

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028



A T M E
College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN



LECTURE NOTES

COURSE: MULTIMEDIA SYSTEM AND DESIGN

COURSE CODE: BCS515B

SEMESTER: V

COURSE COORDINATOR: Mrs. DIVYA N

(ACADEMIC YEAR 2025-26)

INSTITUTIONAL VISION AND MISSION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT VISION AND MISSION

Vision

To be a global leader in Computer Science and Design Engineering striving for design excellence, cultural awareness, a profound commitment to environmental stewardship, and societal responsibility.

Mission

M1: To establish a technology-enabled experiential learning environment, prioritizing and cultivating problem-solving and design thinking skills among students.

M2: To foster collaboration with industries, research and development organizations, jointly addressing socially relevant challenges in Computer Science with a core emphasis on design.

Program Educational Objective (PEO's)

- Demonstrate the proficiency and adaptability needed to seamlessly integrate with the latest advancements and changes in Computer Science and Design.
- Our graduates will possess the adaptability to explore various career paths, such as Research & Development, Manufacturing, and Entrepreneurship with independent design firms, Design Services, and Consultancies.
- Our students will be well-prepared to excel in collaborative, multi-disciplinary environments, showcasing their versatility and skills.

Program Specific Outcomes (PSO's)

- To develop stand-alone, embedded, and web-based solutions with easy-to-operate interfaces using software engineering practices and contemporary programming languages.
- Design and develop computer-based systems in various areas of Multimedia, Graphics data visualization and computer vision.

MODULE - 5

Inference in First Order Logic:**5.1 Backward Chaining**

Logical inference algorithms use the backward chaining approach. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof. We describe the basic algorithm, and then we describe how it is used in logic programming, which is the most widely used form of automated reasoning.

5.1.1 A backward-chaining algorithm

Figure 5.1 shows a backward-chaining algorithm for definite clauses. FOL-BC-ASK(KB, goal) will be proved if the knowledge base contains a clause of the form $\text{lhs} \Rightarrow \text{goal}$, where lhs (left-hand side) is a list of conjuncts. An atomic fact like `American(West)` is considered as a clause whose lhs is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query `Person(x)` could be proved with the substitution $\{x/\text{John}\}$ as well as with $\{x/\text{Richard}\}$. So we implement FOL-BC-ASK as a generator—a function that returns multiple times, each time giving one possible result.

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the rhs of the clause does indeed unify with the goal, proving every conjunct in the lhs, using FOL-BC-AND. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 5.2 is the proof tree for deriving `Criminal(West)`. Backward chaining, as we have written it, is clearly a depth-first search algorithm.

This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. We will discuss these problems and some potential solutions, but first we show how backward chaining is used in logic programming systems.

```

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 5.1: A simple backward-chaining algorithm for first-order knowledge bases.

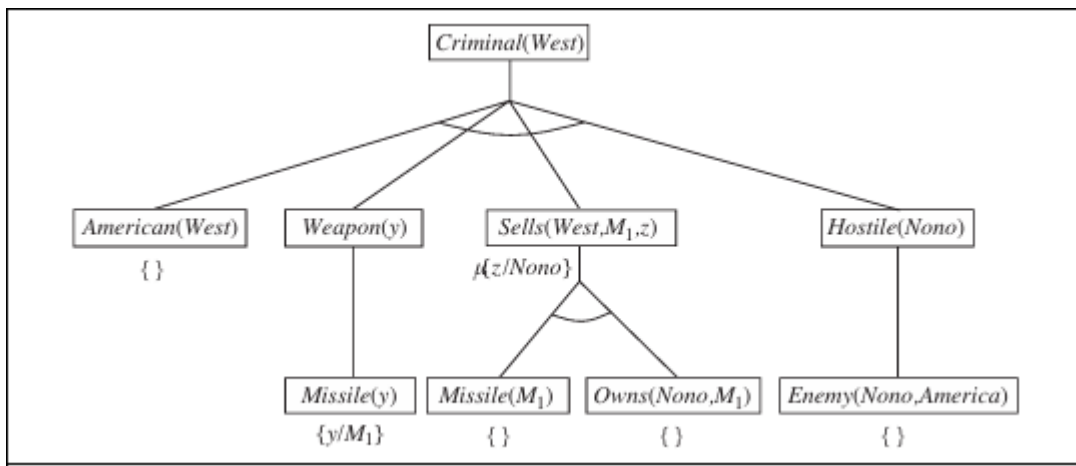


Figure 5.2: Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove Criminal (West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally Hostile(*z*), *z* is already bound to Nono.

5.1.2 Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative, that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is

summed up in Robert Kowalski's equation,

Algorithm = Logic + Control .

Prolog is the most widely used logic programming language. It is used primarily as a rapid prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains. Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. 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 $\text{append}(X,Y,Z)$, 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).

In English, 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. In most high-level languages we can write a similar recursive function that describes how to append two lists. The Prolog definition is actually much more powerful, however, because it describes a relation that holds among three arguments, rather than a function computed from two arguments. For example, we can ask the query $\text{append}(X,Y,[1,2])$: what two lists can be appended to give $[1,2]$? We get back the solutions

$X=[]$

$Y=[1,2];$

$X=[1] \ Y=[2];$

$X=[1,2] \ Y=[]$

The execution of Prolog programs is done through depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some aspects

of Prolog fall outside standard logical inference:

- Prolog uses the database semantics of rather than first-order semantics, and this is apparent in its treatment of equality and negation.
- There is a set of built-in functions for arithmetic. Literals using these function symbols are “proved” by executing code rather than doing further inference. For example, the goal “ X is $4+3$ ” succeeds with X bound to 7. On the other hand, the goal “ $X+Y$ ” fails, because the built-in functions do not do arbitrary equation solving.
- There are built-in predicates that have side effects when executed. These include input output predicates and the assert/retract predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results for example, if facts are asserted in a branch of the proof tree that eventually fails.
- The occur check is omitted from Prolog’s unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones.

5.1.3 Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled. Interpretation essentially amounts to running the FOL-BC-ASK algorithm from Figure 5.3, with the program as the knowledge base. We say “essentially” because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two **First, our implementation** had to explicitly manage the iteration over possible results generated by each of the subfunctions. Prolog interpreters have a global data structure a stack of choice points, to keep track of the multiple possibilities that we considered in FOL-BC-OR. This global stack is more efficient, and it makes debugging easier, because the debugger can move up and down the stack.

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. At any point in time, every variable in the

program either is unbound or is bound to some value. Together, these variables and values implicitly define the substitution for the current branch of the proof. Extending the path can only add new variable bindings, because an attempt to add a different binding for an already bound variable results in a failure of unification. 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. As each new variable is bound by UNIFY-VAR, the variable is pushed onto the trail. When a goal fails and it is time to back up to a previous choice point, each of the variables is unbound as it is removed from the trail.

Even the most efficient Prolog interpreters require several thousand machine instructions per inference step because of the cost of index lookup, unification, and building the recursive call stack. In effect, the interpreter always behaves as if it has never seen the program before; for example, it has to find clauses that match the goal. 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. Prolog basically generates a miniature theorem prove for each different predicate, thereby eliminating much of the overhead of interpretation. It is also possible to open-code the unification routine for each different call, thereby avoiding explicit analysis of term structure. (For details of open-coded unification, see Warren et al. (1977).)

For example, the definition of the Append predicate can be compiled into the code shown in Figure 5.3. Several points are worth mentioning:

- Rather than having to search the knowledge base for Append clauses, the clauses become a procedure and the inferences are carried out simply by calling the procedure.
- As described earlier, the current variable bindings are kept on a trail. The first step of the procedure saves the current state of the trail, so that it can be restored by RESET-TRAIL if the first clause fails. This will undo any bindings generated by the first call to UNIFY.
- The trickiest part is the use of continuations to implement choice points. You can think of a continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds. It would not do just to return from a procedure like APPEND when the goal succeeds,

because it could succeed in several ways, and each of them has to be explored. The continuation argument solves this problem because it can be called each time the goal succeeds. In the APPEND code, if the first argument is empty and the second argument unifies with the third, then the APPEND predicate has succeeded. We then CALL the continuation, with the appropriate bindings on the trail, to do whatever should be done next. For example, if the call to APPEND were at the top level, the continuation would print the bindings of the variables.

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, called AND-parallelism, comes from the possibility of solving each conjunct in the body of an implication in 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.

