Slide 0

Genetic Programming I

Reference book: "Genetic Programming" by John R. Koza (The MIT Press)

What is GP

Slide 1

- an automatic programming (automatic program synthesis or program induction)
- using the principles of Darwinian natural selection like GAs
- with mainly parse tree representation unlike GAs (normally fixed-length strings)
  - program can be easily represented by tree
  - fixed-length strings do not readily provide the hierarchical structure and computational procedure
  - there is no predetermination of the size and shape of solutions (i.e. programs)
- but not restrict to tree nowadays, linear and graph are also used

What is GP (Cont'd)

Slide 2

- LISP is a useful language to implement GP because
  - both programs and data have the same form (i.e., S-expressions)
  - →a computer program can be treated as data in the genetic population
  - a LISP program is its own parse tree (easy tree representation)
  - easy program execution by EVAL function of LISP
  - LISP facilitates the programming of structures whose size and shape change dynamically
  - PRINT function provides ways to present parse trees

What is GP (Cont'd)

Slide 3

- LISP S-expression

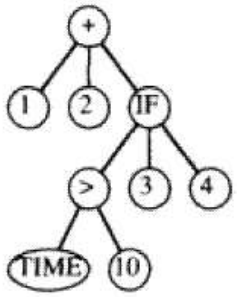


Figure 4.1
The LISP S-expression
`(+ 1 2 (IF (> TIME 10) 3 4))`
depicted as a rooted, point-labeled
tree with ordered branches.

Overview of GP

Slide 4

- Viewpoints
  - for artificial intelligence
  - requirements
    - automatically discovery of a computer program that produces some desired output for particular inputs
  - one approach
    - search a highly fit individual computer program in the space of possible computer programs
    - i.e., machine learning problem →search problem
    - solving problems becomes equivalent to searching a space of possible computer programs for the fittest individual computer program
    - search problem: search algorithm + data or elements
    - in GP: GAs as search algorithm + computer programs as elements (in population pool)

Overview of GP (Cont'd)

Slide 5

- GP
  - evolves individual computer programs in population pool
  - provides a way to search for this fittest individual computer program (i.e., an automatic programming tool)
  - breed populations of hundreds or thousands of computer programs by Darwinian principle with a genetic recombination (crossover)

Overview of GP (Cont'd)

Slide 6

- GP procedure
  - starts with an initial population of randomly generated computer programs composed of functions and terminals appropriate to the problem domain
    - functions
      - standard arithmetic operations
      - standard programming operations
      - standard mathematical functions
      - logical functions
      - domain-specific functions
    - terminals
      - Boolean-valued
      - integer-valued
      - real-valued
      - complex-valued
      - vector-valued
      - symbolic-valued
      - multiple-valued

Overview of GP (Cont'd)

Slide 7

  - each individual computer program is measured in terms of how well it performs (fitness measure)
  - each computer program is run over a number of different fitness cases so that its fitness is measured as a sum or an average over a variety of different situations
    - fitness cases
      - a sampling of different values of an independent variable
      - a sampling of different initial conditions of a system
      - for example,

        fitness: the sum of the value of differences between the output produced by the program and the correct answer

        fitness cases: a sampling of 50 different inputs to the program randomly or systematically

Overview of GP (Cont'd)

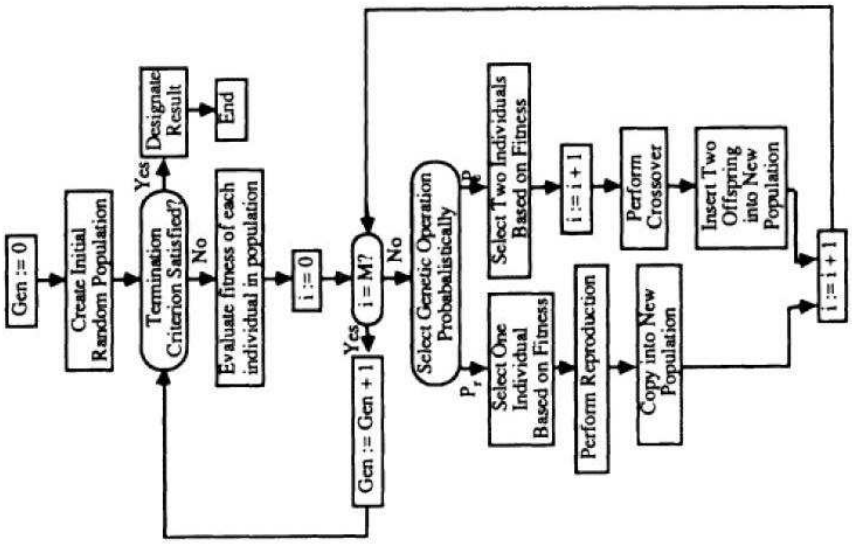
Slide 8

- create a new offspring by Darwinian principle of reproduction and survival of the fittest and genetic operation of recombination (crossover)
  - reproduction operation
    - selecting a computer program in proportion to fitness
    - allowing it to survive by copying it into the new population
  - offsprings programs
    - composed of subexpressions (subtrees, subprograms, subroutines, building blocks)
    - have different sizes and shapes
- offsprings after reproduction and crossover replace the old population
- go to the second step until termination condition is satisfied
- ☛ the best computer program (the best-so-far individual) appeared in any generation is a solution to the problem

