Genetic Algorithm

Motivation

- Genetic algorithms (GAS) provide a learning method motivated by an analogy to biological evolution. Rather than search from general-to-specific hypotheses, or from simple-to-complex, GAS generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses.
- At each step, a collection of hypotheses called the current population is updated by replacing some fraction of the population by offspring of the most fit current hypotheses

Motivation

- 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

- The problem addressed by GAs is to search a space of candidate hypotheses 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.***
- ***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.

Structure of GA

- The algorithm operates by iteratively updating a pool of hypotheses, called the population. 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 intact.
- Others are used as the basis for creating new offspring individuals by applying genetic operations such as crossover and mutation.

A prototypical genetic algorithm

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.

A prototypical genetic algorithm

- A population containing ***p*** *hypotheses is maintained. On each iteration,* the successor population *Ps 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 prototypical genetic algorithm

- The inputs tothis algorithm include the fitness function for ranking candidate hypotheses, a threshold defining an acceptable level of fitness for terminating the algorithm, the size of the population to be maintained, and parameters that determine how successor populations are to be generated: the fraction of the population to be replaced at each generation and the mutation rate.

- in this algorithm each iteration through the main loop produces a new generation of hypotheses based on the current population. First, a certain numberof hypotheses from the current population are selected for inclusion in the nextgeneration. These are selected *probabilistically, where the probability of selecting* hypothesis *hi is given by*

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

Thus, the probability that a hypothesis will be selected is proportional to its own fitness and is inversely proportional to the fitness of the other competing
hypotheses in the current population.

- Once these members of the current generation have been selected for inclusion in the next generation population, additional members are generated using a crossover operation.
- It takes two parent hypotheses from the current generation and creates two offspring hypotheses by recombining portions of both parents. The parent hypotheses are chosen probabilistically from the current population, again using the probability function
- After new members have been created by this crossover operation, the new generation population now contains the desired number of members.
- At this point, a certain fraction m of these members are chosen at random, and random mutations all performed to alter these members

Representing Hypotheses

- Hypotheses in GAS are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover. The hypotheses represented by these bit strings can be quite complex. For example, sets of if-then rules can easily be represented in this way, by choosing an encoding of rules that allocates specific substrings for each rule precondition and postcondition.

- Use a bit string to describe a constraint on the value of a single attribute.
- 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.
- Placing a 1 in some position indicates that the attribute is allowed to take on the corresponding value. For example, 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).*
- 11 1 represents the most general possible constraint, indicating that we don't care which of its possible values the attribute takes on.

- Conjunctions of constraints on multiple attributes can easily be represented by concatenating the corresponding bit strings. For example, consider a second attribute, ***Wind, that can take on the value Strong or Weak. A rule precondition such as (Outlook = Overcast V Rain) A (Wind = Strong)*** can then be represented by the following bit string of length five:

Outlook	***Wind***
01 1	10

- Rule postconditions (such as ***PlayTennis = yes) can be represented in a*** similar fashion. Thus, an entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule postcondition. For example, the rule
- ***IF Wind = Strong THEN PlayTennis = yes*** would be represented by the string

Outlook	***Wind***	***PlayTennis***
111	10	10

- where the first three bits describe the "don't care" constraint on ***Outlook, the next*** two bits describe the constraint on ***Wind, and the final two bits describe the rule*** postcondition (here we assume ***PlayTennis can take on the values Yes or No).***
- Note the bit string representing the rule contains a substring for each attribute in the hypothesis space, even if that attribute is not constrained by the rule preconditions.
- This yields a fixed length bit-string representation for rules, in which substrings at specific locations describe constraints on specific attributes. Given this representation for single rules, we can represent sets of rules by similarly concatenating the bit string representations of the individual rules.
- In some GAS, hypotheses are represented by symbolic descriptions rather than bit strings.

