C_1$  and  $C_2$  (it works at the variable level).
  - **C'** is a ground instance of **C**.
- **$C_1$  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  $C'_2$ .

# Handling Equality in Inference system

- 1. Axiomatizing Equality**
- 2. Demodulation: Adding Inference Rules**
- 3. Paramodulation**

# 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).

# Demodulation: Adding Inference Rules

- If  $x=y$  (a unit clause) and a **clause  $\alpha$  contains  $x$** , we replace  **$x$  with  $y$**  in  **$\alpha$** .
- Demodulation simplifies expressions in one direction (e.g.,  **$x+0=x$  allows  $x+0$  to simplify to  $x$** ).
- **Example:**  
Given:
  - **$\text{Father}(\text{Father}(x))=\text{PaternalGrandfather}(x)$**
  - **$\text{Birthdate}(\text{Father}(\text{Father}(\text{Bella})), 1926)$**
  - We use demodulation to derive:
    - **$\text{Birthdate}(\text{PaternalGrandfather}(\text{Bella}), 1926)$**



# Demodulation: Adding Inference Rules

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

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

where  $\text{SUBST}$  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

- A more general rule, **paramodulation**, extends demodulation to handle cases where **equalities are part of more complex clauses**.
- **Formal Rule:**
- For any terms **x, y, and z**:
  - If **z** appears in a clause **m** and **x** unifies with **z**,
  - Replace **x with y in m**, while preserving other parts of the clause.

# Paramodulation

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

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

# Applications of Resolution Theorem Provers

- **Hardware Design and Verification**
- **Software Verification and Synthesis**

# 5.3 Classical Planning:

- 1. The Definition of Classical Planning**
  1. Representing States in Classical Planning
  2. Defining Actions with Schemas
  3. Planning Domains and Problems
- 2. Algorithms for Planning as State Space Search,**
  - 1. Forward (Progression) State-Space Search**
  - 2. Backward (Regression) Relevant-States Search**
  - 3. Heuristics for Planning**
- 3. Planning Graphs.**
  1. Definition
  2. Construction of Planning graphs
  3. Planning Graphs for Heuristic Estimation
  4. The GRAPHPLAN Algorithm

## 5.3.1 What is 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.

## 5.3.1.1 Representing States in Classical Planning

- States are represented as **conjunctions of fluents—ground, functionless atomic facts.**

**For example:**

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

# The representation follows

- **Closed-world assumption:** Any fluent not explicitly mentioned is considered false.
- **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)**.

