Graduate Course

Genetic Programming I

Chia-Hsuan Yeh

§: The lecture note is based on Banzhaf et al. (1998) and Koza (1992).

Topic 1: Introduction to Genetic Programming

1. Motivation (1):

   - In Biology, there four essential preconditions for the occurrence of evolution by natural selection:
     - Reproduction of individuals in the population;
     - Variation that affects the likelihood of survival of individuals;
     - Heredity in reproduction (that is, like begets like)
     - Finite resources causing competition
   - Test Tube Evolution – the enzyme $Q\beta$ replicase and RNA:
     - Experiment Results (1):
       * The structure and function of the RNA in the test tubes evolves.
       * The mix of RNA in the last test tube varies – depends on the initial conditions.
       * Different initial conditions result in a final mix specifically adapted to those conditions.
       * The RNA that evolves in these test tube experiments would have been extremely unlikely to evolve by random chance.
     - Experiment Results (2):
       * The end product, "fast RNA", is only 218 bases long and is very *short* compared to the original RNA template. The shorter the RNA, the faster it replicates.
       * The three-dimensional structure of fast RNA makes it especially easy for $Q\beta$ replicase to copy quickly.
     - Question:
       If multiplication and variation are supposedly two necessary preconditions for the occurrence of evolution, how is it that fast RNA evolves?
     - Answer:
       The copies of RNA produced by $Q\beta$ replicase are *not* always perfect copies. It is these errors that introduce variability into the population of RNA. Variants of RNA that reproduce faster in a $Q\beta$ replicase solution have an evolutionary advantage, produce more copies than other types of RNA.

- Implication:
  We may safely conclude that evolutionary search can, therefore, learn good solutions much more rapidly than random search and with no knowledge about what the final product should look like.
- Important lessons:
  * A simple system may evolve as long as the elements of multiplication, variance, and heredity exist.
  * Evolutionary learning may occur in the absent of life or of self-replicating entities.
  * Evolutionary learning may be a very efficient way to explore learning landscapes.
  * Evolution may stagnate unless the system retains the ability to continue to evolve.
  * The selection mechanism for evolutionary learning may be implicitly in the experimental setup or may be explicitly defined by the experimenter.

2. Motivation (2):

   - Representation schemes based on fixed-length character strings do not readily provide the hierarchical structure central to the organization of computer programs (into programs and subroutines) and the organization of behavior (into tasks and subtasks).
   - Representation schemes based on fixed-length character strings do not provide any convenient way of representing arbitrary computational procedures or of incorporating iteration or recursion when these capabilities are desirable or necessary to solve a problem.
   - For many problems, the most natural representation for a solution is a hierarchical computer program rather than a fixed-length character string. The size and the shape of the hierarchical computer program will solve a given problem are generally not known in advance, so the program should have the potential of changing its size and shape.
   - Moreover, such representation schemes do not have dynamic variability. The initial selection of string length limits in advance the number of internal states of the system and limits what the system can learn.
   - "The predetermination of the size and shape of solutions and preidentification of the particular components of solutions has been a bane of machine learning systems from the earliest times." (Samuel, 1959; cited in Koza, 1992. p. 63)
   - The pre-assigned structure and complexity are unable to describe the possible outcomes sufficiently in the real world.

3. Basic Concept:

   - The new technique, Genetic Programming (GP), evolves hierarchical computer programs of dynamically varying size and shape. Computer programs are automatically programed by means of natural selection and evolution in GP.
   - The important feature of GP is that all computer programs can be represented by the parse tree forms. For example, a parse tree form of the computer program,

$$(1 + (\texttt{if } (\texttt{A} > 10) \texttt{ then 3 else 4})),$$

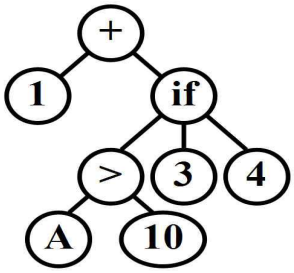

Figure 1: The parse tree of (`1 + (if (A > 10) then 3 else 4)`).

is shown in Figure 1.

- Depending on the particular problem, GP can include several functional forms, for example,
  standard arithmetic operators: +, -, ×, etc.,
  mathematical functions: sin, cos, `exp`, `log`, etc.,
  Boolean operators: `and`, `or`, `not`,
  conditional operators: `if-then-else`,
  and any other domain-specific or pre-defined functions.
- The process of solving a problem can be reformulated as a search for a highly fit individual computer program in the space of possible computer programs. In this way, the process of solving these problems becomes equivalent to searching a space of possible computer programs for the fittest individual computer program. In particular, the search space is the space of all possible computer programs composed of functions and terminals appropriate to the problem domain.
- In GP, populations of hundreds or thousands of computer programs are genetically bred. This breeding is done using the Darwinian principle of survival and reproduction of the fittest along with a genetic recombination (crossover) operator appropriate for mating computer programs.
- Each individual computer program in the population is measured in terms of how well it performs in the particular problem environment. The measure is called *fitness measure*.
- Another important feature of GP is that the absence or relatively minor role of preprocessing of inputs and postprocessing of outputs.

4. Basic steps of executing genetic programming: (Figure 2 is the flowchart for the genetic programming paradigm.)
   - Generate an initial population of random compositions of the functions and terminals of the problem.
   - Iteratively perform the following substeps until the termination criterion has been satisfied:
     - Execute each program in the population and assign it a fitness value according to how well it solves the problem.
     - Create a new population of computer programs by applying the following two primary operators. The operations are applied to computer program(s) in the population chosen with a probability based on fitness.

    - Copy existing computer programs to the new population.
    - Create new computer programs by genetically recombining randomly chosen parts of two existing programs.
- The best computer program that appeared in any generation (i.e., the best-so-far individual) is designated as the result of genetic programming. This result may be a solution (or approximate solution) to the program.

5. Detailed Description of Genetic Programming

- The Structure of Genetic Programming
  - The set of possible structure in genetic programming is the set of all possible compositions of functions that can be composed recursively from the set of N_{func} functions from $F = \{f_1, f_2, ..., f_{N_{\text{func}}}\}$ and the set of N_{term} terminals from $T = \{a_1, a_2, ..., a_{N_{\text{term}}}\}$. Each particular function f_i in the function set F takes a specified number $z(f_i)$ of arguments $z(f_1)$, $z(f_2)$,...,$z(f_{N_{\text{func}}})$.
  - The terminals are typically either variable atoms (representing, perhaps, the inputs, sensors, detector, or state variables of some system) or constant atoms (such as the number "3" or the Boolean constant `NIL`).
- Closure of the Function Set and Terminal Set
  - The *closure* property requires that each of the functions in the function set be able to accept, as its arguments, any value and data type that may possible be returned by any function in the function set and any value and data type that may possibly be assumed by any terminal in the terminal set. That is, each function in the function set should be well defined and closed for any combination of arguments that it may encounter.
  - For example,
    - arithmetic operations operating on numerical variables are sometimes undefined (e.g., division by zero),
    - many common mathematical functions operating on numerical variables are also sometimes undefined (e.g., logarithm of zero),
    - the value returned by many common mathematical functions operating on numerical variables is sometimes a data type that is unacceptable in a particular program (e.g., square root or logarithm of a negative number),
    - the Boolean value (i.e., `TRUE` or `NIL`) typically returned by a conditional operator is generally not acceptable as the argument to an ordinary arithmetic operation.
  - Therefore, we have to modify these operators described above:
    - protected division function (%): consider $A, B \in R$,

$$A\%B = \begin{cases} 1 & \text{if } B = 0, \\ A/B & \text{if } B \neq 0 \end{cases}$$

    - protected natural logarithm function (`Rlog`): Consider $A \in R$,

$$\texttt{Rlog}(A) = \begin{cases} 0 & \text{if } A = 0, \\ \ln|A| & \text{if } A \neq 0 \end{cases}$$