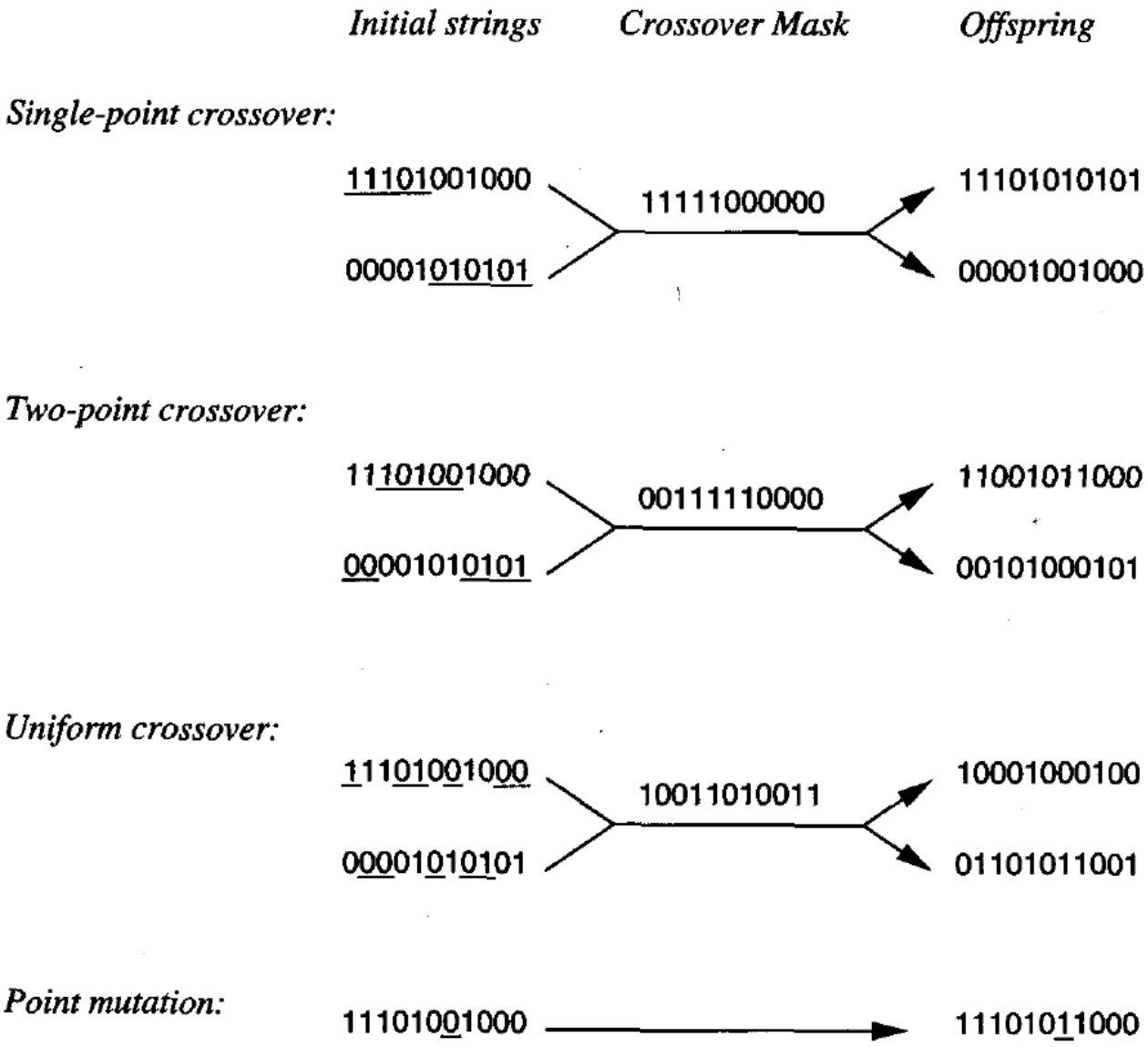
Genetic Operators

- The generation of successors in a GA is determined by a set of operators that recombine and mutate selected members of the current population.
- These operators correspond to idealized versions of the genetic operations found in biological evolution. The two most common operators are ***crossover and mutation***
- The ***crossover operator produces two new offspring from two parent strings,*** by copying selected bits from each parent. 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*** consider the ***single-point crossover operator***
- This offspring takes its first five bits from the first parent and its remaining six bits from the second parent, because the crossover mask 11 11 1000000 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

Crossover

- In single-point crossover, the crossover mask is always constructed so that it begins with a string containing ***n contiguous Is,*** 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 ***no zeros, followed by a contiguous*** string of ***nl 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 ***no and nl. For instance, in the example*** shown the offspring are created using a mask for which ***no = 2 and n 1 = 5. Again, the two offspring are created by switching the roles played by the*** two parents.