## 5.3.1.2 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.

# Example

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))
```

# Planning Domains and Problems

- A planning domain is defined by a **set of action schemas**. A specific problem within the domain includes:
  - **Initial state**: A conjunction of ground fluents.
  - **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  **$\text{Plane}(\text{Plane1}) \wedge \text{At}(\text{Plane1}, \text{SFO})$**  satisfies the goal  **$\text{At}(p, \text{SFO}) \wedge \text{Plane}(p)$** .

# Limitations in Early Approaches

- **Atomic State Representations**
- **Ground Propositional Inference**

# Overcoming Limitations with Factored Representations

- **Structured State Representation**
- **Use of PDDL (Planning Domain Definition Language):**
- **Improved Computational Efficiency**

## Example 1: Air cargo transport

*Init*( $At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)$   
 $\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$   
 $\wedge Airport(JFK) \wedge Airport(SFO)$ )  
*Goal*( $At(C_1, JFK) \wedge At(C_2, SFO)$ )  
*Action*(*Load*( $c, p, a$ ),  
  PRECOND:  $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$   
  EFFECT:  $\neg At(c, a) \wedge In(c, p)$ )  
*Action*(*Unload*( $c, p, a$ ),  
  PRECOND:  $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$   
  EFFECT:  $At(c, a) \wedge \neg In(c, p)$ )  
*Action*(*Fly*( $p, from, to$ ),  
  PRECOND:  $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$   
  EFFECT:  $\neg At(p, from) \wedge At(p, to)$ )

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

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  $a$ .



**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.

## Example2 : The Spare Tire Problem

In this scenario, there are only four actions available:

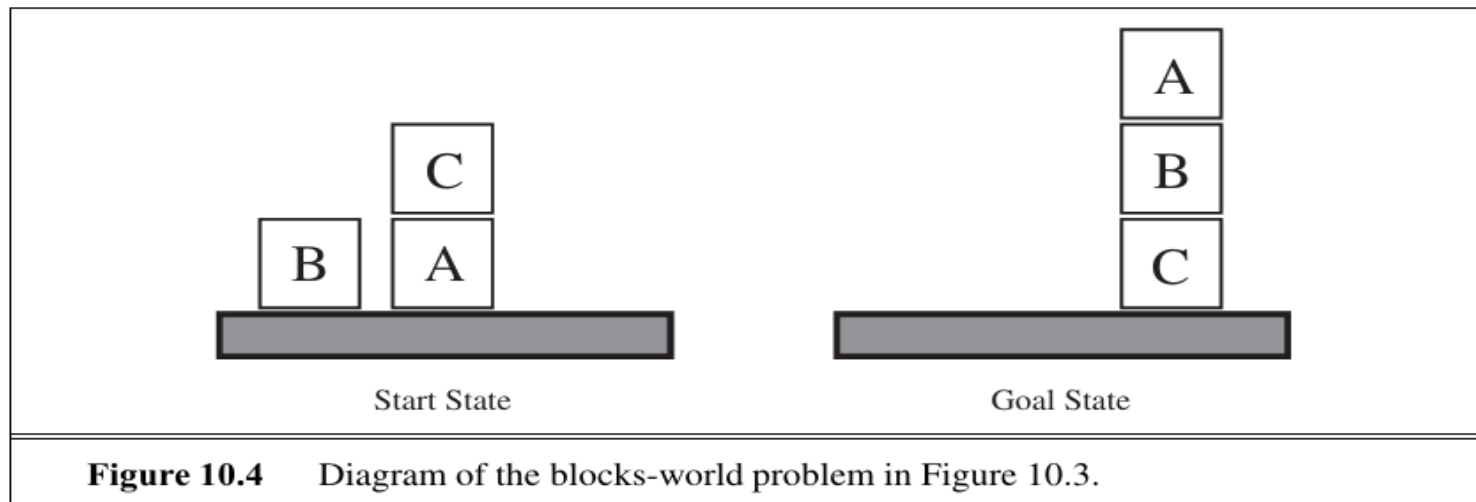
- **Removing the spare tire from the trunk.**
- **Removing the flat tire from the axle.**
- **Mounting the spare tire onto the axle.**
- **Leaving the car unattended overnight.**

$Init(Tire(Flat) \wedge Tire(Spare) \wedge At(Flat, Axle) \wedge At(Spare, Trunk))$   
 $Goal(At(Spare, Axle))$   
 $Action(Remove(obj, loc),$   
     PRECOND:  $At(obj, loc)$   
     EFFECT:  $\neg At(obj, loc) \wedge At(obj, Ground))$   
 $Action(PutOn(t, Axle),$   
     PRECOND:  $Tire(t) \wedge At(t, Ground) \wedge \neg At(Flat, Axle)$   
     EFFECT:  $\neg At(t, Ground) \wedge At(t, Axle))$   
 $Action(LeaveOvernight,$   
     PRECOND:  
     EFFECT:  $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$   
              $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk))$

**Figure 10.2** The simple spare tire problem.

# Example 3: 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: Setup, Goal and Predicates

- **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.

- **Goal:**

The goal is defined by a specific arrangement of blocks, e.g., block A on B and block B on C.

- **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.

# Key Concepts: Actions

## Actions:

- **Move( $b, x, y$ ):** Moves block  $b$  from  $x$  to  $y$  (either another block or the table).
  - **Preconditions:**
    - **$\text{On}(b, x) \wedge \text{Clear}(b) \wedge \text{Clear}(y)$**
    - (Block  $b$  is on  $x$ , block  $b$  is clear, and the destination  $y$  is clear.)