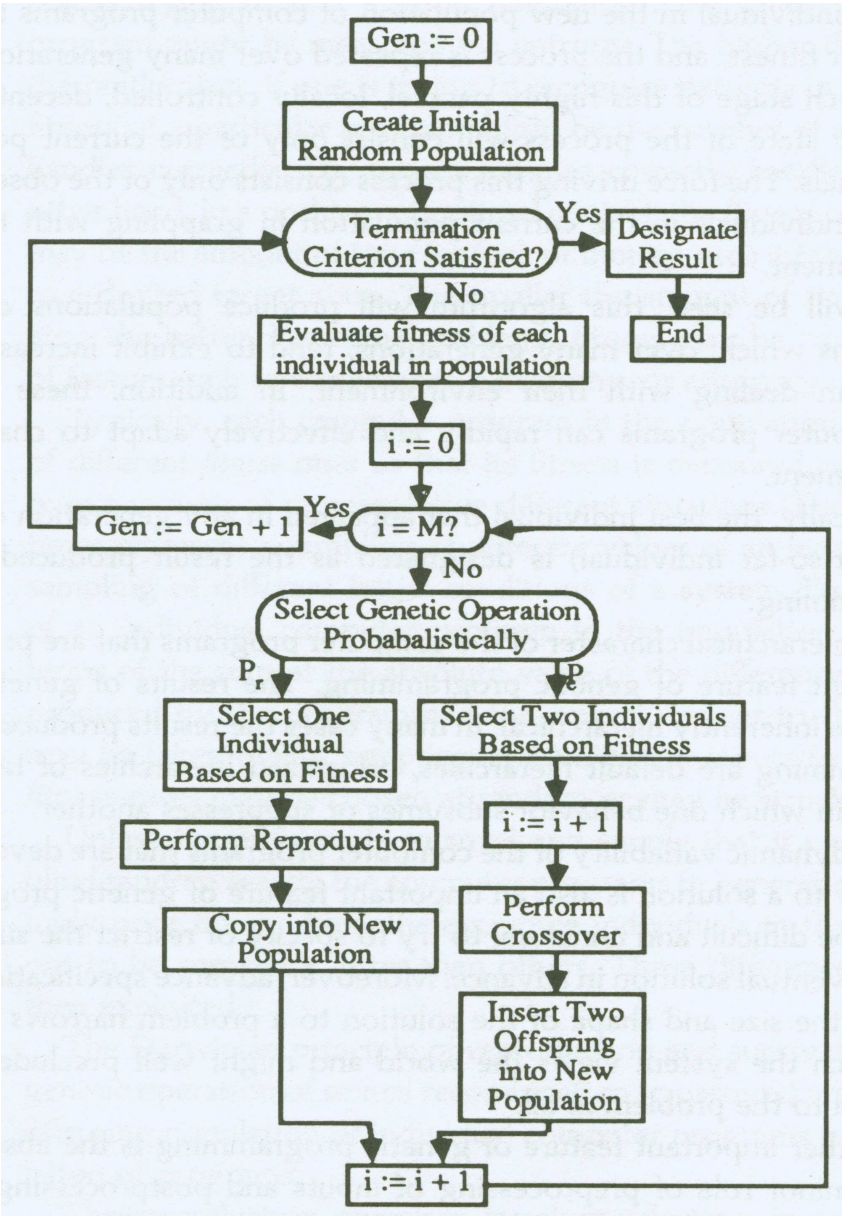

Figure 2: Flowchart for the genetic programming paradigm.

* If a program contains a conditional operator in a problem where the Boolean value that would ordinarily be returned is unacceptable, then the conditional operator can be modified in any one of the following three ways:
  - Numerical-valued logic can be used.
    If numerical-valued logic is used, a numerical-valued conditional comparative operator is defined so as to return numbers (such as +1 and -1 or perhaps 1 and 0) instead of returning Boolean values. For example, the numerical-valued greater-than function `GT` over two arguments would be defined so as to return +1 if its first argument is greater than its second argument and return -1 other.
  - Conditional comparative operators can be redefined.
    A conditional comparative operator can be defined as as to first perform the desired comparison and to then execute an alternative depending on the outcome of the comparison test. For example, the conditional comparative operator `IFLTZ` (If Less Than Zero) can be defined over three argument so as to execute its second argument if its first argument is less 0, but to execute its third argument.
  - Conditional branching operators can be redefined.
    A conditional branching operator can be defined so as to access some state or condition external to the program and then execute an alternative depending on that state or condition. Such an operator returns the result of evaluating whichever argument is actually selected on the basis of the outcome of the test. For example, we define a conditional branching operator to sense for food directly in front of the ant as required in the artificial ant problem, e.g., `(If-Food-Ahead (Move) (Turn-Right))`.
* Sufficiency of the Function Set and Terminal Set
  - The *sufficiency* property requires that the set of terminals and the set of primitive function be capable of expressing a solution to the problem.
  - The user of GP should know or believe that some composition of the functions and terminals he supplied can yield a solution to the problem.

- The Initial Structure
  - The generation of each individual in the initial population is done by randomly generating a rooted, point-labeled tree with ordered branches.
  - The generative process can be implemented in several different ways resulting in initial random trees of different size and shapes. Two of the basic ways are called the "full" method and "grow" method. The depth of a tree is defined as the length of the longest nonbacktracking path from the root to the endpoint.
    * The "full" method of generating the initial random population involves creating trees for which the length of every nonbacktracking path between an endpoint and the root is equal to the specified maximum depth. That is accomplished by restricting the selection of the label for points at depths less than the maximum to the function set, and then restricting the selection of the label for points at the maximum depth to the terminal set.
    * The "grow" method of generating the initial random population involves growing trees that are variably shaped. The length of a path between an endpoint and the root is no greater than the specified maximum depth. This

is accomplished by making the random selection of the label for points at depths less than the maximum from the function set and terminal set, while restricting the random selection of the label for points at the maximum depth to the terminal set. The relative number of functions and the number of terminals determine the expected length of paths between the root and the endpoints of the tree.

  - The generative method that Koza (1992) believe does best over a broad range of problems is a method called "ramped half-and-half". It produces a wide variety of tree of various sizes and shapes.
    * The "ramped half-and-half" generative method is a mixed method that incorporates both the full method and the grow method.
    * It involves creating an equal number of trees using a depth parameter that ranges between 2 and the maximum specified depth.

- Fitness:
  - raw fitness
  - standard fitness
  - adjusted fitness
  - normalized fitness
- Primary Operations for Modifying Structures
  - Reproduction
  - Crossover
    * The *crossover* operation for GP creates variation in the population by producing new offspring that consist of parts taken from each parent.
    * First, two parents are randomly chosen from the population based on the specified selection mechanism.
    * The operation begins by independently selecting, using a uniform probability distribution, one random point in each parent to be the crossover point for that parent.
    * The crossover fragment for a particular parent is the rooted subtree which has as its root the crossover point for that parent and which consists of the entire subtree lying below the crossover point.
    * The two offspring are produced by exchanging two parents' crossover fragments. For example, in the Figure 3, the point 2 and 6 are selected as the crossover points. After exchanging the two crossover fragments, we have two new offspring shown in Figure 4.
  - In the conventional GA, it is very likely to be convergence of the entire population. Once it happened, the only way to change the population is mutation.
  - In contrast, in GP, when an individual incestuously mates with itself, the two resulting offspring will, in general, be different. Thus, convergence of the population is unlikely in GP.
  - A maximum permissible size (measured via the depth of the tree) is established for offspring. This limit prevents the expenditure of large amounts of computer time on a few extremely large individuals.
- Secondary Operations
  - **Mutation**

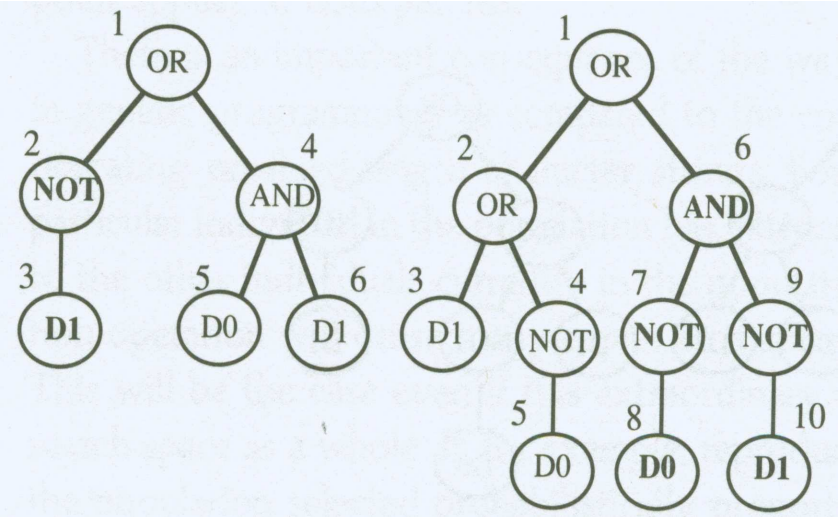

Figure 3: Two parental computer programs.