Overview of GP (Cont'd)

Slide 9

- GP algorithm



Detailed description of GP

Structures undergoing adaptation

- possible compositions of functions
  - composed recursively from the set of N_{func} functions from $F = \{f_1, f_2, \ldots, f_{N_{func}}\}$ and the set of N_{term} terminals from $T = \{a_1, a_2, \ldots, a_{N_{term}}\}$

Slide 10

  - each function f_i takes a specified number of arguments $z(f_1), z(f_2), \ldots, z(f_{N_{func}})$
  - functions
    - arithmetic operations ($+$, $-$, $*$, etc.)
    - mathematical functions (such as sin, cos, exp, and log)
    - Boolean operations (such as AND, OR, NOT)
    - conditional operators (such as If-Then-Else)
    - functions causing iteration (such as Do-Until)
    - functions causing recursion, and
    - any other domain-specific functions that may be defined

Structures undergoing adaptation (Cont'd)

  - terminals
    - either
      - variable atoms (representing, perhaps, the inputs, sensors, detectors, or state variables of some system)
        or
      - constant atoms (such as the number 3 or the Boolean constant NIL)
    - occasionally, functions taking no explicit arguments

Slide 11

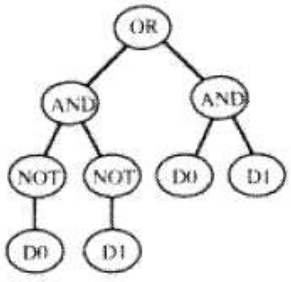
Structures undergoing adaptation (Cont'd)

Slide 12

- an example
  - $F = \{AND, OR, NOT\}$
  - $T = \{D0, D1\}$, where $D0$ and $D1$ are Boolean variable atoms that serve as arguments for the functions
  - combined set C: $C = F \cup T = \{AND, OR, NOT, D0, D1\}$
  - terminals in C can be viewed as zero argument functions
  - consider even-2-parity function (equivalence function) that returns T if an even number of its arguments are T
    - $F = D0D1 + D0'D1'$
    - S-expression: (OR (AND (NOT D0) (NOT D1)) (AND D0 D1))
    - disjunctive normal form (DNF)

Structures undergoing adaptation (Cont'd)

Slide 13



- search space
  - all possible LISP S-expressions recursively created by compositions of the available functions and terminals

Closure of the function set and terminal set

Slide 14

- closure property
  - each of the functions can accept
    - any value and data type returned by any functions in the function set
    - any value and data type assumed by any terminals in the terminal set
- simple example
  - function set: Boolean functions such as AND, OR, and NOT
  - closure property
    - AND, OR, and NOT functions generate Boolean values, T or NIL
    - terminal set consists of Boolean variables

Closure of the function set and terminal set (Cont'd)

Slide 15

- ordinary programs
  - arithmetic operations operating on numerical variables are sometimes undefined (e.g., division by zero)
  - many common mathematical functions operating on numerical variables are sometimes undefined (e.g., logarithm of zero)
  - the value returned by mathematical functions may be an unacceptable data type (e.g., square root or logarithm of a negative number)
  - Boolean value is generally not acceptable as the argument to an ordinary arithmetic operation
  - ☛ in ordinary programs, closure property may be impossible or very complex and restrictive syntactic
  - ☛ but, closure property can be achieved merely by careful handling of a small number of situations

Closure of the function set and terminal set (Cont'd)

Slide 16

- solutions
  - divide by zero
    - define a protected division function
      returns one if division by 0, otherwise returns normal quotient

```
(defun % (numerator denominator)
  "The Protected Division Function"
  (if (= 0 denominator) 1 (/ numerator denominator)))
```

    - return the symbolic value :undefined
    - change ordinary arithmetic functions so as to return :undefined when they encounter :undefined

Closure of the function set and terminal set (Cont'd)

Slide 17

  - square root or logarithm of a negative number and functions are unacceptable complex number
    - define a protected square root function

```
(defun srt (argument)
  "The Protected Square Root Function"
  (sqrt (abs argument)))
```

    - define a protected logarithm function

```
(defun rlog (argument)
  "The Protected Natural Logarithm Function"
  (if (= 0 argument) 0 (log (abs argument))))
```

Closure of the function set and terminal set (Cont'd)

Slide 18

- a function does not accept Boolean value
  - numerical-valued logic (return +1 and -1 or 1 and 0) can be used

```
(defun gt (first-argument second-argument)
  "The numerically-valued greater-than function"
  (if (> first-argument second-argument) 1 -1))
```

  - conditional comparative operators can be redefined
    - for example, IFLTZ (If Less Than Zero) returns T or NIL
    - redefine ifltz with three arguments: (defun ifltz (first-argu second-argu third-argu) ...)
    - but, at function implementation, all arguments are evaluated
      →macro implementation

```
#+TI (setf sys:inhibit-displacing-flag t)

(defmacro ifltz (first-argu then-argu else-argu)
  (if (< (eval first-argu) 0)
      (eval then-argu)
  (eval else-argu)))
```

