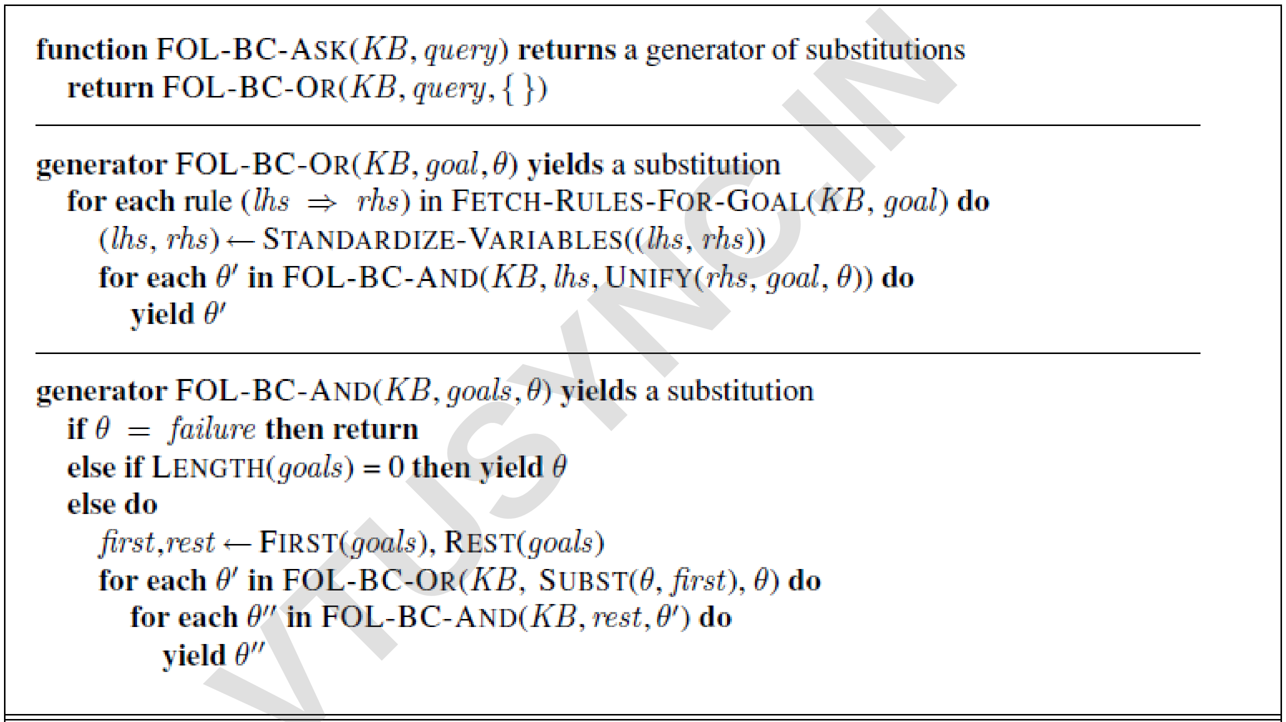
Syllabus:

- **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**
- **Text book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson, 2015**

BACKWARD CHAINING



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

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

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

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

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

Steps:

1.Unification

2.Substitution

3.Standardization of Variables

- Knowledge Base (KB):
- Rule 1: If Criminal(x)←American(x)∧Weapon(y)∧Sells(x,y,z)∧Hostile(z)
- Rule 2: Sells(x,y,z)←Missile(y)∧Owns(z,y)←Missile(y)∧Owns(z,y)
- Fact 1: American(West)

- Fact 2: Weapon(M1)
- Fact 3: Hostile(Nono)

Query: Is West a criminal? (i.e., Criminal(West)

Step 1 (Start with the query):

- The query is Criminal(West)
- We apply FOL-BC-OR(KB, Criminal(West), {}).

Step 2 (Look for applicable rules):

- Rule 1 applies because Criminal(x)is the goal.
- Now we need to check if all the premises of Rule 1 are satisfied for **West**:
    - American(West) (True, from Fact 1).
    - Weapon(M1) (True, from Fact 2).
    - Sells(West,M1,Nono) (We need to check this).
    - Hostile(Nono) (True, from Fact 3).

Step 3 (Check Sells(West,M1,Nono):

- Rule 2 applies to this sub-goal.
- We check the premises:
    - Missile(M1) (True, from Fact 2).
    - Owns(Nono,M1) (Assume this is true).
- If both are true, we can conclude that Sells(West,M1,Nono)is true.
- **Step 4 (Conclude the query):**
- Since all premises of Rule 1 are satisfied, we can conclude Criminal(West) is true.

Step 5 (Return the substitution):

- The final result is a substitution that shows **West** is a criminal.

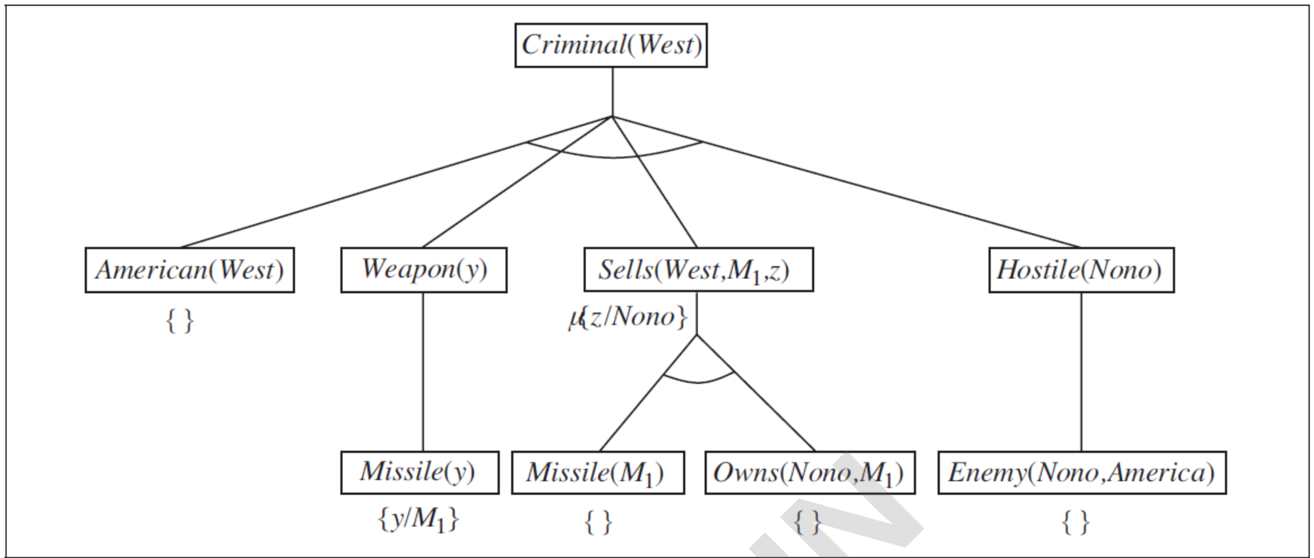


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining.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

Algorithm = Logic + Control

- Prolog is the most widely used logic programming language.
- Prolog uses uppercase letters for variables and lowercase for constants—the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written "backwards" from what we are used to; instead of $A \wedge B \Rightarrow C$ in Prolog we have C :- A, B.
- Here is a typical example:

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

The notation [E|L] denotes a list whose first element is E and whose rest is L.