  - **Effects:**
    - **$\text{On}(b, y) \wedge \text{Clear}(x) \wedge \neg\text{On}(b, x) \wedge \neg\text{Clear}(y)$**
    - (Block  $b$  is on  $y$ ,  $x$  becomes clear,  $b$  is no longer on  $x$ , and  $y$  is no longer clear.)

# Key Concepts: MoveToTable(b, x):

- **Preconditions:**

- **On(b, x)  $\wedge$  Clear(b)**
- (Block b is on x and is clear.)

- **Effects:**

- **On(b, Table)  $\wedge$  Clear(x)  $\wedge$   $\neg$ On(b, x)**
- (Block b is now on the table, x is clear, and b is no longer on x.)

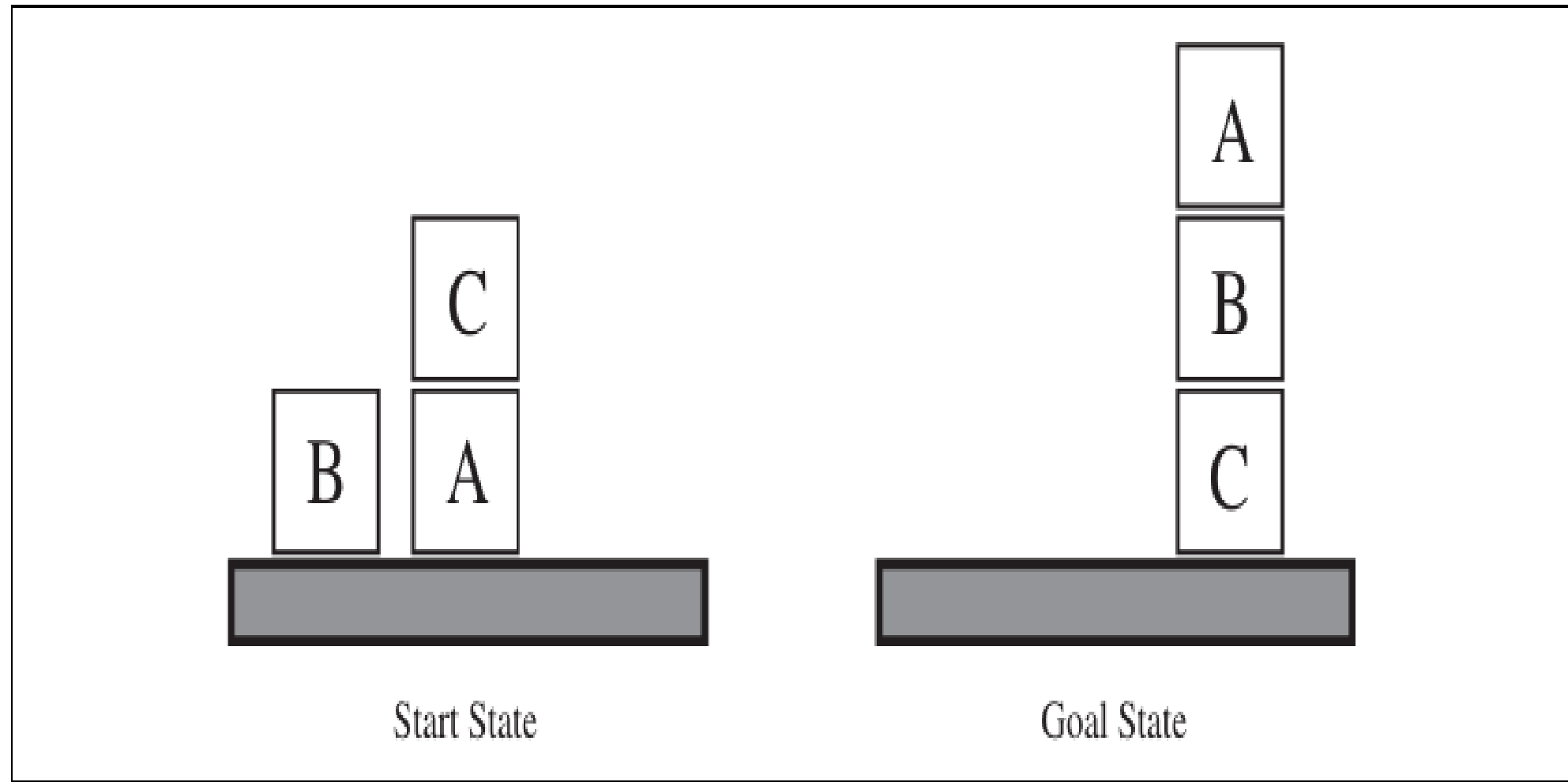
- **Reinterpret Clear(x):**

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

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A)$   
 $\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C))$   
 $Goal(On(A, B) \wedge On(B, C))$   
 $Action(Move(b, x, y),$   
 $PRECOND: On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$   
 $(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$   
 $EFFECT: On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$   
 $Action(MoveToTable(b, x),$   
 $PRECOND: On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),$   
 $EFFECT: On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$

**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.

# The Complexity of Classical Planning

- 1. Key Decision Problems**
- 2. Decidability**
- 3. Complexity Classes**
- 4. Practical Implications**
- 5. Role of Heuristics**

# PlanSAT

- The full form of **PlanSAT** is **Plan Satisfiability**.
- It refers to the decision problem of determining whether a valid plan exists that satisfies the constraints of a given planning problem

# PSPACE

- The full form of **PSPACE** is **Polynomial Space**.
- It represents a complexity class in computational complexity theory.
- Specifically, **PSPACE** includes all decision problems that can be solved using a polynomial amount of memory (space) on a deterministic **Turing machine**, regardless of the time it might take.

# PlanSAT Bounded PlanSAT

- **PlanSAT:** Determines whether there exists any plan that solves a given planning problem.
- **Bounded PlanSAT:** Asks if there is a solution of length  $k$  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.



# Difference Between P and NP Problems

Feature	P Problems	NP Problems
Definition	Problems that can be solved in polynomial time.	Problems whose solutions can be <i>verified</i> in polynomial time.
Key Property	Easy to solve and verify.	Easy to verify, but not necessarily easy to solve.
Relationship	P is a subset of NP.	NP includes all P problems, but may also include harder problems.
Examples	<ul style="list-style-type: none"><li>- Sorting a list (e.g., Merge Sort).</li></ul>	<ul style="list-style-type: none"><li>- Sudoku puzzle: Verifying a completed solution.</li></ul>
	<ul style="list-style-type: none"><li>- Finding the shortest path (Dijkstra's algorithm). </li></ul>	<ul style="list-style-type: none"><li>- Traveling Salesman Problem (TSP): Verifying if a given route meets conditions.</li></ul>

## 5.2.2 Algorithms for Planning as State-Space Search

Two approaches to searching for a plan.

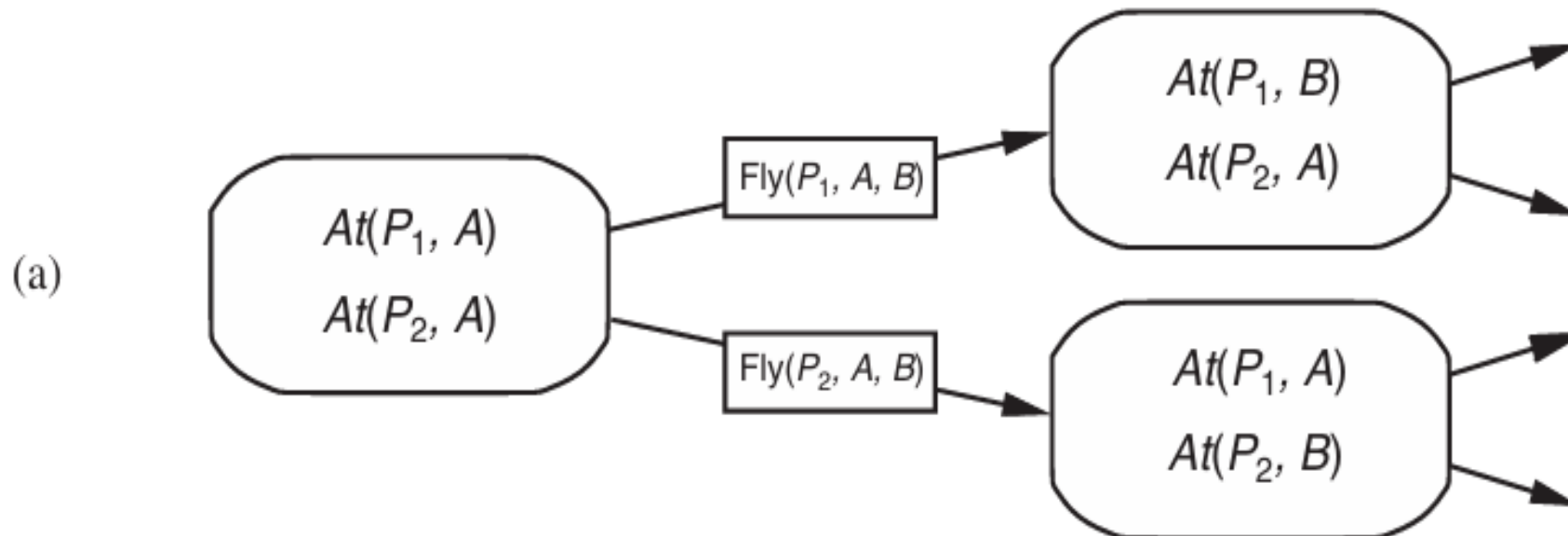
- (a) **Forward (progression).**
- (b) **Backward (regression)**

# 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:**
  - Explores irrelevant actions.
