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**
   a. **Backward Chaining,**
   b. **Resolution**
2. **Classical Planning**
   a. **Definition of Classical Planning**
   b. **Algorithms for Planning as State Space Search**
   c. **Planning Graphs**

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.**

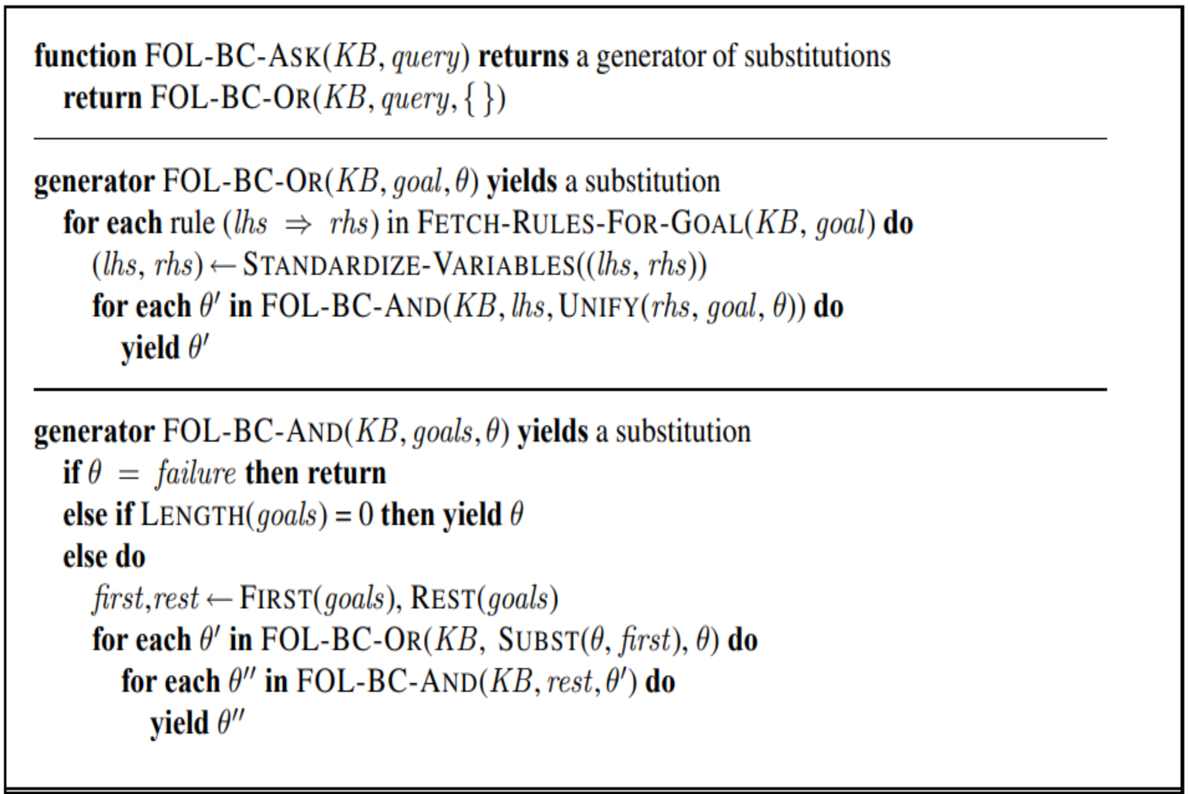
Backward chaining would proceed as follows:

- **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.

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.

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

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**.

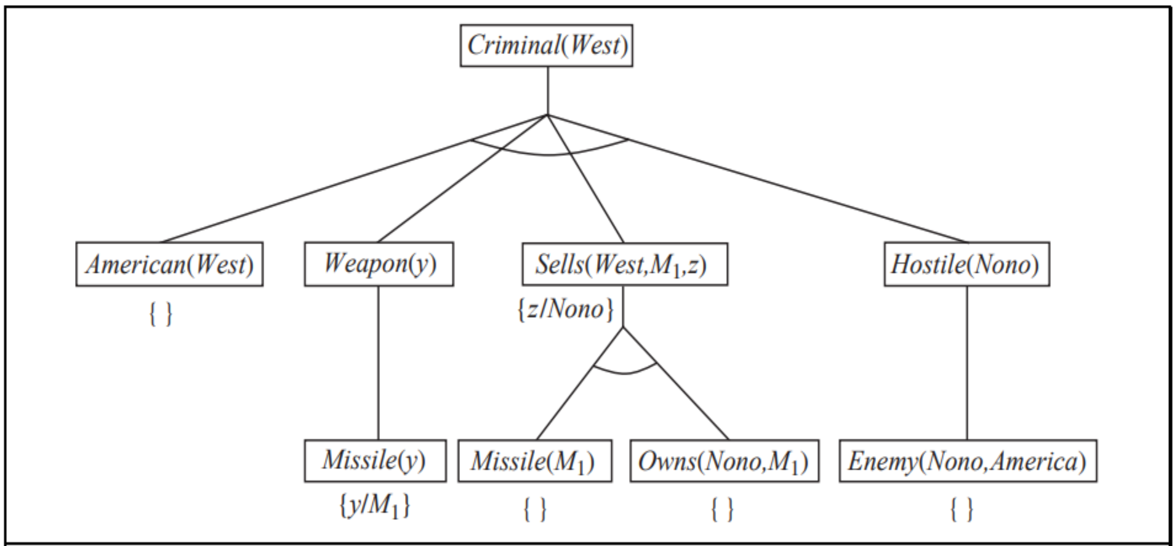


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. 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 $Nono$.

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

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. P∨Q
2. ¬P∨R

Applying resolution: Resolve the statements by eliminating P:

- P∨Q
- ¬P∨R

Resolving P and ¬P: Q∨R
The resulting statement **Q∨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.

Example2:

Clause 1: (P∨Q∨R)
Clause 2:(¬P∨¬Q∨S)
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: **(P∨Q∨R) and ((¬P∨¬Q∨S)**
Resolving P and ¬P gives**: (Q∨R)∨(¬Q∨S)**
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. $(A \lor B) \land (\neg A \lor C)$
   This expression has two clauses: $A \lor B$ and $\neg A \lor C$.
2. $(P \lor Q \lor R) \land (\neg P \lor \neg Q)$
   This expression also has two clauses: $P \lor Q \lor R$ and $\neg P \lor \neg Q$.
3. $(A \lor B \lor \neg C) \land (\neg A \lor \neg B \lor D) \land (C \lor D)$
   This expression has three clauses: $A \lor B \lor \neg C$, $\neg A \lor \neg B \lor D$, and $C \lor D$.
4. $(P \lor Q)$
   This is already in CNF, with one clause: $P \lor Q$.
5. $(A \land B) \lor (\neg C \land D)$
   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 $(A \lor \neg C) \land (A \lor D) \land (B \lor \neg C) \land (B \lor D)$.