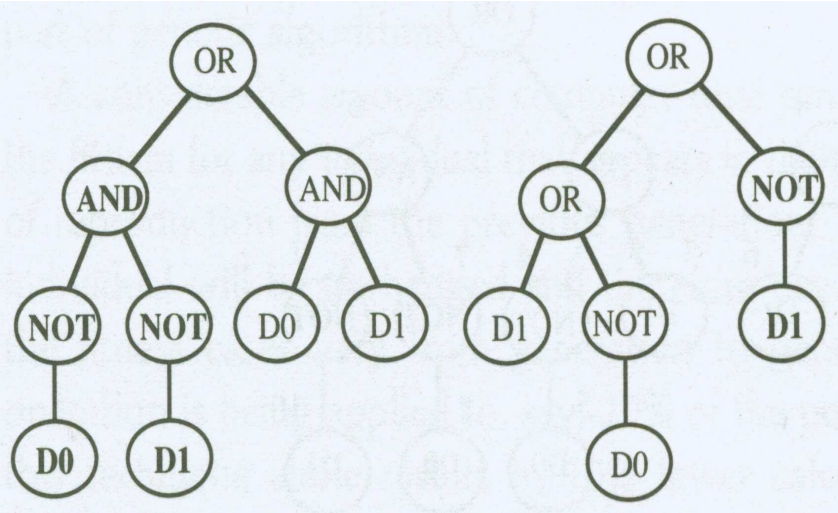

Figure 4: The two offspring produced by crossover.

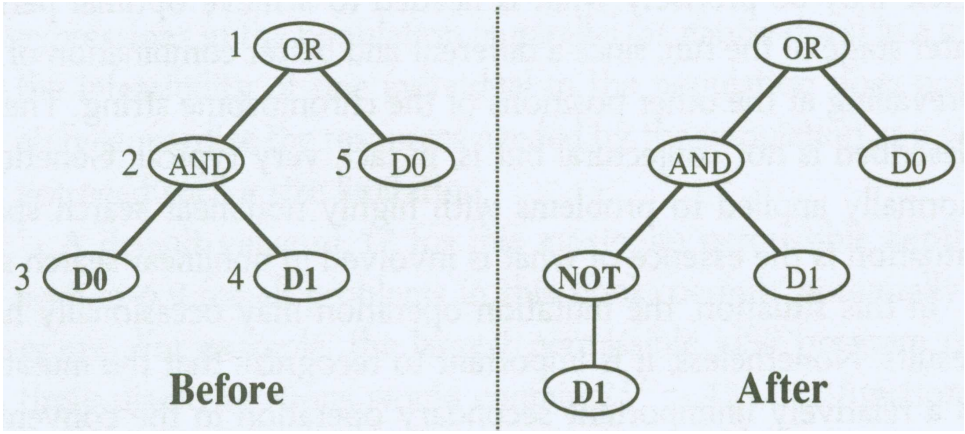

Figure 5: A computer program before and after mutation is performed at point 3.

  - The mutation operation introduces random changes in structures in the population.
  - Tree Mutation:
    - It begins by selecting a point at random within the tree, and then removes whatever is currently at the selected point (*mutation point*) and whatever is below the selected point and inserts a randomly generated subtree at that point. Figure 5 is an example.
    - This operation is controlled by a parameter that specifies the maximum size (measured by depth) for newly created subtree that is to be inserted. This parameter typically has the same value as the parameter for the maximum initial size of trees in the initial random population.
  - Point Mutation:
    - Each node (function node or terminal node) is independently being mutated with a specified probability.
    - The mutation point is replaced by a randomly selected function in the function set or a terminal in the terminal set.
    - However, the replacement should be performed based on the same class member.
    - For example, {+, -, ×, %}, {sin, cos, exp, Rlog}, {x_1, x_2,.., R}.
- Permutation
  - The *permutation* operation is a generalization of the inversion operator for the conventional genetic algorithm operating on strings.
  - It begins by selecting a function point of the program at random. If the function at the selected point has k arguments, a permutation is selected at random from the set of $k!$ possible permutations. Then the arguments of the function at the selected point are permuted in accordance with the random permutation.
- **Encapsulation**
  - The encapsulation operation is a means for automatically identifying a potentially useful subtree and giving it a name so that it can be referenced and used later.

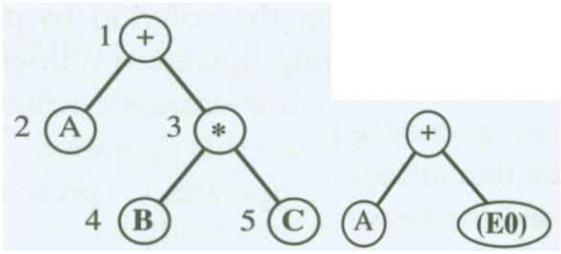

Figure 6: A computer program with point 3 designated as the point for applying the encapsulation operation and its result.

  - ∗ It begins by selecting a function point of the tree at random. The result of this operation is one offspring tree and one new subtree definition, see Figure 6.
  - ∗ The encapsulation operator is called "define building block" operation and the encapsulated function is called the "defined function".
- – Editing
- – Decimation

- Termination Criterion
  - – The genetic programming paradigm parallels nature in that it is a never-ending process. However, as a practical matter, a run of GP terminates when the *termination criterion* is satisfied.
  - – The run terminates when either a prespecified maximum number of generations have been run or some additional problem-specific *success predicate* has been satisfied.
- Result Designation
  - – best-so-far individual,
  - – the population as the whole or a subpopulation selected proportionate to fitness is designated as the result.
- Basic Control Parameters for Genetic Programming:

Population size	N
Maximum number of generation	G
Function set	$\{+, -, \times, \%, \sin, \cos, \texttt{exp}, \texttt{Rlog}, \texttt{Abs}, \texttt{sqrt}\}$
Terminal set	$\{x_1, x_2, ..., x_{10}, R\}$
Probability of reproduction	p_r
Probability of crossover	p_c
Probability of mutation	p_m
Mutation scheme	Tree mutation
Selection mechanism*	Proportional selection
Fitness function*	Sum of square errors
Maximum depth of tree	17
Maximum number in the domain of Exp	1700

Topic 2: Amount of Processing Required to Solve a Problem

1. Both the conventional genetic algorithm and genetic programming involve probabilistic steps at three points in the algorithm, namely

   - creating the initial population,
   - selecting individuals from the population on which to perform each operation,
   - selecting a point within the selected individual at which to perform the genetic operation.

2. Because of these probabilistic steps, anything can happen and nothing is guaranteed for any given run. In particular, there is no guarantee that a given run will yield an individual that satisfies the success predicate of the problem after being run for a particular number of generations.

3. The exponentially increasing allocation of future trials resulting from Darwinian fitness-proportionate selection is both a strength and a weakness of genetic methods. This Darwinian allocation is a weakness of genetic methods because it may result in premature convergence; it is a strength because it is the fundamental reason why genetic methods work in the first place.

4. Non-convergence and premature convergence should be viewed as inherent features of both conventional genetic algorithm and genetic programming, rather than as problems to be cured by altering the fundamental Darwinian nature of the methods.

5. The analogue of premature convergence in genetic methods manifests itself in nature as the so-called niche preemption principle. According to this principle, a biological niche in nature tends to become dominated by a single species. Each independently run of a GP can be viewed as a separate niche which may become dominated by a particular individual species.

