CS601PC: MACHINE LEARNING
III Year B.Tech. CSE II-Sem.
UNIT- IV

Genetic Algorithms – Motivation, Genetic algorithms, an illustrative example, hypothesis space search, genetic programming, models of evolution and learning, parallelizing genetic algorithms.

Learning Sets of Rules – Introduction, sequential covering algorithms, learning rule sets: summary, learning First-Order rules, learning sets of First-Order rules: FOIL, Induction as inverted deduction, inverting resolution.

Reinforcement Learning – Introduction, the learning task, *Q*–learning, non-deterministic rewards and actions, temporal difference learning, generalizing from examples, relationship to dynamic programming.

TEXT BOOKS:

1. Machine Learning–Tom M.Mitchell-MGH (Page Nos. 249 to 303 & 367 to 387)

GENETIC ALGORITHMS

MOTIVATION:

Genetic algorithms (GAS) provide a learning method motivated by an analogy to **biological evolution**.

The popularity of GAS is motivated by a number of factors including:

- Evolution is known to be a successful, robust method for adaptation within biological systems.
- GAS can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
- Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

GENETIC ALGORITHMS (GA)

GA($Fitness$, $Fitness\_threshold$, p, r, m)

Fitness: A function that assigns an evaluation score, given a hypothesis.

Fitness_threshold: A threshold specifying the termination criterion.

p: The number of hypotheses to be included in the population.

r: The fraction of the population to be replaced by Crossover at each step.

m: The mutation rate.

- *Initialize population:* $P \leftarrow$ Generate p hypotheses at random
- *Evaluate:* For each h in P, compute $Fitness(h)$
- While $[\max_h Fitness(h)] < Fitness\_threshold$ do

  Create a new generation, P_S:

  1. *Select:* Probabilistically select $(1 - r)p$ members of P to add to P_S. The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$$

  2. *Crossover:* Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P, according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s.
  3. *Mutate:* Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.
  4. *Update:* $P \leftarrow P_s$.
  5. *Evaluate:* for each h in P, compute $Fitness(h)$

- Return the hypothesis from P that has the highest fitness.

TABLE 9.1
A prototypical genetic algorithm. A population containing p hypotheses is maintained. On each iteration, the successor population P_S is formed by probabilistically selecting current hypotheses according to their fitness and by adding new hypotheses. New hypotheses are created by applying a crossover operator to pairs of most fit hypotheses and by creating single point mutations in the resulting generation of hypotheses. This process is iterated until sufficiently fit hypotheses are discovered. Typical crossover and mutation operators are defined in a subsequent table.

A **collection of hypotheses** called the current population. The hypotheses in the current population are evaluated relative to a given measure of **fitness**, with the most fit hypotheses **selected** probabilistically as seeds for producing the next generation. Members of the current population give rise to the next generation population by means of operations such as **random mutation and crossover**. GAS generate **successor hypotheses** by **repeatedly** mutating and recombining parts of the best currently known hypotheses.

The problem addressed by GAS is to search a space of candidate hypotheses **iteratively to identify the best hypothesis**. In GAS, the "best hypothesis" is defined as the one that optimizes a predefined numerical measure for the problem

at hand, called the **hypothesis Fitness**. On each iteration, all members of the population are **evaluated** according to the fitness function.

A **new population** is then generated by probabilistically **selecting the most fit individuals** from the current population. Some of these selected individuals are carried forward into the **next generation population**.

For example, if the learning task is the problem of approximating an unknown function given training examples of its input and output, then **fitness** could be defined as the **accuracy of the hypothesis** over this training data.

If the task is to learn a strategy for **playing chess**, **fitness** could be defined as the **number of games won** by the individual when playing against other individuals in the current population.

Hypotheses in GAS are often represented by **bit strings**, so that they can be easily manipulated by genetic operators such as **mutation and crossover**

I. REPRESENTING HYPOTHESES:

For example,

1) Consider the attribute **Outlook**, which can take on any of the three values **Sunny, Overcast, or Rain**.

One obvious way to represent a constraint on **Outlook** is to use a **bit string** of **length three**, in which each bit position corresponds to one of its three possible values. The string **010** represents the constraint that Outlook must take on the second of these values or Outlook = Overcast. Similarly, the string **011** represents the more general constraint that allows two possible values, or (Outlook = Overcast v Rain). **111** represents the most general possible constraint, indicating that we don't care which of its possible values the attribute takes on.

2) Consider a second attribute, **Wind** that can take on the value **Strong or Weak**. A rule precondition such as

$$(Outlook = Overcast \vee Rain) \wedge (Wind = Strong)$$

can then be represented by the following bit string of length five:

Outlook	*Wind*
011	10

3) The rule

IF $Wind = Strong$ THEN $PlayTennis = yes$

would be represented by the string

Outlook	*Wind*	*PlayTennis*
111	10	10

II. GENETIC OPERATORS

The crossover operator produces two new offspring from two parent strings, by **copying selected bits from each parent**.