```
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 5.3: Pseudo code 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.

5.1.4 Redundant inference and infinite loops

We now turn to the Achilles heel of Prolog: the mismatch between depth-first search and search trees that include repeated states and infinite paths. Consider the following logic program that decides if a path exists between two points on a directed graph:

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

A simple three-node graph, described by the facts `link(a,b)` and `link(b,c)`, is shown in Figure 5.4(a). With this program, the query `path(a,c)` generates the proof tree shown in Figure 5.5(a). On the other hand, if we put the two clauses in the order

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

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

Then Prolog follows the infinite path shown in Figure 5.4(b). Prolog is therefore incomplete as a theorem prover for definite clauses—even for Datalog programs, as this example shows—because, for some knowledge bases, it fails to prove sentences that are entailed.

Notice that forward chaining does not suffer from this problem: once `path(a,b)`, `path(b,c)`, and `path(a,c)` are inferred, forward chaining halts. Depth-first backward chaining also has problems with redundant computations. For example, when finding a path from A1 to J4 in Figure 5.5(b), Prolog performs 877 inferences, most of which involve finding all possible paths to nodes from which the goal is unreachable. The total amount of inference can be exponential in the number of ground facts that are generated. If we apply forward chaining instead, at most n^2 `path(X,Y)` facts can be generated linking n nodes. For the problem in Figure 5.5(b), only 62 inferences are needed.

Forward chaining on graph search problems is an example of dynamic programming, in which the solutions to subproblems are constructed incrementally from those of smaller subproblems and are cached to avoid recomputation. We can obtain a similar effect in a backward chaining system using memoization—that is, caching solutions to subgoals as they are found and then reusing those solutions when the subgoal recurs, rather than repeating the previous computation. This is the approach taken by tabled logic programming systems, which use efficient storage and retrieval mechanisms to perform memoization. Tabled logic programming combines the goal-directedness of backward chaining with the dynamic programming efficiency of forward chaining. It is also complete for Datalog knowledge bases, which means that the programmer need worry less about infinite loops.

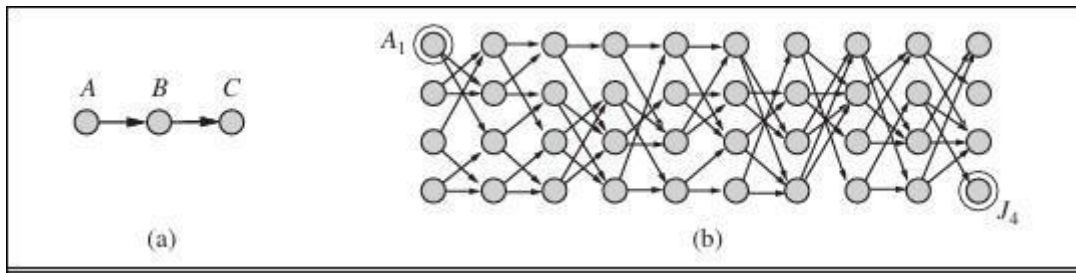


Figure 5.4: (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A1 to J4 requires 877 inferences.

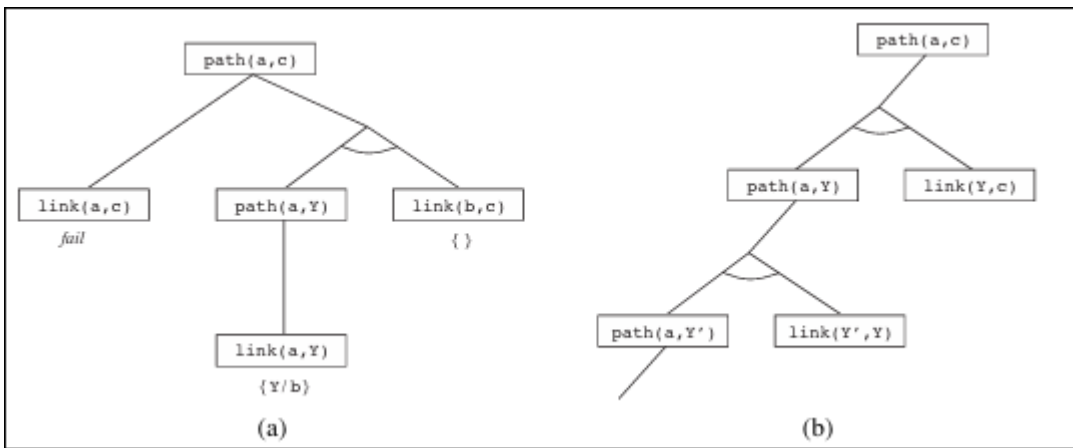


Figure 5.5: (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the “wrong” order.

5.1.5 Database semantics of Prolog

Prolog uses database semantics. The unique names assumption says that every Prolog constant and every ground term refers to a distinct object, and the closed world assumption says that the only sentences that are true are those that are entailed by the knowledge base. There is no way to assert that a sentence is false in Prolog. This makes Prolog less expressive than first-order logic, but it is part of what makes Prolog more efficient and more concise. Consider the following Prolog assertions about some course offerings:

Course(CS,101), Course(CS,102), Course(CS,106), Course(EI,101)

Under the unique names assumption, CS and EI are different (as are 101, 102, and 106), so this means that there are four distinct courses. Under the closed-world assumption there are no other courses, so there are exactly four courses. But if these were assertions in FOL rather than in Prolog, then all we could say is that there are somewhere between one and infinity courses. That's because the assertions (in FOL) do not deny the possibility that other

unmentioned courses are also offered, nor do they say that the courses mentioned are different from each other. If we wanted to translate Equation (9.11) into FOL, we would get this:

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

$$\vee (d=CS \wedge n=106) \vee (d=EE \wedge n=101).$$

This is called the completion of Equation. It expresses in FOL the idea that there are at most four courses. To express in FOL the idea that there are at least four courses, we need to write the completion of the equality predicate:

$$x = y \Leftrightarrow (x=CS \wedge y=CS) \vee (x=EE \wedge y=EE) \vee (x=101 \wedge y=101)$$

$$\vee (x=102 \wedge y=102) \vee (x=106 \wedge y=106).$$

The completion is useful for understanding database semantics, but for practical purposes, if your problem can be described with database semantics, it is more efficient to reason with Prolog or some other database semantics system, rather than translating into FOL and reasoning with a full FOL theorem prover.

5.1.6 Constraint logic programming

Backtracking enumerates the domains of the variables, it works only for finite domain CSPs. In Prolog terms, there must be a finite number of solutions for any goal with unbound variables. Infinite-domain CSPs—for example, with integer or real-valued variables—require quite different algorithms, such as bounds propagation or linear programming.

Consider the following example. We define $\text{triangle}(X,Y,Z)$ as a predicate that holds if the three arguments are numbers that satisfy the triangle inequality:

$$\text{triangle}(X,Y,Z) : X > 0, Y > 0, Z > 0, X + Y > Z, Y + Z > X, X + Z > Y.$$

If we ask Prolog the query $\text{triangle}(3,4,5)$, it succeeds. On the other hand, if we ask $\text{triangle}(3,4,Z)$, no solution will be found, because the subgoal $Z > 0$ cannot be handled by Prolog; we can't compare an unbound value to 0. 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. For example, the solution to the $\text{triangle}(3,4,Z)$ query is the constraint $7 > Z > 1$. Standard logic programs are just a special case of CLP in which the solution constraints must be equality constraints—that is, bindings.