Closure of the function set and terminal set (Cont'd)

Slide 19

  - conditional branching operators can be redefined
    - (IF-FOOD-AHEAD (MOVE) (TURN-RIGHT)) →occur both move
      →macro definition

```
#+TI (setf sys:inhibit-displacing-flag t)

(defmacro if-food-ahead (then-argu else-argu)
  (if *food-directly-in-front-of-ant-p*
      (eval then-argu)
  (eval else-argu)))
```

- conclusions
  - closure property is desirable, but not absolutely required
  - if impossible, then one method is to discard individuals or give some penalty

Sufficiency of the function set and terminal set

Slide 20

- the set of functions and terminals are capable of expressing a solution
- i.e., some composition of the functions and terminals can yield a solution
- for example,
  - in the domain of Boolean functions
    - function set: $F = \{AND, OR, NOT\}$
    - remove OR →still possible
    - remove NOT →no longer sufficient

Universality of selecting primitive functions and terminals

Slide 21

- irrelevant and extraneous functions will probably be degraded the performance
- for example
  - three function sets are all sufficient
    - $\{AND,\ OR,\ NOT\}$
    - $\{IF,\ AND,\ OR,\ NOT\}$: good for human understanding
    - $\{NAND,\ NOR\}$: good for semiconductor layouts
  - XOR can be an extraneous function
- discussion
  - the effect on performance of extraneous functions is complex
  - in general, numerous extraneous functions degrade performance more or less
  - but, a particular additional function may dramatically improve performance
    - for example, adding an extraneous function IF to $\{OR, NOT\}$ improves the performance
  - the effect on performance of extraneous terminals is clearer than the effect of extraneous functions →usually, extraneous terminals reduce performance

the initial structure

Slide 22

- consist of the individuals of S-expressions
- the S-expressions are generated by randomly generating a rooted, point-labeled tree
- steps
  - select one of the functions from the set F at random (using a uniform random probability distribution)

    restrict the selection to the function set F in order to generate a hierarchical structure, not a degenerate structure consisting of a single terminal
  - combined set $C = F \cup T$ of functions and terminals is randomly selected to be the label for $z(f)$ lines of the root function
  - if a function is chosen, then the generating process continues recursively
  - if a terminal is chosen, that points becomes an endpoint of the tree and the generating process is terminated for that point

the initial structure (Cont'd)

Slide 23

Figure 6.2
Beginning of the creation of a random program tree, with the function + with two arguments chosen for the root of the tree.

the initial structure (Cont'd)

Slide 24

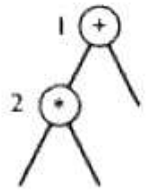


Figure 6.3
Continuation of the creation of a random program tree, with the function * with two arguments chosen for point 2.

the initial structure (Cont'd)

Slide 25

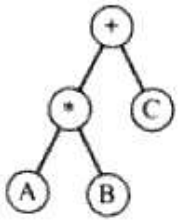


Figure 6.4
Completion of the creation of a random program tree, with the terminals A, B, and C chosen.

the initial structure (Cont'd)

Slide 26

- two methods
  - full method
    - the length of every path from root to endpoint is equal to the specified maximum depth
    - selecting from a function set at less depth than maximum and from a terminal set at maximum depth
  - grow method
    - the length of path from root to endpoint is no greater than the maximum depth
    - selecting from a combined set $C = F \cup T$ at less depth than maximum and from a terminal set at maximum depth
  - ramped half-and-half method
    - mixed method of full and grow (used for all problems in this book)
    - for example (maximum depth: 6)
      - 20% for depth 2, 20% for depth 3, ..., 20% for depth 6
      - 50% of the trees are created via full and 50% via grow
    - create tree having a wide variety of sizes and shapes

the initial structure (Cont'd)

Slide 27

- duplicate individuals
  - are unproductive deadwood
  - waste computational resources
  - undesirably reduces the genetic diversity
  - ☛ avoiding duplication is desirable, but not necessary
  - ☛ newly created S-expression is checked for uniqueness

fitness

- is the driving force of Darwinian natural selection
- types
  - raw fitness
  - standardized fitness
  - adjusted fitness
  - normalized fitness

Slide 28

- raw fitness
  - evaluated over a set of fitness cases
  - effect of selecting particular fitness cases can be minimized by using a different fitness cases in each generation
  - the most common definition is that raw fitness is error

    →the sum of the distances between the point of S-expression and the correct point over all fitness cases

fitness (Cont'd)

- two cases
  - ∗ S-expression is integer-valued or floating-point-valued
    - $r(i,t) = \sum_{j=1}^{N_e} |S(i,j) - C(j)|$ or
    - $r(i,t) = \sum_{j=1}^{N_e} \sqrt{(S(i,j) - C(j))^2}$, where $S(i,j)$ is the value returned by S-expression i for fitness cases j and $C(j)$ is the correct value for fitness case j
  - ∗ S-expression is Boolean-valued or symbolic-valued

  → the sum of distances is equivalent to the number of mismatches
  - ∗ S-expression is complex-valued or vector-valued or multiple-valued

  → the sum of distances separately obtained from each component
- the better, the smaller (error) the larger (food eaten, benefit achieved, etc.)