Here is a Prolog program for append(X,Y,Z), which succeeds if list Z is the result of appending lists X and Y:

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

We can read these clauses as

(1) appending an empty list with a list Y produces the same list Y and

(2) [A|Z] is the result of appending [A|X] onto Y, provided that Z is the result of appending X onto Y.

Query append(X,Y,[1,2]):Possible solutions are

X=[] Y=[1,2];

X=[1] Y=[2];

X=[1,2] Y=[]

Efficient implementation of Logic Programming

- The execution of a Prolog program can happen in two modes: interpreted and compiled.
- Interpretation essentially amounts to running the FOL-BC-ASK algorithm with the program as the knowledge base.
- Prolog interpreters have a global data structure, a stack of choice points, to keep track of the multiple possibilities.
- Second, our simple implementation of FOL-BC-ASK spends a good deal of time generating substitutions.
- Instead of explicitly constructing substitutions, Prolog has logic variables that remember their current binding.
- When a path in the search fails, Prolog will back up to a previous choice point, and then it might have to unbind some variables.
- This is done by keeping track of all the variables that have been bound in a stack called the trail.
- A compiled Prolog program, on the other hand, is an inference procedure for a specific set of clauses, so it *knows* what clauses match the goal.
- The instruction sets of today's computers give a poor match with Prolog's semantics, so most Prolog compilers compile into an intermediate language rather than directly into machine language.
- The most popular intermediate language is the Warren Abstract Machine, or WAM, named after David H. D. Warren, one of the implementers of the first Prolog compiler.
- The WAM is an abstract instruction set that is suitable for Prolog and can be either interpreted or translated into machine language.

- **The definition of the Append predicate can be compiled into the code**

```
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($x, y, z, continuation$)
```

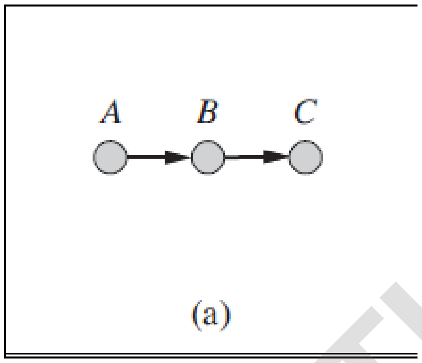
Figure 9.8 Pseudocode representing the result of compiling the Append predicate. The function NEW-VARIABLE returns a new variable, distinct from all other variables used so far. The procedure CALL(*continuation*) continues execution with the specified continuation.

- **Unification: This is a central operation in logic programming that attempts to make two terms (variables or constants) equal.**
- **Backtracking: If any unification fails, the procedure will revert to the previous state using**

the global trail pointer, enabling it to attempt alternative solutions.

- **Recursion: The procedure uses recursion to continue traversing the lists and append elements one by one.**
- **Parallelization can also provide substantial speedup. There are two principal sources of parallelism.**
- **The first, called OR-parallelism, comes from the possibility of a goal unifying with many different clauses in the knowledge base.**
- **Each gives rise to an independent branch in the search space that can lead to a potential solution and all such branches can be solved in parallel.**
- **The second is AND-parallelism comes from the possibility of solving each conjunct of the body in the implication of parallel.**
- **AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables.**
- **Each conjunctive branch must communicate with the other branches to ensure a global solution**

Redundant Inferences and infinite loop



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

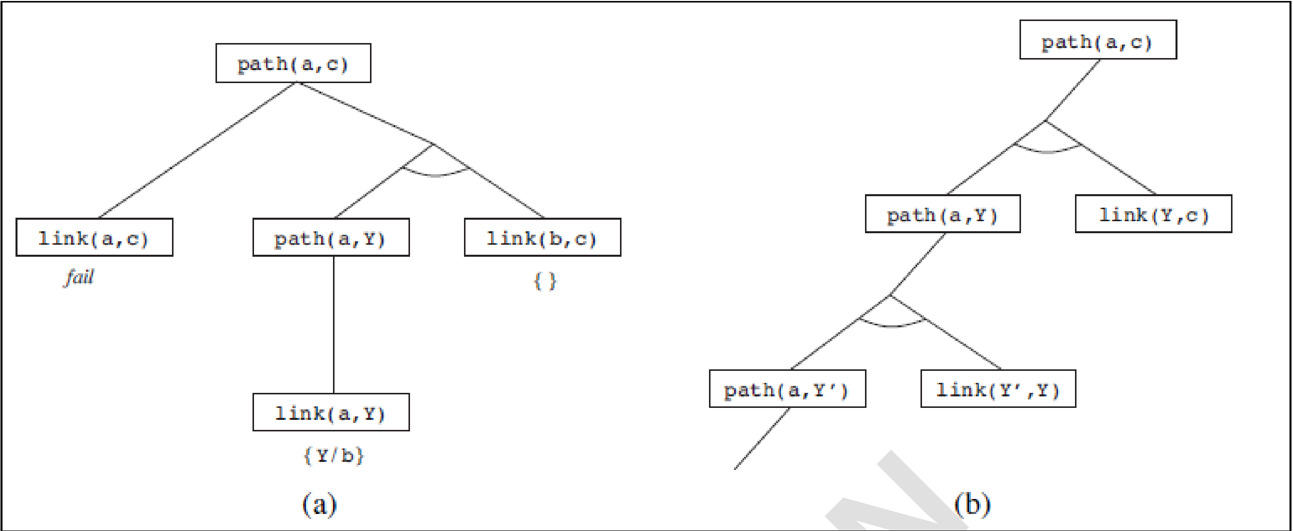


Figure 9.10 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

a)path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z).
b)path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X,Z).---Fails

Database Semantics of Prolog

Course(CS, 101), Course(CS, 102), Course(CS, 106), Course(EE, 101)--1

In First Order Logic representation

- Course(d, n) ⇔ (d=CS ∧ n = 101) ∨ (d=CS ∧ n = 102) ∨ (d=CS ∧ n = 106) ∨ (d=EE ∧ n = 101) .
- This is called the completion of Equation 1.
- To express in FOL the idea that there are at least four courses, we need to write the completion of the equality predicate:
- x = y ⇔ (x = CS ∧ y = CS) ∨ (x = EE ∧ y = EE) ∨ (x = 101 ∧ y = 101) ∨ (x = 102 ∧ y = 102) ∨ (x = 106 ∧ y = 106) .
- The completion is useful for understanding database semantics.