5.2 RESOLUTION

The last of our three families of logical systems is based on resolution. We saw on page 250 that propositional resolution using refutation is a complete inference procedure for propositional logic.

5.2.1 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.⁶ Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$$\forall x \text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$$

becomes, in CNF,

$$\neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x, y, z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x).$$

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. In particular, the CNF sentence will be unsatisfiable just when the original sentence is unsatisfiable, so we have a basis for doing proofs by contradiction on the CNF sentences.

The procedure for conversion to CNF is similar to the propositional case, which we saw on page 253. 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

$$\forall x [\forall y \text{Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{Loves}(y, x)].$$

The steps are as follows:

- Eliminate implications:

$$\forall x [\neg \forall y \neg \text{Animal}(y) \vee \text{Loves}(x, y)] \vee [\exists y \text{Loves}(y, x)].$$

- Move $\neg$ -inwards: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$$\neg \forall x p \text{ becomes } \exists x \neg p$$

$$\neg \exists x p \text{ becomes } \forall x \neg p.$$

Our sentence goes through the following transformations:

$$\forall x [\exists y \neg(\neg \text{Animal}(y) \vee \text{Loves}(x, y))] \vee [\exists y \text{Loves}(y, x)].$$

$$\forall x [\exists y \neg \neg \text{Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists y \text{Loves}(y, x)].$$

$$\forall x [\exists y \text{Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists y \text{Loves}(y, x)].$$

- Standardize variables: For sentences like $(\exists x P(x)) \vee (\exists x Q(x))$ which use the same

variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$$\forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists z \text{ Loves}(z, x)] .$$

- **Skolemize:** Skolemization is the process of removing existential quantifiers by elimination. translate $\exists x P(x)$ into $P(A)$, where A is a new constant. However, we can't apply Existential Instantiation to our sentence above because it doesn't match the pattern $\exists v \alpha$; only parts of the sentence match the pattern. If we blindly apply the rule to the two matching parts we get

$$\forall x [\text{Animal}(A) \wedge \neg \text{Loves}(x, A)] \vee \text{Loves}(B, x) ,$$

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 :

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

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:

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

- **Distribute $\vee$ over $\wedge$:**

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

This step may also require flattening out nested conjunctions and disjunctions.

5.2.2 The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule given on page 253. 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

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

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

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

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

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

This rule is called the binary resolution rule because it resolves exactly two literals. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable. An alternative approach is to extend factoring— the removal of redundant literals— to the first-order case. Propositional factoring reduces two literals to one if they are identical; first-order factoring reduces two literals to one if they are unifiable. The unifier must be applied to the entire clause. The combination of binary resolution and factoring is complete.

5.2.3 Example proofs

Resolution proves that $\text{KB} \models \alpha$ by proving $\text{KB} \wedge \neg \alpha$ unsatisfiable, that is, by deriving the empty clause. The algorithmic approach is identical to the propositional case. Instead, we give two example proofs. The first is the crime example, The sentences in CNF are

$$\begin{array}{ll} \neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x, y, z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x) & \\ \neg \text{Missile}(x) \vee \neg \text{Owns}(\text{Nono}, x) \vee \text{Sells}(\text{West}, x, \text{Nono}) & \\ \neg \text{Enemy}(x, \text{America}) \vee \text{Hostile}(x) & \\ \neg \text{Missile}(x) \vee \text{Weapon}(x) & \\ \text{Owns}(\text{Nono}, M_1) & \text{Missile}(M_1) \\ \text{American}(\text{West}) & \text{Enemy}(\text{Nono}, \text{America}) . \end{array}$$

We also include the negated goal $\neg \text{Criminal}(\text{West})$. The resolution proof is shown in Figure 5.6. Notice the structure: single “spine” beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases.

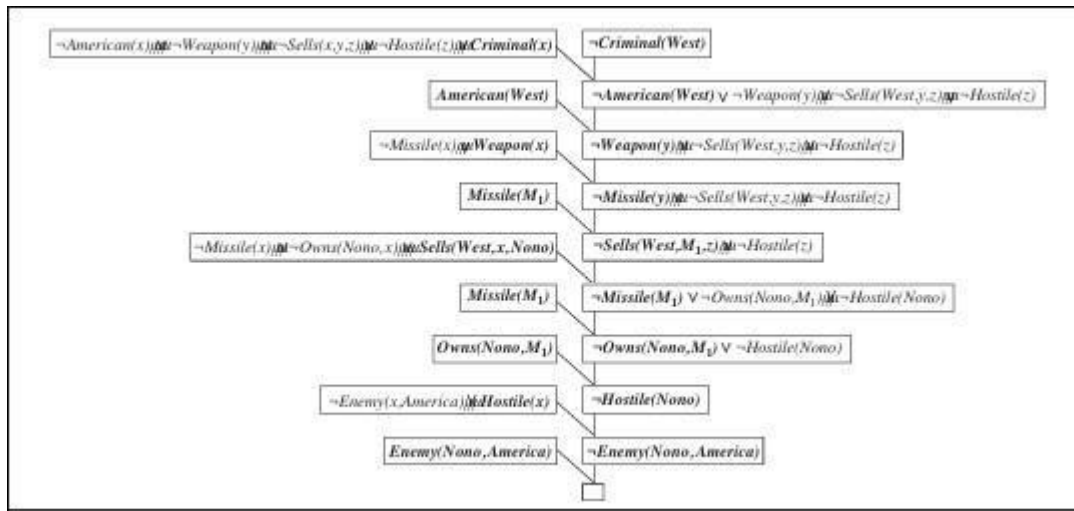


Figure 5.6: Are solution proof that West is a criminal. At each step, the literals that unify are in bold.

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is as follows:

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

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

- A1. $\text{Animal}(F(x)) \vee \text{Loves}(G(x), x)$
- A2. $\neg \text{Loves}(x, F(x)) \vee \text{Loves}(G(x), x)$
- B. $\neg \text{Loves}(y, x) \vee \neg \text{Animal}(z) \vee \neg \text{Kills}(x, z)$
- C. $\neg \text{Animal}(x) \vee \text{Loves}(\text{Jack}, x)$
- D. $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E. $\text{Cat}(\text{Tuna})$
- F. $\neg \text{Cat}(x) \vee \text{Animal}(x)$
- $\neg G$. $\neg \text{Kills}(\text{Curiosity}, \text{Tuna})$

The resolution proof that Curiosity killed the cat is given in Figure 5.7. In English, the proof could be paraphrased as follows:

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

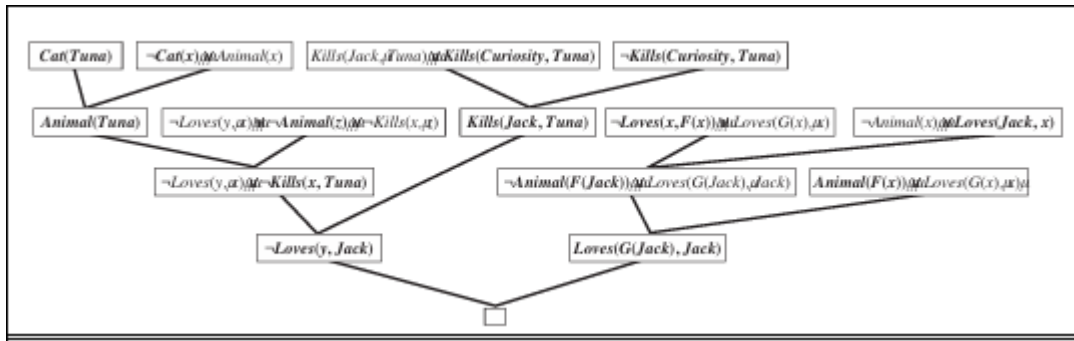


Figure 5.7: A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $\text{Loves}(G(\text{Jack}), \text{Jack})$. Notice also in the upper right, the unification of $\text{Loves}(x, F(x))$ and $\text{Loves}(\text{Jack}, x)$ can only succeed after the variables have been standardized apart.