Slide 29

fitness (Cont'd)

Slide 30

- standardized fitness
  - restates the raw fitness so that a lower value is always a better value
  - the better, the smaller (error): standardized fitness equals the raw fitness $s(i,t) = r(i,t)$
  - the better, the larger: $s(i,t) = r_{max} - r(i,t)$, where r_{max} is possible maximum value of raw fitness
- adjusted fitness
  - $a(i,t) = \frac{1}{1+s(i,t)}$
  - lies between 0 and 1 and bigger for better individual
  - good for
    - if standard fitness can range between 0 (the best) and 64 (the worst)
    - adjust fitness of scoring 64 and 63 is 0.0154 and 0.0159
    - adjust fitness of scoring 4 and 3 is 0.20 and 0.25
    - focus on the good individuals

fitness (Cont'd)

Slide 31

- normalized fitness
  - if the selection is fitness proportionate, then the normalized fitness is also needed
  - $n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}$
  - desirable characteristics
    - it ranges between 0 and 1
    - it is larger for better
    - the sum of normalized fitness is 1

fitness (Cont'd)

Slide 32

- Greedy over-selection
  - motivation
    - the population size M of 500 is sufficient about two-thirds of the problems in this book
    - more complex problems generally require larger population size
    - greedy over-selection considerably enhances the performance of GP
  - method
    - the fitter individuals are given a better chance of selection
    - apply only when the population size is 1,000 or larger but not 500 or below
  - implementation
    - for a population size of 1,000
    - individuals are sorted in order of their normalized fitness $n(i,t)$
    - the 32% of fittest individuals are placed in group I, the remaining are placed in group II
    - select 80% of the time from group I in proportion to its normalized fitness and 20% of the time from group II

Primary operations for modifying structures

Slide 33

- two primary operations
  - Darwinian reproduction
  - crossover (sexual recombination)
- reproduction
  - the basic engine of Darwinian natural selection and survival of the fittest
  - asexual operation in that it operates on only one parental S-expression
  - two steps
    - a single S-expression is selected from the population according to selection method based on fitness
    - selected individual is copied without alteration from the current population into the new population

Primary operations for modifying structures (Cont'd)

Slide 34

- selection based on fitness
  - * the probability that individual s_i will be copied $\frac{f(s_i(t))}{\sum_{j=1}^{M} f(s_j(t))}$, where $f(s_i(t))$ is the normalized fitness
  - ☛ called *fitness-proportionate reproduction*
- rank selection
  - * based on the rank (not the numerical value) of fitness
  - * reduce the potentially dominating effects of high-fitness individuals
  - * select individuals in proportion to the rank (the higher rank, the larger probability)
- tournament selection
  - * a specified group of individuals (from 2 to population size) are chosen at random
  - * select the best individual from the group and discard all others

Primary operations for modifying structures (Cont'd)

Slide 35

- crossover
  - the first and second parent are chosen by fitness-based selection
  - select one random point in each parent using uniformly distribution
  - offsprings are produced by exchange the crossover fragments of two parents

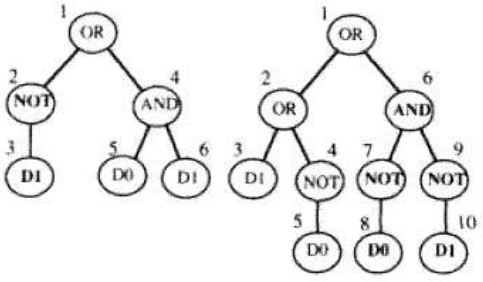


Figure 6.5
Two parental computer programs.

Primary operations for modifying structures (Cont'd)

Slide 36

- S-expressions
  - parents
    (OR **(NOT D1)** (AND D0 D1))
    (OR (OR D1 (NOT D0)) **(AND (NOT D0) (NOT D1))**)
  - offsprings
    (OR **(AND (NOT D0) (NOT D1))** (AND D0 D1))
    (OR (OR D1 (NOT D0)) **(NOT D1)**)

Primary operations for modifying structures (Cont'd)

Slide 37

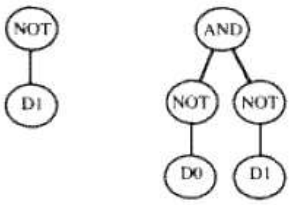


Figure 6.6
The crossover fragments resulting from selection of point 2 of the first parent and point 6 of the second parent as crossover points.

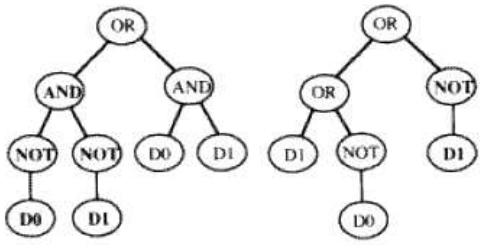


Figure 6.7
The two offspring produced by crossover.