Constraint Logic Programming

- Constraint logic programming (CLP) allows variables to be *constrained* rather than *bound*.
- A CLP solution is the most specific set of constraints on the query variables that can be derived from the knowledge base.
- Consider the following example. We define triangle(X,Y,Z) as a predicate that holds if the three arguments are numbers that satisfy the triangle inequality:

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

- If we ask Prolog the query triangle(3,4,5), it succeeds.
- On the other hand, if we ask triangle(3,4,Z), no solution will be found, because the subgoal Z>=0

cannot be handled by Prolog; we can’t compare an unbound value to 0.

RESOLUTION

Conjunctive normal form of 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) .
- “Everyone who loves all animals is loved by someone,” or
- ∀ x [∀ y Animal(y) ⇒ Loves(x, y)] ⇒ [∃ y Loves(y, x)] .
- The steps are as follows:
- **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) ∧¬Love
- The sentence now reads “Either there is some animal that x doesn’t love, or (if this is not the case) someone loves x.”
- Clearly, the meaning of the original sentence has been preserved.s(x, y)] ∨ [∃ y Loves(y, x)] .
- **Standardize variables**: For sentences like (∃xP(x))∨(∃xQ(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.In the simple case, it is just like the Existential Instantiation rule.
- If we blindly apply the rule to the two matching parts we get
- ∀ x [Animal (A) ∧ ¬Loves(x,A)] ∨ Loves(B, x) ,
- which has the wrong meaning entirely: it says that everyone either fails to love a particular animal

A or is loved by some particular entity B.

- In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person.
- Thus, we want the Skolem entities to depend on x and z:
- $\forall$ x [Animal (F(x)) $\land \neg$Loves(x, F(x))] $\lor$ Loves(G(z), x) .
- Here F and G are **Skolem functions**.
- The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears.
- As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.
- **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)) $\land \neg$Loves(x, F(x))] $\lor$ Loves(G(z), x) .
- **Distribute $\lor$ over $\land$**:
- [Animal (F(x)) $\lor$ Loves(G(z), x)] $\land$ [$\neg$Loves(x, F(x)) $\lor$ Loves(G(z), x)] .
- This step may also require flattening out nested conjunctions and disjunctions.

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.
- For example, we can resolve the two clauses
- [Animal (F(x)) $\lor$ Loves(G(x), x)] and [$\neg$Loves(u, v) $\lor$ $\neg$Kills(u, v)]
- by eliminating the complementary literals Loves(G(x), x) and $\neg$Loves(u, v), with unifier θ={u/G(x), v/x}, to produce the **resolvent** clause
- [Animal (F(x)) $\lor$ $\neg$Kills(G(x), x)] .

<u>Proofs Resolution</u>

<u>Example:</u>

- Everyone who loves all animals is loved by someone.
- Anyone who kills an animal is loved by no one.
- Jack loves all animals.
- Either Jack or Curiosity killed the cat, who is named Tuna.
- Did Curiosity kill the cat?
- First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. ∀ x [∀ y Animal (y) ⇒ Loves(x, y)] ⇒ [∃ y Loves(y, x)]
B. ∀ x [∃ z Animal (z) ∧ Kills(x, z)] ⇒ [∀ y ¬Loves(y, x)]
C. ∀ x Animal(x) ⇒ Loves(Jack, x)
D. Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
E. Cat(Tuna)
F. ∀ x Cat(x) ⇒ Animal (x)
¬G. ¬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. Becauseanyone 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.

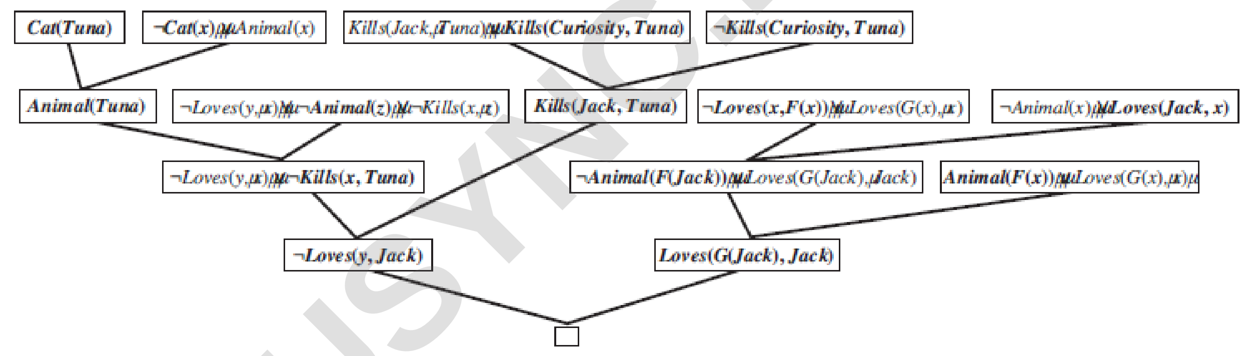

Cat(Tuna)
¬Cat(x) ∨ Animal(x)
Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
¬Kills(Curiosity, Tuna)
Animal(Tuna)
¬Loves(y, x) ∨ ¬Animal(z) ∨ ¬Kills(x, z)
Kills(Jack, Tuna)
¬Loves(x, F(x)) ∨ Loves(G(x), x)
¬Animal(x) ∨ Loves(Jack, x)
¬Loves(y, x) ∨ ¬Kills(x, Tuna)
¬Animal(F(Jack)) ∨ Loves(G(Jack), Jack)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