The proof answers the question “Did Curiosity kill the cat?” but often we want to pose more general questions, such as “Who killed the cat?” Resolution can do this, but it takes a little more work to obtain the answer. The goal is $\exists w \text{ Kills}(w, \text{Tuna})$, which, when negated, becomes $\neg \text{Kills}(w, \text{Tuna})$ in CNF. Repeating the proof in Figure 5.7, with the new negated goal, we obtain a similar proof tree, but with the substitution $\{w/\text{Curiosity}\}$ in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof.

Unfortunately, resolution can produce nonconstructive proofs for existential goals. For example, $\neg \text{Kills}(w, \text{Tuna})$ resolves with $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$ to give $\text{Kills}(\text{Jack}, \text{Tuna})$, which resolves again with $\neg \text{Kills}(w, \text{Tuna})$ to yield the empty clause. Notice that w has two different bindings in this proof; resolution is telling us that, yes, someone killed Tuna—either Jack or Curiosity. This is no great surprise! One solution is to restrict the allowed resolution steps so that the query variables can be bound only once in a given proof; then we need to be able to backtrack over the possible bindings. Another solution is to add a special answer literal to the negated goal, which becomes $\neg \text{Kills}(w, \text{Tuna}) \vee \text{Answer}(w)$.

Now, the resolution process generates an answer whenever a clause is generated containing just a single answer literal. For the proof in Figure 5.7, this is $\text{Answer}(\text{Curiosity})$. The nonconstructive proof would generate the clause $\text{Answer}(\text{Curiosity}) \vee \text{Answer}(\text{Jack})$, which does not constitute an answer.

5.2.4 Completeness of resolution

The resolution is refutation-complete, which means that if a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences. Hence, it can be used to find all answers to a given question, $Q(x)$, by proving that $KB \wedge \neg Q(x)$ is unsatisfiable. We take it as given that any sentence in first-order logic (without equality) can be rewritten as a set of clauses in CNF. This can be proved by induction on the form of the sentence, using atomic sentences as the base case (Davis and Putnam, 1960). Our goal therefore is to prove the following: if S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction.

Our proof sketch follows Robinson's original proof with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof (Figure 5.8) is as follows:

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:

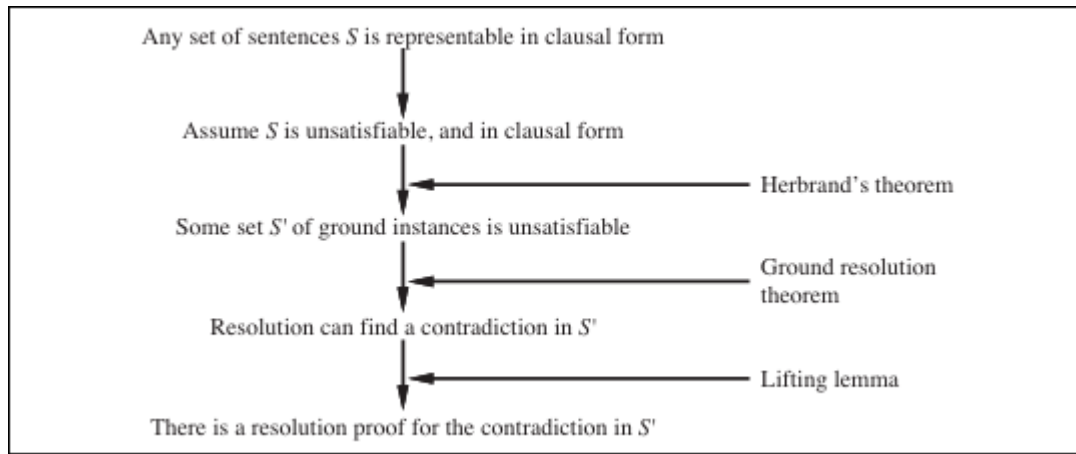


Figure 5.8: Structure of a completeness proof for resolution.

- **Herbrand universe:** If S is a set of clauses, then H_S , 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 $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, then H_S is the following infinite set of ground terms:

$$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \dots\}.$$

- **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 $H_S(S)$. For example, if S contains solely the clause just given, then $H_S(S)$ is the infinite set of clauses

$$\begin{aligned} &\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B), \\ &\neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B), \\ &\neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B), \\ &\neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \dots\} \end{aligned}$$

These definitions allow us to state a form of Herbrand's theorem (Herbrand, 1930): If a set S of clauses is unsatisfiable, then there exists a finite subset of $H_S(S)$ that is also unsatisfiable. Let S be this finite subset of ground sentences. Now, we can appeal to the ground resolution theorem (page 255) to show that the resolution closure $RC(S)$ contains the empty clause. That is, running propositional resolution to completion on S will derive a contradiction. Now that we have established that there is always a resolution proof involving some finite subset of the Herbrand base of S , the next step is to show that there is a resolution proof using the clauses of S itself, which are not necessarily ground clauses. We start by considering a single application of the resolution rule. Robinson stated this lemma:

Let C_1 and C_2 be two clauses with no shared variables, and let C'_1 and C'_2 be ground instances of C_1 and C_2 . If C' is a resolvent of C'_1 and C'_2 , then there exists a clause C such that (1) C is a resolvent of C_1 and C_2 and (2) C' is a ground instance of C .

This is called a lifting lemma, because it lifts a proof step from ground clauses up to general first-order clauses. In order to prove his basic lifting lemma, Robinson had to invent unification and derive all of the properties of most general unifiers. Rather than repeat the proof here, we simply illustrate the lemma:

$$\begin{aligned}C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\C'_1 &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\C'_2 &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B) .\end{aligned}$$

We see that indeed C is a ground instance of C' . In general, for C_1 and C_2 to have any resolvents, they must be constructed by first applying to C_1 and C_2 the most general unifier of a pair of complementary literals in C_1 and C_2 . From the lifting lemma, it is easy to derive a similar statement about any sequence of applications of the resolution rule:

For any clause C' in the resolution closure of S' there is a clause C in the resolution closure of S such that C' is a ground instance of C and the derivation of C is the same length as the derivation of C' .

From this fact, it follows that if the empty clause appears in the resolution closure of S , it must also appear in the resolution closure of S' . This is because the empty clause cannot be a ground instance of any other clause. To recap: we have shown that if S is unsatisfiable, then there is a finite derivation of the empty clause using the resolution rule. The lifting of theorem proving from ground clauses to first-order clauses provides a vast increase in power. This increase comes from the fact that the first-order proof need instantiate variables only as far as necessary for the proof, whereas the ground-clause methods were required to examine a huge number of arbitrary instantiations.

5.2.5 Equality

None of the inference methods described so far in this chapter handle an assertion of the form $x=y$. Three distinct approaches can be taken. The first approach is to axiomatize equality—to write down sentences about the equality relation in the knowledge base. We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute equals for equals in any predicate or function. So we need three basic axioms, and then one

for each predicate and function:

$$\begin{aligned} &\forall x \ x=x \\ &\forall x, y \ x=y \Rightarrow y=x \\ &\forall x, y, z \ x=y \wedge y=z \Rightarrow x=z \\ \\ &\forall x, y \ x=y \Rightarrow (P_1(x) \Leftrightarrow P_1(y)) \\ &\forall x, y \ x=y \Rightarrow (P_2(x) \Leftrightarrow P_2(y)) \\ &\vdots \\ &\forall w, x, y, z \ w=y \wedge x=z \Rightarrow (F_1(w, x) = F_1(y, z)) \\ &\forall w, x, y, z \ w=y \wedge x=z \Rightarrow (F_2(w, x) = F_2(y, z)) \\ &\vdots \end{aligned}$$

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations. However, these axioms will generate a lot of conclusions, most of them not helpful to a proof. So there has been a search for more efficient ways of handling equality. One alternative is to add inference rules rather than axioms. The simplest rule, demodulation, takes a unit clause $x=y$ and some clause α that contains the term x , and yields a new clause formed by substituting y for x within α . It works if the term within α unifies with x ; it need not be exactly equal to x .