Primary operations for modifying structures (Cont'd)

Slide 38

- cases
  - if a terminal is located at the crossover point of first parent
    - subtree of second parent is inserted at the location of first parent
    - the terminal of first parent is inserted at the location of subtree
  - if terminals are located at both crossover points
    - merely swaps these terminals from tree to tree
  - if the root of one parent is selected as a crossover point
    - the entire first parent is inserted into the second parent at the point
    - subtree of second parent become the other offspring
  - rarely, if the root of first parent and a terminal of second parent are selected
    - the entire first parent is inserted into the second parent at the point
    - a terminal of second parent become the other offspring
  - if the roots of two parents are selected →reproduction of those two parents
  - if an individual mates with itself or two identical individuals mate
    - two offsprings will be different if the crossover points are different
    - two offsprings will be same at GAs →make GAs premature convergence
    - this gives less effect to GP

Primary operations for modifying structures (Cont'd)

Slide 39

- maximum permissible size
  - an allowable depth of tree
  - prevent the large amount of computer time on a few extremely large individuals
  - if an offspring is impermissible size
    - the offspring is ignored and the first of its parents is inserted into the new population
    - if the other offspring is permissible, then it is inserted into the new population
  - if two offsprings are impermissible size →both parents are inserted into the new population
  - default value is 17
    - the largest permissible LISP program contains $2^{17} = 131,072$ functions and terminals
    - if four LISP functions and terminals are roughly equivalent to one line of a conventional program →33,000 lines
    - for the problems in this book, results contain about 500 functions and terminals (125 lines)

Secondary operations

Slide 40

- five optional secondary operations
  - mutation
  - permutation
  - editing
  - encapsulation
  - decimation

Secondary operations (Cont'd)

Slide 41

- mutation
  - random changes in structures
  - usefulness
    - in GAs, mutation gives benefit in diversity preventing from premature convergence but relatively unimportant operation (small)
    - in GP, GP keeps diversity and crossover operation of GP is sufficient (very very small)
    - GP in this book does not use mutation
  - steps
    - select an individual with a probability proportional to the normalized fitness
    - select a point at random within S-expression
    - remove the subtree and inserts a randomly generated subtree at that point
    - maximum size
      - allowable depth for newly created subtree
      - usually, the same as the maximum initial size in initial population

Secondary operations (Cont'd)

Slide 42

– an example

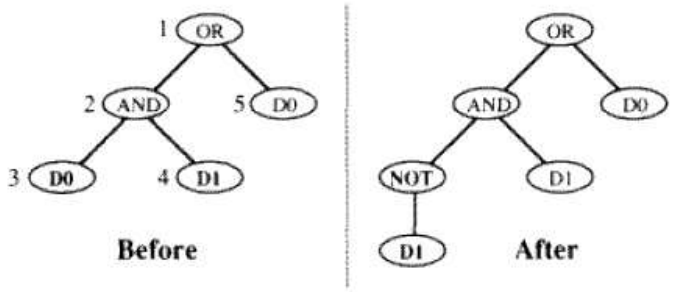


Figure 6.8
A computer program before and after the mutation operation is performed at point 3.

Secondary operations (Cont'd)

Slide 43

- permutation
  - generalization of the inversion operation for GAs operating on strings
  - inversion
    - inversely reorder characters between two selected points of an individual
  - steps
    - select one S-expression in the same way for reproduction and crossover (a probability proportional to the normalized fitness)
    - select a function (internal) point of S-expression at random
    - if the function has k arguments, select a permutation from the $k!$ possible permutations
    - if the function is commutative, then there is no effect on this permutation
  - GP in this book does not use permutation

Secondary operations (Cont'd)

Slide 44

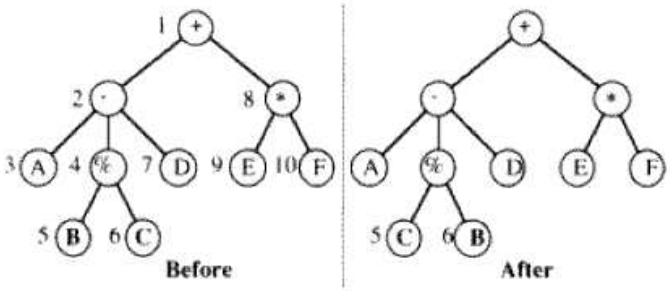


Figure 6.9
An S-expression before and after the permutation operation is performed at point 4 containing the protected division function %.

Secondary operations (Cont'd)

Slide 45

- editing
  - provide a means to edit and simplify S-expression
  - asexual operation
  - recursively applies a pre-established set of domain-independent and domain-specific editing rules
  - universal domain-independent rule
    - if any function has no side effects and is not context dependent and has only constant atoms as arguments, it replaces with the value obtained from evaluation
    - for example
      - (+ 1 2) →3
      - (AND T T) →T

Secondary operations (Cont'd)

Slide 46

- pre-established domain-specific rules
  - for numeric problems, insert 0 whenever a subexpression is subtracted from itself
  - in Boolean domains, one might use editing rules
    - (AND X X) →X
    - (OR X X) →X
    - (NOT (NOT X)) →X
    - De Morgan's laws
- two distinct ways
  - used entirely external to the run to make the output of displayed individuals more readable
  - used during the run in an attempt either to produce simplified output or to improve the overall performance