Common operators for genetic algorithms



Crossover

- *Uniform crossover combines bits sampled uniformly from the two parents,* 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 addition to recombination operators that produce offspring by combining parts of two parents, a second type of operator produces offspring from a single parent. 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 our prototypical algorithm

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. Often other criteria may be included as well, such as the complexity or generality of the rule. More generally, when the bit-string hypothesis is interpreted as a complex procedure (e.g., when the bit string represents a collection of if-then rules that will be chained together to control a robotic device), the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.
- 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
- This method is sometimes called ***jitness proportionate selection, or roulette wheel selection.***
- ***Other methods*** for using fitness to select hypotheses have also been proposed. ***tournament selection, two hypotheses are first chosen at random from the current***population. With some predefined probability p the more fit of these two is then selected, and with probability (1 - p) the less fit hypothesis is selected. Tournament selection often yields a more diverse population than fitness proportionate selection
- In another method called ***rank selection, the*** hypotheses in the current population are first sorted by fitness. The probability that a hypothesis will be selected is then proportional to its rank in this sorted list, rather than its fitness.

GA EXAMPLE

- A genetic algorithm can be viewed as a general optimization method that searches a large space of candidate objects seeking one that performs best according to the fitness function.
- GAS often succeed in finding an object with high fitness. GAS have been applied to a number of optimization problems outside machine learning, including problems such as circuit layout and job-shop scheduling. Within machine learning, they have been applied both to function-approximation problems and to tasks such as choosing the network topology for artificial neural network learning systems
- GABIL system described by DeJong et al. (1993).
- GABIL uses a GA to learn boolean concepts represented by a disjunctive set of propositional rules. 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

- **Representation:** Each hypothesis in GABIL corresponds to a disjunctive set of propositional rules
- The hypothesis space of rule preconditions consists of a conjunction of constraints on a fixed set of attributes
- To represent a set of rules, the bit-string representations of individual rules are concatenated.
- 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
- IFal=Tr\az=F THEN c=T; IF a2=T 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

The length of the bit string grows with the number of rules in the hypothesis. This variable bit-string length requires a slight modification to the crossover operator

Genetic operators

- GABIL uses the standard mutation operator in which a single bit is chosen at random and replaced by its complement.
- The crossover operator that it uses is a fairly standard extension to the two-point crossover operator described in Table 9.2. In particular, to accommodate the variable-length bit strings that encode rule sets, and to constrain the system so that crossover occurs only between like sections of the bit strings that encode rules, the following approach is taken. To perform a crossover operation on two parents, two crossover points are first chosen at random in the first parent string. Let dl (dz) 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 dl and ***d2 value 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 "1" indicate crossover points, then dl = 1 and dz = 3. Hence the allowed pairs of crossover points for the second parent include the pairs of bit positions (1,3), (1,8), and **(6,8). If the pair (1,3) happens to be** chosen,

	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

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

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

- In experiments comparing the behavior of GABIL to decision tree learning algorithms such as C4.5 and ID5R, and to the rule learning algorithm AQ14, DeJong et al. (1993) report roughly comparable performance among these systems, tested on a variety of learning problems. For example, over a set of 12 synthetic problems, GABIL achieved an average generalization accuracy of 92.1 %, whereas the performance of the other systems ranged from 91.2 % to 96.6 %.**9.3.1**

GENETIC PROGRAMMING

- Genetic programming (GP) is a form of evolutionary computation in which the individuals in the evolving population are computer programs rather than bit strings. Koza (1992) describes the basic genetic programming approach and presents a broad range of simple programs that can be successfully learned by GP.
- **Representing Programs :** Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program. Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes.

Representation for the function

$$\sin(x) + \sqrt{x^2 + y}.$$

- To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., sin, cos, *J,+, -, exponential~),* as well as the terminals (e.g., x, ***y, constants such as 2).***
- ***The genetic*** programming algorithm then uses an evolutionary search to explore the vast space of programs that can be described using these primitives.