Completeness of Resolution

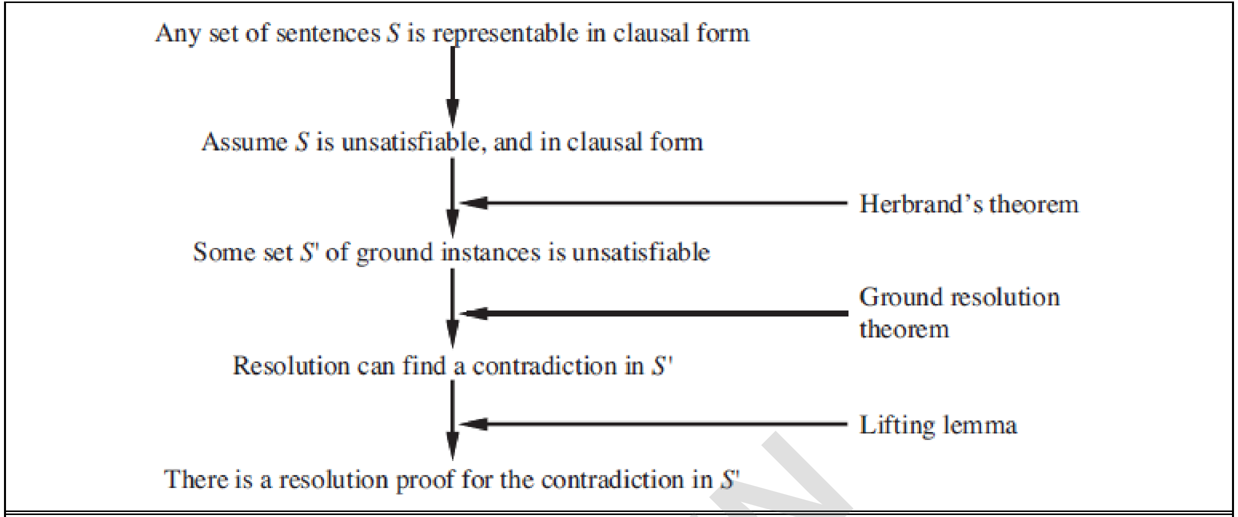

Any set of sentences S is representable in clausal form
Assume S is unsatisfiable, and in clausal form
Herbrand's theorem
Some set S' of ground instances is unsatisfiable
Ground resolution theorem
Resolution can find a contradiction in S'
Lifting lemma
There is a resolution proof for the contradiction in S'

Figure 9.13 Structure of a completeness proof for resolution.

1. First, we observe that if S is unsatisfiable, then there exists a particular set of *ground instances* of the clauses of S such that this set is also unsatisfiable (Herbrand's theorem).

   2. We then appeal to the **ground resolution theorem** , which states that propositional resolution is complete for ground sentences.

   3. We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

- To carry out the first step, we need three new concepts: **Herbrand universe**: If S is a set of clauses, then HS, the Herbrand universe of S, is the set of all ground terms constructable from the following:
  a. The function symbols in S, if any.
  b. The constant symbols in S, if any; if none, then the constant symbol A.
  For example, if S contains just the clause ¬P(x, F(x,A))∨¬Q(x,A)∨R(x,B), then
  HS is the following infinite set of ground terms: {A,B, F(A,A), F(A,B), F(B,A), F(B,B), F(A,F(A,A)), . . .} .
- **Saturation**: If S is a set of clauses and P is a set of ground terms, then P(S), the saturation of S with respect to P, is the set of all ground clauses obtained by applying all possible consistent substitutions of ground terms in P with variables in S.
  • **Herbrand base**: The saturation of a set S of clauses with respect to its Herbrand universe is called the Herbrand base of S, written as HS(S). For example, if S contains solely the clause just given, then HS(S) is the infinite set of clauses
- {¬P(A, F(A,A)) ∨ ¬Q(A,A) ∨ R(A,B),
- ¬P(B,F(B,A)) ∨¬Q(B,A) ∨ R(B,B),
- ¬P(F(A,A), F(F(A,A),A)) ∨¬Q(F(A,A),A) ∨ R(F(A,A),B),

- ¬P(F(A,B), F(F(A,B),A)) ∨ ¬Q(F(A,B),A) ∨ R(F(A,B),B), . . . }

CLASSICAL PLANNING :
Chapter 10.1,10.2,10.3
Definition of Classical Planning, Algorithms for Planning as State-Space Search, Planning Graphs

Definition of Classical Planning

- The system is unable to infer the correct behavior of the XOR gate for certain input combinations, like 1 and 0.
- This failure to infer is due to the lack of knowledge about the relationship between those inputs (i.e., that 1 and 0 are different).
- By examining the axiom for the XOR gate and testing for the output at each gate, it becomes clear that the system needs to be explicitly told about the condition $1 \neq 0$ in order to deduce the correct output.
- Once this information is provided, the system can correctly infer that Signal (Out(1, X1)) = 1 when the inputs are 1 and 0.
  In essence, the problem is a missing or forgotten assertion that would allow the system to properly deduce the XOR gate's output.

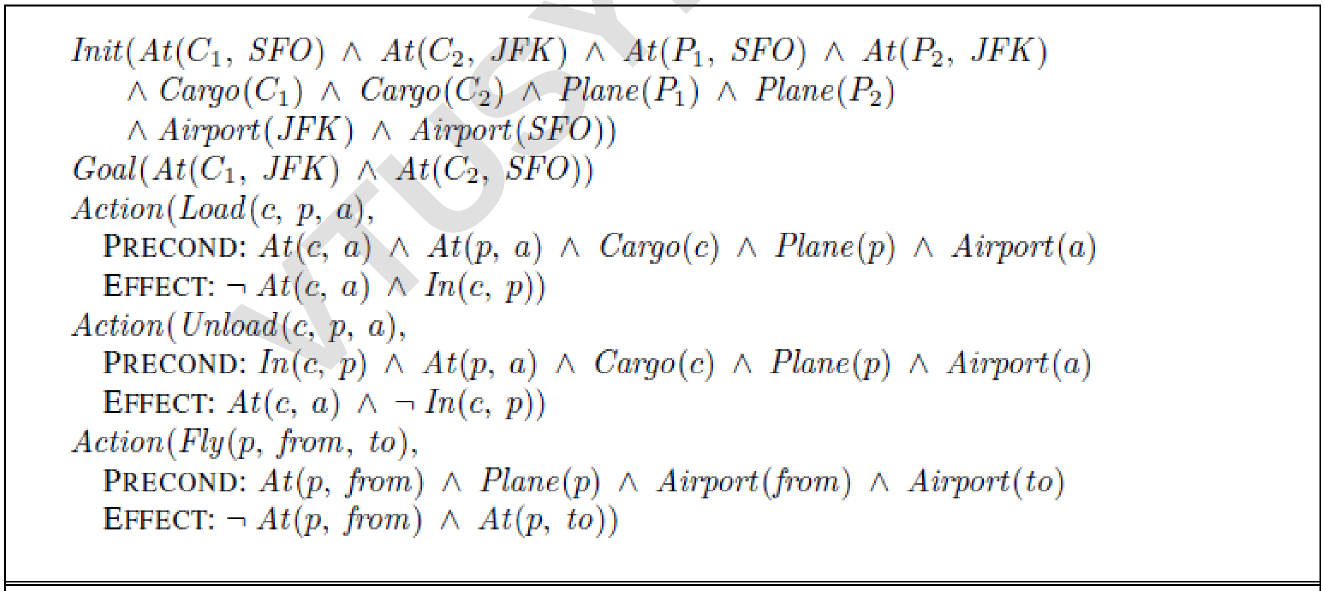
- **We use a language called PDDL, the Planning Domain Definition Language, that allows us to express all $4Tn^2$ actions with one action schema. There have been several versions of PDDL.**
- **We now show how PDDL describes the four things we need to define a search problem: the initial state, the**
  actions that are available in a state, the result of applying an action, and the goal test.
- **Each state is represented as a conjunction of fluents that are ground, functionless atoms.**
- **For example, Poor ∧ Unknown might represent the state of a hapless agent, and a state in a package delivery problem might be At(Truck 1, Melbourne) ∧ At(Truck 2, Sydney).**
- **The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a *set* of fluents, which can be manipulated with set operations.**
- **Actions are described by a set of action schemas that implicitly define the ACTIONS(s) and RESULT(s, a) functions needed to do a problem-solving search.**
- **Classical planning concentrates on problems where most actions leave most things unchanged.**
- **A set of ground (variable-free) actions can be represented by a single action schema.**

- **The schema is a lifted representation—it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic.**
- For example, here is an action schema for flying a plane from one location to another:
- Action(Fly(p, from, to), PRECOND:At(p, from) ∧ Plane(p) ∧ Airport (from) ∧ Airport (to) EFFECT:¬At(p, from) ∧ At(p, to))
- The schema consists of the action name, a list of all the variables used in the schema, a **precondition** and an **effect**.