Secondary operations (Cont'd)

Slide 47

- operations
  - operate on each individual in the population
  - frequency parameter f_{ed}
    - if 1, then apply to all generations
    - if 0, then apply to no generations
    - if n, where $n > 1$, apply to every n generations

Secondary operations (Cont'd)

Slide 48

- encapsulation
  - a means for automatically identifying a potentially useful subtree
  - give it a name and referenced and used later
  - a key issue in AI
    - how to scale up promising techniques to solve subproblems to larger problems
    - one way is to decompose it into hierarchy of smaller subproblems
    - identifying smaller subproblems is the key step
    - automating this identifying subproblems is important goal of AI
  - steps
    - select an individual with a probability proportional to the normalized fitness
    - select a function (internal) point of S-expression
    - remove the subtree located at the point
    - define a new function to permit references to the deleted tree
    - the new encapsulated function has no arguments
    - named $E0, E1, E2, E3, \ldots,$

Secondary operations (Cont'd)

Slide 49

  - for example
    - a LISP S-expression: (+ A (* B C))
    - point 3 was selected
    - encapsulated function $E0$

```
(defun E0 ()
  (* B C)
)
```

    - (* B C) subtree is replaced by $E0$ with no arguments
    - result S-expression: (+ A (E0))
    - in effect, the call ($E0$) has become a new individual atom (terminal)

Secondary operations (Cont'd)

Slide 50

- effect of encapsulation
  - * the selected subtree has no longer disruptive effects of crossover because it is a single point
  - * a potential building block for future generations
  - * original parent is not changed by the operation and participate in additional genetic operations
- name
  - * in earlier, called "define building block" and encapsulated function "defined function"
  - * now, called "automatically defined function" (ADF)

Secondary operations (Cont'd)

Slide 51

- decimation
  - situation
    - * in initial random population
      - · a very large percentage of the individuals have very poor fitness
        →enormous amounts of computer time may wasted for poor individuals
      - · the few individuals with marginally better fitness begin to dominate the population
  - a faster way to deal with this situation
  - two parameters controlling when the operation is to be invoked
    - * a percentage and a condition
    - * for example
      - · 10% and generation 0
        →after the fitness calculation for generation 0, 10% of the population is deleted
  - operation is done probabilistically on the basis of fitness

Termination criterion

Slide 52

- GP terminates
  - when either a prespecified maximum number G have been run (generational predicate)
  - or some additional problem-specific success predicate has been satisfied
- cases
  - success predicate →some individual has a standardized fitness of 0
  - appropriate lower criterion for success
    - where we may not recognize a solution (e.g., optimization problems)
    - where we do not ever expect an exact solution (e.g., creating a mathematical model for noisy empirical data)
  - no success predicate
    - we merely analyze the results after running for G generations

Result designation

Slide 53

- in this book
  - use the best individual that ever appeared in any generation (i.e., the best-so-far individual)
  - no use of so-called elitism
- an alternative method
  - the best-of-generation individual at the time of termination
  - no caching is required
  - usually produce the same results as the best-so-far method

Control parameters

Slide 54

- GP is controlled by 19 control parameters
  - two major numerical parameters
  - 11 minor numerical parameters
  - six qualitative variables
- two major numerical parameters
  - the population size M is 500
  - the maximum number G of generations is 51 (initial called 0, plus 50 generations)

Control parameters (Cont'd)

Slide 55

- eleven minor numerical parameters
  - the probability of crossover p_c is 0.90 (e.g., $M = 500$, 450 individual (225 pairs))
  - the probability of reproduction p_r is 0.10 (e.g., $M = 500$, 50 individual with reselection allowed)
  - in selecting crossover points, 90% for internal (function) points and 10% for external (terminal) points
    - uniform probability distribution makes merely swapping of terminals
  - a maximum size $D_{created}$ is 17 for individuals created by the crossover
  - a maximum size $D_{initial}$ is 6 for initial individuals
  - the probability of mutation p_m is 0
  - the probability of permutation p_p is 0
  - the frequency f_{ed} for editing is 0
  - the probability of encapsulation p_{en} is 0
  - the condition for invoking the decimation is NIL
  - the decimation percentage p_d is NIL

Control parameters (Cont'd)

Slide 56

- six qualitative variables
  - the generative method for initial random population is ramped half-and-half
  - the method of selecting for reproduction and for the first parent in crossover is fitness-proportionate reproduction
  - the method of selecting the second parent for a crossover is the same as the method for selecting the first parent
  - optional adjusted fitness measure is used
  - over-selection is not used for population of 500 and below and is used for populations of 1,000 and above
  - the elitism strategy is not used

Four introductory examples of GP

Slide 57

- examples
  - optimal control: evolve a control strategy that brings a card moving along a track to rest at a designated target point in minimal time
  - robotic planning: evolve a robotic action plan that enables an artificial ant to find all the food along a trail
  - symbolic regression: evolve a mathematical expression that closely fits a given finite sample of data
  - Boolean 11-multiplexer: evolve a Boolean expression that performs the Boolean 11-multiplexer function
- five major steps for using GP
  - determining the set of terminals
  - determining the set of functions
  - determining the fitness measure
  - determining the parameters and variables
  - determining the method of designating a result and the termination criterion