6. Therefore, one way to minimize the effect of niche preemption, premature convergence, initial conditions, and other random events when using genetic methods is to **make multiple independent runs** of a problem.

7. The amount of computational resource required by GP depends primarily on the product of

   - the number of individuals M in the population,
   - the number of generations executed in the run, and
   - the amount of processing required to measure the fitness of an individual over all the applicable fitness cases.

8. The third factor above is often not uniform during a run, in fact, vary in complex and unobvious ways. It also depends on the problem environment.

Topic 3: Four Introductory Examples of Genetic Programming

1. This topic contains examples of the GP paradigm applied to four simple introductory problems. The goal is to genetically breed a computer program to solve one illustrate example problem from each of the following four fields:

   - **Symbolic regression**
     Evolve a mathematical expression that closely fits a given finite sample of data.
   - **Robotic planning**
     Evolve a robotic action plan that will enable an artificial ant to find all the food along a trail containing various gaps and irregularities.
   - **Boolean 11-multiplexer**
     Evolve a Boolean expression that performs the Boolean 11-multiplexer function.
   - **Optimal control**
     Evolve a control strategy that will apply a force so as to bring a cart moving along a track to rest at a designated target point in minimum time.

2. There are five major steps in preparing to use the genetic programming paradigm to solve a problem:

   - determining the set of terminals,
   - determining the set of functions,
   - determining the fitness measure,
   - determining the parameters and variables for controlling the run, and
   - determining the method of designating a result and the criterion for terminating a run.

3. Simple Symbolic Regression

   - Consider a simple form of the problem if linear regression, one is given a set of values of various independent variable(s) and the corresponding values for the dependent variable(s).
   - The goal is to discover a set of numerical coefficients for a linear combination of the independent variable(s) that minimizes some measure of error (such as the square root of the sum of the squares of the differences) between the given values and computed values of the dependent variable(s).
   - The issue is deciding what type of function most appropriately fits the data, not merely computing the numerical coefficients after the type of function for the model has already been chosen. In other words, the real problem is often *both* the discovery of the correct functional form that fits the data and the discovery of the appropriate numerical coefficients that go with that functional form.
   - For example, suppose we are given a sampling of numerical values from a target curve over 20 points in some domain, such as the real interval [-1.0, 1.0]. That is, we are given a sample of data in the form of 20 pairs (x_i, y_i), where x_i is a value of the independent variable in the interval [-1.0, 1.0] and y_i is the associated value of the dependent variable. These 20 pairs (x_i, y_i) are the fitness cases that will be used to evaluate the fitness of any proposed computer program.

- The first major step in preparing to use GP is to identify the set of terminals. **Here, the information which the mathematical expression must process is the value of the independent variable `X`. Thus, the terminal set is**
  `T = {X}`
- The second major step is to identify the set of functions that are used to generate the mathematical expressions that attempt to fit the given finite sample of data. **If we wanted to use our knowledge that the answer is $x^4 + x^3 + x^2 + x$, a function set consisting only of the addition and multiplication operations would be sufficient for this problem**. However, we employ the more general function set:
  $\texttt{F} = \{+, -, \times, \%, \sin, \cos, \texttt{exp}, \texttt{Rlog}\}$
- The third major step is to identify the fitness function. Here, the raw fitness is the sum , taken over the 20 fitness cases, of the absolute value of the difference (error) between the value in the real-valued range space produced by the computer program for a given value of the independent variable x_i, and the correct y_i in the range space. The closer this sum of error is to 0, the better the computer program.
- The hit measure counts the number of fitness cases for which the numerical value returned by the computer program comes from within a small tolerance (called the *hit criterion*) of the correct value.
- Figure 7 is the tableau for the simple symbolic regression problem, and Figure 8, 9, 10 have shown the median, second-best, and best individuals from generation 0 in a typical run. And, Figure 11 is the parse tree form of 100% correct best-of-run individual.

4. Artificial Ant

- Consider the task of navigating an artificial ant attempting to find all food lying along an irregular trial as described in Figure 12. The problem involves primitive operations enabling the ant to move forward, turn right, turn left, and sense food along the irregular Santa Fe trail.
- In this problem, the information we want to process is the information coming in from the outside world via the ant's very limited sensor. Thus, one reasonable approach to this problem is to place the **conditional branching operator** `IF-FOOD-AHEAD` into the function set.
- The `IF-FOOD-AHEAD` conditional branching operator takes two arguments and executes the first argument if (and only if) the ant senses food directly in front of it, but executes the second argument if (and only if) the ant does not sense any food directly in front of it.
- The terminal set should contain the actions which the ant should execute based on the outcome of this information processing. Thus, the terminal set is
  $\texttt{T} = \{(\texttt{MOVE}), (\texttt{RIGHT}), (\texttt{LEFT})\}$
- However, we also need the **unconditional** operator. For example, we define an two-arguments `PROGN` connective: `(PROGN (RIGHT) (LEFT))`, which causes the ant to **unconditionally** perform the sequence of turning to the right and then turning to the left. Here, we use the following function set:
  `F = {IF-FOOD-AHEAD, PROGN2, PROGN3}`
- The natural measure of the fitness of a given computer program in this problem is the amount of food eaten within some reasonable amount of time by an ant executing the

Objective:	Find a function of one independent variable and one dependent variable, in symbolic form, that fits a given sample of 20 (x_i, y_i) data points, where the target function is the quartic polynomial $x^4 + x^3 + x^2 + x$.
Terminal set:	X (the independent variable).
Function set:	+, −, *, %, SIN, COS, EXP, RLOG.
Fitness cases:	The given sample of 20 data points (x_i, y_i) where the x_i come from the interval $[-1, +1]$.
Raw fitness:	The sum, taken over the 20 fitness cases, of the absolute value of difference between value of the dependent variable produced by the S-expression and the target value y_i of the dependent variable.
Standardized fitness:	Equals raw fitness for this problem.
Hits:	Number of fitness cases for which the value of the dependent variable produced by the S-expression comes within 0.01 of the target value y_i of the dependent variable.
Wrapper:	None.
Parameters:	$M = 500$. $G = 51$.
Success predicate:	An S-expression scores 20 hits.

Figure 7: Tableau for the simple symbolic regression problem.

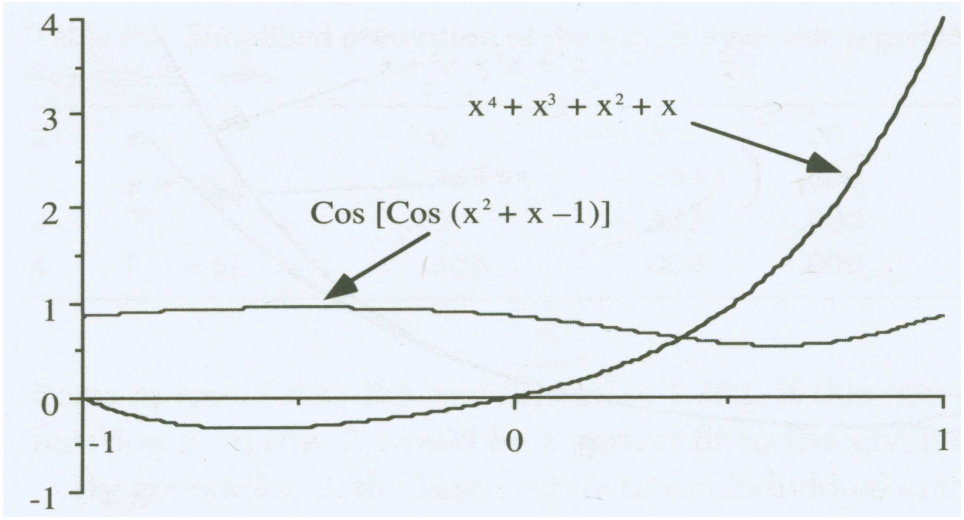

Figure 8: Median individual from generation 0 compared to target curve $x^4 + x^3 + x^2 + x$.