Summary

1. **Load C1 onto P1 at SFO.**
2. **Fly P1 from SFO to JFK.**
3. **Unload C1 at JFK.**
4. **Load C2 onto P2 at JFK.**
5. **Fly P2 from JFK to SFO.**
6. **Unload C2 at SFO.**

After completing these actions,
C1 will be at JFK, and C2 will be at SFO, achieving the goal state.



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

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

- Steps to achieve the Goal:
- To achieve the goal where C1 is at JFK and C2 is at SFO, you can follow these steps:

1. Move Cargo C1 from SFO to JFK:
    1. Step 1a: Load C1 onto P1 at SFO.
        1. Preconditions: C1 is at SFO, P1 is at SFO.
        2. Action: Load(C1, P1, SFO).

3. Effects: C1 is in P1, C1 is no longer at SFO.

- Step 1b: Fly P1 from SFO to JFK.
- Preconditions: P1 is at SFO.
- Action: Fly(P1, SFO, JFK).
- Effects: P1 is at JFK.
- Step 1c: Unload C1 at JFK.
- Preconditions: C1 is in P1, P1 is at JFK.
- Action: Unload(C1, P1, JFK).
- Effects: C1 is at JFK, C1 is no longer in P1.

2.Move Cargo C2 from JFK to SFO:

- Step 2a: Load C2 onto P2 at JFK.
    - Preconditions: C2 is at JFK, P2 is at JFK.
    - Action: Load(C2, P2, JFK).
    - Effects: C2 is in P2, C2 is no longer at JFK.
- Step 2b: Fly P2 from JFK to SFO.
    - Preconditions: P2 is at JFK.
    - Action: Fly(P2, JFK, SFO).
    - Effects: P2 is at SFO.
- Step 2c: Unload C2 at SFO.
- Preconditions: C2 is in P2, P2 is at SFO.
- Action: Unload(C2, P2, SFO).
- Effects: C2 is at SFO, C2 is no longer in P2.
- **Steps to Achieve the Goal:**
- To achieve the goal where the spare tire is on the axle, the following sequence of actions can be followed:
- **Remove the flat tire from the axle**:
- **Action**: Remove(Flat, Axle)
- **Preconditions**: The flat tire is on the axle.
- **Effects**: The flat tire is no longer on the axle and is now on the ground (At(Flat, Ground)).
- **Put the spare tire on the axle**:
- **Action**: PutOn(Spare, Axle)
- **Preconditions**: The spare tire is on the ground (At(Spare, Ground)) and the flat tire is no longer on the axle.
- **Effects**: The spare tire is now on the axle (At(Spare, Axle)), and it is no longer on the ground (¬ At(Spare, Ground)).

$Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(obj, loc),$
  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.

-

Summary of the Plan:

- **Remove(Flat, Axle)**: Move the flat tire off the axle to the ground.
- **PutOn(Spare, Axle)**: Place the spare tire on the axle.
- Once these actions are performed, the goal will be achieved, with the spare tire placed on the axle.

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

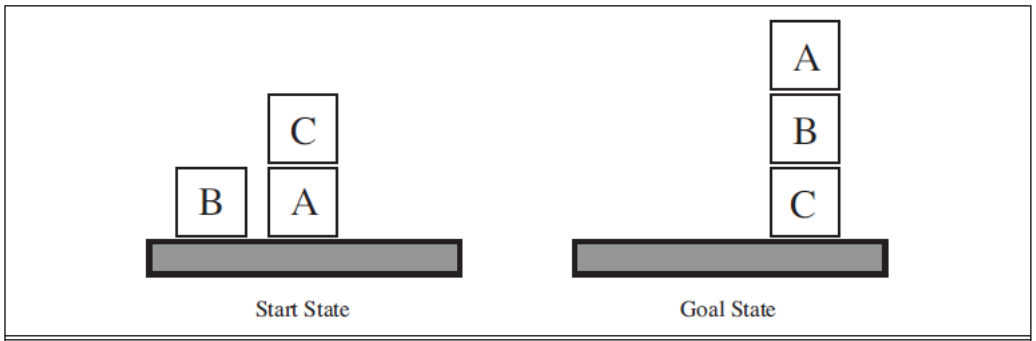


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

-
- **1. Move Block C from A to the Table**:

- **Action**: Move(C, A, Table)
- **Preconditions**: Block C is on Block A (On(C, A)), Block C is clear (Clear(C)), and Block A is clear.
- **Effects**: Block C will be on the table (On(C, Table)), and Block A will be clear (Clear(A)).
- **2. Move Block B from the Table to Block C**:
- **Action**: Move(B, Table, C)
- **Preconditions**: Block B is on the table (On(B, Table)), Block B is clear (Clear(B)), and Block C is clear.
- **Effects**: Block B will be on Block C (On(B, C)), and Block C will no longer be clear (¬ Clear(C)).
- **3. Move Block A from the Table to Block B**:
- **Action**: Move(A, Table, B)
- **Preconditions**: Block A is on the table (On(A, Table)), Block A is clear (Clear(A)), and Block B is clear.
- **Effects**: Block A will be on Block B (On(A, B)), and Block B will no longer be clear (¬ Clear(B)).
- **Summary of the Plan:**
- **Move(C, A, Table)**: Move Block C from Block A to the table.
- **Move(B, Table, C)**: Move Block B from the table to Block C.
- **Move(A, Table, B)**: Move Block A from the table to Block B.
- After completing these steps, the goal state will be reached with:
- Block A on Block B (On(A, B)).
- Block B on Block C (On(B, C)).

Complexity of Classical Planning

- **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.
- The first result is that both decision problems are decidable for classical planning.
- The proof follows from the fact that the number of states is finite. But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semidecidable: an algorithm exists that will terminate with the correct answer for any solvable problem, but may not terminate on unsolvable problems.
- The Bounded PlanSAT problem remains decidable even in the presence of function symbols.

Algorithms for Planning a State Space search

<u>Forward (Progression) state space search</u>

- In **forward state space search**, we start from the **initial state** and explore all possible actions leading to new states until we reach the **goal state**. The key idea is to start from the initial conditions and expand the search in the forward direction by applying actions, generating new states.