Four introductory examples of GP (Cont'd)

Slide 58

- Boolean 11-multiplexer
  - input
    - N-multiplexer consists of k address bits and 2^k data bits d_i, where $N = k + 2^k$

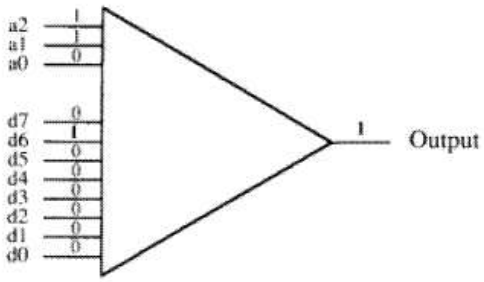


Figure 7.27
Boolean 11-multiplexer with input of 11001000000 and output of 1.

Four introductory examples of GP (Cont'd)

Slide 59

  - steps
    - determining the set of terminals: $T = \{A0, A1, A2, D0, D1, \ldots, D7\}$
    - determining the set of functions: $F = \{AND, OR, NOT, IF\}$
    - determining the fitness measure
      - use 2048 fitness cases (combinations of inputs)
      - no sampling
      - raw fitness
        correct Boolean output value
        from 0 to 2048
      - standard fitness
        r_{max} - raw fitness
    - determining the parameters and variables
      - population size: 4,000
    - determining the method of designating a result and the criterion for terminating
      - terminate a run after a specified maximum number of generation G
      - or earlier if the standard fitness of 0 is found

Four introductory examples of GP (Cont'd)

Slide 60

- search space
  - if $k = 3$, then $k + 2^k = 11$, 11 inputs
  - 2^{11} possible inputs →possible functions $2^{2^{11}} = 2^{2048} \approx 10^{616}$
- initial individuals
  - merely constant such as contradictory: (AND A0 (NOT A0))
  - pass a single input: (NOT (NOT A1))
  - inefficient: (OR D7 D7)
  - wrong arguments: (IF D0 A0 A2)
  - nonsense: (IF (IF (IF D2 D2 D2) D2 D2) D2 D2)
  - the worst individual for generation 0
    →(OR (NOT A1) (NOT (IF (AND A2 A0) D7 D3)))
  - high scoring: (IF A0 D1 D2)

Four introductory examples of GP (Cont'd)

Slide 61

- at generation 0
  - average standard fitness is 985.4
  - hits histogram

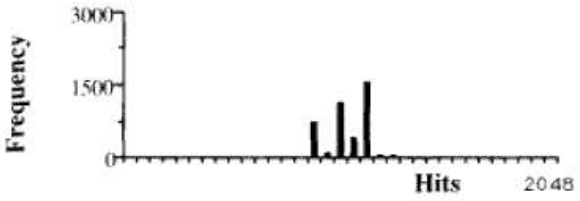


Figure 7.29
Hits histogram for generation 0
of the 11-multiplexer problem.

Four introductory examples of GP (Cont'd)

Slide 62

- after generation 1
  - average standard fitness

generation	1	2	3	4	5	6	7	8	9
value	891	845	823	763	731	651	558	459	382

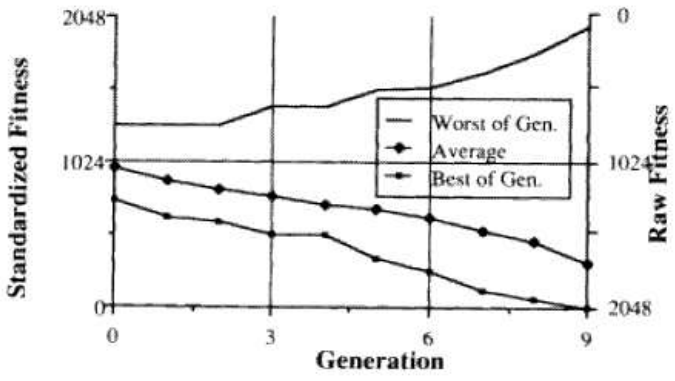
  - best-of-generation

generation	1	2	3	4	5	6	7	8	9
value	640	640	576	384	384	256	256	128	0

Four introductory examples of GP (Cont'd)

Slide 63

- fitness curves



Four introductory examples of GP (Cont'd)

Slide 64

- at generation 1
  - standard fitness of best-of-generation is 640
  - S-expression: (IF A0 (IF A2 D7 D3) D0)
    - consider two address bits
    - incorporate three of the eight data bits
- at generation 2
  - no better best-of-generation individual than generation 1, but population is improved
- at generation 3
  - standard fitness of best-of-generation is 576
  - S-expression: (IF A2 (IF A0 D7 D4) (AND (IF (IF A2 (NOT D5) A0) D3 D2) D2))
    - contain both AND and NOT functions

Four introductory examples of GP (Cont'd)