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

we can conclude by demodulation

$Birthdate(PaternalGrandfather(Bella), 1926)$.

More formally, we have

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

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

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

The rule can also be extended to handle non-unit clauses in which an equality literal appears:

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

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

For example, from

$P(F(x, B), x) \vee Q(x)$ and $F(A, y) = y \vee R(y)$

we have $\theta = UNIFY(F(A, y), F(x, B)) = \{x/A, y/B\}$, and we can conclude by paramodulation the sentence

$P(B, A) \vee Q(A) \vee R(B)$.

A third approach handles equality reasoning entirely within an extended unification algorithm. That is, terms are unifiable if they are provably equal under some substitution, where “provably” allows for equality reasoning. For example, the terms $1+2$ and $2+1$ normally are not unifiable, but a unification algorithm that knows that $x + y = y + x$ could unify them with the empty substitution.

5.2.6 Resolution strategies

we examine strategies that help find proofs efficiently.

- A) **Unit preference:** This strategy prefers to do resolutions where one of the sentences is a single literal (also known as a unit clause). The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses. Resolving a unit sentence (such as P) with any other sentence (such as $\neg P \vee \neg Q \vee R$) always yields a clause (in this case, $\neg Q \vee R$) that is shorter than the other clause. When the unit preference strategy was first tried for propositional inference in 1964, it led to a dramatic speedup, making it feasible to prove theorems that could not be handled without the preference. **Unit resolution** is a restricted form of resolution in which every resolution step must involve a unit clause. Unit resolution is incomplete in

general, but complete for Horn clauses. Unit resolution proofs on Horn clauses resemble forward chaining.

- B) **Set of support:** Preferences that try certain resolutions first are helpful, but in general it is more effective to try to eliminate some potential resolutions altogether. For example, we can insist that every resolution step involve at least one element of a special set of clauses—the set of support. The resolvent is then added into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced dramatically. We have to be careful with this approach because a bad choice for the set of support will make the algorithm incomplete. However, if we choose the set of support S so that the remainder of the sentences are jointly satisfiable, then set-of-support resolution is complete. For example, one can use the negated query as the set of support, on the assumption that the original knowledge base is consistent.
- C) **Input resolution:** In this strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The proof in Figure 9.11 on page 348 uses only input resolutions and has the characteristic shape of a single “spine” with single sentences combining onto the spine. Clearly, the space of proof trees of this shape is smaller than the space of all proof graphs. In Horn knowledge bases, Modus Ponens is a kind of input resolution strategy, because it combines an implication from the original KB with some other sentences. Thus, it is no surprise that input resolution is complete for knowledge bases that are in Horn form, but incomplete in the general case. The linear resolution strategy is a slight generalization that allows P and Q to be resolved together either if P is in the original KB or if P is an ancestor of Q in the proof tree. Linear resolution is complete.
- D) **Subsumption:** The subsumption method eliminates all sentences that are subsumed by (that is, more specific than) an existing sentence in the KB. For example, if $P(x)$ is in the KB, then there is no sense in adding $P(A)$ and even less sense in adding $P(A) \vee Q(B)$. Subsumption helps keep the KB small and thus helps keep the search space small.

5.3 Classical Planning

The hybrid propositional logical agent can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. But it relies on ground (variable-free) propositional inference, which means that it may be swamped when there are many actions and states. For example, in the wumpus world, the

simple action of moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations. In response to this, planning researchers have settled on a factored representation— one in which a state of the world is represented by a collection of variables. 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 select simple version and alter its syntax to be consistent with the rest of the book. 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 fluent that are ground, functionless atoms. For example, $\text{Poor} \wedge \text{Unknown}$ might represent the state of a hapless agent, and a state in a package delivery problem might be $\text{At}(\text{Truck1}, \text{Melbourne}) \wedge \text{At}(\text{Truck2}, \text{Sydney})$. Database semantics is used: the closed-world assumption means that any fluent that are not mentioned are false, and the unique names assumption means that *Truck1* and *Truck2* are distinct. The following fluents are not allowed in a state: $\text{At}(x,y)$ (because it is non-ground), $\neg \text{Poor}$ (because it is a negation), and $\text{At}(\text{Father}(\text{Fred}), \text{Sydney})$ (because it uses a function symbol). 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. The set semantics is sometimes easier to deal with.

Actions are described by a set of action schemas that implicitly define the $\text{ACTIONS}(s)$ and $\text{RESULT}(s,a)$ functions needed to do a problem-solving search, to say what changes and what stays the same as the result of the action. Classical planning concentrates on problems where most actions leave most things unchanged. Think of a world consisting of a bunch of objects on a flat surface. The action of nudging an object causes that object to change its location by a vector Δ . A concise description of the action should mention only Δ ; it shouldn't have to mention all the objects that stay in place. PDDL does that by specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned.

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*) $\wedge$ *Plane*(*p*) $\wedge$ *Airport*(*from*) $\wedge$ *Airport*(*to*)
 EFFECT: $\neg$ *At*(*p*, *from*) $\wedge$ *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. Although we haven't said yet how the action schema converts into logical sentences, think of the variables as being universally quantified. We are free to choose whatever values we want to instantiate the variables. For example, here is one ground action that results from substituting values for all the variables:

Action(*Fly*(*P*₁, *SFO*, *JFK*),
 PRECOND: *At*(*P*₁, *SFO*) $\wedge$ *Plane*(*P*₁) $\wedge$ *Airport*(*SFO*) $\wedge$ *Airport*(*JFK*)
 EFFECT: $\neg$ *At*(*P*₁, *SFO*) $\wedge$ *At*(*P*₁, *JFK*)

The precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences). The precondition defines the states in which the action can be executed, and the effect defines the result of executing the action. An action *a* can be executed in state *s* if *s* entails the precondition of *a*. Entailment can also be expressed with the set semantics: $s \models q$ iff every positive literal in *q* is in *s* and every negated literal in *q* is not. In formal notation we say

$$(a \in \text{ACTIONS}(s)) \Leftrightarrow s \models \text{PRECOND}(a),$$

where any variables in *a* are universally quantified. For example,

$$\forall p, \text{from}, \text{to} \quad (\text{Fly}(p, \text{from}, \text{to}) \in \text{ACTIONS}(s)) \Leftrightarrow s \models (\text{At}(p, \text{from}) \wedge \text{Plane}(p) \wedge \text{Airport}(\text{from}) \wedge \text{Airport}(\text{to}))$$

We say that action *a* is applicable in state *s* if the preconditions are satisfied by *s*. When an action schema *a* contains variables, it may have multiple applicable instantiations. For example, with the initial state defined in Figure 10.1, the *Fly* action can be instantiated as *Fly*(*P*₁, *SFO*, *JFK*) or as *Fly*(*P*₂, *JFK*, *SFO*), both of which are applicable in the initial state. If an action *a* has *v* variables, then, in a domain with *k* unique names of objects, it takes $O(vk)$ time in the worst case to find the applicable ground actions.

```

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 5.9: APDDL description of an air cargo transportation planning problem.

5.3.1 Example: Air cargo transport

Figure 5.9 shows an air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: Load, Unload, and Fly. The actions affect two predicates: *In*(*c*,*p*) means that cargo *c* is inside plane *p*, and *At*(*x*,*a*) means that object *x* (either plane or cargo) is at airport *a*. Note that some care must be taken to make sure the *At* predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane; the cargo only becomes *At* the new airport when it is unloaded. So *at* really means “available for use at a given location.” The following plan is a solution to the problem:

```

[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK),
 Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)] .

```

Finally, there is the problem of spurious actions such as *Fly*(*P*₁,*JFK*,*JFK*), which should be a no-op, but which has contradictory effects (according to the definition, the effect would include *At*(*P*₁,*JFK*) ∧ ¬*At*(*P*₁,*JFK*)). It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is to add inequality preconditions saying that the from and to airports must be different; see another example of this in Figure 5.10.