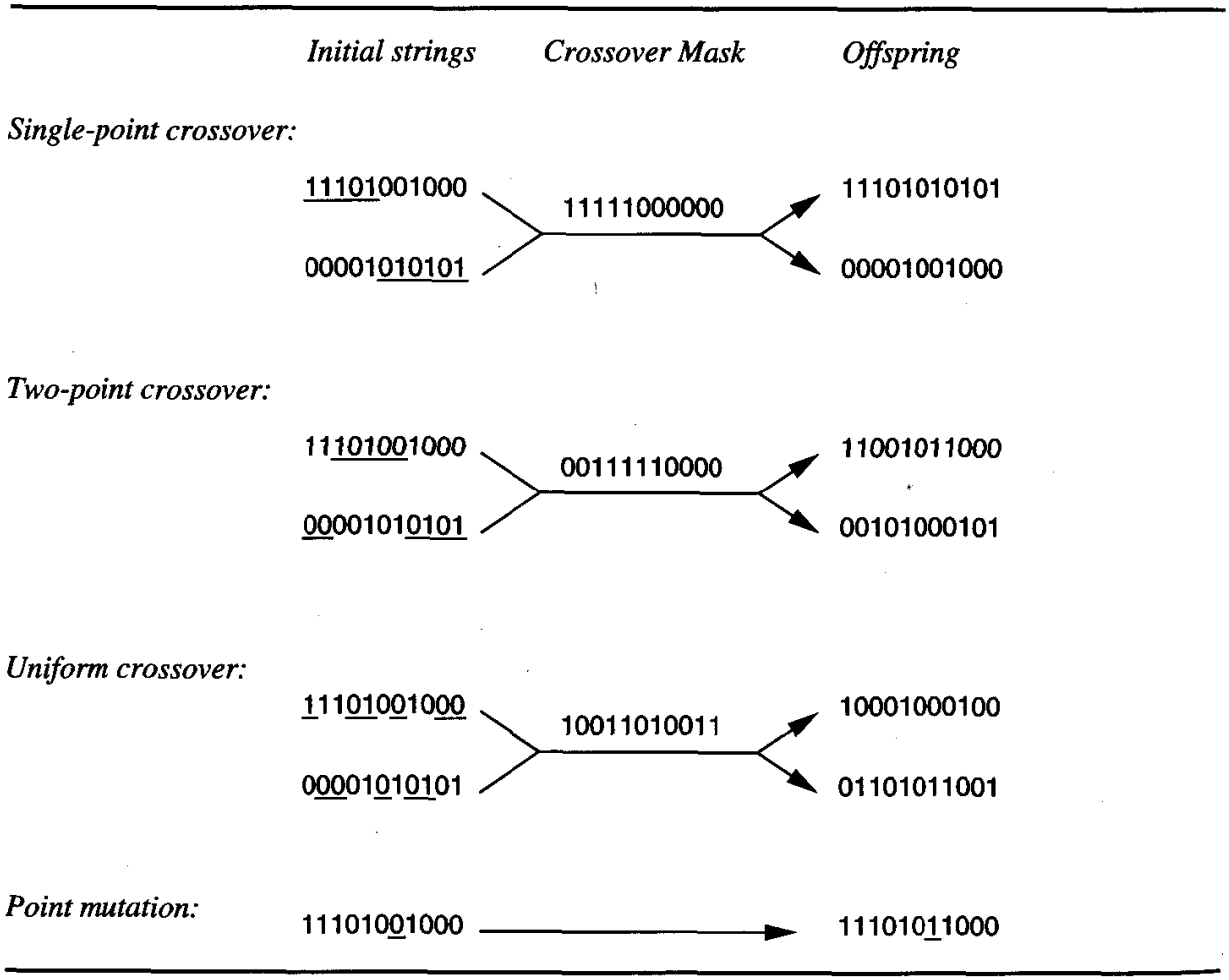


TABLE 9.2
Common operators for genetic algorithms. These operators form offspring of hypotheses represented by bit strings. The crossover operators create two descendants from two parents, using the crossover mask to determine which parent contributes which bits. Mutation creates a single descendant from a single parent by changing the value of a randomly chosen bit.

The bit at position i in each offspring is copied from the **bit at position i** in one of the two parents. The choice of which parent contributes the bit for position i is determined by an additional string called the **crossover mask**.

To illustrate, consider the **single-point crossover operator** at the top of Table 9.2. Consider the topmost of the two offspring in this case. This offspring takes its first five bits from the first parent and its remaining six bits from the second parent, because the **crossover mask 11111000000** specifies these choices for each of the bit positions. The second offspring uses the same crossover mask, but switches the roles of the two parents. Therefore, it contains the bits that were not used by the first offspring. In single-point crossover, the crossover mask is always constructed so that it begins with a string containing **n contiguous 1s**, followed by the **necessary number of 0s** to complete the string. This results in offspring in which the first n bits are contributed by one parent and the remaining bits by the second parent. Each time the single-point crossover operator is applied, the crossover point n is chosen at random, and the crossover mask is then created and applied.

In **two-point crossover**, offspring are created by substituting intermediate segments of one parent into the middle of the second parent string. Put another way, the crossover mask is a string beginning with n_o zeros, followed by a contiguous string of n_l ones, followed by the necessary number of zeros to complete the string. Each time the two-point crossover operator is applied, a mask is generated by randomly choosing the integers n_o and n_l. For instance, in the example shown in Table 9.2 the offspring are created using a mask for which $\mathbf{n_0 = 2}$ **and** $\mathbf{n_1 = 5}$. Again, the two offspring are created by switching the roles played by the two parents.

Uniform crossover combines bits sampled uniformly from the two parents, as illustrated in Table 9.2. In this case the crossover mask is generated as a random bit string with each bit chosen at random and independent of the others.

In particular, the **mutation operator** produces small random changes to the bit string by choosing a **single bit at random**, then changing its value. Mutation is often performed after crossover has been applied as in Table 9.1.

III. FITNESS FUNCTION AND SELECTION :

The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population. If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples.

In Table 9.1, the probability that a hypothesis will be selected is given by the ratio of its fitness to the fitness of other members of the current population as seen in Equation (9.1).

AN ILLUSTRATIVE EXAMPLE

To illustrate the use of **GAS for concept learning**, we briefly summarize the **GABIL system**. GABIL uses a GA to learn **boolean concepts** represented by a **disjunctive set of propositional rules**. The learning tasks in this study included both **artificial learning** tasks designed to explore the systems' **generalization accuracy** and the **real world problem of breast cancer diagnosis**.

As per table 9.1, the **parameter r**, which determines the fraction of the parent population replaced by crossover, was set to **0.6**. The **parameter m**, which determines the mutation rate, was set to **0.001**. These are typical settings for these parameters. The **population size p** was varied from **100 to 1000**, depending on the specific learning task.

The specific instantiation of the GA algorithm in GABIL can be summarized as follows:

- **Representation:** To illustrate, consider a hypothesis space in which rule **preconditions** are **conjunctions** of constraints over two boolean attributes, **a1 and a2**. The **rule postcondition** is described by a single bit that indicates the predicted value of the **target attribute c**. Thus, the hypothesis consisting of the two rules

  $$\text{IF } a_1 = T \wedge a_2 = F \text{ THEN } c = T; \qquad \text{IF } a_2 = T \text{ THEN } c = F$$

  would be represented by the string

a_1	a_2	c	a_1	a_2	c
10	01	1	11	10	0

- **Genetic operators:** The crossover operator that it uses is a fairly standard extension to the two-point **crossover operator**. GABIL uses the standard **mutation operator**. A single bit is chosen at random and replaced by its complement.

To perform a crossover operation on two parents, two crossover points are first chosen at random in the first parent string. Let d_1 (d_2) denote the distance from the leftmost (rightmost) of these two crossover points to the rule boundary immediately to its left. The crossover points in the second parent are now randomly chosen, subject to the constraint that they must have the same d_1 and d_2 value. For example, if the two parent strings are

	a_1	a_2	c	a_1	a_2	c
h_1 :	10	01	1	11	10	0

and

	a_1	a_2	c	a_1	a_2	c
h_2 :	01	11	0	10	01	0

and the crossover points chosen for the first parent are the points following bit positions 1 and 8,

	a_1	a_2	c	a_1	a_2	c
h_1 :	1[0	01	1	11	1]0	0

where "[" and "]" indicate crossover points.

The allowed pairs of crossover points for the second parent bit positions (1,3),

	a_1	a_2	c	a_1	a_2	c
h_2 :	0[1	1]1	0	10	01	0

Then the two resulting offspring will be

	a_1	a_2	c
h_3 :	11	10	0

and

	a_1	a_2	c	a_1	a_2	c	a_1	a_2	c
h_4 :	00	01	1	11	11	0	10	01	0

As this example illustrates, this **crossover operation** enables offspring to contain a different number of rules than their parents, while assuring that all bit strings generated in this fashion represent **well-defined rule sets**.

- **Fitness function:** The fitness of each hypothesized rule set is based on its **classification accuracy** over the training data. In particular, the function used
  to measure fitness is

$$Fitness(h) = (correct(h))^2$$

  where correct (h) is the percent of all training examples correctly classified by hypothesis h.

GABIL achieved an average generalization accuracy of 92.1 %, whereas the performance of the other systems ranged from 91.2 % to 96.6 % over a set of 12 synthetic problems.

HYPOTHESIS SPACE SEARCH

The schema theorem of Holland is based on the concept of schemas, or patterns **that describe sets of bit strings**. The approach is to **restrict** the kinds of individuals allowed to recombine to form offspring. For example, by allowing only the **most similar individuals to recombine**, we can encourage the formation of **clusters of similar individuals**.

The schema theorem reduces the problem of crowding. **Crowding** is a phenomenon in which some individual that is more highly fit than others in the population quickly reproduces, so that copies of this individual and very similar individuals take over a large fraction of the population. The negative impact of crowding is that it reduces the diversity of the population, thereby slowing further progress by the GA.