Steps to Convert a Formula to CNF:

1. **Eliminate Biconditionals (⇔):** Replace each biconditional (↔) with an equivalent expression in terms of conjunction (∧), disjunction (V), 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 V B

3. **Move Negations Inward** (¬): Apply De Morgan's laws and distribute negations inward to literals.
   - ¬(¬A) ≡ A
   - ¬(A V B) ≡(¬A ∧ ¬B)
   - ¬(A∧ B) ≡ (¬A V ¬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)V(R∧S∧T)** .
   Now, let's distribute disjunctions over conjunctions: **H=(PV(R∧S∧T))∧(QV(R∧S∧T))**

Proof By Resolution Process includes the following steps in general

1. **Initial Set of Clauses (Knowledge Base**)
2. **Negate the Conclusion**:
3. **Apply Resolution**
4. **Continue Resolving**
5. **Conclusion**
6. **Termination**

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]**

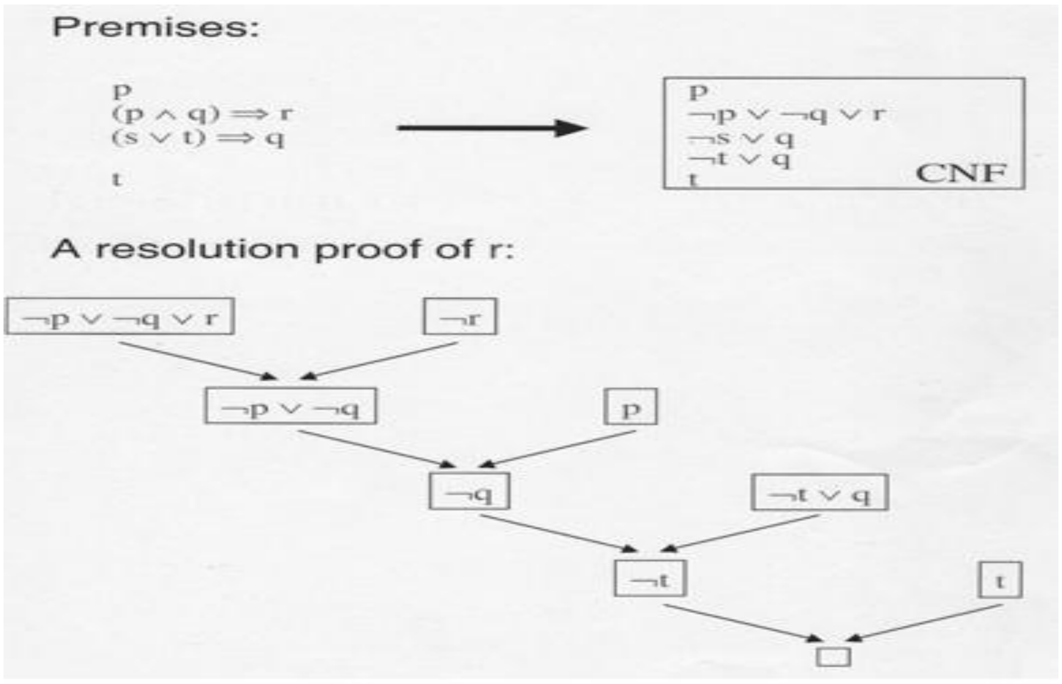
- **Apply Resolution:**

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] V P[1,2] , ¬P[1,2] resolves into ¬B[1,2]**
4. **B[1,2] V¬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 **False** and **P[1,2]** is true for the given knowledge base.

Example 2:



A grammar for conjunctive normal form

$$
\begin{array}{rcl}
\mathit{CNFSentence} & \rightarrow & \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} & \rightarrow & \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Literal} & \rightarrow & \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} & \rightarrow & \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} & \rightarrow & (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} & \rightarrow & (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{array}
$$

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

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

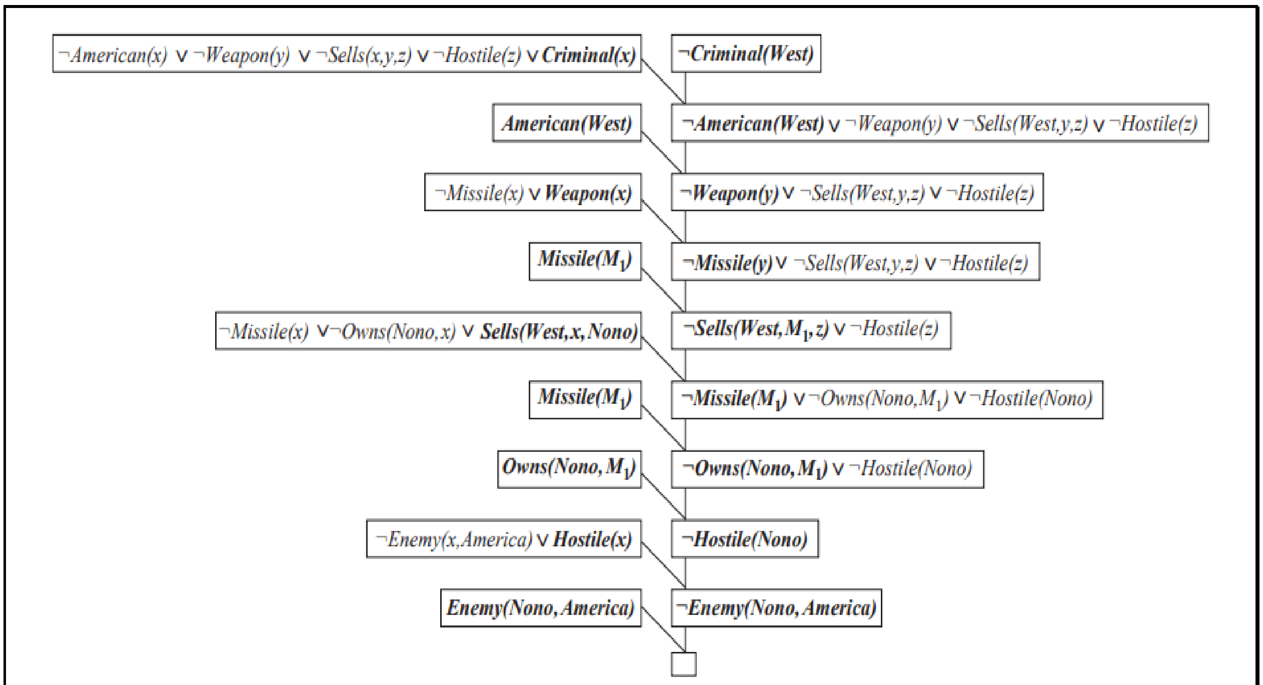

¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x,y,z) ∨ ¬Hostile(z) ∨ Criminal(x)
¬Criminal(West)
American(West)
¬American(West) ∨ ¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ Weapon(x)
¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ ¬Owns(Nono,x) ∨ Sells(West,x,Nono)
¬Sells(West,M1,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(M1) ∨ ¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
Owns(Nono,M1)
¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
¬Enemy(x,America) ∨ Hostile(x)
¬Hostile(Nono)
Enemy(Nono,America)
¬Enemy(Nono,America)

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

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

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

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

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. $Animal(F(x)) \lor Loves(G(x),x)$

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

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

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

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

E. $Cat(Tuna)$

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

¬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.