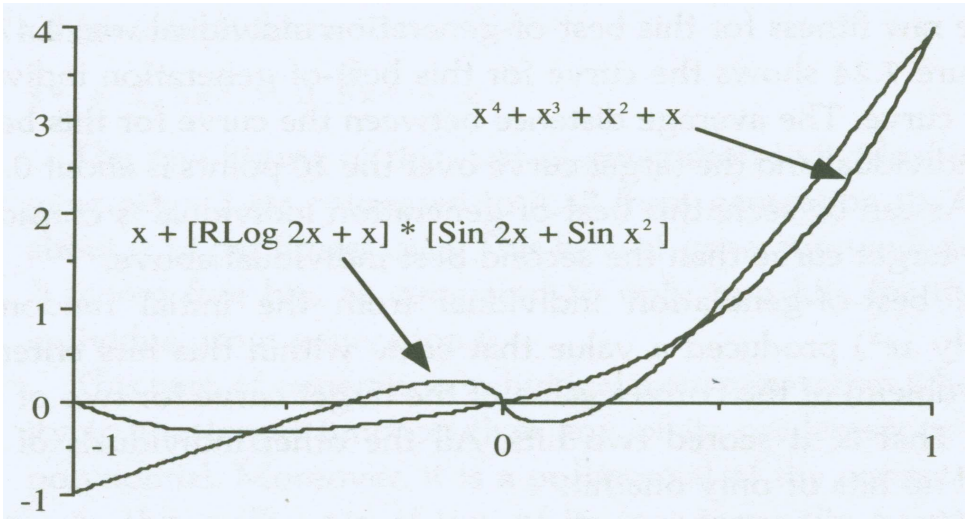

Figure 9: Second-best individual from generation 0 compared to target curve $x^4 + x^3 + x^2 + x$.

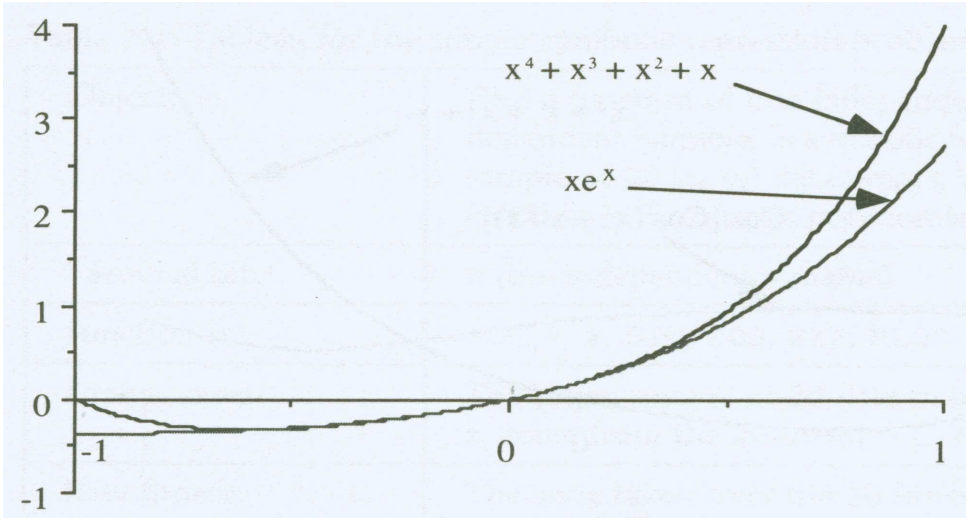

Figure 10: Best-of-generation individual from generation 0 compared to target curve $x^4 + x^3 + x^2 + x$.

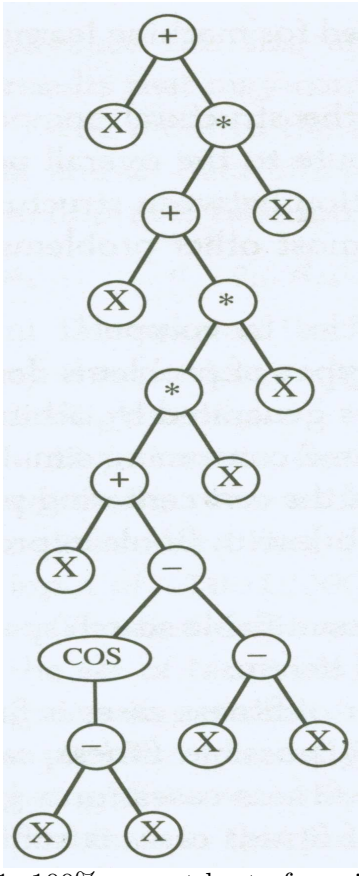

Figure 11: 100% correct best-of-run individual

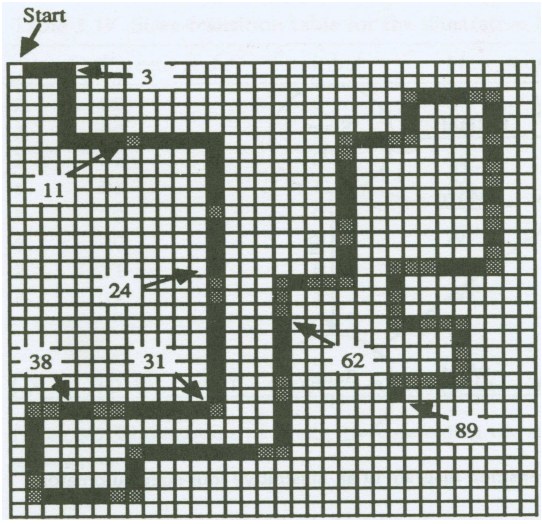

Figure 12: The Santa Fe trail for the artificial ant problem.

given program. Each move operation and each turn operation takes one step whereas the `IF-FOOD-AHEAD` operator and the unconditional connectives `PROGN2` and `PROGN3` each takes no time steps to execute.

- Here, we limited the ant to the 400 time steps. The time-out limit is sufficiently small in relation to 1024 to prevent a random walk or a tessellating movement from covering all 1024 squares of the grid before timing out.
- Figure 13 is the tableau for the artificial ant problem for the Santa Fe trail.
- One randomly generated computer program which is called the "quilter" path moves and turns without looking. It consists of nine points:

```
(PROGN3 (RIGHT)
        (PROGN3 (MOVE) (MOVE) (MOVE))
        (PROGN2 (LEFT) (MOVE))).
```

  Figure 14 shows the first part of the quilter's path.
- One randomly generated computer finds the first 11 pieces of food on the trail and then goes into an infinite loop when it encounters the first gap in the trail. In Figure 15, the looper's path is marked by `X`'s.
- One randomly generated computer program actually correctly takes note of the portion of food along the trail before finding the first gap in the trail, then actively avoids this food by carefully moving around it until it returns to its starting point, and never eats any food. The computer program for the avoider has seven points:

```
(IF-FOOD-AHEAD (RIGHT)
               (IF-FOOD-AHEAD (RIGHT)
               (PROGN2 (MOVE) (LEFT))).
```

  Figure 16 is the avoider's path.
- On generation 21, a computer program scoring 89 out of 89 emerged for the first time on this run. The computer program has 18 points and is shown below:

```
(IF-FOOD-AHEAD (MOVE)
               (PROGN3 (LEFT)
                       (PROGN2 (IF-FOOD-AHEAD (MOVE)
```

Objective:	Find a computer program to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe trail.
Terminal set:	(LEFT), (RIGHT), (MOVE).
Function set:	IF-FOOD-AHEAD, PROGN2, PROGN3.
Fitness cases:	One fitness case.
Raw fitness:	Number of pieces of food picked up before the ant times out with 400 operations.
Standardized fitness:	Total number of pieces of food (i.e., 89) minus raw fitness.
Hits:	Same as raw fitness for this problem.
Wrapper:	None.
Parameters:	$M = 500$. $G = 51$.
Success predicate:	An S-expression scores 89 hits.