```

Init(On(A, Table) ∧ On(B, Table) ∧ On(C, A)
    ∧ Block(A) ∧ Block(B) ∧ Block(C) ∧ Clear(B) ∧ Clear(C))
Goal(On(A, B) ∧ On(B, C))
Action(Move(b, x, y),
    PRECOND: On(b, x) ∧ Clear(b) ∧ Clear(y) ∧ Block(b) ∧ Block(y) ∧
        (b ≠ x) ∧ (b ≠ y) ∧ (x ≠ y),
    EFFECT: On(b, y) ∧ Clear(x) ∧ ¬On(b, x) ∧ ¬Clear(y))
Action(MoveToTable(b, x),
    PRECOND: On(b, x) ∧ Clear(b) ∧ Block(b) ∧ (b ≠ x),
    EFFECT: On(b, Table) ∧ Clear(x) ∧ ¬On(b, x))

```

Figure 5.10: A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C,A), Move(B,Table,C), Move(A,Table,B)]$.

5.3.2 Example: The spare tire problem

Consider the problem of changing a flat tire (Figure 10.2). The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is an abstract one, with no sticky lug nuts or other complications. There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. A solution to the problem is $[Remove(Flat,Axle), Remove(Spare,Trunk), PutOn(Spare,Axle)]$.

```

Init(Tire(Flat) ∧ Tire(Spare) ∧ At(Flat, Axle) ∧ At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(obj, loc),
    PRECOND: At(obj, loc)
    EFFECT: ¬At(obj, loc) ∧ At(obj, Ground))
Action(PutOn(t, Axle),
    PRECOND: Tire(t) ∧ At(t, Ground) ∧ ¬At(Flat, Axle)
    EFFECT: ¬At(t, Ground) ∧ At(t, Axle))
Action(LeaveOvernight,
    PRECOND:
    EFFECT: ¬At(Spare, Ground) ∧ ¬At(Spare, Axle) ∧ ¬At(Spare, Trunk)
        ∧ ¬At(Flat, Ground) ∧ ¬At(Flat, Axle) ∧ ¬At(Flat, Trunk))

```

Figure 5.11: A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C,A), Move(B,Table,C), Move(A,Table,B)]$.

5.3.3 Example: The blocks world

One of the most famous planning domains is known as the blocks world. This domain consists of a set of cube-shaped blocks sitting on a table.² The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it

to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block B on C. From Figure 10.4. We use $On(b,x)$ to indicate that block b is on x, where x is either another block or the table. The action for moving block b from the top of x to the top of y will be $Move(b,x,y)$. Now, one of the preconditions on moving b is that no other block be on it. In first-order logic, this would be $\neg \exists x On(x,b)$ or, alternatively, $\forall x \neg On(x,b)$. Basic PDDL does not allow quantifiers, so instead we introduce a predicate $Clear(x)$ that is true when nothing is on x. The action $Move$ moves a block b from x to y if both b and y are clear. After the move is made, b is still clear but y is not. A first attempt at the $Move$ schema is

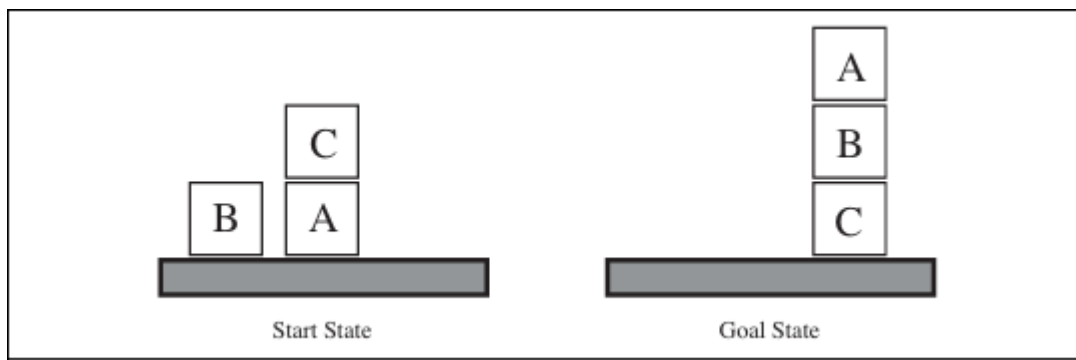


Figure 5.12: Diagram of the blocks-world problem.

```

Action(Move(b, x, y),
  PRECOND: On(b, x) ∧ Clear(b) ∧ Clear(y),
  EFFECT: On(b, y) ∧ Clear(x) ∧ ¬On(b, x) ∧ ¬Clear(y)) .

```

Unfortunately, this does not maintain $Clear$ properly when x or y is the table. When x is the Table, this action has the effect $Clear(Table)$, but the table should not become clear; and when $y=Table$, it has the precondition $Clear(Table)$, but the table does not have to be clear for us to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block b from x to the table:

```

Action(MoveToTable(b, x),
  PRECOND: On(b, x) ∧ Clear(b),
  EFFECT: On(b, Table) ∧ Clear(x) ∧ ¬On(b, x)) .

```

Second, we take the interpretation of $Clear(x)$ to be “there is a clear space on x to hold a block.” Under this interpretation, $Clear(Table)$ will always be true. The only problem is that nothing prevents the planner from using $Move(b,x,Table)$ instead of $MoveToTable(b,x)$. We

could live with this problem—it will lead to a larger-than-necessary search space, but will not lead to incorrect answers—or we could introduce the predicate *Block* and add $\text{Block}(b) \wedge \text{Block}(y)$ to the precondition of *Move*.

10.4 Algorithms for Planning as State-Space Search

The description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state. Figure 10.4 compares forward and backward searches.

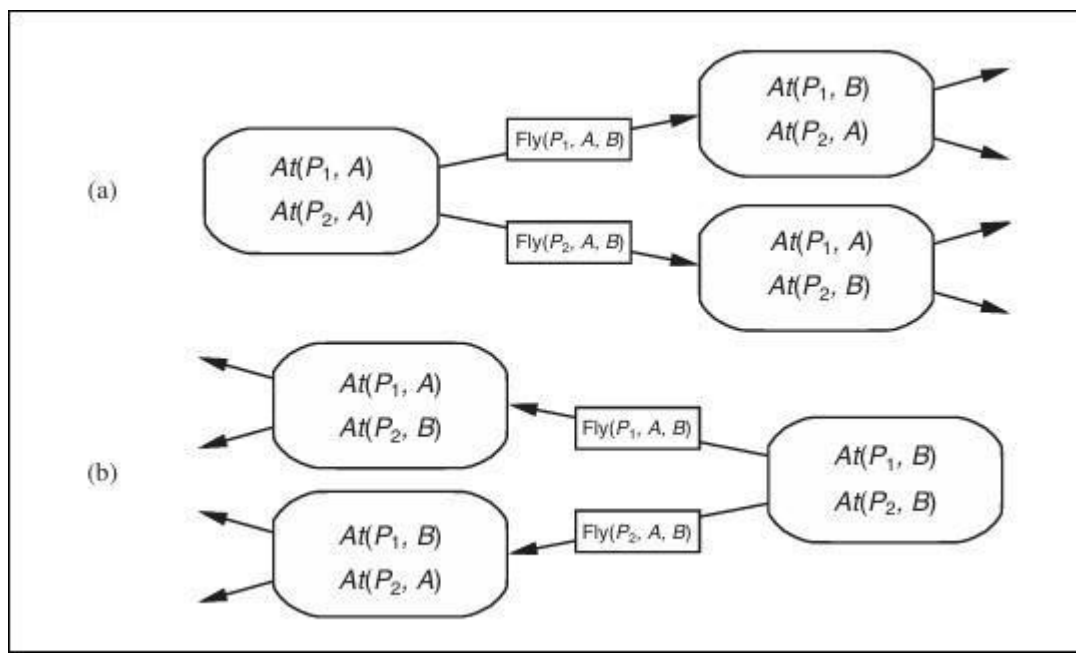


Figure 10.4: Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

10.4.1 Forward (progression) state-space search

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is an action schema $\text{Buy}(\text{isbn})$ with effect $\text{Own}(\text{isbn})$. ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal. **Second**, planning problems often have large state spaces. Consider an air cargo problem with 10 airports, where

each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 200041 nodes.

10.4.2 Backward (regression) relevant-states search

backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the n-queens problem because there is no easy way to describe the states that are one move away from the goal. Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it. Given a ground goal description g and a ground action a , the regression from g over a gives us a state description g defined by

$$g = (g - ADD(a)) \cup Precond(a) .$$

That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed. To get the full advantage of backward search, we need to deal with partially uninstantiated actions and states, not just ground ones. For example, suppose the goal is to deliver a specific piece of cargo to SFO: $At(C2, SFO)$. That suggests the action $Unload(C2, p, SFO)$:

Action($Unload(C_2, p', SFO)$,
PRECOND: $In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$
EFFECT: $At(C_2, SFO) \wedge \neg In(C_2, p')$.

10.4.3 Heuristics for planning

Neither forward nor backward search is efficient without a good heuristic function. 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. 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.

It is also possible to ignore only selected preconditions of actions. Consider the sliding block puzzle (8-puzzle or 15-puzzle) from Section 3.2. We could encode this as a planning problem involving tiles with a single schema Slide:

Action(Slide(t, s_1, s_2),
PRECOND: On(t, s_1) $\wedge$ Tile(t) $\wedge$ Blank(s_2) $\wedge$ Adjacent(s_1, s_2)
EFFECT: On(t, s_2) $\wedge$ Blank(s_1) $\wedge$ $\neg$ On(t, s_1) $\wedge$ $\neg$ Blank(s_2))

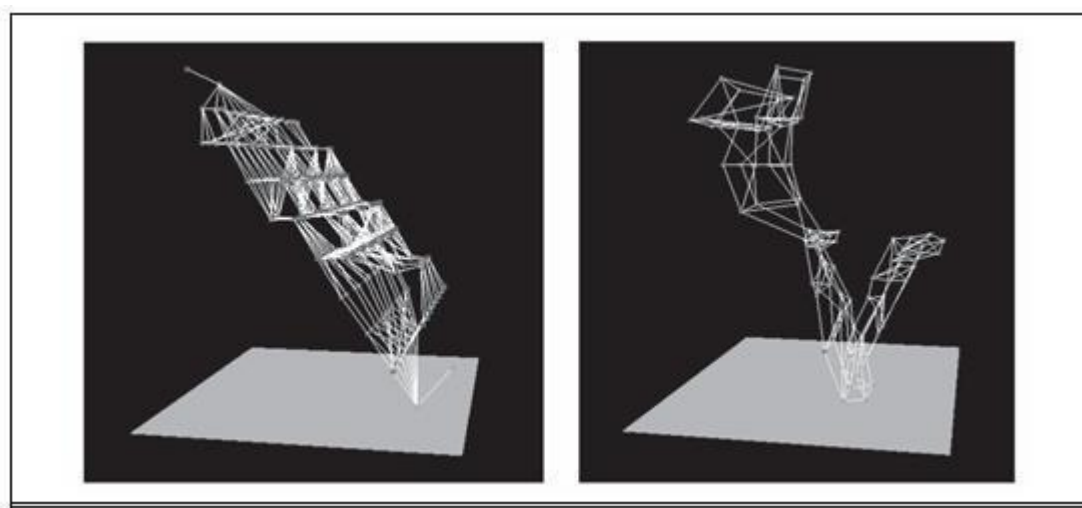


Figure 10.4.3.1: Two state spaces from planning problems with the ignore-delete-lists heuristic. The height above the bottom plane is the heuristic score of a state; states on the bottom plane are goals. There are no local minima, so search for the goal is straightforward. From Hoffmann (2005).

Figure 10.4.3.1, 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 hillclimbing search will easily find a solution to these problems (although it may not be an optimal solution).

10.5 Planning Graphs

All of the heuristics we have suggested can suffer from inaccuracies. This section shows how 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 graph is a directed graph organized into levels: first a level S_0 for the initial state, consisting of nodes representing each fluent that holds in S_0 ; then a level A_0 consisting of nodes for each ground action that might be applicable in S_0 ; then alternating levels S_i followed by A_i ; until we reach a termination condition.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned on page 368, it is straightforward to propositionalize a set of action schemas. Despite the resulting increase in the size of the problem description, planning graphs have proved to be effective tools for solving hard planning problems. Figure 10.7 shows a simple planning problem, and Figure 10.4.3.1 shows its planning graph. Each action at level A_i is connected to its preconditions at S_i and its effects at S_{i+1} .

So a literal appears because an action caused it, but we also want to say that a literal can persist if no action negates it. This is represented by a persistence action (sometimes called a no-op). For every literal C , we add to the problem a persistence action with precondition C and effect C . Level A_0 in Figure 10.8 shows one “real” action, $\text{Eat}(\text{Cake})$, along with two persistence actions drawn as small square boxes. Level A_0 contains all the actions that could occur in state S_0 , 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, $\text{Eat}(\text{Cake})$ is mutually exclusive with the persistence of either $\text{Have}(\text{Cake})$ or $\neg\text{Eaten}(\text{Cake})$. We shall see shortly how mutex links are computed.

Level S_1 contains all the literals that could result from picking any subset of the actions in A_0 , as well as mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions. For example, $\text{Have}(\text{Cake})$ and $\text{Eaten}(\text{Cake})$ are mutex: depending on the choice of actions in A_0 , either, but not both, could be the result. In other words, S_1 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 S_i and action level A_i 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 10.4.3.1 levels off at S_2 . What we end up with is a structure where every A_i level contains all the actions that are applicable in S_i , along with constraints saying that two actions cannot both be executed at the same level. Every S_i level contains all the literals that could result from any possible choice of actions in A_{i-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.

We now define mutex links for both actions and literals. 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, $\text{Eat}(\text{Cake})$ and the persistence of $\text{Have}(\text{Cake})$ have inconsistent effects because they disagree on the effect $\text{Have}(\text{Cake})$.
- Interference: one of the effects of one action is the negation of a precondition of the other. For example $\text{Eat}(\text{Cake})$ interferes with the persistence of $\text{Have}(\text{Cake})$ by negating its precondition.
- Competing needs: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, $\text{Bake}(\text{Cake})$ and $\text{Eat}(\text{Cake})$ are mutex because they compete on the value of the $\text{Have}(\text{Cake})$ precondition.

A mutex relation holds between two literals at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive.

This condition is called inconsistent support. For example, Have(Cake) and Eaten(Cake) are mutex in S1 because the only way of achieving Have(Cake), the persistence action, is mutex with the only way of achieving Eaten(Cake), namely Eat(Cake). In S2 the two literals are not mutex, because there are new ways of achieving them, such as Bake(Cake) and the persistence of Eaten(Cake), that are not mutex. A planning graph is polynomial in the size of the planning problem. For a planning problem with l literals and a actions, each S_i has no more than l nodes and l^2 mutex links, and each A_i has no more than $a + l$ nodes (including the no-ops), $(a + l)^2$ mutex links, and $2(a + l)$ precondition and effect links. Thus, an entire graph with n levels has a size of $O(n(a + l)^2)$. The time to build the graph has the same complexity.

10.5.1 Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. First, if any goal literal fails to appear in the final level of the graph, then the problem is unsolvable. Second, we can estimate the cost of achieving any goal literal g_i from state s as the level at which g_i first appears in the planning graph constructed from initial state s . We call this the level cost of g_i . In Figure 10.4.3.1, Have(Cake) has level cost 0 and Eaten(Cake) has level cost 1.

The estimate might not always be accurate, however, because planning graphs allow several actions at each level, whereas the heuristic counts just the level and not the number of actions. For this reason, it is common to use a serial planning graph for computing heuristics. A serial graph insists that only one action can actually occur at any given time step; this is done by adding mutex links between every pair of non persistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

To estimate the cost of a conjunction of goals, there are three simple approaches. The max-level heuristic simply takes the maximum level cost of any of the goals; this is admissible, but not necessarily accurate. The level sum heuristic, following the subgoal independence

assumption, returns the sum of the level costs of the goals; this can be inadmissible but works well in practice for problems that are largely decomposable.

For our problem, the level-sum heuristic estimate for the conjunctive goal $Have(Cake) \wedge Eaten(Cake)$ will be $0+1=1$, where as the correct answer is 2, achieved by the plan $[Eat(Cake), Bake(Cake)]$. That doesn't seem so bad. A more serious error is that if $Bake(Cake)$ were not in the set of actions, then the estimate would still be 1, when in fact the conjunctive goal would be impossible.

Finally, the set-level heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. This heuristic gives the correct values of 2 for our original problem and infinity for the problem without $Bake(Cake)$. It is admissible, it dominates the max-level heuristic, and it works extremely well on tasks in which there is a good deal of interaction among subplans. It is not perfect, of course; for example, it ignores interactions among three or more literals.

10.5.2 The GRAPHPLAN algorithm

This subsection shows how to extract a plan directly from the planning graph, rather than just using the graph to provide a heuristic. The GRAPHPLAN algorithm repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