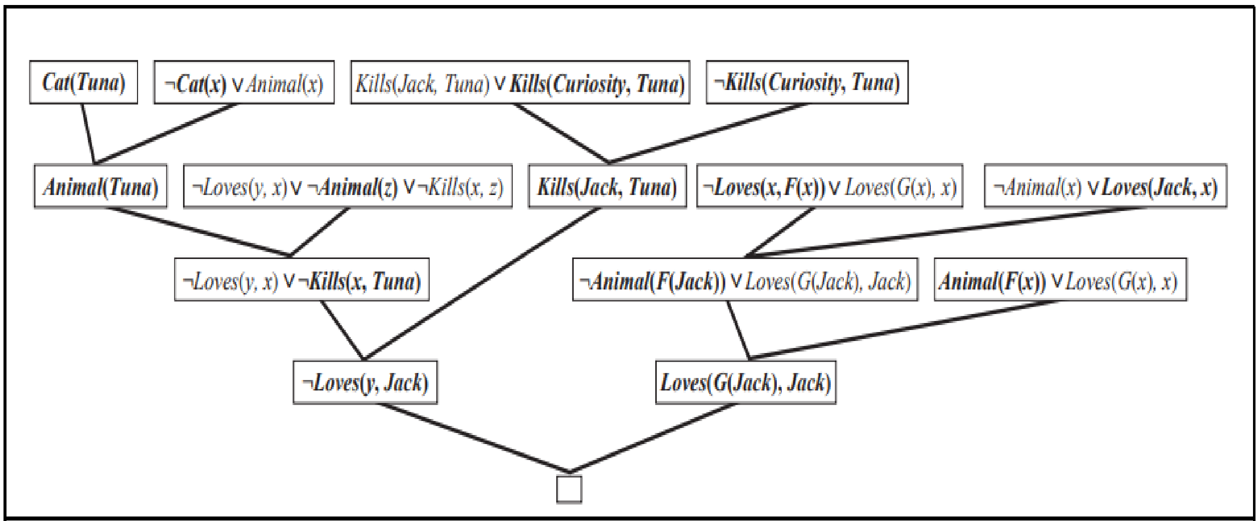

Cat(Tuna)
¬Cat(x) ∨ Animal(x)
Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
¬Kills(Curiosity, Tuna)
Animal(Tuna)
¬Loves(y, x) ∨ ¬Animal(z) ∨ ¬Kills(x, z)
Kills(Jack, Tuna)
¬Loves(x, F(x)) ∨ Loves(G(x), x)
¬Animal(x) ∨ Loves(Jack, x)
¬Loves(y, x) ∨ ¬Kills(x, Tuna)
¬Animal(F(Jack)) ∨ Loves(G(Jack), Jack)
Animal(F(x)) ∨ Loves(G(x), x)
¬Loves(y, Jack)
Loves(G(Jack), Jack)

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***.

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.

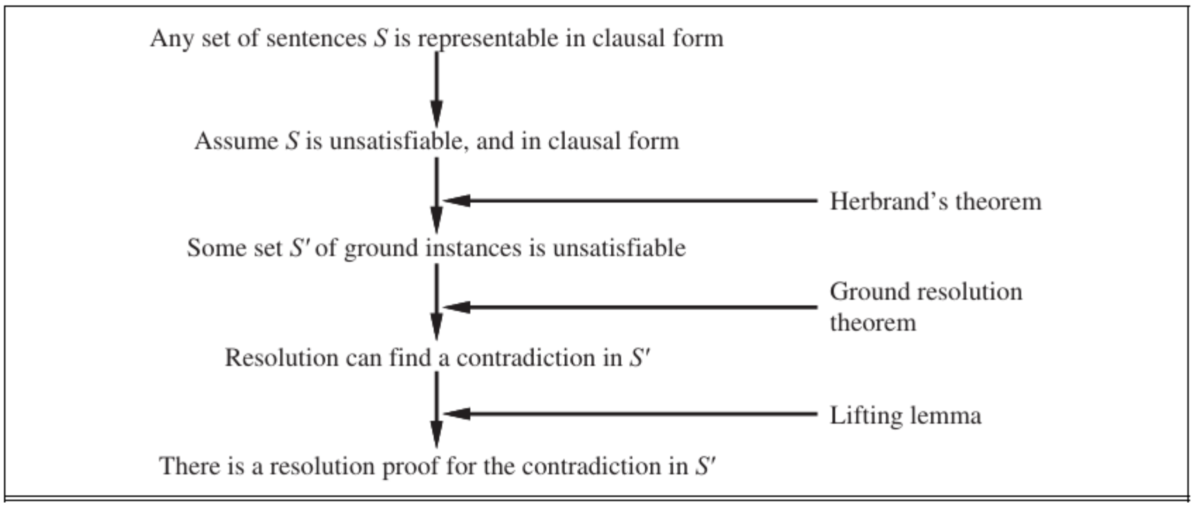
Key Idea

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:


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′

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**: $x=y \land y=z \Rightarrow x=z$

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)
- $w=y \land x=z \Rightarrow F(w,x)=F(y,z)$(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 $\textsc{Unify}(x, z) = \theta$,

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \qquad m_1 \vee \cdots \vee m_n}{\textsc{Sub}(\textsc{Subst}(\theta, x), \textsc{Subst}(\theta, y), \textsc{Subst}(\theta, \ell_1 \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_n)} .$$

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.

5.3 Classical Planning:

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

5.3.1 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.

5.3.1.1 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).

5.3.1.2 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.

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)**.

5.3.1.3 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)**.

5.3.1.4 Limitations in Early Approaches:

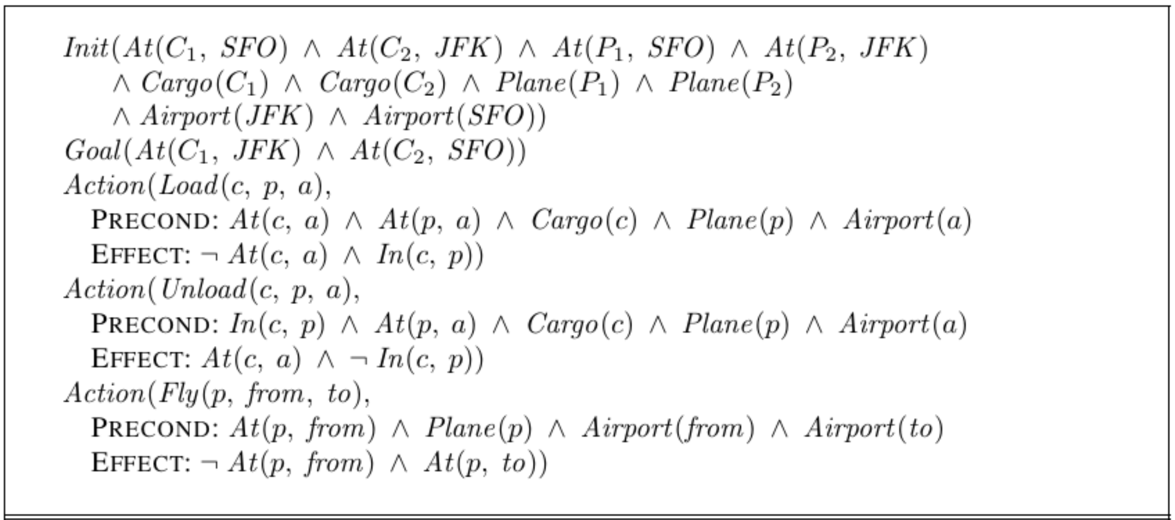
1. **Atomic State Representations (Chapter 3 Problem-Solving Agent):**
   - States are treated as indivisible entities, leading to a lack of structure in representations.
   - This approach relies heavily on **domain-specific heuristics** for effective problem-solving, limiting its generalizability and 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.