Figure 13: Tableau for the artificial ant problem for the Santa Fe trail.

```
                                (RIGHT))
                    (PROGN2 (RIGHT)
                            (PROGN2 (LEFT)
                                    (RIGHT))))
          (PROGN2 (IF-FOOD-AHEAD (MOVE)
                                 (LEFT))
                  (MOVE))))).
```

Figure 17 graphically depicts the 100%-correct best-of-run individual that emerged on generation 21 of this run of this problem.

5. Boolean Multiplexer

- Consider the problem of Boolean concept learning (i.e., discovering a composition of Boolean functions that can return the correct value of a Boolean function after seeing a certain number of examples consisting of arguments).
- The input to the Boolean N-multiplexer function consists of k address bits a_i and 2^k data bits d_i, where $N = k + 2^k$. That is, the input to the Boolean multiplexer function consists of the $k + 2^k$ bits:
  $a_{k-1}, ..., a_1, a_0, d_{2^k-1}, ...d_1, d_0$
- The value of the Boolean multiplexer function is the Boolean value (0 or 1) of the particular data bit that is singled out by the k address bits of the multiplexer.
- For example, Figure 18 shows a Boolean 11-multiplexer (i.e., k=3) in which the three address bits $a_2a_1a_0$ are currently 110. The multiplexer singles out data bit 6 (i.e., d_6) to be the output of the multiplexer. Specifically, for an input of **110**01000000, the output of the multiplexer is **1**.
- In this problem, the information that must be processed by a computer program corresponds to the 11 inputs to the Boolean 11-multiplexer. That is, the terminal set contains the 11 arguments as shown below:

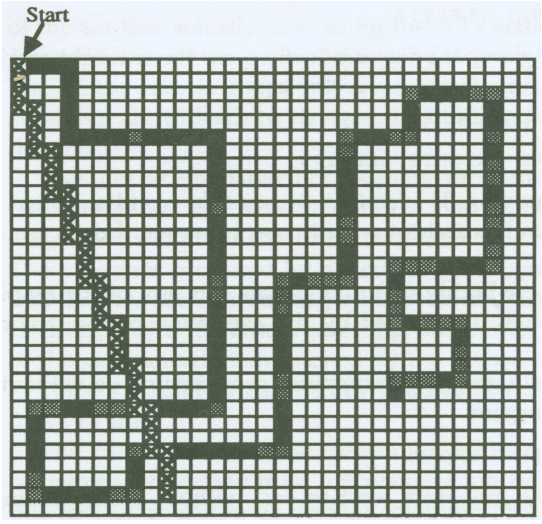

Figure 14: Path of the quilter from generation 0 of the artificial ant problem.

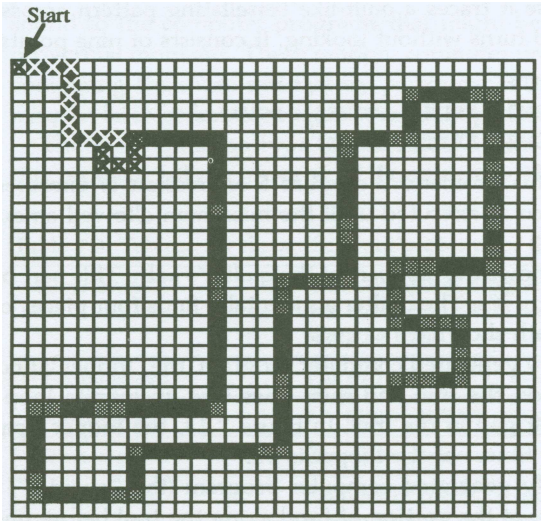

Figure 15: Path of the looper from generation 0 of the artificial ant problem.

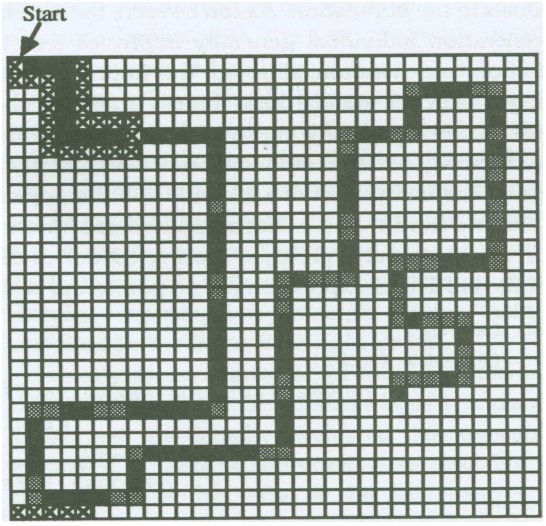

Figure 16: Path of the avoider from generation 0 of the artificial ant problem.

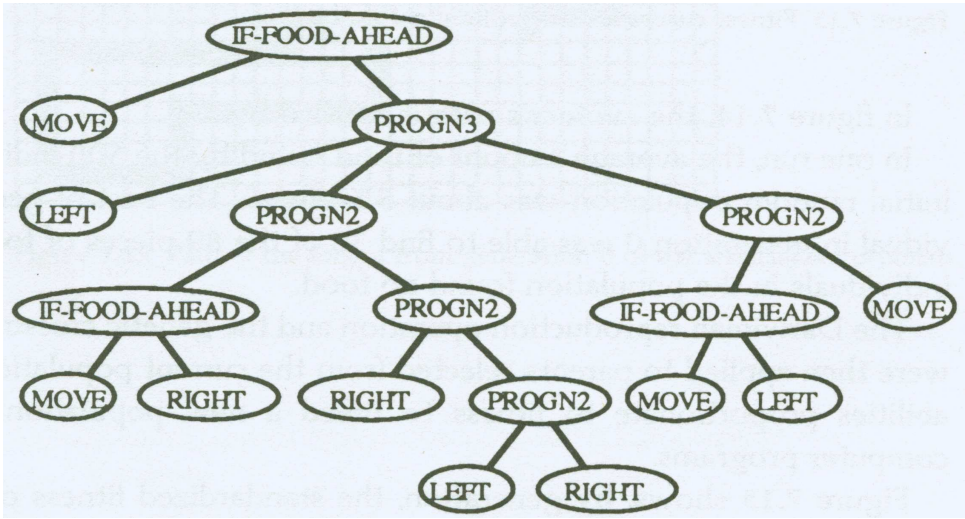

Figure 17: Best-of-run individual for the artificial ant problem.

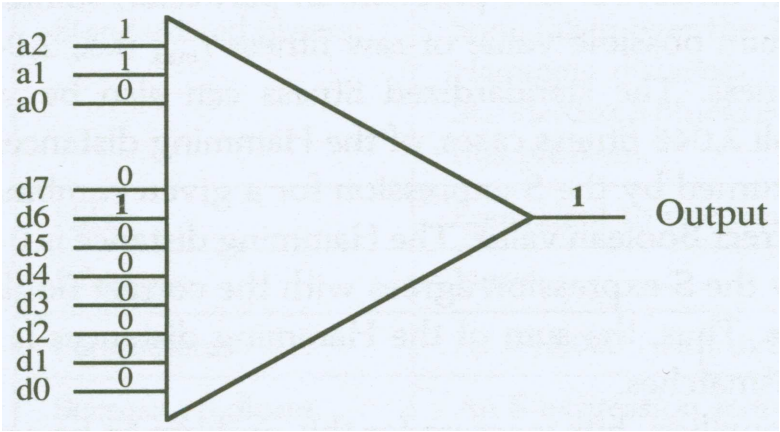

Figure 18: Boolean 11-multiplexer with input of 11001000000 and output 1.

Objective:	Find a Boolean S-expression whose output is the same as the Boolean 11-multiplexer function.
Terminal set:	A0, A1, A2, D0, D1, D2, D3, D4, D5, D6, D7.
Function set:	AND, OR, NOT, IF.
Fitness cases:	The $2^{11} = 2{,}048$ combinations of the 11 Boolean arguments.
Raw fitness:	Number of fitness cases for which the S-expression matches correct output.
Standardized fitness:	Sum, taken over the $2^{11} = 2{,}048$ fitness cases, of the Hamming distances (i.e., number of mismatches). Standardized fitness equals 2,048 minus raw fitness for this problem.
Hits:	Equivalent to raw fitness for this problem.
Wrapper:	None.
Parameters:	$M = 4{,}000$ (with over-selection). $G = 51$.
Success predicate:	An S-expression scores 2,048 hits.