  - 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.**

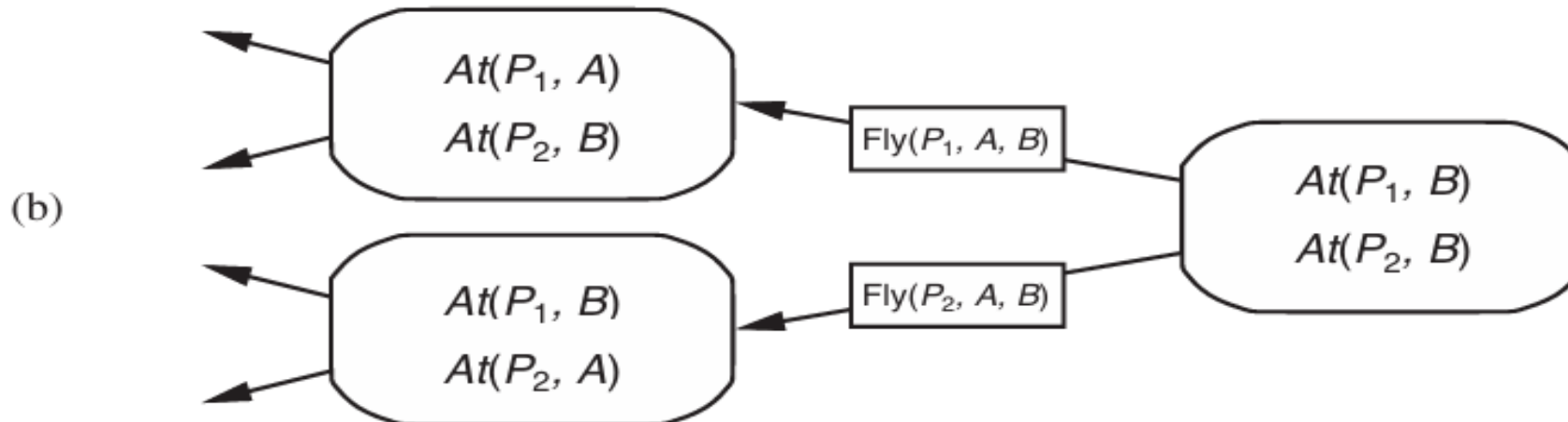


# Backward (Regression) Relevant-States Search

- **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.

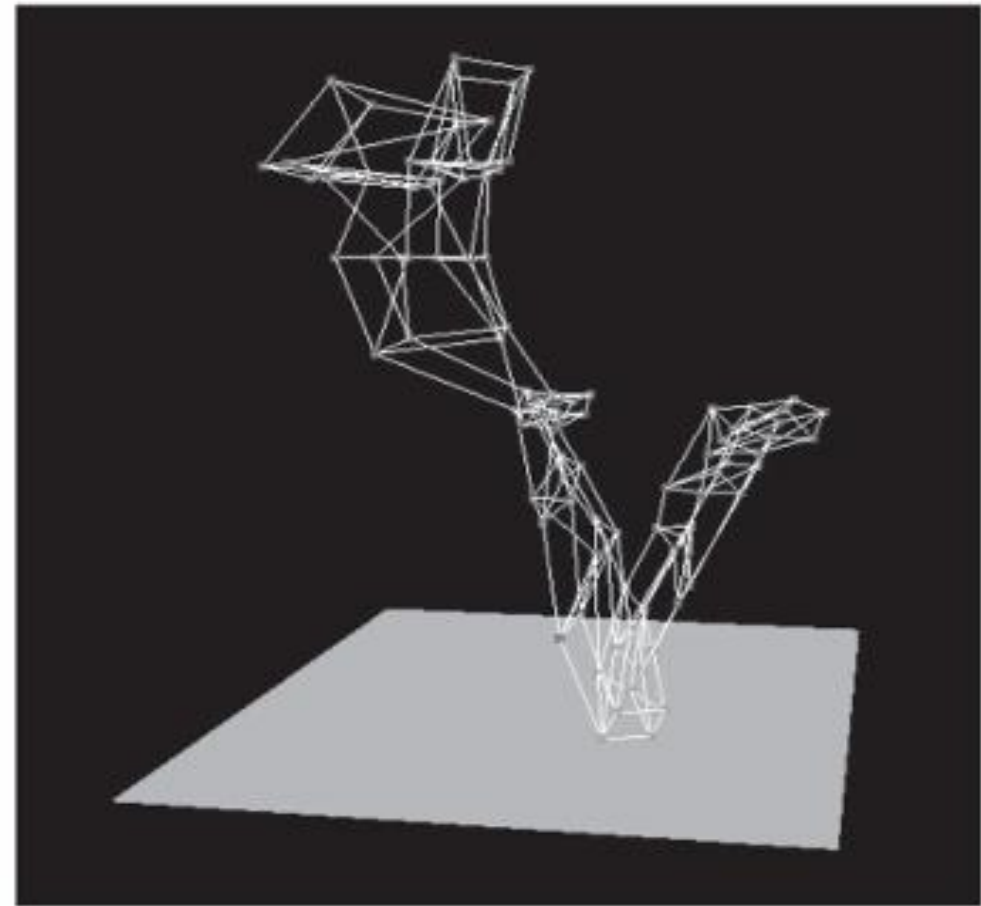
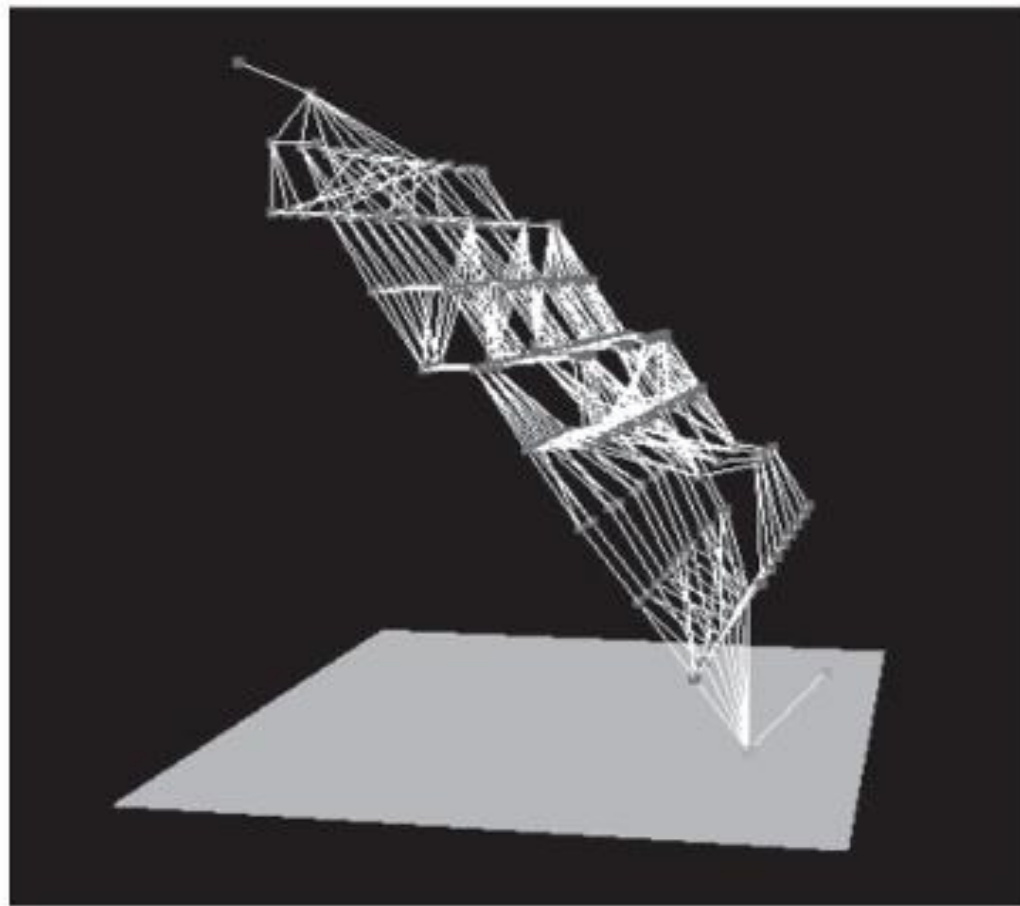
# Example

- If the goal is **At(C2, SFO)**, the algorithm considers the action **Unload(C2, p, SFO)**:
  - **Precondition:**  $\text{In}(\text{C2}, p) \wedge \text{At}(p, \text{SFO})$ .
  - **Effect:**  $\text{At}(\text{C2}, \text{SFO})$ .
- It regresses to find the **predecessor state where these preconditions** are true.



# Heuristics for Planning

- **Purpose:** Estimate the **cost of reaching the goal** from the current state to guide search algorithms like  $A^*$ .
- **Types of Heuristics:**
  - **Ignore Preconditions:**
    - **Drops preconditions, making every action applicable.**
    - **Example:** Simplifies the 8-puzzle by ignoring adjacency requirements for moves.
  - **Ignore Delete Lists:**
    - **Assumes actions cannot undo progress, making the problem monotonic.**
    - **Example:** In a transportation problem, **unloading an item is never undone.**
  - **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** 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).



## 5.2.3 Planning Graphs

- 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:**
  - **Levels:**
    - $S_0$ : Represents the initial state.
    - $A_0$ : Represents actions applicable in  $S_0$ .
    - Alternates between **states** ( $S_1, S_2, \dots$ ) and **actions** ( $A_1, A_2, \dots$ ).
  - **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)*

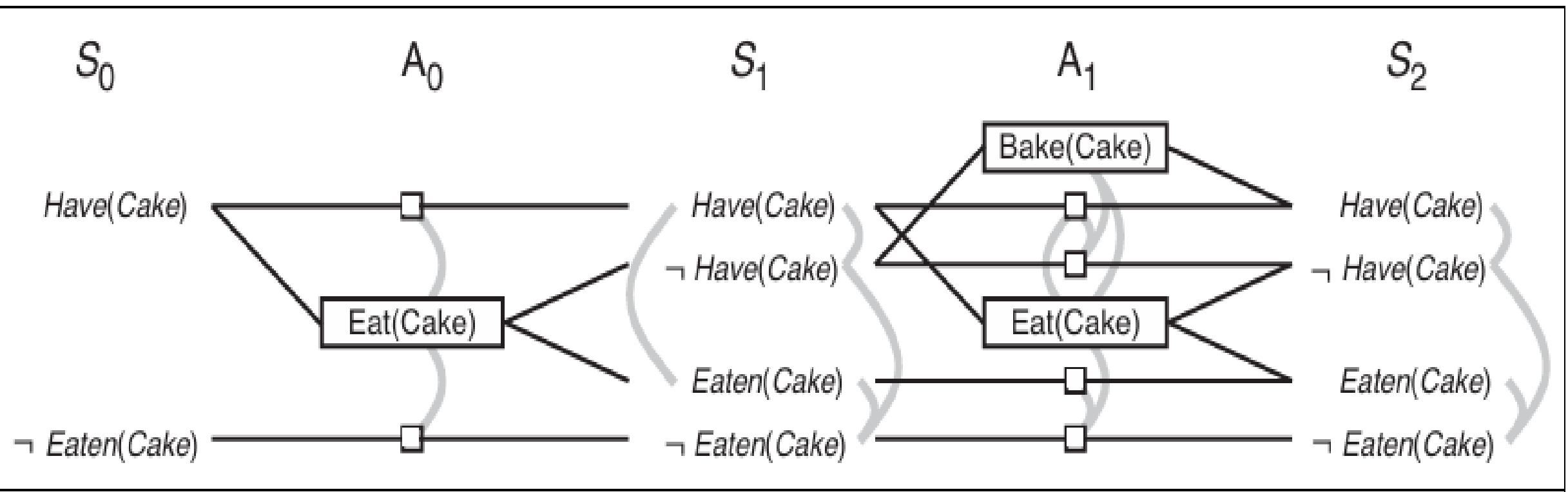
EFFECT:  $\neg$  *Have(Cake)  $\wedge$  Eaten(Cake)*

*Action(Bake(Cake))*

PRECOND:  $\neg$  *Have(Cake)*

EFFECT: *Have(Cake)*

**Figure 10.7** The “have cake and eat cake too” problem.



**Figure 10.8** 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

1. A **planning graph** provides valuable insights into a problem once constructed
2. **Unsolvability Check**
3. **Estimating Goal Costs:**
4. **Accuracy and Serial Graphs:**
5. **Estimating Conjunction Costs:**
  1. **Max-Level Heuristic**
  2. **Level-Sum Heuristic**
  3. **Set-Level Heuristic**
6. **Planning Graphs as Relaxed Problems**

## 5.2.3.2 The GRAPHPLAN Algorithm

- The **GRAPHPLAN algorithm** 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.

**function** GRAPHPLAN(*problem*) **returns** solution or failure

*graph*  $\leftarrow$  INITIAL-PLANNING-GRAPH(*problem*)

*goals*  $\leftarrow$  CONJUNCTS(*problem*.GOAL)

*nogoods*  $\leftarrow$  an empty hash table

**for** *tl* = 0 **to**  $\infty$  **do**

**if** *goals* all non-mutex in  $S_t$  of *graph* **then**

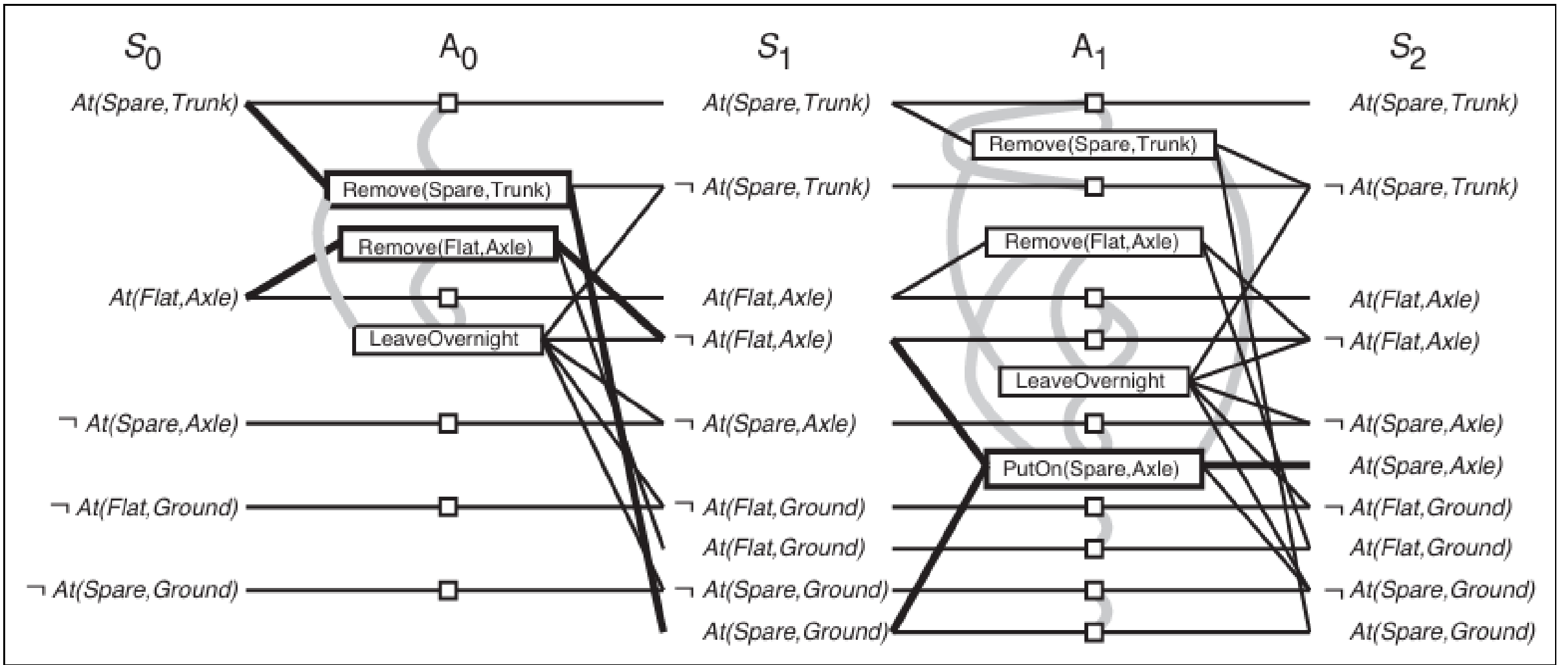
*solution*  $\leftarrow$  EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)

**if** *solution*  $\neq$  failure **then return** *solution*

**if** *graph* and *nogoods* have both leveled off **then return** failure

*graph*  $\leftarrow$  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.



**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.



Note: In the context of planning graphs used in classical planning let's define literals, mutexes, and no-goods:

- **Literals** : A literal is a **propositional variable** (fact) that can either be **true or false**. In planning graphs, literals represent the **conditions or states that are either currently true or can potentially be achieved at a particular level of the graph**.
- **Mutexs** : **Mutexes** (short for **mutual exclusions**) indicate pairs of actions or literals that **cannot coexist** at a particular level of the planning graph due to conflicts.
- **No Goods**: A **no-good** is a set of literals (or conditions) that are **known to be unsatisfiable**. No-goods represent combinations of literals or partial plans that **cannot lead to a solution**.

## 5.2.3.3 Termination of GRAPHPLAN

- GRAPHPLAN ensures termination and returns failure when no solution exists.
- **When to Terminate?**
- The algorithm continues expanding the graph as long as new possibilities arise:
- **No-Goods:** If EXTRACT-SOLUTION fails, it indicates that some goals are unachievable and are marked as no-goods.
- **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.

# Finite Nature in Planning Graphs:

- **Monotonic Increase:**
- **Literals:** Once a literal is introduced at a level, it persists in all subsequent levels.
- **Actions:** Similarly, actions that are applicable at a level will continue to be applicable at later levels.
- Since the set of all possible literals and actions is finite, the graph eventually stabilizes.

# Finite Nature in Planning Graphs:

- **Monotonic Decrease:**
- **Mutexes:** Mutexes decrease as more actions and literals become reachable and interact.
- **No-Goods:** No-goods also decrease as more solutions are found and fewer unsatisfiable combinations remain.

# Termination Condition

- When the graph stabilizes and either:
  - **A goal is missing, or**
  - **Any goal is mutex with another,**
- the algorithm terminates, returning failure. This guarantees that further expansion would not yield a solution.

## 5.2.4 Logic Programming

- **Logic programming** is a method of building systems by writing **rules and facts** in a formal language.
- 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.
- **Variables and Constants:** Variables are uppercase (e.g., X), and constants are lowercase (e.g., john).
- **Clause Structure:** Instead of  $A \wedge B \Rightarrow C$ , 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."**
- **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:**

- Appending an empty list to **Y** gives **Y**.
- 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!

- **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.

## 5.2.4.1 Efficient Implementation of Logic Programs

Prolog programs can be executed in **two modes**(Each mode has distinct characteristics and optimizations):

1. **Interpreted** and
2. **Compiled.**

# 1. Interpreted Mode

- In this mode, Prolog functions like the **FOL-BC-ASK algorithm**, **treating the program as a knowledge base.**
- Prolog interpreters include optimizations for better efficiency. Two key improvements are:
  1. **Global Stack of Choice Points:** Prolog uses a **global stack of choice points** to track alternatives
  2. **Logic Variables and Trails:** Prolog uses **logic variables** that dynamically remember their current bindings.

## 2. Compiled Mode

1. Optimized Clause Matching
2. Open-Coded Unification
3. Intermediate Language Compilation
4. High-Level Language Translation