5.3.1.5 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.

5.3.1.6 Example : Air cargo transport



```
Init(At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK)
    ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)
    ∧ Airport(JFK) ∧ Airport(SFO))
Goal(At(C1, JFK) ∧ At(C2, SFO))
Action(Load(c, p, a),
  PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
  EFFECT: ¬ At(c, a) ∧ In(c, p))
Action(Unload(c, p, a),
  PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
  EFFECT: At(c, a) ∧ ¬ In(c, p))
Action(Fly(p, from, to),
  PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
  EFFECT: ¬ At(p, from) ∧ At(p, to))
```

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.

5.3.1.7 **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)].`

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

Figure 10.2 The simple spare tire problem.

5.3.1.8 **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.

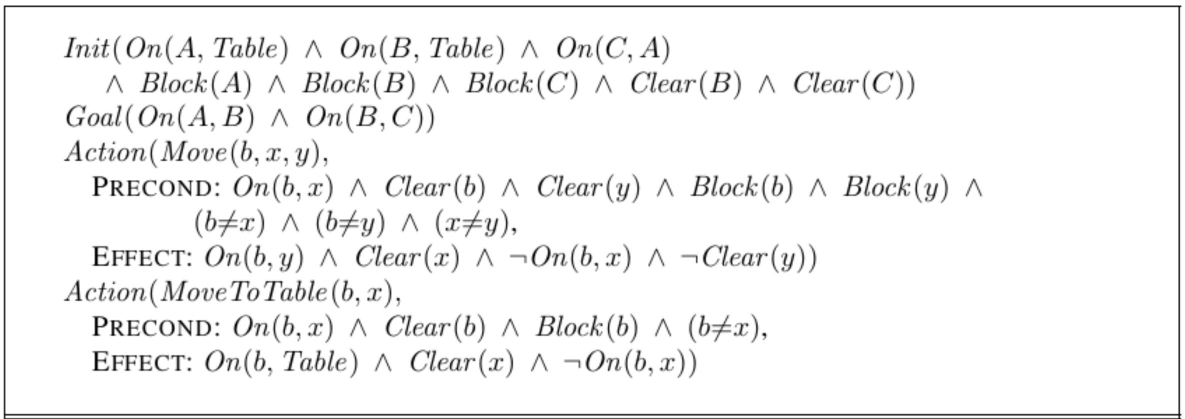
Key Concepts:

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.



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

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

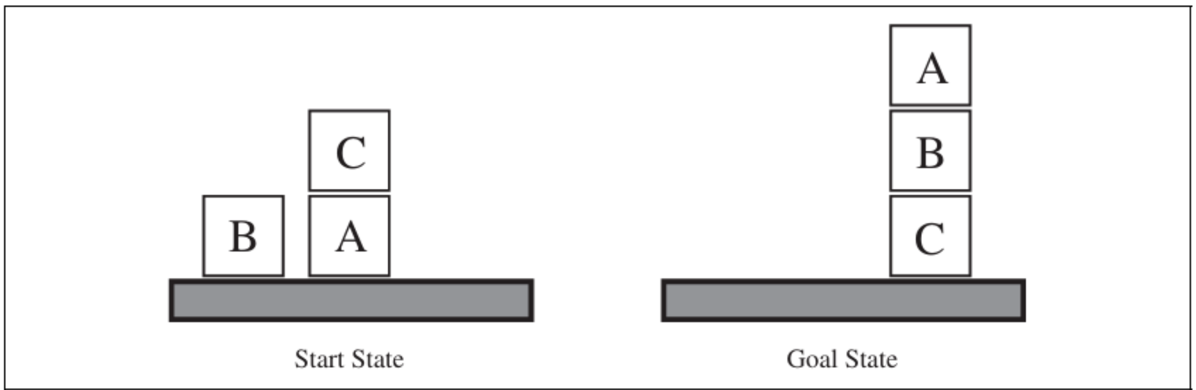


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

5.3.1.9 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.

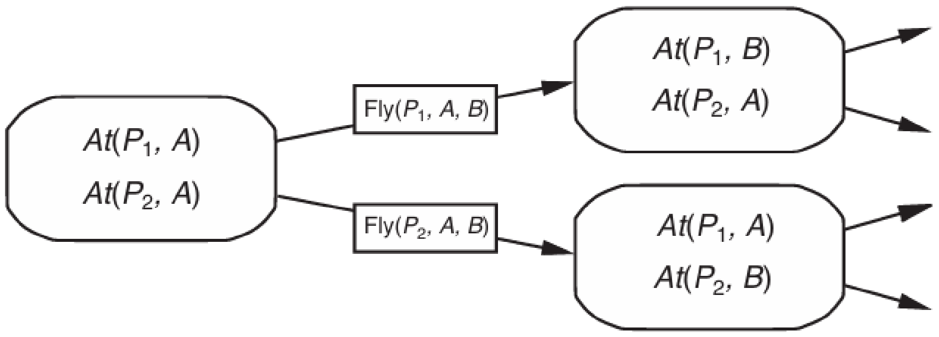
5.3.2 Algorithms for Planning as State-Space Search

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)

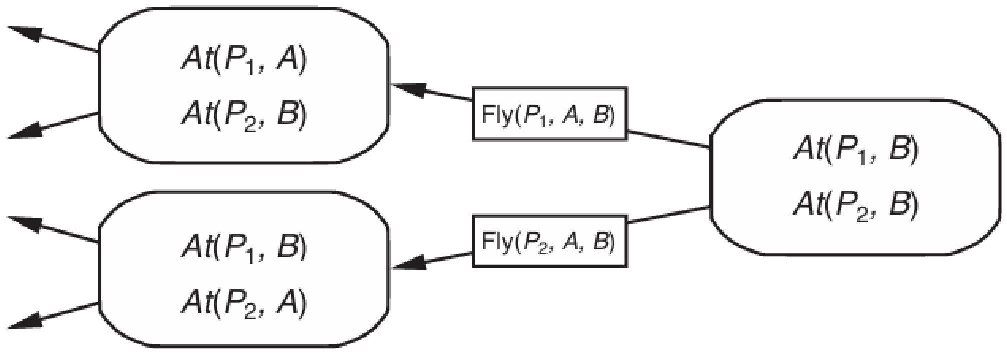


Backward (Regression) Relevant-States Search

- **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).