Figure 19: Tableau for the Boolean 11-multiplexer Problem.

$\mathtt{T} = \{\mathtt{A}_0, \mathtt{A}_1, \mathtt{A}_2, \mathtt{D}_0, \mathtt{D}_1, ..., \mathtt{D}_7\}$
Note that these terminals are not distinguished as being address lines versus data lines.

- There are many possible choices of sufficient function sets for this problem. Here, we use
$\mathtt{F} = \{\mathtt{AND}, \mathtt{OR}, \mathtt{NOT}, \mathtt{IF}\}$
where `IF` means the `if-then-else` operator.
- There are $2^{11} = 2048$ possible combinations of the 11-multiplexer function. For this particular problem, we use the entire set of 2048 combinations of arguments as the fitness cases for evaluating fitness.
- The raw fitness in this problem is simply the number of fitness where the Boolean value returned by the computer program for a given combination of argument is the correct Boolean value.
- Figure 19 summarizes the key features of the Boolean 11-multiplexer problem.
- Figure 20 is the solution to the 11-multiplexer problem. It is a hierarchical conditional composition of two 6-multiplexers.
- Hierarchies
  - Hierarchies are an efficient and often highly understandable way of presenting the steps and substeps (tasks and subtasks, routines and subroutines) that constitute

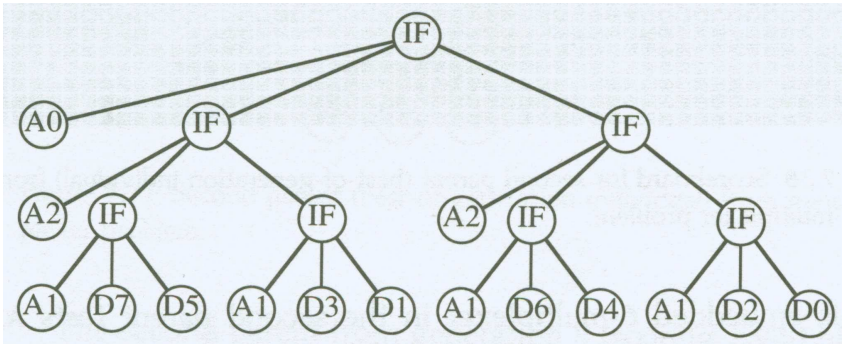

Figure 20: The solution to the 11-multiplexer problem.

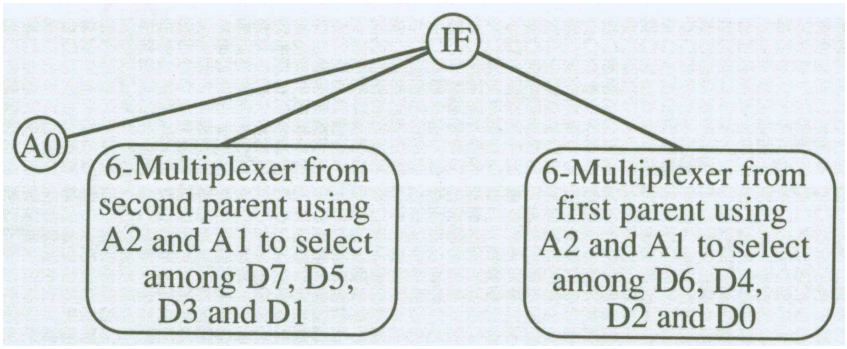

Figure 21: The solution to the 11-multiplexer problem is a hierarchical conditional composition of two 6-multiplexers.

the solution to a problem. Moreover, hierarchical structures are amenable to scaling up to large problems.

– In many cases, the hierarchies produced can be very informative.

6. Cart Centering

- The cart centering problem involve a cart that can move to the left or the right on a frictionless one-dimensional track. The problem is to center the cart, in minimal time, by applying a force of **fixed** magnitude (a *bang-bang force*) so as to accelerate the cart toward the left or the right.
- The cart centering problem is a problem of optimal control. Such problems involve a system whose state is described by state variables. The choice of the control variable causes the state of the system to change. The goal is to choose the value of the control variable so as to cause the system to go to a specified target state with an optimal cost. The cost may be measured in time, distance, fuel, or dollars.
- There are two state variables for the system in the cart centering problem: the current position $x(t)$ of the cart along the track and the current velocity $v(t)$ of the cart.
- There is one control variable for this system: the direction from which a rocket applies a bang-bang force F to the center of mass of the cart so as to accelerate the cart in either the positive or the negative direction along the track.
- The target state for the system is the state for which the cart is at rest and centered at the origin.

- Basic mathematical equations:
  - $a(t) = \frac{F(t)}{m}$,
    where m is the mass of the cart, $a(t)$ is the acceleration. Then, the velocity $v(t+1)$ of the cart at time step $t+1$ (which occurs a small amount of time τ after time step t) becomes:
  - $v(t+1) = v(t) + \tau a(t)$,
    where τ is the size of the time step.
  - At time step $t+1$, the position $x(t+1)$ of the cart becomes
    $x(t+1) = x(t) + \tau v(t)$
- The problem is to find a time-optimal control strategy for centering the cart that satisfies the following three conditions:
  - The control strategy satisfies how to apply the bang-bang force for any given current position $x(t)$ and current velocity $v(t)$ of the cart at each time step.
  - The cart approximately comes to rest at the origin.
  - The time required is minimal.
- The exact time-optimal solution is , for any given current position and velocity, to apply the bang-bang force $F(t)$ to accelerate the cart in the positive direction if

$$-x(t) > \frac{v(t)^2 \texttt{Sign}\, v(t)}{2|F|/m}$$

  or, otherwise, to apply the bang-bang force F to accelerate the cart in the negative direction. The `Sign` function returns +1 for a positive argument and -1 otherwise.
- In this problem, the inputs to the computer program are the two state variables of the system (i.e., x and v). The output from the computer program is interpreted as the single control variable of the system (the direction, +1 or -1, for the bang-bang force F).
- The physics of the problem dictate that the variables having explanatory power for the problem are the position x of the cart along the track and the velocity v of the cart. Thus, the terminal set for the cart centering problem is
  $\texttt{T} = \{\texttt{X}, \texttt{V}, -1\}$
  Note that the numerical constant -1 was included in the terminal set above because we **thought** it might be useful.
- Because the exact time-optimal solution in terms of LISP form
  `(GT (× -1 X) (× V (ABS V))`
  contains "greater than" condition and absolute value, therefore, we include the `GT` (greater than) operator and `ABS` function in the function set.
  $\texttt{F} = \{+, -, \times, \texttt{GT}, \texttt{ABS}\}$
- Here, the fitness measure is the total time required to center the cart after starting at a representative sampling of random initial condition points (x, v) in the domain specified for this problem.
- Figure 22 is the tableau for the cart centering problem, the best-of-run individual is shown in Figure 23.

Objective:	Find a time-optimal bang-bang control strategy to center a cart on a one-dimensional frictionless track.
Terminal set:	The state variables of the system: x (positive X of the cart) and v (velocity V of the cart).
Function set:	+, −, *, %, ABS, GT.
Fitness cases:	20 initial condition points (x, v) for position and velocity chosen randomly from the square in position-velocity space whose opposite corners are $(-0.75, 0.75)$ and $(0.75, -0.75)$.
Raw fitness:	Sum of the time, over the 20 fitness cases, taken to center the cart. When a fitness case times out, the contribution is 10.0 seconds.
Standardized fitness:	Same as raw fitness for this problem.
Hits:	Number of fitness cases that did not time out.
Wrapper:	Converts any positive value returned by an S-expression to $+1$ and converts all other values (negative or zero) to -1.
Parameters:	$M = 500$. $G = 51$.
Success predicate:	None.

Figure 22: Tableau for the cart centering problem.