Steps in Forward Search:

1. **Start from the initial state** (the given configuration of the world).
2. **Generate possible successor states** by applying available actions to the current state.
3. **Repeat** the process for each successor state until a state satisfying the goal condition is found.
4. Use a **search strategy** (such as breadth-first search, depth-first search, or heuristic search) to decide which state to expand next.
5. For the previous block stacking problem, a forward search would start with the initial state where Block A is on the table, Block B is on the table, and Block C is on Block A. It would then generate successor states by moving blocks around until it reaches the goal state where:
6. Block A is on Block B.
7. Block B is on Block C.

Backward(regression) Relevant state space search

- **Backward state space search**, we start from the **goal state** and work backward toward the **initial state**. The idea is to try to reverse-engineer the solution by considering which actions could have led to the goal and then working backward to identify the previous states.
- **Steps in Backward Search:**

1. **Start from the goal state** (the desired configuration of the world).
2. **Identify possible predecessor states** by determining which actions could have resulted in the goal state.
3. **Repeat** the process for each predecessor state, working backward until the initial state is reached.
4. Like forward search, use a **search strategy** to decide which state to explore next.
5. For the block stacking problem, backward search would begin with the goal state (Block A on Block B, and Block B on Block C). It would then look at actions that could have resulted in this goal and work backward:
6. Determine that Block A must have been moved to Block B and Block B must have been moved to Block C.
7. Eventually, the search would backtrack to the initial state where Block A is on the table, Block B is on the table, and Block C is on Block A.

Heuristics Planning

- We look first at heuristics that add edges to the graph. For example, the **ignore preconditions heuristic** drops all preconditions from actions.
- Every action becomes applicable in every state, and any single goal fluent can be achieved in one step (if there is an applicable
- action—if not, the problem is impossible). This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because
- (1) some action may achieve multiple goals and

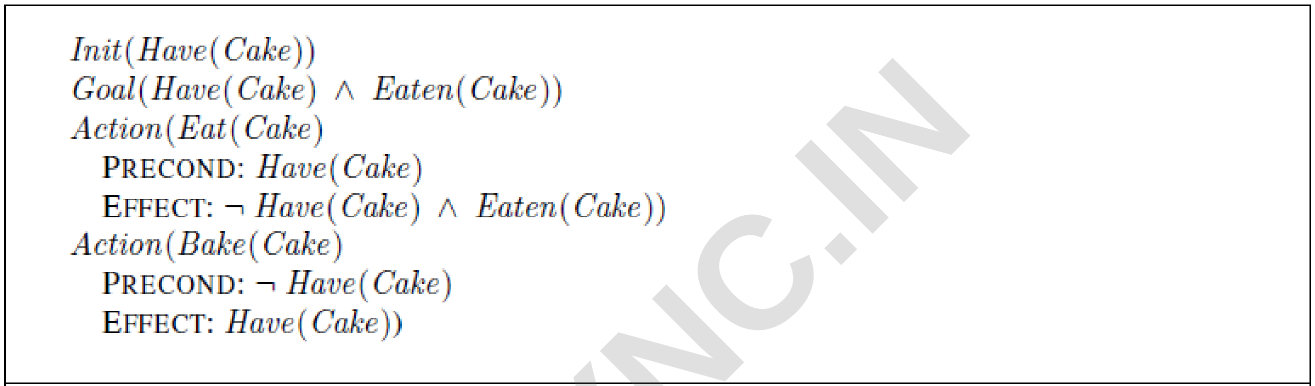
- (2) some actions may undo the effects of others.
- For many problems an accurate heuristic is obtained by considering
- (1) and ignoring
- (2). First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal. Then, we count the minimum number of actions required such that the union of those actions' effects satisfies the goal. This is an instance of the **set-cover problem**.
- There is one minor irritation: the set-cover problem is NP-hard.
- Fortunately, a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of log n of the true minimum covering, where n is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

- A key idea in defining heuristics is decomposition: dividing a problem into parts, solving each part independently, and then combining the parts.
- The subgoal independence assumption is that the cost of solving a conjunction of subgoals is approximated by the sum
- of the costs of solving each subgoal *independently*.
- The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each subgoal—for example, when an action in one subplan deletes a goal
- achieved by another subplan.
- It is pessimistic, and therefore inadmissible, when subplans contain redundant actions—for instance, two actions that could be replaced by a single action in the merged plan.
- It is clear that there is great potential for cutting down the search space by forming abstractions.
- The trick is choosing the right abstractions and using them in a way that makesthe total cost—defining an abstraction, doing an abstract search, and mapping the abstraction back to the original problem—less than the cost of solving the original problem.
- Pattern databases can be useful here.

PLANNING GRAPHS

- A special data structure called a planning graph can be used to give better heuristic estimates.
- These heuristics can be applied to any of the search techniques we have seen so far.
- Alternatively, we can search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.
- A planning problem asks if we can reach a goal state from the initial state.
- Suppose we are given a tree of all possible actions from the initial state to successor states, and their successors,
- and so on.
- If we indexed this tree appropriately, we could answer the planning question "can we reach state G from state S0" immediately, just by looking it up.
- Of course, the tree is of exponential size, so this approach is impractical.

- A planning graph is polynomialsize approximation to this tree that can be constructed quickly.
- The planning graph can't answer definitively whether G is reachable from S0, but it can *estimate* how many steps it takes to reach G.
- The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.
- A planning graph is a directed graph organized into **levels**: first a level S0 for the initial state, consisting of nodes representing each fluent that holds in S0; then a level A0 consisting of nodes for each ground action that might be applicable in S0; then alternating levels Si followed by Ai; until we reach a termination condition.



$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
  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.

Planning Problem Analysis:

1. **Initial state**: You have the cake (Have(Cake)).
2. **Goal**: You need to both have the cake and have eaten the cake.

To achieve the goal, you can proceed with the following steps:

- **Step 1**: You already have the cake (from the initial state), so you can **Eat(Cake)**.
  - **Precondition for Eat(Cake)**: You need to have the cake, which you do.
  - **Effect**: After eating, you no longer have the cake (¬Have(Cake)) but have eaten it (Eaten(Cake)).
- **Step 2**: Now, you no longer have the cake (¬Have(Cake)), so you can't directly eat the cake again. However, since **Eaten(Cake)** is part of the goal and is already achieved, you are done. The goal **Have(Cake) ∧ Eaten(Cake)** is satisfied because **Eaten(Cake)** is true.

Thus, the sequence of actions you would take to achieve the goal is:

1. **Eat(Cake)**, which leads to **Eaten(Cake)** and **¬Have(Cake)**.

This is the simplest solution based on the given actions and conditions.