A schema is any string composed of 0s, 1s, and *'s. Each schema represents the set of bit strings containing the indicated 0s and 1s, with each **"*"** interpreted as a **"don't care"**. For example, the schema **0*10** represents the set of bit strings that includes exactly **0010 and 0110**. An individual bit string can be viewed as a representative of each of the different schemas that it matches. For example, the bit string 0010 can be thought of as a representative of 2^4 distinct schemas including 00**, 0*10, ****, etc. Similarly, a population of bit strings can be viewed in terms of the set of schemas that it represents and the number of individuals associated with each of these schema.

The schema theorem characterizes the evolution of the population within a GA in terms of the number of instances representing each schema. Let m(s, t) denote the number of instances of schema s in the population at time t (i.e., **during the t^{th} generation**). The schema theorem describes the expected value of **m(s, t + 1)** in terms of m(s, t).

The evolution of the population in the GA depends on the

selection step,

the recombination step, and

the mutation step.

Let,

f(h) - fitness of the individual bit string h

$\bar{f}$(t) - average fitness of all individuals in the population at time t.

n - total number of individuals in the population

$\hat{u}$ (s, t) - average fitness of instances of schema s in the population at time t

E[m(s, t + 1)] - Expected value of m(s, t + l)

The expected value of m(s, t + l) is calculated from the given the equation,

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$$

$$\Pr(h) = \frac{f(h)}{\sum_{i=1}^{n} f(h_i)}$$

$$= \frac{f(h)}{n\bar{f}(t)}$$

Therefore, the **expected number of instances of s** resulting from the n independent selection steps that create the entire new generation is just n times this probability.

$$E[m(s, t+1)] = \frac{\hat{u}(s, t)}{\bar{f}(t)} m(s, t)$$

The above Equation states that the expected number of instances of schema s at generation t + 1 is proportional to the average fitness $\hat{u}$ (s, t) of instances of this schema at time t , and inversely proportional to the average fitness $\bar{f}$(t) of all members of the population at time t. **This equation indicates that more fit schemas will grow in influence over time.**

GENETIC PROGRAMMING (GP)

Genetic programming (GP) is a form of evolutionary computation in which the individuals in the evolving population are **computer programs rather than bit strings.**

Programs manipulated by a GP are **typically represented by trees**. Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes.

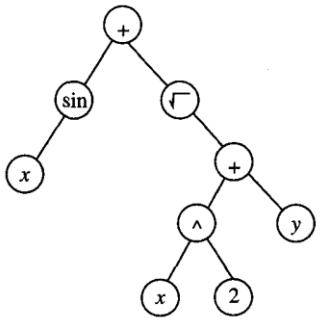


FIGURE 9.1
Program tree representation in genetic programming. Arbitrary programs are represented by their parse trees.

Figure 9.1 illustrates this tree representation for the function $\sin(x) + \sqrt{x^2 + y}$

The fitness of a given individual program in the population is typically determined by executing the program on a set of training data. On each iteration, it produces a new generation of individuals using selection, crossover, and mutation. Crossover operations are performed by replacing a randomly chosen subtree of one parent program by a subtree from the other parent program. Figure 9.2 illustrates a typical crossover operation

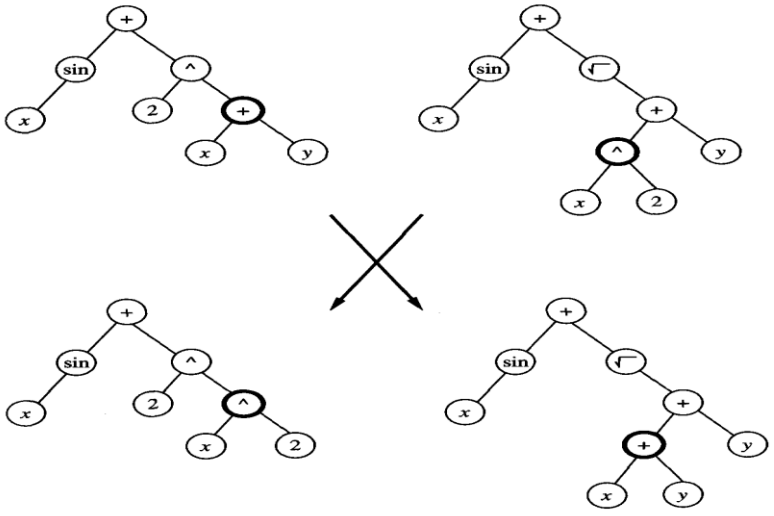


FIGURE 9.2
Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom).

Illustrative Example:

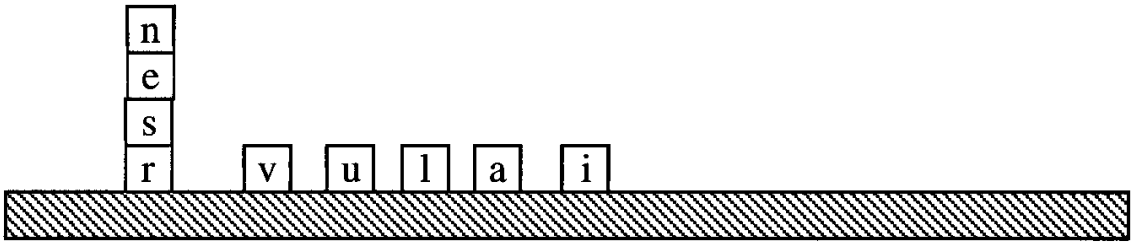


FIGURE 9.3
A block-stacking problem. The task for GP is to discover a program that can transform an arbitrary initial configuration of blocks into a stack that spells the word "universal." A set of 166 such initial configurations was provided to evaluate fitness of candidate programs (after Koza 1992).

One illustrative example presented by **Koza** (1992) involves learning an **algorithm** for stacking the blocks shown in Figure 9.3. The task is to develop a general algorithm for stacking the blocks into a single stack that spells the word **"universal,"** independent of the initial configuration of blocks in the world. The actions available for manipulating blocks allow moving only a single block at a time. In particular, **the top block on the stack can be moved to the table surface, or a block on the table surface can be moved to the top of the stack**.

In Koza's formulation, the primitive functions used to compose programs for this task include the following three terminal arguments:

- **CS (current stack)**, which refers to the name of the top block on the stack, or F if there is no current stack.
- **TB (top correct block)**, which refers to the name of the topmost block on the stack, such that it and those blocks beneath it are in the correct order.
- **NN (next necessary)**, which refers to the name of the next block needed above TB in the stack, in order to spell the word "universal," or F if no more blocks are needed.

In addition to these terminal arguments, the program language in this application included the following primitive functions:

- **(MS x)** (move to stack), if block x is on the table, this operator moves x to the top of the stack and returns the value T. Otherwise, it does nothing and returns the value F.

- **(MT x)** (move to table), if block x is somewhere in the stack, this moves the block at the top of the stack to the table and returns the value T. Otherwise, it returns the value F.
- **(EQ x y)** (equal), which returns T if x equals y, and returns F otherwise.
- **(NOT x)**, which returns T if x = F, and returns F if x = T.
- **(DU x y)** (do until), which executes the expression x repeatedly until expression y returns the value T.

To allow the system to evaluate the fitness of any given program, **Koza provided a set of 166 training example problems** representing a broad variety of initial block configurations, including problems of differing degrees of difficulty.