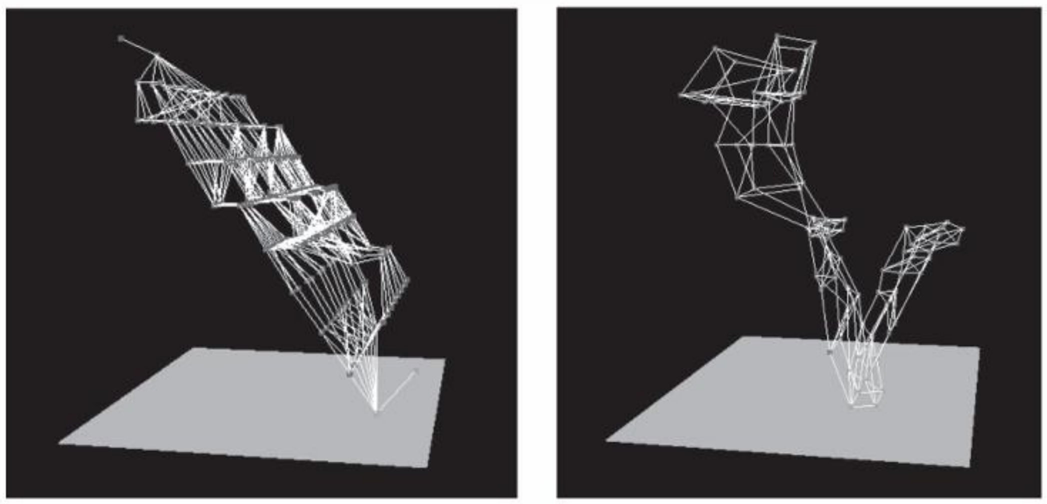


Figure 10.6 Two state spaces from planning problems with the ignore-delete-lists heuristic. The height above the bottom plane is the heuristic score of a state; states on the bottom plane are goals. There are no local minima, so search for the goal is straightforward. From Hoffmann (2005).

5.3.3 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 (leveled 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 **Eaten(Cake) ¬ ¬ Have(Cake) Eaten(Cake) Eaten(Cake)**. The 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.

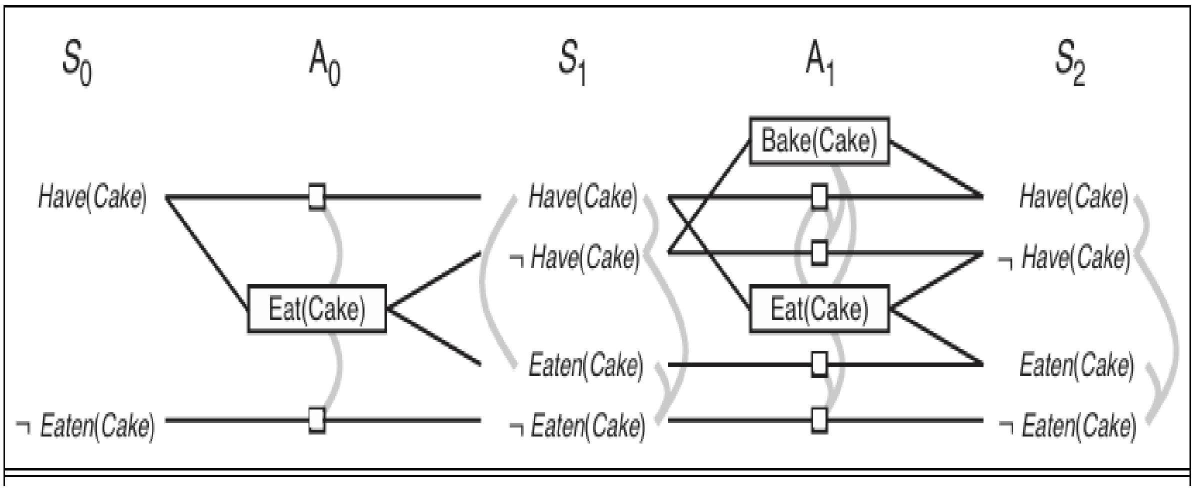


Fig 10.8

Planning Graphs for Heuristic Estimation

- **Level Cost:** The level at which a literal first appears in the graph estimates the cost to achieve it.
    - Example: `Have(Cake)` appears at **S_0**; `Eaten(Cake)` appears at **S_1**.
- **Heuristic Approaches:**
    1. **Max-Level Heuristic:** Maximum level cost of individual goals (admissible but less accurate).
    2. **Level-Sum Heuristic:** Sum of level costs (not admissible but practical).
    3. **Set-Level Heuristic:** Level at which all goals appear without mutual exclusion (accurate and admissible).