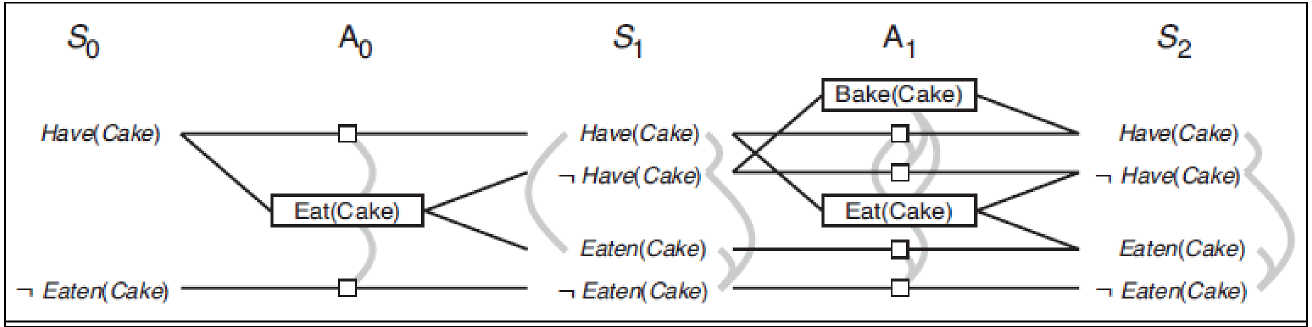
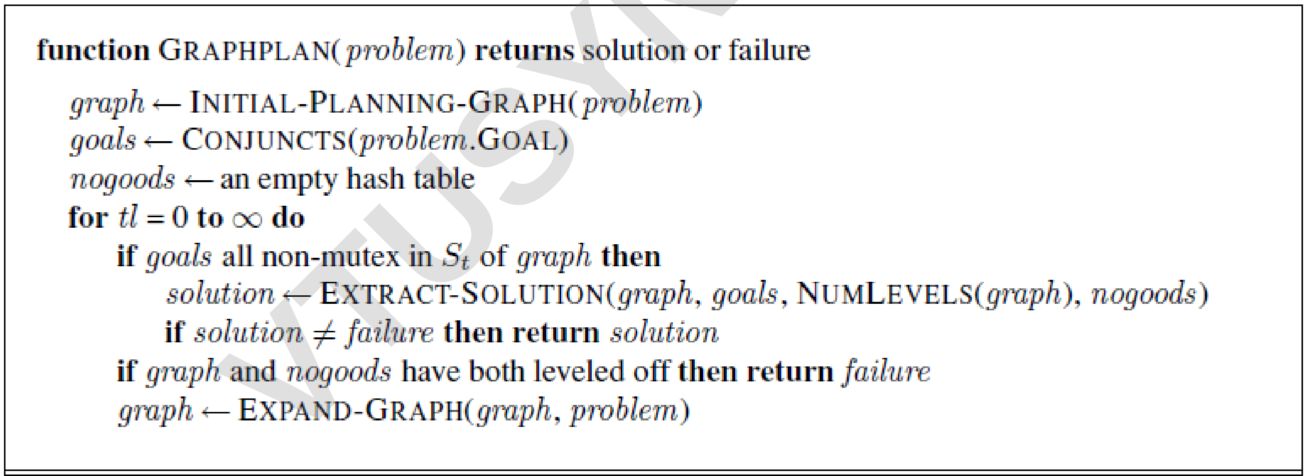


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because 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.

- Figure 10.7 shows a simple planning problem, and Figure 10.8 shows its planning graph.
- Each action at level Ai is connected to its preconditions at Si and its effects at Si+1.
- So a literal appears because an action caused it, but we also want to say that a literal canpersist if no action negates it. This is represented by a **persistence action** .
- For every literal C, we add to the problem a persistence action with precondition C and effect C.
- Level A0 in Figure 10.8 shows one "real" action, Eat (Cake), along with two persistence actions drawn as small square boxes.
- Level A0 contains all the actions that *could* occur in state S0, but just as important it records conflicts between actions that would prevent them from occurring together.
- The gray lines in Figure 10.8 indicate **mutual exclusion** (or **mutex**) links.
- For example, Eat (Cake) is mutually exclusive with the persistence of either Have(Cake) or ¬ Eaten(Cake).
- Level S1 contains all the literals that could result from picking any subset of the actions in A0, as well as mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions. For example, Have(Cake) and Eaten(Cake) are mutex:
- depending on the choice of actions in A0, either, but not both, could be the result.
- In other words, S1 represents a belief state: a set of possible states.
- The members of this set are all subsets of the literals such that there is no mutex link between any members of the subset.
- We continue in this way, alternating between state level Si and action level Ai until we reach a point where two consecutive levels are identical. At this point, we say that the graph has **leveled off**.
- The graph in Figure 10.8 levels off at S2.
- What we end up with is a structure where every Ai level contains all the actions that are applicable in Si, along with constraints saying that two actions cannot both be executed at the same level.
- Every Si level contains all the literals that could result from any possible choice of actions in Ai−1, along with constraints saying which pairs of literals are not possible.

- It is important to note that the process of constructing the planning graph does *not* require choosing among actions, which would entail combinatorial search. Instead, it just records the impossibility of certain choices using mutex links.
- A mutex relation holds between two *actions* at a given level if any of the following three conditions holds:
- ***Inconsistent effects****:* one action negates an effect of the other.
- For example, Eat (Cake) and the persistence of Have(Cake) have inconsistent effects because they disagree on the effect Have(Cake).
- ***Interference****:* one of the effects of one action is the negation of a precondition of the other. For example Eat (Cake interferes with the persistence of Have(Cake) by negatingits precondition.
- ***Competing needs****:* one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, Bake(Cake) and Eat (Cake) are mutex because they compete on the value of the Have(Cake) precondition.

THE GRAPHPLAN ALGORITHM



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

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

- The purpose of GRAPHPLAN is to generate a sequence of actions that lead from the initial state to a goal state, while respecting the constraints and conditions of the problem.
- The algorithm builds a **planning graph** incrementally and uses a backward search to extract the solution.
- **1.Initial Planning Graph**
- The first step is to construct an initial planning graph from the given **problem**. This graph will encode both the actions and propositions (conditions) that are relevant to the problem. The planning graph will be expanded in subsequent steps.
- **2.Extracting the Goal**

- This line extracts the conjuncts of the goal.
- Goals are often a conjunction of several conditions (e.g., Have(Cake) ∧ Eaten(Cake)).
- We separate them so we can handle each individual goal condition separately.
- **3.Hash Table for no goods**
- The **nogoods** table is used to store sets of conditions that are impossible to achieve. If a particular set of conditions (or actions) has been determined to be unsolvable, it's stored in the **nogoods** hash table to avoid repeating unnecessary work.
- **4.Main Loop**
- This begins an infinite loop that will continue expanding the planning graph until a solution is found or it's determined that no solution exists.