The fitness of any given program was taken to be the number of these examples **solved by the algorithm**. The population was initialized to a set of 300 random programs. After 10 generations, **the system discovered the following program**, which solves all **166 problems**.

(EQ (DU (MT CS)(NOT CS)) (DU (MS NN)(NOT NN)))

Notice this program contains a sequence of two DU, or **"Do Until"** statements. The first repeatedly **moves the current top of the stack** onto the table, until the stack becomes empty. The **second "Do Until"** statement then repeatedly **moves the next necessary block** from the table onto the stack. The role played by the top level EQ expression here is to provide a syntactically legal way to sequence these two "Do Until" loops.

After only a few generations, this GP was able to discover a program that solves all 166 training problems.

MODELS OF EVOLUTION AND LEARNING

In many natural systems, individual organisms learn to adapt significantly during their lifetime. At the same time, biological and social processes allow their species to adapt over a time frame of many generations.

1. **Lamarckian Evolution –** If an organism changes during its life to adapt to environment, then those changes are passed to its offsprings. Ex. Giraffe

Larnarck was a scientist who, in the late nineteenth century, proposed that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime.

2. **Baldwin Effect – Baldwin explains two learning behaviour.**
   a) Geno type – genetic code (DNA) and global search
   b) Pheno type – Your characteristics(behavior) and local search

Although Lamarckian evolution is not an accepted model of biological evolution, other mechanisms have been suggested by which individual learning can alter the course of evolution. One such mechanism is called the Baldwin effect.

If a species is evolving **in a changing environment**, there will be evolutionary pressure to favor individuals with the capability to **learn during their lifetime**. For example, if a new **predator** appears in the environment, then individuals **capable of learning to avoid the predator** will be more successful than individuals who cannot learn. In effect, the ability to learn allows an individual to perform a **small local search during its lifetime to maximize its fitness**. **In contrast, nonlearning individuals** whose fitness is fully determined by their **genetic makeup** will operate at a relative disadvantage.

As the proportion of self-improving individuals in the population grows, the population will be able to support a more diverse gene pool, allowing evolutionary processes to adapt more rapidly. This accelerated adaptation may in turn **enable standard evolutionary processes** to more quickly evolve a genetic **(nonlearned) trait to avoid the predator**.

Individual learning supports more rapid evolutionary progress, thereby increasing the **chance that the species will evolve genetic, nonlearned traits that better fit the new environment**.

PARALLELIZING GENETIC ALGORITHMS

Coarse grain approaches to parallelization subdivide the population into somewhat distinct groups of individuals, called demes. Each deme is assigned to a different computational node, and a standard GA search is performed at each node.

Fine-grain approaches to parallelization typically assign one processor per individual in the population. Recombination then takes place among neighboring individuals. Several different types of neighborhoods have been proposed, ranging from planar grid to torus.

LEARNING SETS OF RULES

Introduction:

In Leaning Sets of Rules, the algorithms are designed to learn sets of **first-order rules** that contain variables and also use **sequential covering algorithms** that learn one rule at **a** time to incrementally grow the final set of rules.

As an example of **first-order rule sets**, consider the following two rules that jointly describe the target concept ***Ancestor.*** Here we use the predicate ***Parent(x,*** y) to indicate that y is the mother or father of ***x,*** and the predicate ***Ancestor(x, y)*** to indicate that ***y*** is an ancestor of ***x*** related by an arbitrary number of family generations.

IF $Parent(x, y)$ THEN $Ancestor(x, y)$

IF $Parent(x, z) \wedge Ancestor(z, y)$ THEN $Ancestor(x, y)$

Note these two rules compactly describe a recursive function that would be very difficult to represent using a decision tree or other propositional representation. The above two, **first order rules** are also called **Horn clauses**.

One way to see the representational power of first-order rules is to consider the general purpose programming language **PROLOG**. In PROLOG programs are sets of first-order rules.

SEQUENTIAL COVERING ALGORITHMS

SEQUENTIAL-COVERING(*Target_attribute*, *Attributes*, *Examples*, *Threshold*)

- *Learned_rules* ← {}
- *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- while PERFORMANCE(*Rule*, *Examples*) > *Threshold*, do
  - *Learned_rules* ← *Learned_rules* + *Rule*
  - *Examples* ← *Examples* − {examples correctly classified by *Rule*}
  - *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- *Learned_rules* ← sort *Learned_rules* accord to PERFORMANCE over *Examples*
- return *Learned_rules*

TABLE 10.1
The sequential covering algorithm for learning a disjunctive set of rules. LEARN-ONE-RULE must return a single rule that covers at least some of the *Examples*. PERFORMANCE is a user-provided subroutine to evaluate rule quality. This covering algorithm learns rules until it can no longer learn a rule whose performance is above the given *Threshold*.

LEARN-ONE-RULE that accepts a set of positive and negative training examples as input, then **outputs a single rule** that covers many of the positive examples and few of the negative examples. We require that this output rule have high accuracy, but not necessarily high coverage. **By high accuracy**, we mean the predictions it makes should be correct. By accepting low coverage, we mean it need not make predictions for every training example.

Given this LEARN-ONE-RULE subroutine for learning a single rule, one obvious approach to learning a set of rules is to invoke LEARN-ONE-RULE on all the available training examples, remove any positive examples covered by the **rule** it learns, then invoke it again to learn a **second rule** based on the remaining training examples. This procedure can be **iterated** as many times as desired to learn a **disjunctive set of rules** that together cover any desired fraction of the positive examples. **This is called a sequential covering algorithm because it sequentially learns a set of rules that together cover the full set of positive examples.** The final set of rules can then be sorted so that more accurate rules will be considered first when a new instance must be classified.

General to Specific Beam Search:

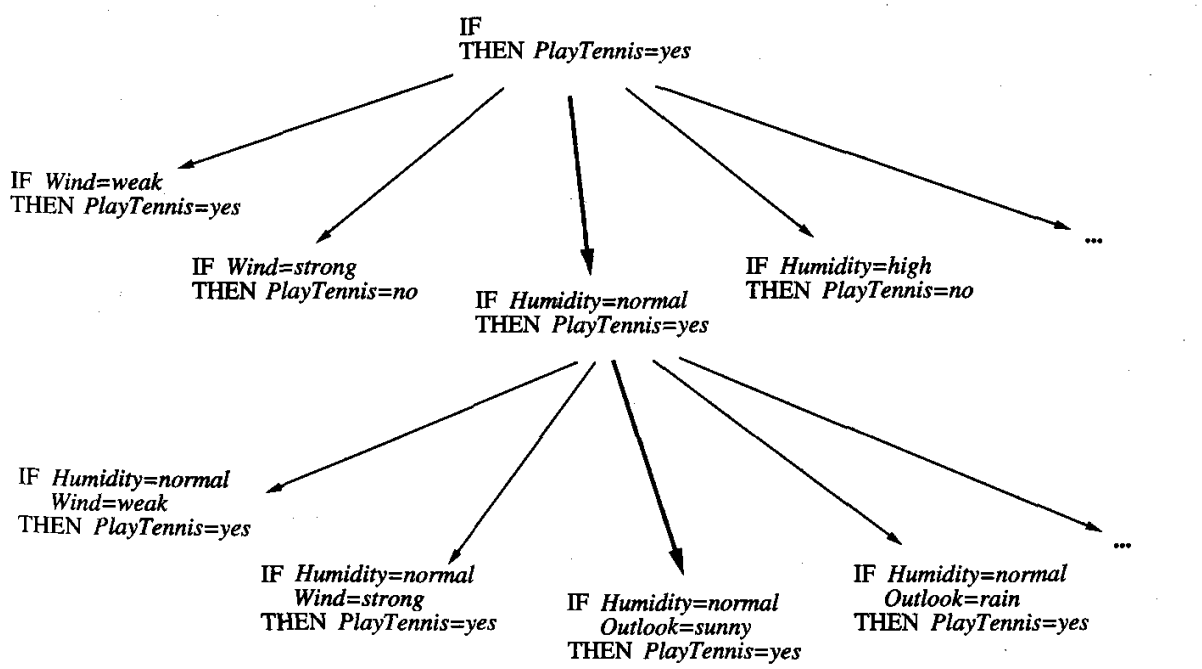


FIGURE 10.1
The search for rule preconditions as LEARN-ONE-RULE proceeds from general to specific. At each step, the preconditions of the best rule are specialized in all possible ways. Rule postconditions are determined by the examples found to satisfy the preconditions. This figure illustrates a beam search of width 1.