The GRAPHPLAN Algorithm

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)
```

Description:

GRAPHPLAN uses planning graphs to extract a valid plan or determine that none exists.

Steps:

1. **Expand Graph:** Build levels of the planning graph until all goals are present and non-mutex.
2. **Extract Solution:** Search for a valid plan using actions from the graph.
3. **Iterate:** If a solution is not found, expand the graph further.

Example:

For the spare tire problem:

- **Initial State (S_0):** `At(Spare, Trunk) ∧ At(Flat, Axle).`
- **Goal:** `At(Spare, Axle).`
- **Plan Extraction:**
  - **A_0:** Remove(Flat, Axle) ∧ Remove(Spare, Trunk).
  - **A_1:** PutOn(Spare, Axle).

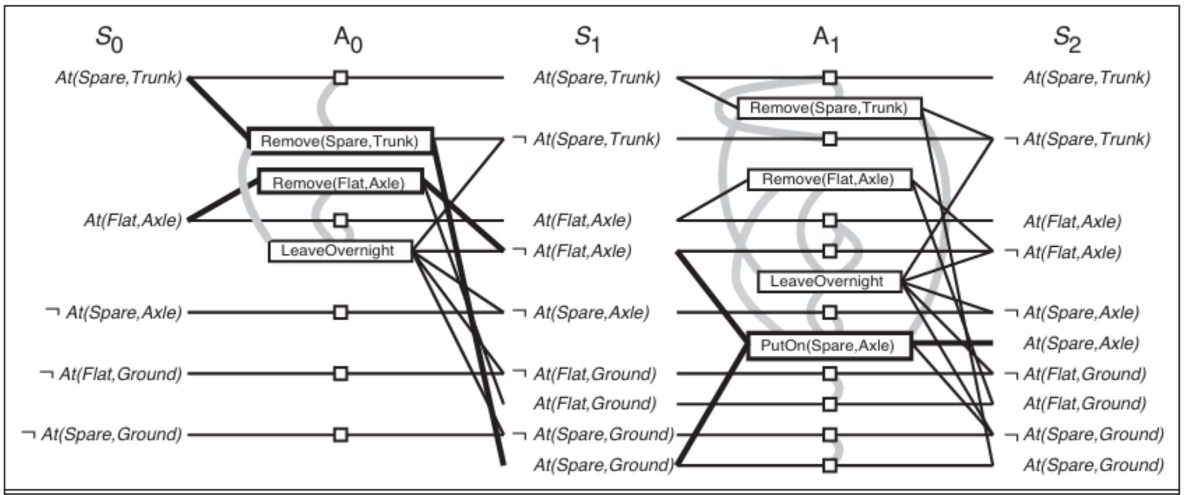


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

Planning Graph Mutex Relations and EXTRACT-SOLUTION

Mutex Relations in Planning Graphs

1. **Interference:**
   - Example: **Remove(Flat, Axle)** and **LeaveOvernight** are mutex because:
     - **Remove(Flat, Axle)** requires `At(Flat, Axle)` as a precondition.
     - **LeaveOvernight** negates `At(Flat, Axle)` as an effect.
2. **Competing Needs:**
   - Example: **PutOn(Spare, Axle)** and **Remove(Flat, Axle)** are mutex because:
     - **PutOn(Spare, Axle)** requires `At(Flat, Axle)` as a precondition.
     - **Remove(Flat, Axle)** negates `At(Flat, Axle).`

3. **Inconsistent Support:**
   - Example: `At(Spare, Axle)` and `At(Flat, Axle)` in **S_2** are mutex because:
     - **PutOn(Spare, Axle** is required to achieve `At(Spare, Axle)`.
     - It conflicts with the persistence of `At(Flat, Axle)`.

   These mutex relations ensure conflicts like placing two objects in the same location are detected.

EXTRACT-SOLUTION Algorithm

Purpose: Determines if a solution exists by analyzing goals and resolving conflicts in the planning graph.

Process:

1. **Initial State:**
   - Start at the last level (**S_n**) of the planning graph with a set of unsatisfied goals.
2. **Actions Available:**
   - At level **S_i**, choose a **conflict-free subset** of actions from **A_{i-1}**:
     - Actions whose effects cover the current goals.
     - Actions where neither the actions nor their preconditions are mutex.
3. **Goal State:**
   - Reach **S_0** where all goals are satisfied.
4. **Cost:**
   - Each action has a cost of **1**.

Example Solution: Spare Tire Problem

1. **Start at S_2:** Goal is `At(Spare, Axle)`.
   - Only action: **PutOn(Spare, Axle)**.
   - Leads to **S_1** with goals:
     - `At(Spare, Ground)`
     - `¬At(Flat, Axle)`.

2. **At S_1:**
   - `At(Spare, Ground)` is achieved by **Remove(Spare, Trunk)**.
   - `¬At(Flat, Axle)` is achieved by **Remove(Flat, Axle)**.
   - Conflict: **LeaveOvernight** is mutex with **Remove(Spare, Trunk)**.
3. **Reach S_0:** Goals are `At(Spare, Trunk)` and `At(Flat, Axle)`. Both are present.
   - Solution:
     - **A_0:** `Remove(Spare, Trunk),Remove(Flat, Axle).`
     - **A_1:** `PutOn(Spare, Axle).`

No-Good States

- If no solution is found for a specific `(level, goals)` pair, mark it as **no-good**.
- Prevent redundant searches by storing and reusing no-good results.

Greedy Heuristic for Backward Search

1. **Select Literal:** Start with the literal with the highest level cost.
2. **Choose Action:** Prefer actions with simpler preconditions (sum or max level cost of preconditions is smallest).

This approach provides efficient guidance to resolve goals during backward search.

Termination of GRAPHPLAN

Monotonic Properties**:**

1. **Literals:** Increase monotonically across levels (once added, they persist).
2. **Actions:** Increase monotonically with available preconditions.
3. **Mutexes:** Decrease monotonically as conflicts resolve.
4. **No-Goods:** Decrease as solutions become unachievable.

Guarantee of Termination:

- The graph levels off when no new literals, actions, or mutexes are added.
- If goals remain unreachable after leveling off, GRAPHPLAN terminates with failure.

Termination of GRAPHPLAN (Simplified Rewrite)

The GRAPHPLAN algorithm guarantees termination and correctly reports failure when no solution exists. Below is an explanation of why this is true.

Why Can't We Stop at Leveling Off?

A planning graph "levels off" when two consecutive levels are identical, indicating that no new literals, actions, or mutexes are added. However, leveling off does not always mean a solution is present:

- **Example:**
  In an air cargo problem with one plane and *n* pieces of cargo at airport A, destined for airport B:
  - A single piece of cargo can be transported in three steps: **Load, Fly, Unload**.
  - The graph levels off at level 4, but a full solution requires **4n - 1 steps** (accounting for return trips to pick up additional cargo).

How Long Should We Expand the Graph?

- **EXTRACT-SOLUTION Failure:**
  If the function fails to find a solution, it marks certain goal sets as **no-goods** (unachievable sets of goals).
  - Continue expanding if fewer no-goods might exist at the next level.
  - Stop expanding when both the graph and the no-goods have leveled off, as no further solutions can emerge.

Proof of Termination

Certain properties of planning graphs ensure they will always level off:

1. **Literals Increase Monotonically:**
    - Once a literal appears at a level, it persists in all subsequent levels due to **persistence actions** (no-op actions that maintain literals).
2. **Actions Increase Monotonically:**
    - If an action's preconditions are present at a level, the action appears at that level and all subsequent levels.
3. **Mutexes Decrease Monotonically:**
    - **Mutexes** (mutual exclusions) never reappear once resolved.
    - **Reason:** If actions or literals are mutex at a level, they remain so in prior levels. However, as more actions and literals appear, mutexes naturally decrease.
4. **No-Goods Decrease Monotonically:**
    - A goal set marked as a no-good at one level cannot become achievable in previous levels.
    - **Proof by Contradiction:** If a no-good set was achievable at an earlier level, persistence actions could extend it to the current level, contradicting its status as a no-good.

Key Termination Point

- **Finite Properties:**
    - The number of actions and literals is finite.
    - Eventually, there will be a level where the number of actions, literals, mutexes, and no-goods remains unchanged.
- **Final Check:**
    - If any goal is missing or mutex with another goal, the GRAPHPLAN algorithm terminates and returns failure.

Conclusion