GP

- As in a genetic algorithm, the prototypical genetic programming algorithm maintains a population of individuals (in this case, program trees). On each iteration, it produces a new generation of individuals using selection, crossover, and mutation.
- The fitness of a given individual program in the population is typically determined by executing the program on a set of training data. Crossover operations are performed by replacing a randomly chosen subtree of one parent program by a subtree from the other parent program

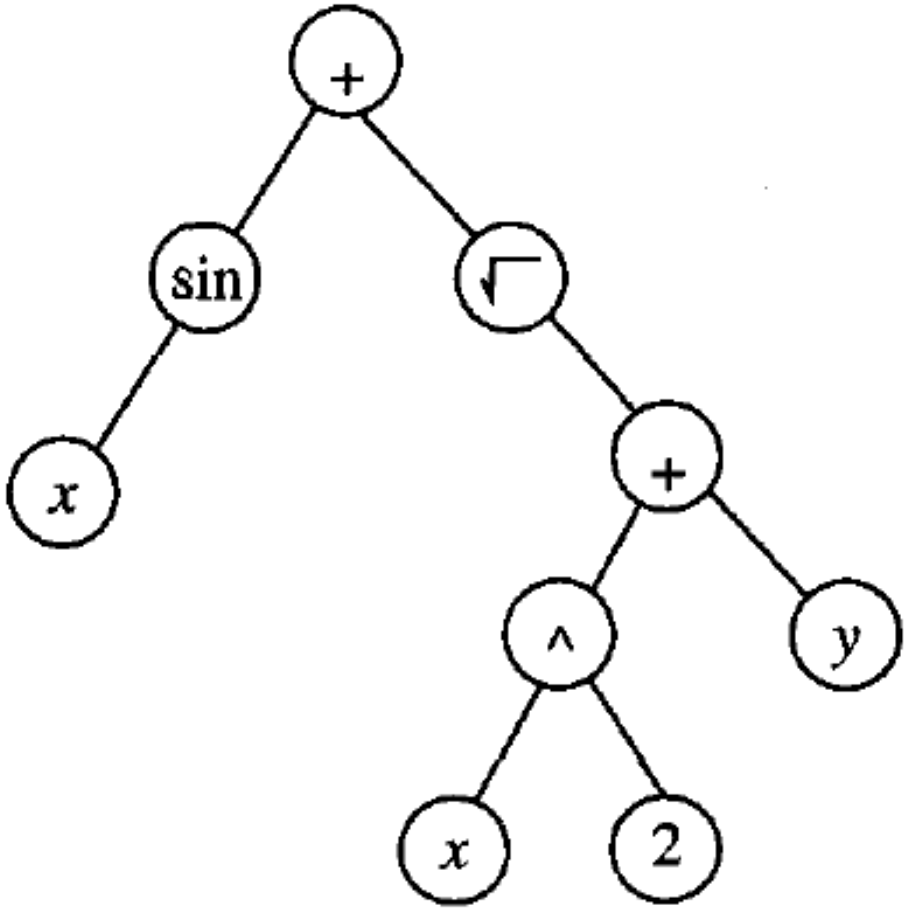
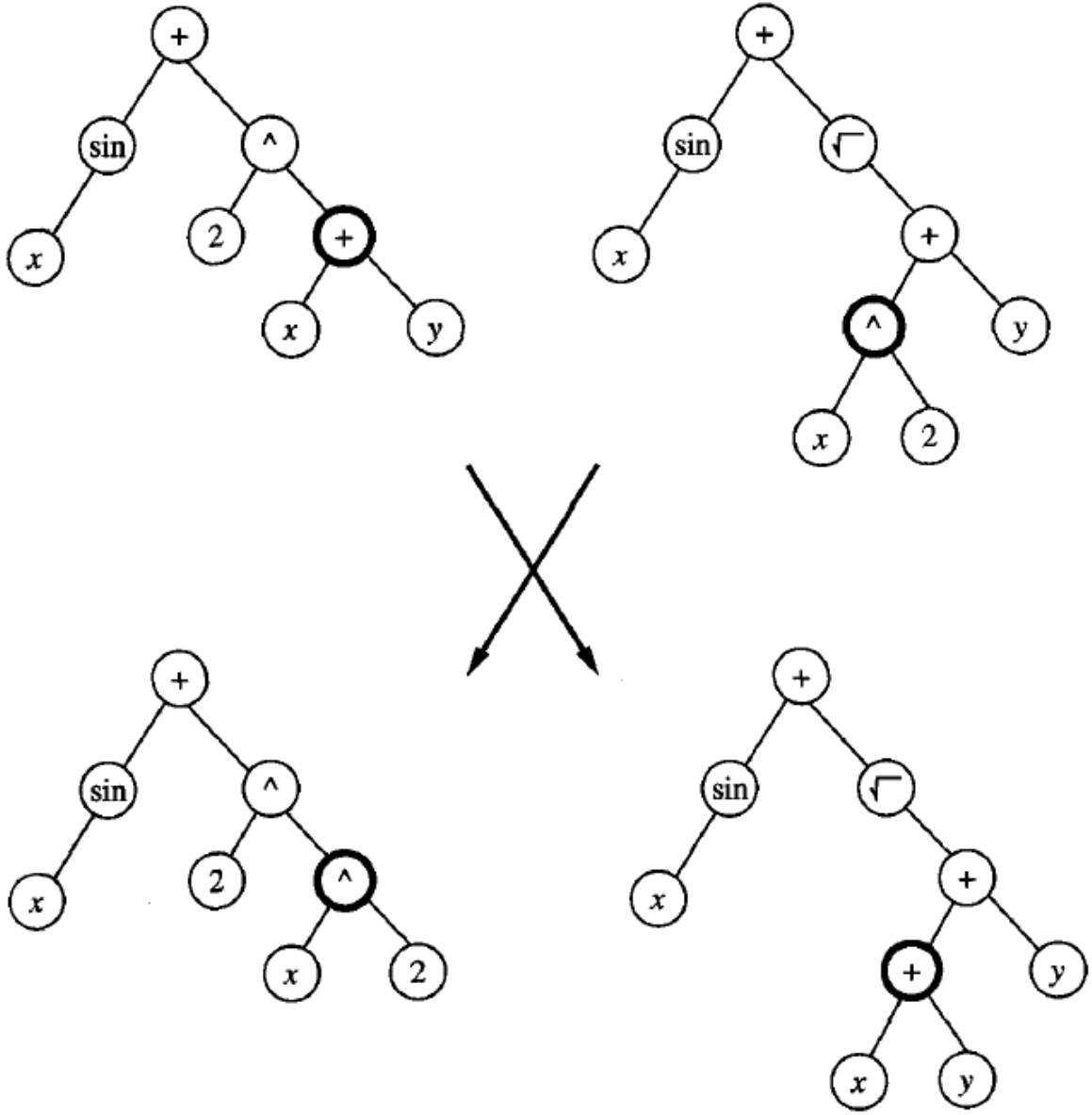
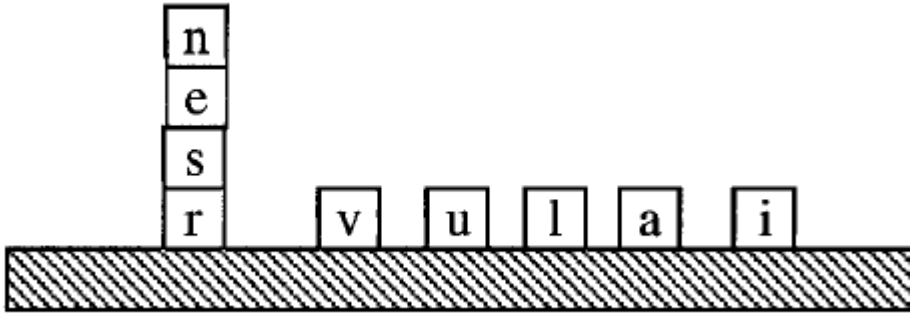


FIGURE 9.1
Program tree representation in genetic programming. Arbitrary programs are represented by their parse trees.


+
sin
^
x
2
+
x
y
+
sin
√
x
+
^
y
x
2
+
sin
^
x
2
^
x
2
+
sin
√
x
+
+
y
x
y

Example

- One illustrative example presented by Koza (1992) involves learning an algorithm for stacking the blocks 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.



stacking the blocks

- 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.
- This particular choice of terminal arguments provides a natural representation for describing programs for manipulating blocks for this task.

primitive functions:

- 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.