One effective approach to implementing LEARN-ONE-RULE is to organize the hypothesis space search, but to follow only the most promising branch in the tree at each step.

As shown in Figure 10.1, the search begins by considering the most general rule precondition possible, then greedily adding the attribute test that most improves rule performance measured over the training examples. **Once this test has been added, the process is repeated by greedily adding a second attribute test, and so on. This process grows the hypothesis by greedily adding new attribute tests until the hypothesis reaches an acceptable level of performance.**

LEARN-ONE-RULE(*Target_attribute, Attributes, Examples, k*)

Returns a single rule that covers some of the Examples. Conducts a general_to_specific greedy beam search for the best rule, guided by the PERFORMANCE *metric.*

- Initialize *Best_hypothesis* to the most general hypothesis $\emptyset$
- Initialize *Candidate_hypotheses* to the set {*Best_hypothesis*}
- While *Candidate_hypotheses* is not empty, Do
  1. *Generate the next more specific candidate_hypotheses*
     - *All_constraints* ← the set of all constraints of the form $(a = v)$, where a is a member of *Attributes*, and v is a value of a that occurs in the current set of *Examples*
     - *New_candidate_hypotheses* ←
       for each h in *Candidate_hypotheses*,
       for each c in *All_constraints*,
       - create a specialization of h by adding the constraint c
     - Remove from *New_candidate_hypotheses* any hypotheses that are duplicates, inconsistent, or not maximally specific
  2. *Update Best_hypothesis*
     - For all h in *New_candidate_hypotheses* do
       - If (PERFORMANCE(h, *Examples*, *Target_attribute*)
         > PERFORMANCE(*Best_hypothesis*, *Examples*, *Target_attribute*))
         Then *Best_hypothesis* ← h
  3. *Update Candidate_hypotheses*
     - *Candidate_hypotheses* ← the k best members of *New_candidate_hypotheses*, according to the PERFORMANCE measure.
- Return a rule of the form
  "IF *Best_hypothesis* THEN *prediction*"
  where *prediction* is the most frequent value of *Target_attribute* among those *Examples* that match *Best_hypothesis*.

PERFORMANCE(h, *Examples*, *Target_attribute*)

- *h_examples* ← the subset of *Examples* that match h
- return $-Entropy(h\_examples)$, where entropy is with respect to *Target_attribute*

TABLE 10.2
One implementation for LEARN-ONE-RULE is a general-to-specific beam search. The frontier of current hypotheses is represented by the variable *Candidate_hypotheses*. This algorithm is similar to that used by the CN2 program, described by Clark and Niblett (1989).

This implementation of LEARN-ONE RULE follows only a single descendant at each search step-the attribute-value pair yielding the best performance. This general-to-specific search for the LEARN-ONE-RULE algorithm is a greedy depth-first search **with no backtracking**.

Beam Search is a search in which the algorithm maintains a list of the k best candidates at each step, rather than a single best candidate. On each search step, descendants are generated for each of these k best candidates, and the resulting set is again reduced to the k most promising members. Beam search keeps track of the most promising alternatives to the current top-rated hypothesis, so that all of their successors can be considered at each search step. This general to specific beam search algorithm is described in Table 10.2.

LEARNING RULE SETS: SUMMARY

Five key dimensions in the design space of rule learning algorithms. ie, Discussions of a variety of possible methods for learning sets of rules

1. **First dimension:**

Sequential covering algorithms (ex. CN2 algorithm) learn one rule at a time, removing the covered examples and repeating the process on the remaining examples. In **Decision tree algorithms** (ex. ID3 algorithm) learn the entire set of disjuncts simultaneously as part of the single search for an acceptable decision tree. ID3 algorithm is also called Simultaneous covering decision tree learning algorithms.

Which should we prefer?

It may depend on how much training data is available. If **data is plentiful**, then it may support the larger number of independent decisions required by the **sequential covering algorithm.** If it is desirable that different rules test the same attributes, it is possible by **Simultaneous covering decision tree learning algorithms**.

2. **Second dimension:**

G**eneral to specific search** is that there is a single maximally general hypothesis from which to begin the search, whereas there are very many specific hypotheses in most hypothesis spaces (i.e., one for each possible instance). Given many maximally specific hypotheses, it is unclear which to select as the starting point of the search.

Specific-to-general search addresses this issue by choosing several positive examples at random to initialize and to guide the search. The best hypothesis obtained through multiple random choices is then selected.

3. **Third dimension:**

Generate and test approach is that each choice in the search is based on the hypothesis performance over many examples, so that the impact of noisy data is minimized.

Example-driven algorithms (FIND-S and CANDIDATE-ELIMINATlON algorithms) that refine the hypothesis based on individual examples are more easily misled by a single noisy training example and are therefore less robust to errors in the training data.

4. **Fourth dimension:**

As in **decision tree learning**, it is possible for LEARN-ONE-RULE to formulate rules that perform very well on the training data, but less well on subsequent data. As in decision tree learning, one way to address this issue is to **post-prune each rule** after it is learned from the training data. In particular, preconditions can be removed from the rule whenever this leads to improved performance over a set of pruning examples distinct from the training examples.

5. **Fifth dimension:**

A final dimension is the particular definition of rule PERFORMANCE used to guide the search in LEARN-ONE-RULE Various evaluation functions have been used. Some common evaluation functions include:

a) Relative frequency - The relative frequency estimate of rule performance is $\frac{n_c}{n}$.

n_c - denote the number of these that it classifies correctly

n - denote the number of examples the rule matches

b) m-estimate of accuracy - The m-estimate of rule accuracy is

$$\frac{n_c + mp}{n + m}$$

m – weight

p - prior probability that a randomly drawn example from the entire data set will have the classification assigned by the rule

c) Entropy - This is the measure used by the PERFORMANCE subroutine in the algorithm of Table 10.2.

$$-Entropy(S) = \sum_{i=1}^{c} p_i \log_2 p_i$$

S be the set of examples that match the rule preconditions.
Negative of the entropy - better rules will have higher scores.
c is the number of distinct values the target function may take on
p_i is the proportion of examples from S for which the target function takes on the i^{th} value

LEARNING FIRST-ORDER RULES

In the previous sections we discussed algorithms for learning sets of propositional (i.e., variable-free) rules. In this section, we consider **learning rules** that **contain variables**-in particular, **learning first-order Horn theories**.