```

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  $\infty$  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 10.5.2.1 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.

Let us now trace the operation of GRAPHPLAN on the spare tire problem from page 370. The graph is shown in Figure 10.10. The first line of GRAPHPLAN initializes the planning graph to a one-level (S_0) 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 $\text{Tire}(\text{Spare})$) and the irrelevant negative literals.

The goal $\text{At}(\text{Spare}, \text{Axle})$ is not present in S_0 , so we need not call EXTRACT-SOLUTION—we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds into A_0 the three actions whose preconditions exist at level S_0 (i.e., all the actions except $\text{PutOn}(\text{Spare}, \text{Axle})$), along with persistence actions for all the literals in S_0 . The effects of the actions are added at level S_1 . EXPAND-GRAPH then looks for mutex relations and adds them to the graph. $\text{At}(\text{Spare}, \text{Axle})$ is still not present in S_1 , so again we do not call EXTRACT-SOLUTION.

We call EXPAND-GRAPH again, adding A_1 and S_1 and giving us the planning graph shown in Figure 10.5.2.2, 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:

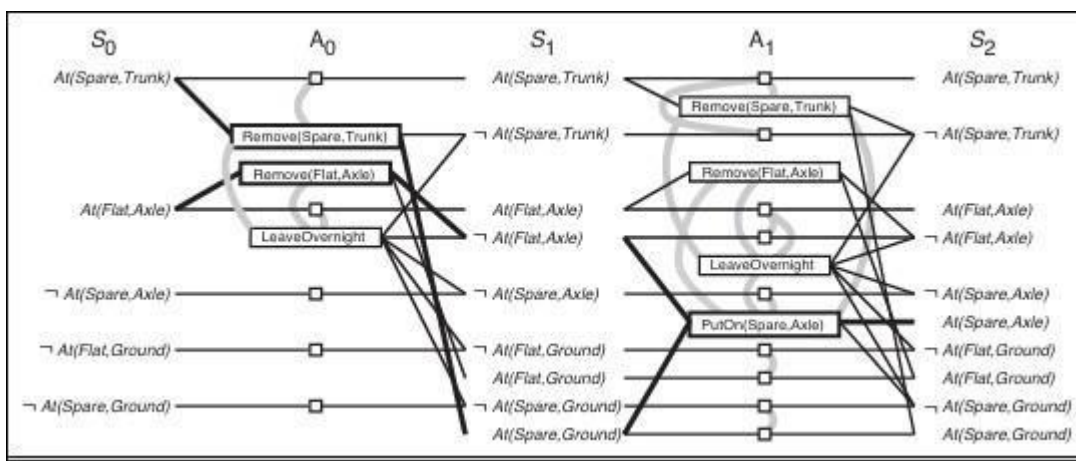


Figure 10.5.2.2 The planning graph for the spare tire problem after expansion to level S_2 .

Mutexlinks 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.

- 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.

This time, when we go back to the start of the loop, all the literals from the goal are present in S2, and none of them is mutex with any other. That means that a solution might exist, and EXTRACT-SOLUTION will try to find it. We can formulate EXTRACT-SOLUTION as a Boolean constraint satisfaction problem (CSP) where the variables are the actions at each level, the values for each variable are in or out of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.

Alternatively, we can define EXTRACT-SOLUTION as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals. We define this search problem as follows:

- The initial state is the last level of the planning graph, S_n , along with the set of goals from the planning problem.
- The actions available in a state at level S_i are to select any conflict-free subset of the actions in A_{i-1} whose effects cover the goals in the state. The resulting state has level S_{i-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 S_0 such that all the goals are satisfied.

- The cost of each action is 1. For this particular problem, we start at S_2 with the goal

At(Spare,Axle). The only choice we have for achieving the goal set is PutOn(Spare,Axle). That brings us to a search state at S1 with goals At(Spare,Ground) and $\neg$ At(Flat,Axle). The former can be achieved only by Remove(Spare,Trunk), and the latter by either Remove(Flat,Axle) or LeaveOvernight.

But LeaveOvernight is mutex with Remove(Spare,Trunk), so the only solution is to choose Remove(Spare,Trunk) and Remove(Flat,Axle). That brings us to a search state at S0 with the goals At(Spare,Trunk) and At(Flat,Axle). Both of these are present in the state, so we have a solution: the actions Remove(Spare, Trunk) and Remove(Flat, Axle) in level A0, followed by PutOn(Spare, Axle) in A1. In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the (level,goals) pair as a no-good, just as we did in constraining learning for CSPs (page 220). Whenever EXTRACT-SOLUTION is called again with the same level and goals, we can find the recorded no-good and immediately return failure rather than searching again. We see shortly that no-goods are also used in the termination test.

We know that planning is PSPACE-complete and that constructing the planning graph takes polynomial time, so it must be the case that solution extraction is intractable in the worst case. Therefore, we will need some heuristic guidance for choosing among actions during the backward search. One approach that works well in practice is a greedy algorithm based on the level cost of the literals. 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.

10.5.3 Termination of GRAPHPLAN

How long do we have to keep expanding after the graph has leveled off? If the function EXTRACT-SOLUTION fails to find a solution, then there must have been at least one set of goals that were not achievable and were marked as a no-good. So if it is possible that there might be fewer no-goods in the next level, then we should continue. As soon as the graph itself and the no-goods have both leveled off, with no solution found, we can terminate with failure because there is no possibility of a subsequent change that could add a solution. Now

all we have to do is prove that the graph and the no-goods will always level off. The key to this proof is that certain properties of planning graphs are monotonically increasing or decreasing.

“ X increases monotonically” means that the set of X s at level $i + 1$ is a superset (not necessarily proper) of the set at level i . The properties are as follows:

- **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.
- **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.
- **Mutexes decrease monotonically:** If two actions are mutex at a given level A_i , 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 S_i nor actions that cannot be executed at level A_i . We can see that “mutexes decrease monotonically” is true if you consider that these invisible literals and actions are mutex with everything. The proof can be handled by cases: if actions A and B are mutex at level A_i , it must be because of one of the three types of mutex. The first two, inconsistent effects and interference, are properties of the actions themselves, so if the actions are mutex at A_i , they will be mutex at every level. The third case, competing needs, depends on conditions at level S_i : that level must contain a precondition of A that is mutex with a precondition of B . Now, these two preconditions can be mutex if they are negations of each other (in which case they would be mutex in every level) or if all actions for achieving one are mutex with all actions for achieving the other. But we already know that the available actions are increasing monotonically, so, by induction, the mutexes must be decreasing.
- **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.