The monotonic properties of planning graphs—literals and actions increasing, mutexes and no-goods decreasing—ensure that the graph levels off in finite time. At this point, GRAPHPLAN either finds a solution or confirms that none exists. For additional details, refer to Ghallab et al. (2004).

Logic Programming

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 $A \wedge B \Rightarrow C$, 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."

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!

Efficient Implementation of Logic Programs

Prolog programs can run in two ways: **interpreted mode** and **compiled mode**. Here's how Prolog handles efficiency and parallel processing:

How Prolog Works

1. **Interpreted Mode**
   - In this mode, Prolog works like a problem-solving engine. It searches for solutions step by step in the program (called the knowledge base).
   - Prolog uses optimizations to speed things up, like:
     - **Choice Points**: A global stack keeps track of alternative paths when solving a problem. This makes execution faster and debugging easier.
     - **Trail**: When Prolog assigns a value to a variable, it remembers it in a "trail." If the current path fails, Prolog can quickly undo these assignments to try other options.
2. **Compiled Mode**
   - In compiled mode, Prolog creates a specialized program for a specific task. This avoids repetitive work, like searching for rules, and makes it faster.
   - Compilers like the **Warren Abstract Machine (WAM)** optimize Prolog code by turning it into an intermediate language that can be executed more efficiently.

Key Features of Prolog Execution

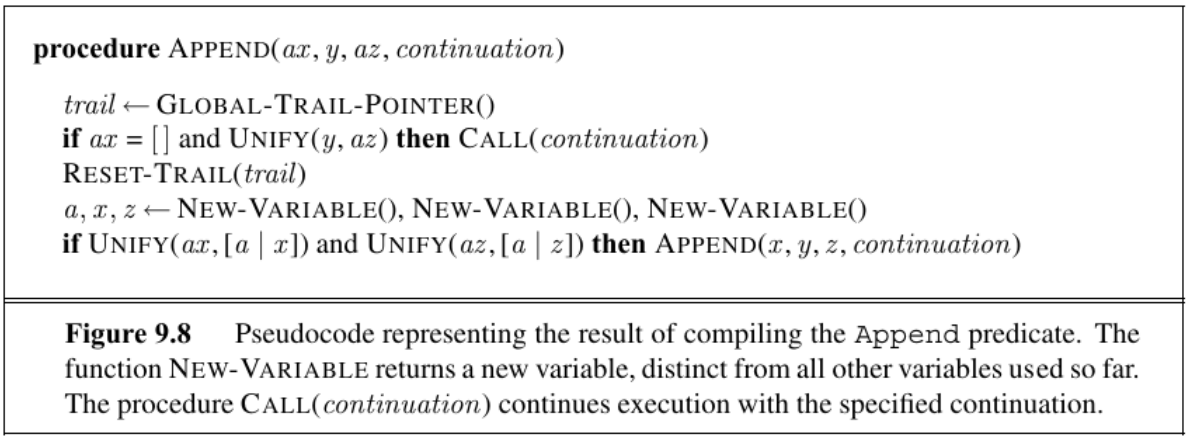
1. **Efficiency**
   - Prolog’s interpreters are slower because they repeatedly analyze and match rules.
   - Compiled Prolog eliminates this overhead by directly using a tailored procedure for each rule.
2. **Choice Points and Continuations**
   - **Choice Points**: Prolog keeps track of decisions made during problem-solving. If a path fails, it can backtrack to an earlier decision.
   - **Continuations**: These help Prolog keep track of "what to do next" when a solution is found. This ensures all possible solutions are explored efficiently.

3. **Parallelization**
   - **OR-Parallelism**: Prolog can explore multiple possible solutions simultaneously when a goal can match multiple rules.
   - **AND-Parallelism**: It can work on different parts of a problem (called conjuncts) at the same time. However, this is more complex because all parts need to agree on variable values.

Why Prolog Is Useful

- Prolog's design lets it quickly handle tasks like planning, natural language processing, and AI research.
- Thanks to optimization techniques, Prolog programs can run as fast as C for many tasks, while being easier to write for logic-based problems.

In short, Prolog uses smart techniques like **choice points**, **trails**, and **parallelization** to solve problems efficiently. This makes it ideal for rapid prototyping in AI and other fields.



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

    $trail \leftarrow$ GLOBAL-TRAIL-POINTER()
    if $ax = [\,]$ and UNIFY($y, az$) then CALL($continuation$)
    RESET-TRAIL($trail$)
    $a, x, z \leftarrow$ NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
    if UNIFY($ax, [a \mid x]$) and UNIFY($az, [a \mid z]$) 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.

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

The pseudocode in the image describes how the **`Append`** predicate works when compiled into an optimized form. Here's a breakdown of each part in simple terms:

Purpose

The `Append(ax, y, az, continuation)` procedure is designed to combine two lists (`ax` and `y`) to produce a new list `az`, while allowing further execution through a **continuation** (a function that specifies "what to do next").

Key Components

1. **Trail Setup**
   - `trail ← GLOBAL-TRAIL-POINTER()`
     - A **trail** is used to keep track of variable bindings. This helps Prolog undo (or "unbind") variable assignments if a path fails and backtracking is required.
2. **Base Case**
   - `if ax = [] and UNIFY(y, az) then CALL(continuation)`
     - If `ax` (the first list) is empty, the result (`az`) should match `y`.
     - Prolog unifies the values of `y` and `az` and executes the next step using the **continuation**.
3. **Backtracking Preparation**
   - `RESET-TRAIL(trail)`
     - If the base case fails, the trail is **reset** to undo bindings made during the failed attempt. This prepares Prolog to try the next possible solution.
4. **Recursive Case**
   - `a, x, z ← NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()`
     - New variables (`a`, `x`, and `z`) are created to handle the decomposition of `ax` (splitting it into a head element `a` and the rest `x`).
   - `if UNIFY(ax, [a | x]) and UNIFY(az, [a | z]) then APPEND(x, y, z, continuation)`
     - The algorithm checks:
       1. If `ax` can be broken into a head (`a`) and tail (`x`), and
       2. If `az` can be built from `a` followed by `z`.
     - If both conditions are met, the procedure **recursively calls itself** to append the rest of the list (`x`) with `y`, building up the final result (`z`).
5. **Continuation Execution**
   - After finding a valid solution, the **continuation** is called. It ensures the program continues with the next steps based on the solution found.

Explanation of Figure

- **NEW-VARIABLE**: Creates fresh variables to avoid conflicts with existing ones.
- **CALL(continuation)**: Executes the continuation, ensuring the program progresses.

- **RESET-TRAIL**: Cleans up variable bindings if a solution fails, enabling backtracking.

Summary: The `Append` predicate in the pseudocode is an optimized version of appending two lists. It uses:

1. **Choice Points** to explore alternatives.
2. **Trail Management** to undo changes when backtracking.
3. **Continuations** to handle the next steps in execution.

This approach allows efficient execution and flexibility in handling multiple solutions.

Redundant Inference and Infinite Loops in Prolog

Key Issue: Prolog's use of depth-first search can cause two major problems when working with graphs:

1. **Infinite Loops**
2. **Redundant Computations**

---

1. Infinite Loops

- Prolog uses **depth-first backward chaining** to answer queries.
- In some cases, Prolog keeps following paths indefinitely, creating infinite loops.

Example of Infinite Loops

The program checks if a path exists between two nodes using this logic:

```
path(X, Z) :- link(X, Z).          % Base case:
Direct link exists
path(X, Z) :- path(X, Y), link(Y, Z). % Recursive
case: Connect through another node
```

- For a graph like in **Figure 9.9(a)** (nodes `A → B → C`), asking Prolog if `path(a, c)` exists:

- If the **base case** is checked first, Prolog finds the path without issues. This is shown in **Figure 9.10(a)**.
- If the **recursive case** is checked first (i.e., the order of the clauses is reversed), Prolog follows paths indefinitely. This is shown in **Figure 9.10(b)**.

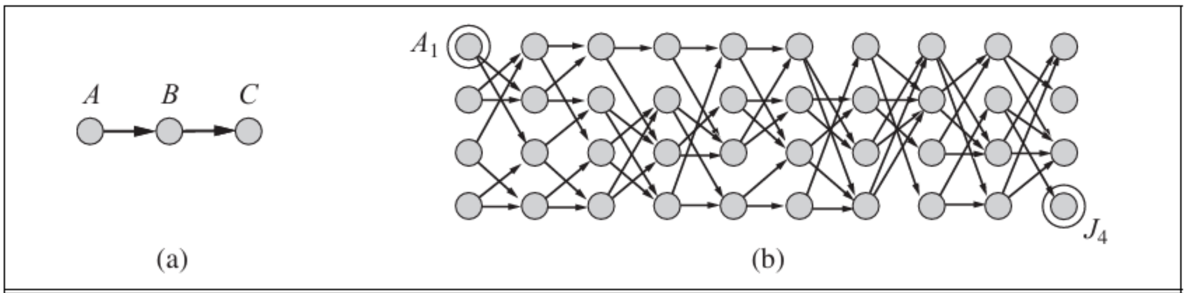


Figure 9.9 (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A_1 to J_4 requires 877 inferences.

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

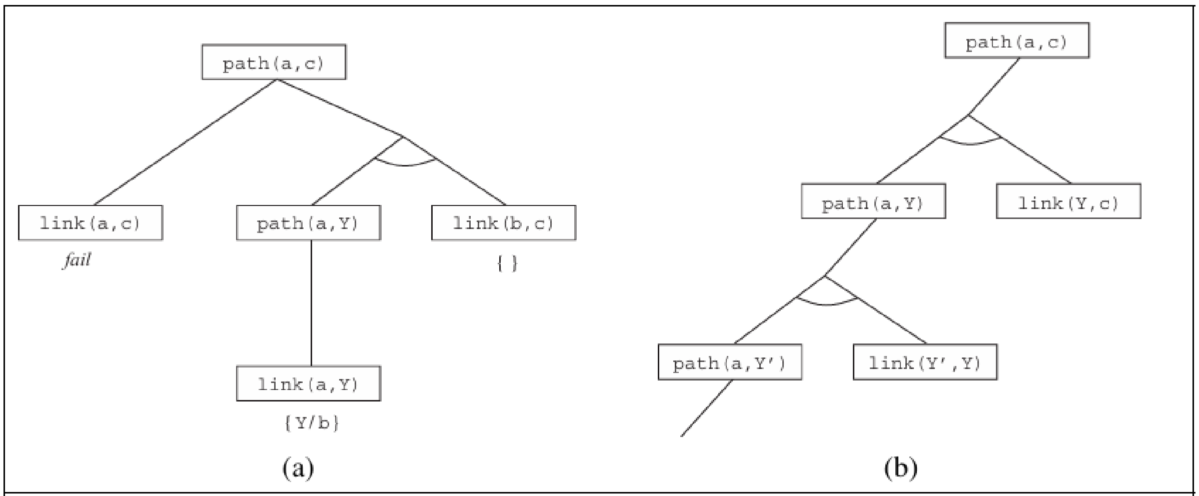


Figure 9.10 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

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

Why It Happens:

Depth-first search keeps exploring deeper paths without checking if a solution already exists or if the current path is looping.

2. Redundant Computations

Even when there are no infinite loops, Prolog's depth-first search can waste time by repeatedly calculating the same paths.

Example of Redundancy

In **Figure 9.9(b)**, finding a path from `A1` to `J4` requires 877 inferences because Prolog keeps checking all possible paths, even those that don't lead to the goal. Most of these paths are unnecessary.

Comparison with Forward Chaining:

- **Forward Chaining:**
  - It computes all paths once and stores the results (dynamic programming). For the same problem, forward chaining only requires 62 inferences.
- **Depth-First Backward Chaining:**
  - It doesn't remember previously computed results, leading to repeated calculations.

Solutions to Prolog's Problems

1. **Memoization**
   - Cache solutions to subproblems (subgoals) as they are found.
   - If a subgoal is encountered again, use the cached result instead of recomputing it.
2. **Tabled Logic Programming**
   - Combines the goal-directed nature of backward chaining with the efficiency of forward chaining.
   - It avoids infinite loops and redundant calculations by remembering results (similar to dynamic programming).

Benefits of Tabled Logic Programming

- **Complete for Datalog Knowledge Bases:**
  - Ensures all valid solutions are found without infinite loops.
- **Efficient:**
  - Reduces redundant computations by caching results.

However, it cannot handle predicates with potentially infinite objects (e.g., `father(X, Y)` for all possible pairs of people).

Database Semantics of Prolog

Prolog uses a **special way of interpreting facts** called **database semantics**, which is simpler and more efficient than full first-order logic (FOL). Here's how it works:

---

1. Unique Names Assumption

- In Prolog, every constant or ground term refers to a **unique, distinct object**.
    - Example: `CS` and `EE` are different, just as `101`, `102`, and `106` are all different.

---

2. Closed World Assumption

- Prolog assumes that **only the facts explicitly stated in the knowledge base are true**.
- If something isn't in the knowledge base, Prolog assumes it is false.
    - For instance, if the Prolog facts are:

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

        - This means there are **exactly four courses**.
        - Prolog assumes there are no other courses unless stated explicitly.

Comparison with First-Order Logic (FOL)

- In FOL, the same facts would mean:
  - **At least one course exists**.
  - There might be more courses that aren't mentioned.
- In Prolog, however, the **closed world assumption** guarantees that there are **exactly four courses**.

Why Prolog is Simpler

1. **Prolog Assumptions Make It Efficient**:
   - Prolog doesn't allow for uncertainty (e.g., unmentioned courses). This keeps reasoning simpler and faster.
2. **Prolog Doesn't Handle "False" Assertions**:
   - Unlike FOL, you can't directly state that something is false in Prolog.

How This Works Mathematically

If we convert the Prolog facts into FOL, we would write something like:

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

- This expresses the idea that **exactly four courses exist**, but the process becomes more complicated.

Practical Takeaway

- **Use Prolog for database-like problems**:
  - If the problem can be described with database semantics (unique names + closed world), Prolog is simpler and more efficient.
  - Translating everything into FOL and reasoning with a full theorem prover is more powerful but slower and less practical.

In short, Prolog's database semantics make it less expressive than FOL but much more efficient for certain tasks, such as reasoning about a fixed set of known facts.

Constraint Logic Programming (CLP)

1. What is Constraint Logic Programming (CLP)?

CLP is a type of logic programming that combines **logic rules** with **constraints**. Unlike standard Prolog, which only works with finite solutions, CLP can handle more complex problems, including those with **infinite domains** like integers or real numbers.

---

2. How Prolog Handles Constraints

- Standard Prolog solves problems using **backtracking** (exploring possibilities one by one), which works for **finite-domain problems**.
  - Example: If three colors are allowed, the query `diff(Q, SA)` (Queensland and South Australia must have different colors) will have **six possible solutions**.
- But Prolog fails with **infinite-domain problems**, like checking conditions involving integers or real numbers.

---

3. Example Problem: Triangle Inequality

Let's define a rule to check if three numbers can form a triangle:

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

- If you ask Prolog `triangle(3, 4, 5)`, it works because all values are known.
- But if you ask `triangle(3, 4, Z)`, Prolog fails because it cannot handle unbound variables in comparisons like `Z >= 0`.

---

4. How CLP Solves This

- **CLP allows variables to remain constrained instead of fully defined.**
- For the query `triangle(3, 4, Z)`, the result will be:
  - `7 >= Z >= 1` (Z must be between 1 and 7 to form a triangle).

This makes CLP more powerful than standard Prolog for solving constraint satisfaction problems (CSPs).

---

5. CLP Algorithms

- CLP uses specialized algorithms to solve different types of constraints.
  - Example: For **real-valued variables** and **linear inequalities**, CLP might use **linear programming**.
- Unlike Prolog's default **depth-first backtracking**, CLP systems use more advanced techniques like:
  - **Heuristic conjunct ordering**: Solving the simplest part of the problem first.
  - **Backjumping**: Skipping unnecessary steps when a dead-end is reached.
  - **Cutset conditioning**: Breaking problems into smaller, manageable parts.

---

6. Flexibility of CLP Systems

- CLP systems allow more control over how problems are solved:
  - Programmers can write rules to specify the order in which constraints are checked.
  - For example, they can choose to first solve constraints with the **fewest unknowns**.
- Tools like the **MRS language** enable this level of customization.

CLP combines the best of logic programming, constraint-solving, and database techniques to solve problems more efficiently, especially for those involving constraints on variables (like ranges or inequalities). It’s more flexible and powerful than standard Prolog for complex scenarios.