Consider,

The task of learning the simple target concept, Daughter (x, y) defined over pairs of people x and y. The value of Daughter(x, y) is True when x is the daughter of y, and False otherwise.

Suppose each person in the data is described by the attributes Name, Mother, Father, Male, Female. Hence, each training example will consist of the description of two people in terms of these attributes, along with the value of the target attribute Daughter.

For example, the following is a positive example in which Sharon is the daughter of Bob:

$$\begin{array}{lll} \langle Name_1 = Sharon, & Mother_1 = Louise, & Father_1 = Bob, \\ Male_1 = False, & Female_1 = True, & \\ Name_2 = Bob, & Mother_2 = Nora, & Father_2 = Victor, \\ Male_2 = True, & Female_2 = False, & Daughter_{1,2} = True \rangle \end{array}$$

where the subscript on each attribute name indicates which of the.two persons is being described.

IF $(Father_1 = Bob) \wedge (Name_2 = Bob) \wedge (Female_1 = True)$
THEN $Daughter_{1,2} = True$

A program using first-order representations could learn the following general rule:

IF $Father(y, x) \wedge Female(y)$, THEN $Daughter(x, y)$

where x and y are variables that can be bound to any person.

- Every well-formed expression is composed of *constants* (e.g., $Mary$, 23, or Joe), *variables* (e.g., x), *predicates* (e.g., $Female$, as in $Female(Mary)$), and *functions* (e.g., age, as in $age(Mary)$).
- A *term* is any constant, any variable, or any function applied to any term. Examples include $Mary$, x, $age(Mary)$, $age(x)$.
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include $Female(Mary)$, $\neg Female(x)$, $Greater\_than(age(Mary), 20)$.
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg Female(Joe)$).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg Female(Joe)$).
- A *positive literal* is a literal with no negation sign (e.g., $Female(Mary)$).
- A *clause* is any disjunction of literals $M_1 \vee \ldots M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

$$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

  where $H, L_1 \ldots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.
- For any literals A and B, the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

$$H \vee \neg L_1 \vee \ldots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z. Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L.
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.

TABLE 10.3
Basic definitions from first-order logic.

LEARNING SETS OF FIRST-ORDER RULES: FOIL

FOIL – First Order Inductive Learning

A variety of algorithms has been proposed for learning first-order rules, or Horn clauses. In this section we consider a program called FOIL.

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* ← those *Examples* for which the *Target_predicate* is *True*
- *Neg* ← those *Examples* for which the *Target_predicate* is *False*
- *Learned_rules* ← {}
- while *Pos*, do

  Learn a NewRule

  - *NewRule* ← the rule that predicts *Target_predicate* with no preconditions
  - *NewRuleNeg* ← *Neg*
  - while *NewRuleNeg*, do

    Add a new literal to specialize NewRule

    - *Candidate_literals* ← generate candidate new literals for *NewRule*, based on *Predicates*
    - *Best_literal* ← $\underset{L \in Candidate\_literals}{\text{argmax}}$ *Foil_Gain*(*L*, *NewRule*)
    - add *Best_literal* to preconditions of *NewRule*
    - *NewRuleNeg* ← subset of *NewRuleNeg* that satisfies *NewRule* preconditions
  - *Learned_rules* ← *Learned_rules* + *NewRule*
  - *Pos* ← *Pos* − {members of *Pos* covered by *NewRule*}
- Return *Learned_rules*

TABLE 10.4
The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.

The two most substantial differences between FOIL and our earlier SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithm follow from the requirement that it accommodate first-order rules. These differences are:

1. In its general-to-specific search to learn each new rule, FOIL employs different detailed steps to generate candidate specializations of the rule. This difference follows from the need to accommodate variables in the rule preconditions.

2. FOIL employs a PERFORMANCE measure, Foil-Gain, that differs from the entropy measure shown for LEARN-ONE-RULE in Table 10.2. This difference follows from the need to distinguish between different bindings of the rule variables and from the fact that FOIL seeks only rules that cover positive examples.

Example of Generating Candidate Specializations in FOIL is

The general-to-specific search in FOIL begins with the most general Rule

$$GrandDaughter(x, y) \leftarrow$$

To specialize this initial rule, the above procedure generates the following literals as candidate additions to the rule preconditions: Equal (x , y) , Female(x), Female(y), Father(x, y), Father(y, x), Father(x, z), Father(z, x), Father(y, z), Father(z, y), and the negations of each of these literals (e.g., Equal(x, y)).

Now suppose that among the above literals FOIL greedily selects Father(y , z) as the most promising, leading to the more specific rule.

$$GrandDaughter(x, y) \leftarrow Father(y, z)$$

If FOIL at this point were to select the literal Father(z, x) and on the next iteration select the literal Female(y),

$$GrandDaughter(x, y) \leftarrow Father(y, z) \wedge Father(z, x) \wedge Female(y)$$

At this point, FOIL will remove all positive examples covered by this new rule. If additional positive examples remain to be covered, then it will begin yet another general-to-specific search for an additional rule.

For illustration, assume the training data includes the following simple set of assertions

$GrandDaughter(Victor, Sharon)$ $Father(Sharon, Bob)$ $Father(Tom, Bob)$
$Female(Sharon)$ $Father(Bob, Victor)$

The constants Victor, Sharon, Bob, and Tom that is not listed above can be assumed to be false

$$\neg GrandDaughter(Tom, Bob), \neg GrandDaughter(Victor, Victor),$$

The evaluation function used by FOIL to estimate the utility of adding a new literal is based on the numbers of positive and negative bindings covered before and after adding the **new literal**.

$$Foil\_Gain(L, R) \equiv t \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right)$$

Where

R – Rule

L – Candidate literal

P_0 - Number of positive bindings of rule R

n_0 - Number of negative bindings of R

p_1 - Number of positive bindings of new rule R^1

n_1 - Number of negative bindings of R^1

t - Number of positive bindings of rule R that are still covered after adding literal L to R.

IF	$Parent(x, y)$	THEN	$Ancestor(x, y)$
IF	$Parent(x, z) \wedge Ancestor(z, y)$	THEN	$Ancestor(x, y)$

INDUCTION AS INVERTED DEDUCTION

An another approach to inductive logic programming is Induction as Inverted Deduction.

Given some data D and some partial background knowledge B, learning can be described as generating a hypothesis h that, together with B, explains D.

$$(\forall \langle x_i, f(x_i) \rangle \in D) \ (B \wedge h \wedge x_i) \vdash f(x_i)$$

Here x_i denotes the i^{th} training instance

$f(x_i)$ denotes its target value

B denotes data set

h denotes hypothesis

D denotes Training data D= $f(x_i, f(x_i))$

$X \vdash Y$ denotes "Y follows deductively from X" or "X entails Y."

As an example,

The predicate **child(u ,v)** represents “child of u is v”.

$$x_i : Male(Bob), Female(Sharon), Father(Sharon, Bob)$$

$$f(x_i) : Child(Bob, Sharon)$$

$$B : Parent(u, v) \leftarrow Father(u, v)$$

In this case, two of the many hypotheses that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$ are

$$h_1 : Child(u, v) \leftarrow Father(v, u)$$

$$h_2 : Child(u, v) \leftarrow Parent(v, u)$$

This example illustrates the role of background knowledge in expanding the set of acceptable hypotheses for a given set of training data.

It also illustrates how new predicates (e.g., Parent) can be introduced into hypotheses (e.g., h_2), even when the predicate is not present in the original description of the instance x_i. This process of augmenting the set of predicates, based on background knowledge, is often referred to as **constructive induction**.

