**Unit III Artificial Intelligence CS6659**

**Forward Chaining**

1. The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made.
2. Here, we explain how the algorithm is applied to first-order definite clauses and how it can be implemented efficiently.
3. Definite clauses such as *Situation Response* are especially useful for systems that make inferences in response to newly arrived information.
4. Many systems can be defined this way, and reasoning with forward chaining can be much more efficient than resolution theorem proving.
5. Therefore it is often worthwhile to try to build a knowledge base using only definite clauses so that the cost of resolution can be avoided.

**First order definite clauses**

1. They are disjunctions of literals of which *exactly one is positive.*
2. *A* definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal.
3. The following are first-order definite clauses:

*King(x) Greedy(x)*  *Evil(x)*. *King( John)* .*Greedy(y)*.

1. Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.
2. Definite clauses are a suitable normal form for use with Generalized Modus Ponens.

**A simple forward-chaining algorithm**

***A*** conceptually straight forward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in *K B .*

**

The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.



**Efficient forward chaining**

1. Simple forward chaining algorithm is designed for ease of understanding rather than for efficiency of operation.
2. There are three possible sources of complexity.
3. First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive.
4. Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration.
5. Finally, the algorithm might generate many facts that are irrelevant to the goal. We will address each of these sources in turn.

**Matching rules against known facts**

The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. For example, suppose we want to apply the rule

Missile(x) -> Weapon(x)



1. Constraint graph for coloring the map of Australia (from Figure. (b) The map-coloring CSP expressed as a single definite clause.

**Incremental forward chaining**

1. When we showed how forward chaining works on the crime example, we cheated; in particular, we omitted some of the rule matching done by the algorithm shown in diagram.
2. For example, on the second iteration, the rule

Missile(x) Weapon(x)

matches against *Missile(M1)* (again), and of course the conclusion *Weapon(M1)*is already known so nothing happens.

1. Such redundant rule matching can be avoided if we make the following observation: ***Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration t*** - ***1.*** This is true because any inference that does not require a new fact from iteration *t* - 1 could have been done at iteration *t* - 1 already.

**Backward Chaining**

1. The second major family of logical inference algorithms uses the **backward chaining** approach.
2. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof.
3. 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.
4. We will also see that backward chaining has some disadvantages com- pared with forward chaining, and we look at ways to overcome them.
5. Finally, we will look at the close connection between logic programming and constraint satisfaction problems.

**A backward chaining algorithm**





Proof tree constructed by backward chaining to prove that West is a criminal.

1. Backward chaining, as we have written it, is clearly a depth-first search algorithm.
2. This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions).
3. It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness.

**Logic programming**

1. Logic programming is a technology that comes fairly close to embodying the declarative ideal: 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.
2. The ideal is summed up in Robert Kowalski's equation,

*Algorithm* = *Logic* +*Control*

1. **Prolog** is by far the most widely used logic programming language.
2. Its users number in the hundreds of thousands. 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).
3. Many expert systems have been written in Prolog for legal, medical, financial, and other domains.
4. 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.
5. Clauses are written with the head preceding the body; ": - " is used for left- implication, commas separate literals in the body, and a period marks the end of a sentence:



1. Prolog includes "syntactic sugar" for list notation and arithmetic. As an example, here is a Prolog program for append (X, Y, Z ) , which succeeds if list Z is the result of appending lists x and Y:



1. The execution of Prolog programs is done via 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:**

1. *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 "5 is X+Y" fails, because the built-in functions do not do arbitrary equation solving.
2. *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 some confusing effects- for example, if facts are asserted in a branch of the proof tree that eventually fails.
3. *Prolog allows a form of negation called* **negation as failure.** A negated goal n o t P is considered proved if the system fails to prove p. Thus, the sentence

*alive(X) :- n o t dead(X).* can be read as "Everyone is alive if not provably dead."

1. *Prolog has an equality operator, =, but it lacks the full power of logical equality.* An equality goal succeeds if the two terms are *unstable* and fails otherwise. So X+Y = 2+3 succeeds with x bound to *2* and Y bound to 3, but m o r n i n g s t a r = e v e n i n g s t a r fails. (In classical logic, the latter equality might or might not be true.) No facts or rules about equality can be asserted.
2. *The* ***occur check*** *is omitted from Prolog's unification algorithm.* This means that some unsound inferences can be made; these are seldom a problem except when using Prolog for mathematical theorem proving.

**Efficient implementation of logic programs**

1. The execution of a Prolog program can happen in two modes: interpreted and compiled.
2. Interpretation essentially amounts to running the FOL-BC-ASK algorithm, with the program as the knowledge base. We say "essentially," because Prolog interpreters contain a variety of improvements designed to maximize speed.
3. Pseudo code representing the result of compiling the Append predicate. The function *NEW-VARIABLE* returns a new variable, distinct from all other variables so far used.
4. The procedure *C A L L ( c o n t i n u a t i o n) c*ontinues execution with the specified continuation.



**Redundant inference and infinite loops**

1. Consider the following logic program that decides if a path exists between two points on a directed graph:



1. A simple three-node graph, described by the facts link (a, b) and link (b, c) , is shown in Figure 1(a). With this program, the query path ( a, c ) generates the proof tree shown in Figure 2(a). On the other hand, if we put the two clauses in the order





Figure 1 (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 Al to ***J4*** requires 877 inferences.



Figure 2 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

**Constraint logic programming**

1. In our discussion of forward chaining, we showed how constraint satisfaction problems (CSPs) can be encoded as definite clauses. Standard Prolog solves such problems in exactly the same way as the backtracking algorithm.
2. Because backtracking enumerates the domains of the variables, it works only for **finite domain** CSP’s. InProlog terms, there must be a finite number of solutions for any goal with unbound variables.
3. Infinite-domain CSPs- for example with integer or real-valued variables require quite different algorithms, such as bounds propagation or linear programming.
4. The following clause succeeds if three numbers satisfy the triangle inequality:



1. If we ask Prolog the query triangle (3 ,4, 5), this works fine. 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. The difficulty is that variables in Prolog must be in one of two states: unbound or bound to a particular term.

1. Binding a variable to a particular term can be viewed as an extreme form of constraint, namely an equality constraint. **Constraint logic programming** (CLP) allows variables to be *constrained* rather than *bound.* A solution to a constraint logic program is the most specific set of constraints on the query variables that can be derived from the knowledge base.