```
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.



# Noteworthy points highlight how this compilation enhances efficiency

- Transforming **Clauses into Procedures**
- Trail Management for **Variable Bindings**
- Use of Continuations for **Choice Points**

# Impact of Prolog Compilation

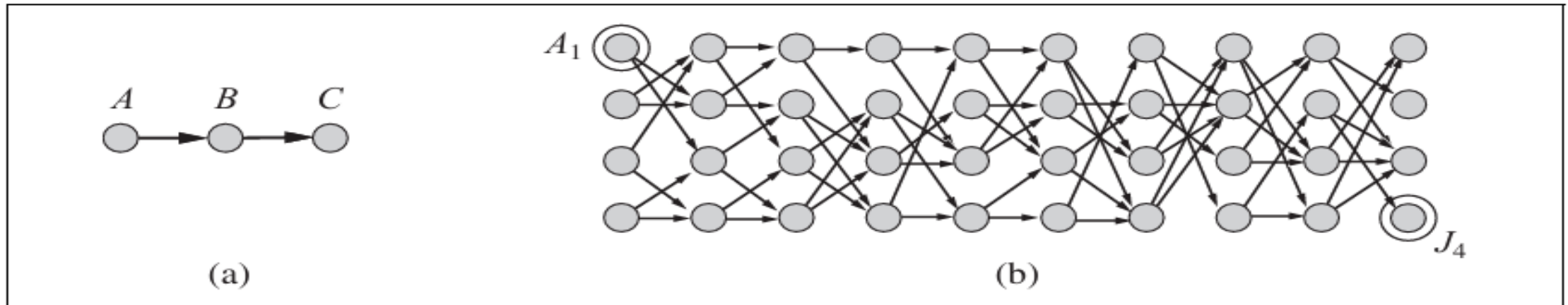
- Before the advancements by Warren and others, Prolog's performance was too slow for practical use. However, compilers like the **Warren Abstract Machine (WAM)** enabled Prolog to achieve execution speeds **competitive with C** on standard benchmarks (Van Roy, 1990).
- **Prolog's ability** to express 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.

