node is a literal from a rule premise. Variable bindings flow through the network and are filtered out when they fail to match a literal. If two literals in a rule share a variable—for example, \( \text{Sells}(x, y, 2) \land \text{Hostile}(z) \) in the crime example—then the bindings from each literal are filtered through an equality node. A variable binding reaching a node for an \( n \)-ary literal such as \( \text{Sells}(x, y, z) \) might have to wait for bindings for the other variables to be established before the process can continue. At any given point, the state of a rete network captures all the partial matches of the rules, avoiding a great deal of recomputation.

Rete networks, and various improvements thereof, have been a key component of so-called production systems, which were among the earliest forward-chaining systems in widespread use. The XCON system (originally called R1; McDermott, 1982) was built with a production-system architecture. XCON contained several thousand rules for designing configurations of computer components for customers of the Digital Equipment Corporation. It was one of the first clear commercial successes in the emerging field of expert systems. Many other similar systems have been built with the same underlying technology, which has been implemented in the general-purpose language OPS-5.

Production systems are also popular in cognitive architectures—that is, models of human reasoning—such as ACT (Anderson, 1983) and SOAR (Laird et al., 1987). In such systems, the "working memory" of the system models human short-term memory, and the productions are part of long-term memory. On each cycle of operation, productions are matched against the working memory of facts. A production whose conditions are satisfied can add or delete facts in working memory. In contrast to the typical situation in databases, production systems often have many rules and relatively few facts. With suitably optimized matching technology, some modern systems can operate in real time with tens of millions of rules.

Irrelevant facts

The final source of inefficiency in forward chaining appears to be intrinsic to the approach and also arises in the propositional context. Forward chaining makes all allowable inferences based on the known facts, even if they are irrelevant to the goal at hand. In our crime example, there were no rules capable of drawing irrelevant conclusions, so the lack of directedness was not a problem. In other cases (e.g., if many rules describe the eating habits of Americans and the prices of missiles), FOL-FC-ASK will generate many irrelevant conclusions.

One way to avoid drawing irrelevant conclusions is to use backward chaining, as described in Section 9.4. Another solution is to restrict forward chaining to a selected subset of rules, as in PL-FC-ENTAILs? (page 258). A third approach has emerged in the field of deductive databases, which are large-scale databases, like relational databases, but which use forward chaining as the standard inference tool rather than SQL queries. The idea is to rewrite the rule set, using information from the goal, so that only relevant variable bindings—those belonging to a so-called magic set—are considered during forward inference. For example, if the goal is \( \text{Criminal}(\text{West}) \), the rule that concludes \( \text{Criminal}(x) \) will be rewritten to include an extra conjunct that constrains the value of \( x \):

\[
\text{Magic}(x) \land \text{American}(x) \land \text{Weapon}(y) \land \text{Sells}(x, y) \land \text{Hostile}(z) \land \text{Criminal}(x).
\]

*The word production in production systems denotes a condition-action rule.*
The fact \textit{Magic( West)} is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process. The complete process for defining magic sets and rewriting the knowledge base is too complex to go into here, but the basic idea is to perform a sort of "generic" backward inference from the goal in order to work out which variable bindings need to be constrained. The magic sets approach can therefore be thought of as a kind of hybrid between forward inference and backward preprocessing.

9.4 BACKWARD CHAINING

The second major family of logical inference algorithms uses the backward chaining approach introduced in Section 7.5 for definite clauses. 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. We also see that backward chaining has some disadvantages compared with forward chaining, and we look at ways to overcome them. Finally, we look at the close connection between logic programming and constraint satisfaction problems.

9.4.1 A backward chaining algorithm

Figure 9.6 shows a backward-chaining algorithm for definite clauses. \textsc{FOL-BC-ASK(KB, goal)} will be proved if the knowledge base contains a clause of the form \( \text{lhs} = \text{goal} \), where \text{lhs} (left-hand side) is a list of conjuncts. An atomic fact like \textit{American( West)} is considered as a clause whose \text{lhs} is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query \textit{Person(x)} could be proved with the substitution \( \{ x/\text{John} \} \) as well as with \( \{ x/\text{Richard} \} \). So we implement \textsc{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 \text{lhs} of a clause must be proved. \textsc{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 \text{lhs}, using \textsc{FOL-B C-AND}. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 9.7 is the proof tree for deriving \textit{Criminal(West)} from sentences \((9.3)\) through \((9.10)\).

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.