It can be represented by inverse entailment operators O(B,D). An inverse entailment operator, O(B, D) takes the training data D = $\{(x_i, f(x_i))\}$ and background knowledge B as input and produces as output a hypothesis h.

$$O(B, D) = h \text{ such that } (\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

INVERTING RESOLUTION

A general method for automated deduction is the resolution rule which is a complete rule for deductive inference in first-order logic. Therefore, it is possible to invert the resolution rule to form an inverse entailment operator.

Example for Resolution rule is

$$\begin{array}{lll} P & \vee & L \\ \neg L & \vee & R \\ \hline P & \vee & R \end{array}$$

Another example, Clause C_1 = A V B V C V ¬D

Clause C_2 = ¬B V E V F

The Result of Resolution rule is A V C V ¬D V E V F.

The general form of the propositional resolution operator is described as

1. Given initial clauses C_1 and C_2, find a literal L from clause C_1 such that $\neg L$ occurs in clause C_2.
2. Form the resolvent C by including all literals from C_1 and C_2, except for L and $\neg L$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

where $\cup$ denotes set union, and "–" denotes set difference.

TABLE 10.5
Resolution operator (propositional form). Given clauses C_1 and C_2, the resolution operator constructs a clause C such that $C_1 \wedge C_2 \vdash C$.

It is easy to invert the resolution operator to form an inverse entailment operator O(C, C1) that performs inductive inference. In general, the inverse entailment operator must derive one of the initial clauses, C2, given the resolvent C and the other initial clause C1.

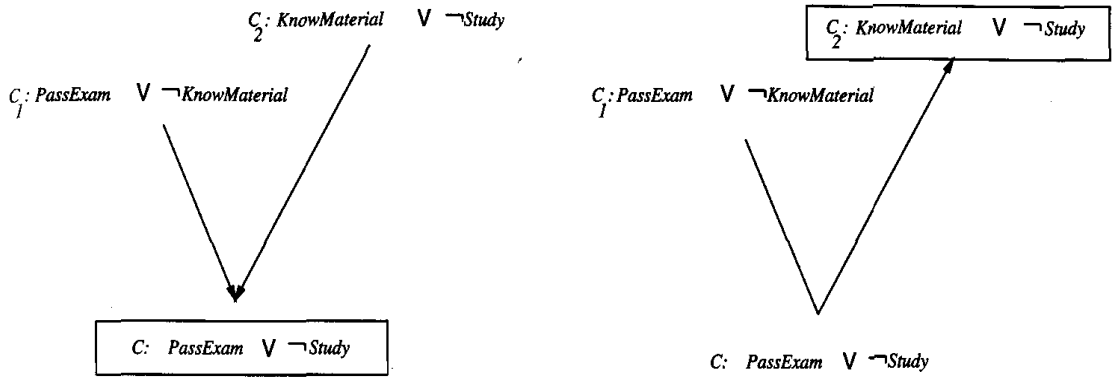


FIGURE 10.2
On the left, an application of the (deductive) resolution rule inferring clause C from the given clauses C_1 and C_2. On the right, an application of its (inductive) inverse, inferring C_2 from C and C_1.

We can develop rule-learning algorithms based on inverse entailment operators such as inverse resolution. The CIGOL program uses inverse resolution

1. Given initial clauses C_1 and C, find a literal L that occurs in clause C_1, but not in clause C.
2. Form the second clause C_2 by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{\neg L\}$$

TABLE 10.6
Inverse resolution operator (propositional form). Given two clauses C and C_1, this computes a clause C_2 such that $C_1 \wedge C_2 \vdash C$.

A multistep application of this inverse resolution rule is

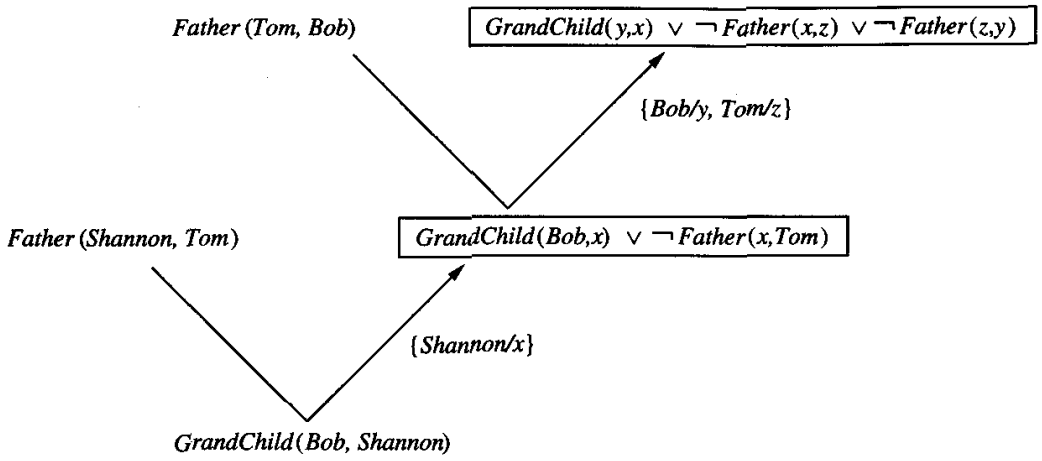
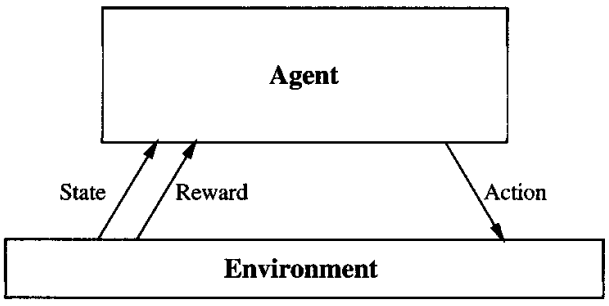


FIGURE 10.3
A multistep inverse resolution. In each case, the boxed clause is the result of the inference step. For each step, C is the clause at the bottom, C_1 the clause to the left, and C_2 the boxed clause to the right. In both inference steps here, θ_1 is the empty substitution {}, and θ_2^{-1} is the substitution shown below C_2. Note the final conclusion (the boxed clause at the top right) is the alternative form of the Horn clause $GrandChild(y, x) \leftarrow Father(x, z) \wedge Father(z, y)$.

REINFORCEMENT LEARNING

Introduction:

Reinforcement learning addresses the question of how an autonomous agent (ex. Mobile Robot) that senses and acts in its environment can learn to choose optimal actions to achieve its goals. The robot, or ***agent,*** has a set of sensors to observe the ***state*** of its environment, and a set of ***actions*** it can perform to alter this state. Each time the agent performs an action in its environment, a trainer may provide a **reward or penalty** to indicate the **desirability of the resulting state**.



For example, a mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn." Its task is to learn a control **strategy, or policy**, for choosing actions that achieve its goals. The agents can learn successful control policies by experimenting in their environment.

For example, the robot may have a goal of docking onto its battery charger whenever its **battery level** is low.

When training an agent to play a game the trainer might provide a **positive reward** when the game is won, **negative reward** when it is lost, and **zero reward** in all other states.

For example, the goal of docking to the battery charger can be captured by assigning a positive reward (e.g., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition. This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot.