## 5.2.4.2 Redundant Inference and Infinite Loops in Prolog

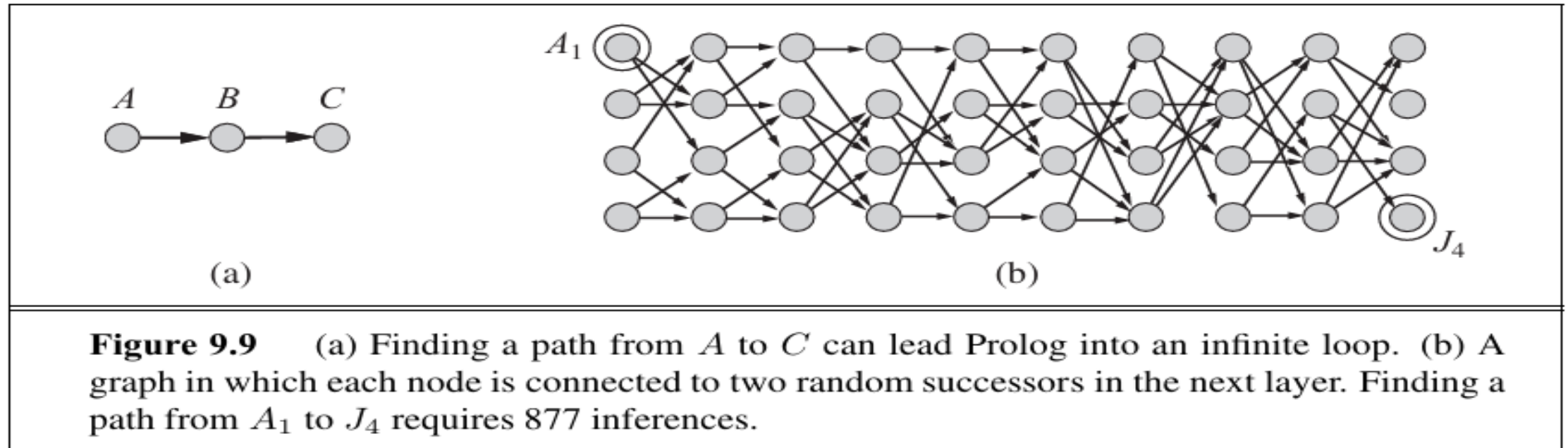
- **Infinite Loops in Depth-First Search**
- Consider the following Prolog program, which checks if a path exists between two points in a directed graph:
  - `path(X, Z) :- link(X, Z).`
  - `path(X, Z) :- path(X, Y), link(Y, Z).`



**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.

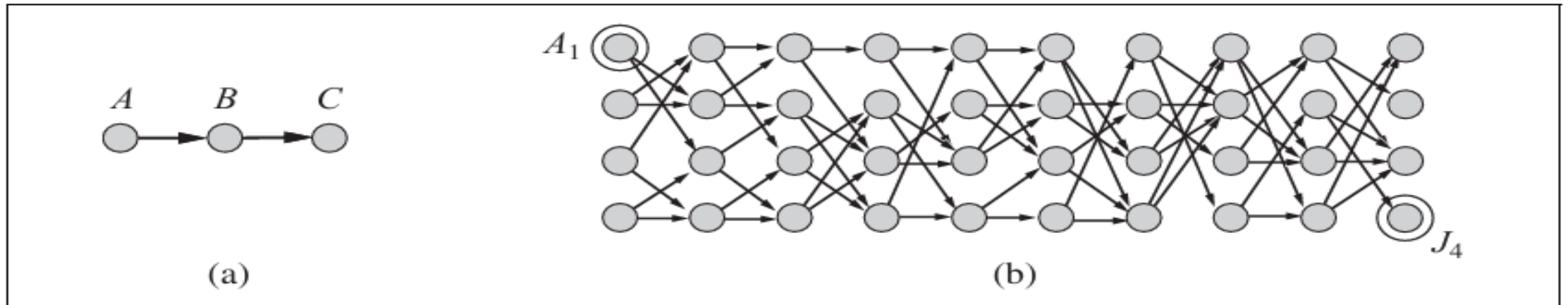
## 5.2.4.2 Redundant Inference and Infinite Loops in Prolog

- Forward chaining, in contrast, avoids this issue by systematically generating facts. Once  $\text{path}(a, b)$ ,  $\text{path}(b, c)$ , and  $\text{path}(a, c)$  are inferred, the process halts.



# Redundant Computations

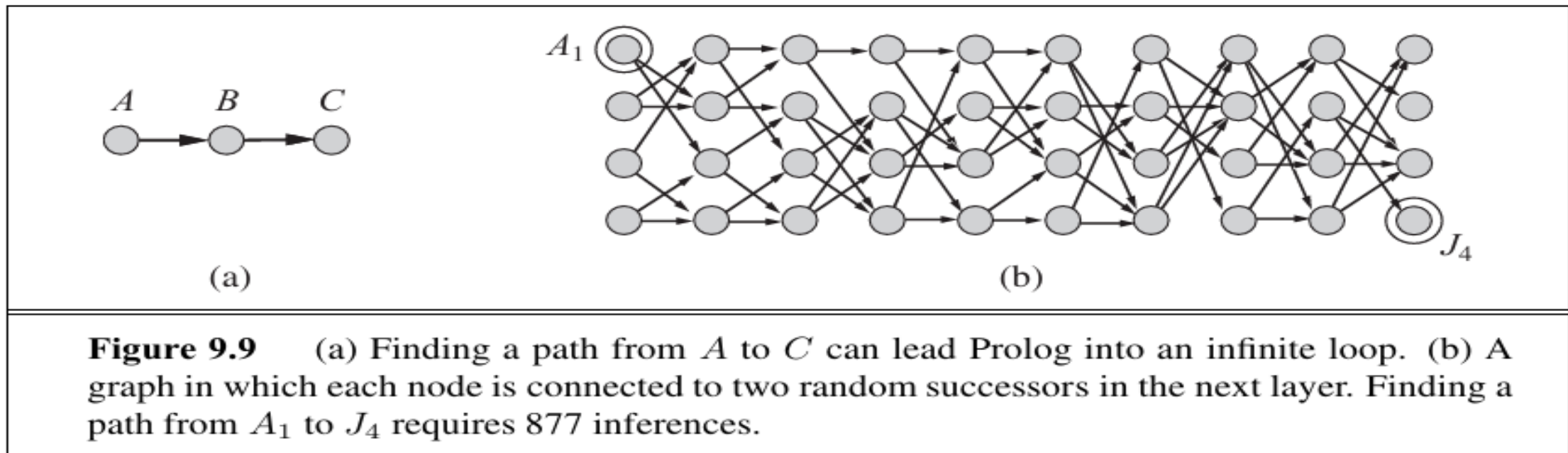
- Another major drawback of DFS in Prolog is redundant computations, particularly in graph problems.
- For example:
- When finding a path from  $A_1$  to  $J_4$  in a more complex graph (Figure 9.9(b)), Prolog performs 877 inferences, exploring multiple unnecessary paths, including those leading to unreachable nodes.
- This redundancy resembles the repeated-state problem