Slide 65

- at generation 4, 5
  - standard fitness of best-of-generation is 384
  - one of the S-expressions of best individuals
    (IF A0 (IF A2 D7 D3) (IF A2 D4 (IF A1 D2 (IF A2 D7 D0))))
    - use all three address bits
    - use only data bits as the second and third arguments
- at generation 6, 7, 8: population is improved
- at generation 9
  - standard fitness of best-of-generation is 0 (finding solution)
  - S-expression:

```
(IF A0 (IF A2 (IF A1 D7 (IF A0 D5 D0))
              (IF A0 (IF A1 (IF A2 D7 D3) D1) D0))
              (IF A2 (IF A1 D6 D4)
              (IF A2 D4 (IF A1 D2 (IF A2 D7 D0)))))
```

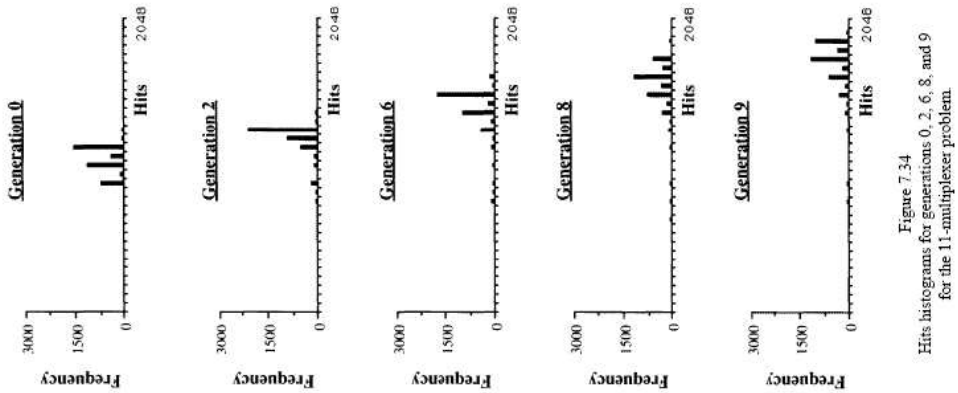
  - simplified either manually or via the editing operation

```
(IF A0 (IF A2 (IF A1 D7 D5) (IF A1 D3 D1))
              (IF A2 (If A1 D6 D4) (IF A1 D2 D0)))
```

Four introductory examples of GP (Cont'd)

Slide 66

- histogram



Figue 7.34
Hits histograms for generations 0, 2, 6, 8, and 9 for the 11-multiplexer problem

Four introductory examples of GP (Cont'd)

Slide 67

- hierarchies
  - * in generation 8, a subtree of an individual consists of 6-multiplexer

```
(IF A0 (IF A0 (IF A2 (IF A1 D7 (IF A0 D5 D0))
                     (IF A0 (IF A1 (IF A2 D7 D3)
                                   D1)
                            D0))
              (IF A1 D6 D4))
       (IF A2 D4
              (IF A1 D2 (IF A0 D7 (IF A2 D4 D0)))))
```

  - → 6-multiplexer

```
(IF A2 (IF A1 D7 (IF A0 D5 D0))
       (IF A0 (IF A1 (IF A2 D7 D3)
                     D1)
              D0))
```

Four introductory examples of GP (Cont'd)

Slide 68

* 11-multiplexer: hierarchical composition of 6-multiplexer

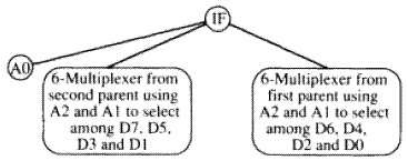


Figure 7.39
The solution to the 11-multiplexer problem is a hierarchical conditional composition of two 6-multiplexers.

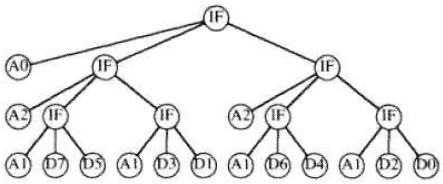


Figure 7.40
The solution to the 11-multiplexer problem is a hierarchical conditional composition of 3-multiplexers (i.e., IF-THEN-ELSE functions).

Four introductory examples of GP (Cont'd)

Slide 69

* 6-multiplexer: hierarchical composition of 3-multiplexer

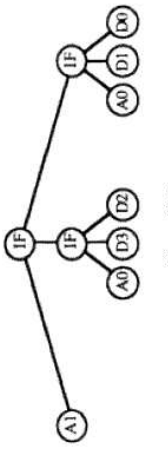


Figure 7.41
The solution to 6-multiplexer is a hierarchy of conditional compositions of 3-multiplexers.

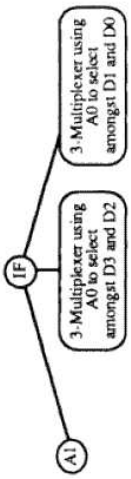


Figure 7.42
The solution to 6-multiplexer is a hierarchical conditional composition of two 3-multiplexers.

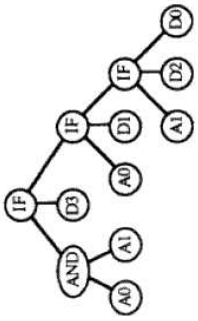


Figure 7.43
A solution to the 6-multiplexer problem containing a default hierarchy.