The task of the robot is to perform sequences of actions, observe their consequences, and learn a control policy. The control policy we desire is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent. **One highly successful application of the reinforcement learning algorithms is game-playing problem.**

The target function to be learned in this case is a control policy, ***Π: S*** $\rightarrow$ ***A,*** that outputs an appropriate action ***a*** from the set ***A,*** given the current state ***s*** from the set ***S.***

An algorithm called *Q* learning that can acquire optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment. Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems.

The learning task:

In a **Markov decision process (MDP)**, the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform. At each discrete time step t, the agent senses the current state s_t, chooses a current action a_t, and performs it. The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by **producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$**. Here the functions **δ and r are part of the environment** and are not necessarily known to the agent. In an MDP, the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state.

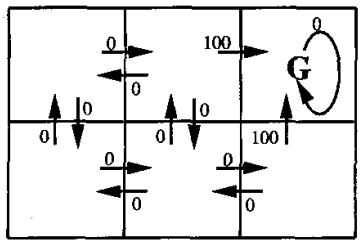
The task of the agent is to learn a policy, : S $\rightarrow$ A, for selecting its next action a, based on the current observed state s_t; that is, $\pi(s_t) = a_t$.

We require that the agent learn a policy π that maximizes $V^{\pi}(s)$ for all states s. We will call such a policy an optimal policy and denote it by π^*.

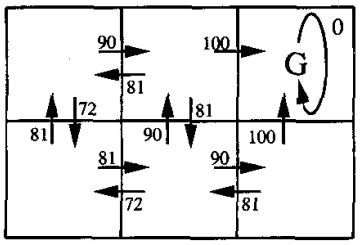
$V^{\pi}(s)$ called the discounted cumulative reward achieved by policy n from initial state s.

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}} V^{\pi}(s), (\forall s)$$

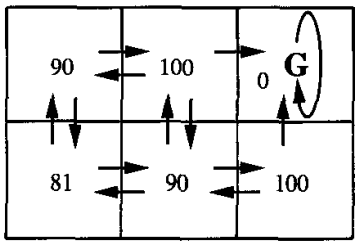
To illustrate these concepts, a simple grid-world environment is depicted in the Figure 13.2.



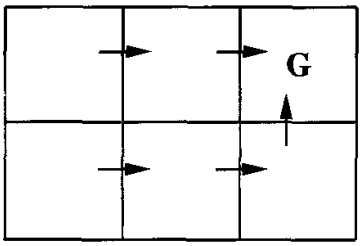
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

FIGURE 13.2
A simple deterministic world to illustrate the basic concepts of Q-learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, $r(s, a)$ gives reward 100 for actions entering the goal state **G**, and zero otherwise. Values of $V^*(s)$ and $Q(s, a)$ follow from $r(s, a)$, and the discount factor $\gamma = 0.9$. An optimal policy, corresponding to actions with maximal Q values, is also shown.

The **six grid squares** in this diagram represent six possible states, or locations, for the agent. Each arrow in the diagram represents a possible action the agent can take to move from one state to another.

The number associated with each arrow represents the immediate reward r(s, a) the agent receives if it executes the corresponding **state-action transition**. Note the immediate reward in this particular environment is defined to be **zero** for all state-action transitions **except for those leading into the state labeled G. G is the goal state**, because the only way the agent can receive reward, in this case, is by entering this state. **G is also called as absorbing state. The optimal policy directs the agent along the shortest path toward the state G.**

Q-LEARNING

The evaluation function Q(s, a) is the maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action. Learning the Q function corresponds to learning the optimal policy. The value of Q is the reward received immediately upon executing action a from state s, plus the value (discounted by γ) of following the optimal policy thereafter.

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

The Learning V* is a useful way to learn the optimal policy only when the agent has perfect knowledge of δ and r. This requires that it be able to perfectly predict the immediate result (i.e., the immediate reward and immediate successor) for every possible state-action transition. The agent can acquire the optimal policy by learning V*, provided it has perfect knowledge of the immediate reward function r and the state transition function δ.

Note that Q(s, a) is exactly the quantity that is maximized in order to choose the optimal action a in state s.

$$\pi^*(s) = \operatorname{argmax} Q(s,a)$$

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s,a)$ to zero.
Observe the current state s
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s,a)$ as follows:

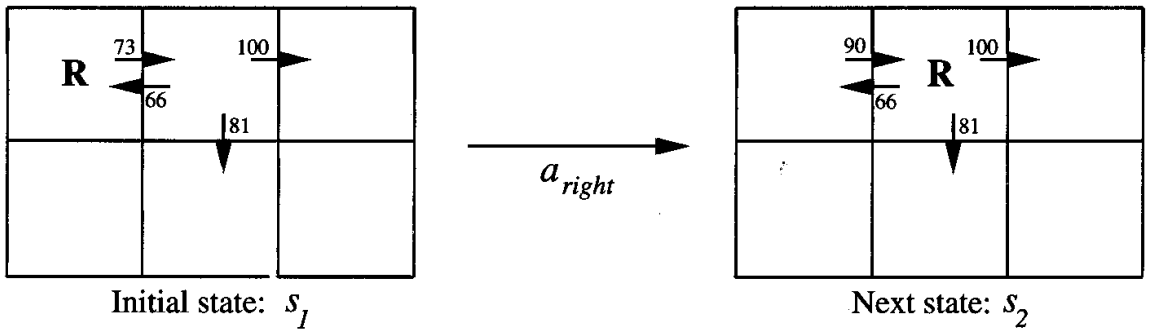
$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

TABLE 13.1
Q learning algorithm, assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$.

The key problem is finding a reliable way to estimate training values for Q, given only a sequence of immediate rewards r spread out over time. This can be accomplished through iterative approximation.

To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement. Each time the agent moves forward from an old state to a new one, Q learning propagates $\hat{Q}$ estimates backward from the new state to the old. At the same time, the immediate reward received by the agent for the transition is used to augment these propagated values of $\hat{Q}$.



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$
$$\leftarrow 0 + 0.9 \max\{66, 81, 100\}$$
$$\leftarrow 90$$

FIGURE 13.3
The update to $\hat{Q}$ after executing a single action. The diagram on the left shows the initial state s_1 of the robot (**R**) and several relevant $\hat{Q}$ values in its initial hypothesis. For example, the value $\hat{Q}(s_1, a_{right}) = 72.9$, where a_{right} refers to the action that moves **R** to its right. When the robot executes the action a_{right}, it receives immediate reward $r = 0$ and transitions to state s_2. It then updates its estimate $\hat{Q}(s_1, a_{right})$ based on its $\hat{Q}$ estimates for the new state s_2. Here $\gamma = 0.9$.

NON-DETERMINISTIC REWARDS AND ACTIONS

Above we considered Q learning in deterministic environments. Here we consider the nondeterministic case, in which the reward function r(s, a) and action transition function δ(s, a) may have probabilistic outcomes.

The functions δ(s, a) and r(s, a) can be viewed as first producing a **probability distribution over outcomes based on s and a**, and then drawing an outcome at random according to this distribution. When these probability distributions depend solely on s and a (e.g., **they do not depend on previous states or actions**), then we call the system a **nondeterministic Markov decision process(MDP)**.

The obvious generalization is to redefine the value V^{π} of a policy π to be the expected value (over these nondeterministic outcomes) of the discounted cumulative reward. We can express Q as

$$Q(s, a) = E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a')$$

Assignment:

1. Describe Temporal Difference Learning
2. Write short note on Generalizing from Examples.
3. Explain briefly about Relationship to Dynamic Programming.