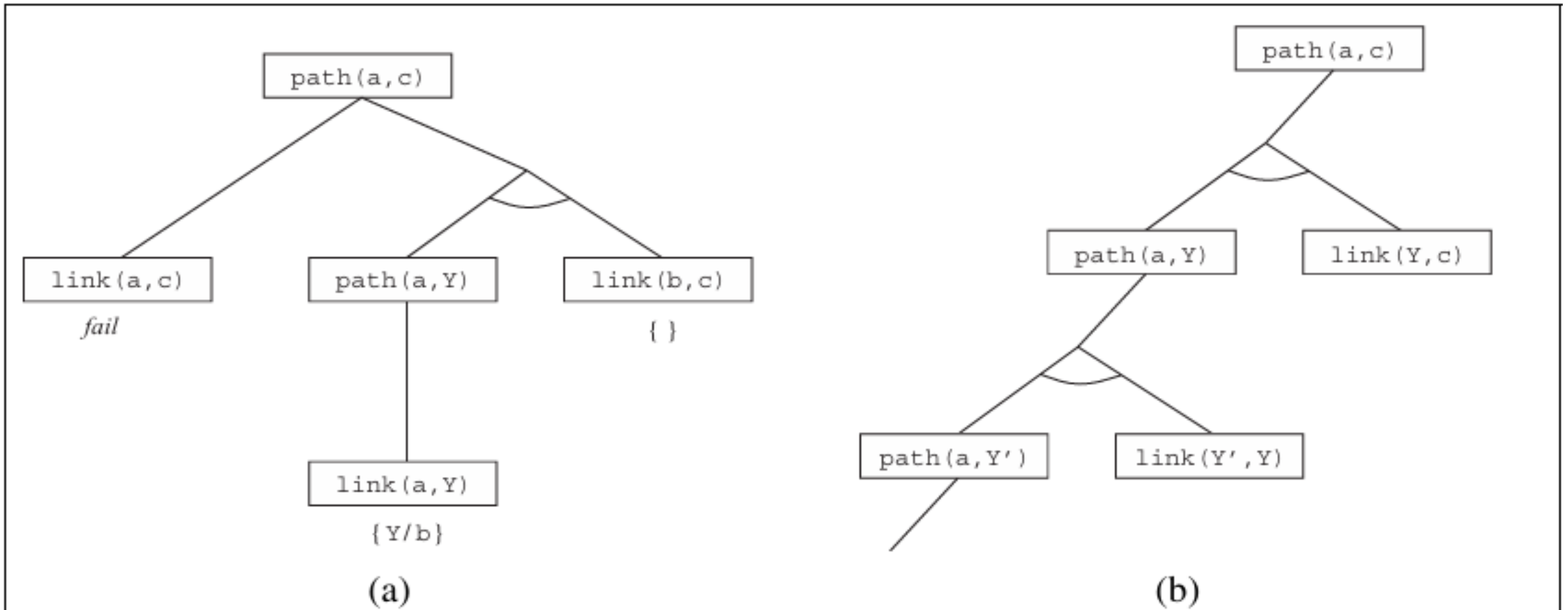


**Figure 9.9** (a) Finding a path from  $A$  to  $C$  can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from  $A_1$  to  $J_4$  requires 877 inferences.

# Redundant Computations

- Forward chaining, applied to the same problem, is far more efficient:
- It generates at most  $n^2$   $\text{path}(X, Y)$  facts for  $n$  nodes, avoiding repeated calculations.





**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.



# 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.

## 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

## 5.2.4.3 Database Semantics of Prolog

Example:

**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.

## 5.2.4.3 Database Semantics of Prolog

Example:

In FOL, the assertions would be expressed as:

$\text{Course}(d, n) \Leftrightarrow (d=\text{CS} \wedge n=101) \vee (d=\text{CS} \wedge n=102) \vee (d=\text{CS} \wedge n=106) \vee (d=\text{EE} \wedge n=101)$ .

To express that there are **at least four courses in FOL**, we would need to expand the equality predicate as follows:

$x = y \Leftrightarrow (x=\text{CS} \wedge y=\text{CS}) \vee (x=\text{EE} \wedge y=\text{EE}) \vee (x=101 \wedge y=101) \vee (x=102 \wedge y=102) \vee (x=106 \wedge y=106)$ .

## 5.2.4.4 Constraint Logic Programming

- Standard Prolog solves Constraint Satisfaction problems (CSP) using a backtracking algorithm, suitable only for **finite-domain CSPs**.
- 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  $\text{triangle}(X, Y, Z)$  is a predicate that holds if the three arguments satisfy the triangle inequality:
- **$\text{triangle}(X, Y, Z) :- X > 0, Y > 0, Z > 0, X + Y \geq Z, Y + Z \geq X, X + Z \geq Y.$**
- When we query  $\text{triangle}(3, 4, 5)$ , Prolog successfully returns true.
- However, when we query  $\text{triangle}(3, 4, Z)$ , **no solution is found because the subgoal  $Z \geq 0$**  cannot be handled by Prolog—Prolog cannot compare an unbound value to 0.
- The solution to the query  $\text{triangle}(3, 4, Z)$  would be the constraint  $7 \geq Z \geq 1$ , which means that  $Z$  must be between 1 and 7.

# Prolog Programs Examples

# 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
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.
```

End of Module 5