5. Checking for Non-Mutex Goals

Here, the algorithm checks if all the goal conditions are non-mutually exclusive (non-mutex) in the state (St) of the planning graph at the current level (tl). A mutex is a mutual exclusion constraint that indicates two actions or propositions cannot be true simultaneously. If the goals are all non-mutex, we proceed to the next step.

6.Extracting the Solution

If all the goals are achievable (non-mutex), the algorithm attempts to **extract the solution** from the planning graph. The function **EXTRACT-SOLUTION** will look for the sequence of actions that satisfy the goals. If a valid solution is found, it returns the sequence of actions.

7. Terminating Condition

If the planning graph and **nogoods** hash table have "leveled off," meaning no further expansion is possible, the algorithm terminates and returns **failure** because no solution can be found.

8.Expanding the Planning Graph

If no solution has been found yet, the algorithm expands the planning graph to the next level. This involves adding new layers of actions and propositions to the graph based on the previous layers.

- **Planning Graph:** A layered structure that encodes the relationships between actions and propositions at different time steps.
- **Mutex:** Constraints that indicate which actions or propositions cannot coexist at the same level in the graph.
- **No-Goods:** A table that helps avoid searching for impossible plans by storing sets of conditions that cannot be satisfied.
- **Backward Search:** Once the graph is expanded, the algorithm searches backward from the goals to find the sequence of actions that can achieve them.

- GRAPHPLAN is efficient because it constructs a planning graph that captures the necessary dependencies and constraints, allowing it to search for solutions systematically while avoiding redundant computations.

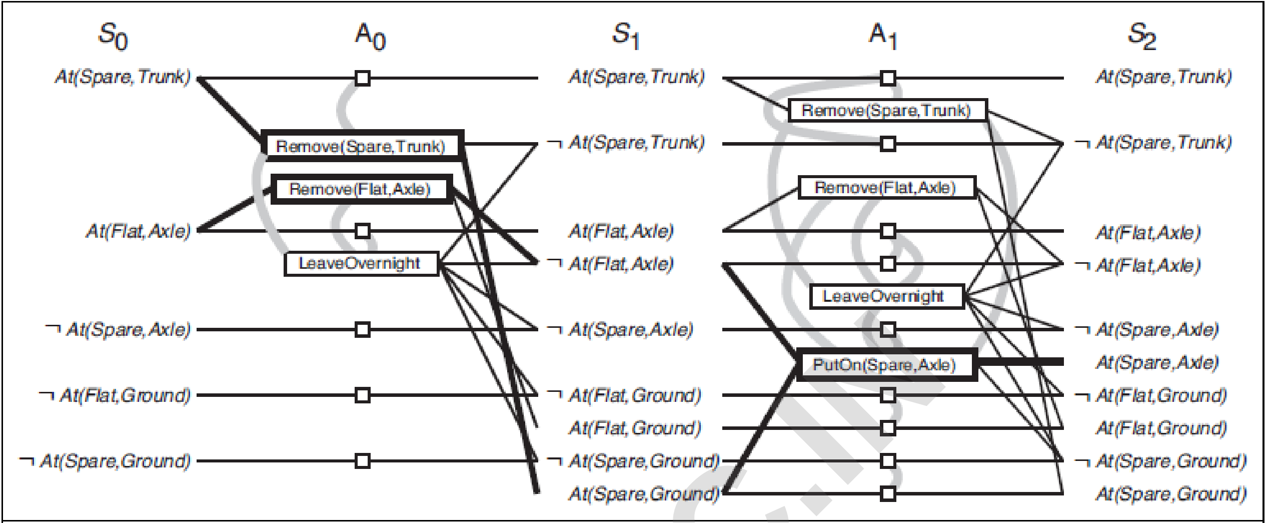


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

- The first line of GRAPHPLAN initializes the planning graph to a one-level (S0) graph representing the initial state.
- The positive fluents from the problem description's initial state are shown, as are the relevant negative fluents.
- Not shown are the unchanging positive literals (such as Tire(Spare)) and the irrelevant negative literals.
- The goal At(Spare, Axle) is not present in S0, so we need not call EXTRACT-SOLUTION— we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds into A0 the three actions whose preconditions exist at level S0 (i.e., all the actions except PutOn(Spare, Axle)), along with persistence actions for all the literals in S0. The effects of the actions are added at level S1. EXPAND-GRAPH then looks for mutex relations and adds them to the graph.
- At(Spare, Axle) is still not present in S1, so again we do not call EXTRACT-SOLUTION.
- We call EXPAND-GRAPH again, adding A1 and S1 and giving us the planning graph shown in Figure 10.10.
- Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:
- ***Inconsistent effects***: Remove(Spare, Trunk) is mutex with LeaveOvernight because one has the effect At(Spare, Ground) and the other has its negation.
- ***Interference***: Remove(Flat, Axle) is mutex with LeaveOvernight because one has the precondition

At(Flat, Axle) and the other has its negation as an effect.

- ***Competing needs***: PutOn(Spare, Axle) is mutex with Remove(Flat, Axle) because one has At(Flat, Axle) as a precondition and the other has its negation.
- ***Inconsistent support***: At(Spare, Axle) is mutex with At(Flat, Axle) in S2 because the only way of achieving At(Spare, Axle) is by PutOn(Spare, Axle), and that is mutex with the persistence action that is the only way of achieving At(Flat, Axle).
- Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.
- We define this search problem as follows:
- The initial state is the last level of the planning graph, Sn, along with the set of goals from the planning problem.
- The actions available in a state at level Si are to select any conflict-free subset of the actions in Ai−1 whose effects cover the goals in the state.
- The resulting state has level Si−1 and has as its set of goals the preconditions for the selected set of actions. By "conflict free," we mean a set of actions such that no two of them are mutex and no two of their preconditions are mutex.
- The goal is to reach a state at level S0 such that all the goals are satisfied.
- The cost of each action is 1.

For any set of goals, we proceed in the following order:
1. Pick first the literal with the highest level cost.
2. To achieve that literal, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

Termination of the GraphPlan

1. ***Literals increase monotonically***: Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; once a literal shows up, persistence actions cause it to stay forever.
2. ***Actions increase monotonically:*** Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of the monotonic increase of literals; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus so will the action.
3. ***Mutexes decrease monotonically:*** If two actions are mutex at a given level Ai, then they will also be mutex for all *previous* levels at which they both appear. The same holds for mutexes between literals. It might not always appear that way in the figures, because the figures have a simplification: they display neither literals that cannot hold at level Si nor actions that cannot be executed at level Ai. We can see that "mutexes decrease monotonically" is true if you consider that these invisible literals and actions are mutex with everything.

4. No-goods decrease monotonically: If a set of goals is not achievable at a given level, then they are not achievable in any previous level. The proof is by contradiction: if they were achievable at some previous level, then we could just add persistence actions to make them achievable at a subsequent level.
+