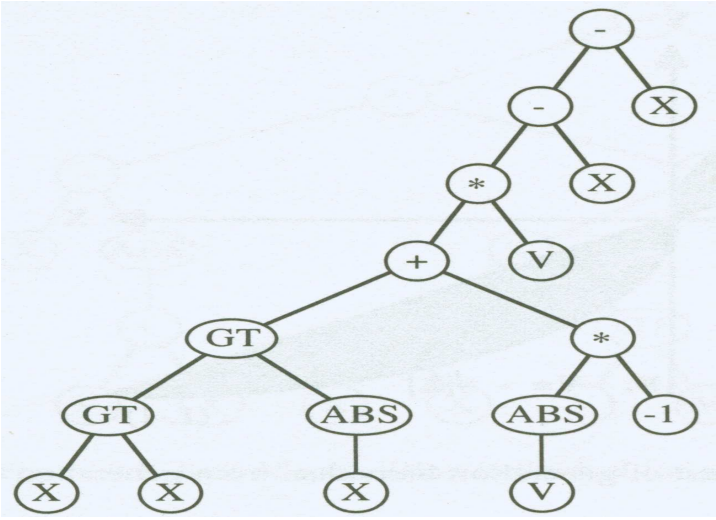

Figure 23: Best-of run individual for the cart centering problem.

Topic 4: Symbolic Regression — Error-Driven Evolution

1. Introduction

   - Problem of symbolic regression require finding a function, in symbolic form, that fits a given finite sampling of data points. Symbolic regression provides a means for function identification. It is error-driven evolution.
   - In this topic, we will show how the techniques of symbolic regression can be used to solve a wide variety of different problems, including
     - discovery of trigonometric identities,
     - symbolic regression involving creation of arbitrary constants,
     - econometric modeling and forecasting,
     - differential equations,
     - integral equations,
     - sequence induction, and
     - programmatic image compression.
   - Discovery of Trigonometric Identities
     - Finding a mathematical identity involves finding a new and unobvious mathematical expression, in symbolic form, that always has the same value as some given mathematical expression.
     - Consider a given mathematical expression, in symbolic form, such as $\cos 2x$. If we can discover a new mathematical expression, such as $1 - 2\sin^2 x$, that equals $\cos 2x$ for all values of x, we will have succeeded in finding an identity.
     - $\cos 2x = \cos(-2x) = 2\cos^2 x - 1 = 1 - 2\sin^2 x = \sin(\pi/2 - 2x)$.
     - The candidate elements of terminal set, x, 1, π.
     - The selection of functions in the function set and terminals in the terminal set will influence what the mathematical expression(s) we will find out.
   - Symbolic Regression with Constant Creation
     - Suppose we are given a sampling of the numerical values from the given curve $2.718x^2 + 3.1416x$ over 20 randomly chosen points in some domain, such as the interval [-1, 1].
     - The problem of constant creation can be solved by expanding the terminal set by adding one special new terminal called *ephemeral random constant* and denoted R
     - R could be the floating-point type ranged between (-1.000 1.000), or integer type ranged between (-5, 5), or Boolean type which produces `True` or `False`.
     - For most of the cases, the determination of the range of R is an *art*.
   - Economic Modeling and Forecasting
     - An important problem area in virtually every area of science is finding the relationship underlying empirically observed values of the variables measuring a system. In practice, the observed data may be noisy and there may be no way to express the relationships in any precise way. Some of the data may be missing.
     - The goal is to discover empirical relationships between dependent variable(s) and independent variables. However, the clear relationships usually don't exist. They also depend on the choice of *independent variables*, their lags, and the functions employed in function set.

- In the economic forecasting, we usually divided the data into two parts, one for training in order to obtain the functional relationships, the other is used to validate the performance of the functional relationships.

- Differential Equations
  - A differential equation is an equation involving one or more derivatives (of some order) of an unknown function. The solution to a differential equation is a function that, when substituted into the given equation, satisfies the equation and any given initial conditions.
  - Example 1
    * Consider the simple differential equation: $\frac{dy}{dx} + y \cos x = 0$, where $y_{\text{initial}} = 1.0$ and $x_{\text{initial}} = 0.0$
    * The goal is to find a function that satisfies this equation and its initial condition, namely the function $e^{-\sin x}$.
    * Terminal set = {`X`}.
    * Function set = {`+`, `-`, $\times$, `%`, sin, cos, `exp`, `Rlog`}.
    * We start by generating 200 random values of the independent variables, x_i over some appropriate domain, such as the unit interval [0, 1]. We then sort these values into ascending order.
    * We are seeking a function $f(x)$ such that, for every one of the 200 values x_i of the variables x, we get 0 when we perform the following computation: $f'(x_i) + f(x_i) \cos x_i$, where $f'(x_i)$ is obtained by numerically differentiation with respect to the independent variable x_i.
    * In solving differential equations, the fitness of a particular genetically produced function should be expressed in terms of two components:
      · how well the function satisfies the differential equation,
      · how well the function satisfies the initial condition of the differential equation.
    * Since a mere linear function passing through the initial condition point will maximize this second component, it seems reasonable that the first component should receive the majority of the weight in calculating fitness.
    * Therefore, the raw fitness of a genetically produced function f_j is 75% of the first component plus 25% of the second component.
    * The first component used in computing the raw fitness of f_j is the sum, for i between 0 and 199, of the absolute values of the differences between the zero function and $f_j'(x_i) + f_j(x_i) \cos x_i$, namely $\sum_{i=0}^{199} |f_j'(x_i) + f_j(x_i) \cos x_i|$.
    * The second component used in computing the raw fitness of f_j is based on the absolute value of the difference between the given value y_{initial} for the initial condition and the value of the function $f_j(x_{\text{initial}})$ for the particular given initial condition point x_{initial}.
    * Note that the initial condition should be chosen so that the zero function does not satisfy the differential equation and the initial condition; otherwise, the zero function will likely be produced as the solution by genetic programming.
    * A hit is defined as a fitness case for which the standardized fitness is less than 0.01. Since numerical differentiation is relatively inaccurate for the endpoints

of an interval, attainment of a hit for 198 of the 200 fitness cases is one of the termination criteria for this problem.

* By generation 6, the best-of-generation S-expression in the population was, when simplified, equivalent to $e^{-\sin x}$. Its raw fitness is a mere 0.057. As it happens, this individual scores 199 hits, thus terminating the run.

– Example 2

* A second example of a differential equation is $\frac{dy}{dx} - 2y + 4x = 0$ with an initial condition such that $y_{\text{initial}} = 4$ when $x_{\text{initial}} = 1$.
* In generation 28 of one run, the S-expression
`(+ (× (exp (-X 1)) (exp (-X 1))) (+ (+ X X) 1))`
emerged. This individual is equivalent to $e^{-2}e^{2x} + 2x + 1$, which is the exact solution to the differential equation.

– Example 3

* A third example of a differential equation is $\frac{dy}{dx} = \frac{2+\sin x}{3(y-1)^2}$, with an initial condition such that $y_{\text{initial}} = 2$ when $x_{\text{initial}} = 0$.
* This problem was run with a function set that included the cube root function `CUBRT`.
* In generation 13 of one run, the S-expression
`(- (CUBRT (CUBRT 1)) (CUBRT (- (- (- (cos X) (+ 1 (CUBRT 1))) X) X)))`
emerged. This individual is equivalent to $1 + (2 + 2x - \cos x)^{1/3}$ which is the exact solution to the differential equation.
* When the initial condition of the differential equation involves only a value of the function itself, any point in the domain of the independent variable x may be used for the initial condition.
* When the initial condition involves a value of a derivative of the function (as may be the case when the differential equation involves second derivatives or higher derivatives), it is necessary that the value of the independent variable x involved in the initial condition be one of the points in the random set of points x_i so that the first derivative (and any required higher derivative) of the genetically produced function is evaluated for the initial condition.
* In addition, it is preferable that the points x_{initial} be an internal point, rather than an endpoint of the domain since numerical differentiation is usually more accurate for the internal points of an interval.

- Integral Equations
  – An integral equation is an equation involving the integral of an unknown function. The solution to an integral equation is a function which, when substituted into the given equation, satisfies the equation.
  – An example of an integral equation is $y(t) - 1 + 2\int_{r=0}^{r=t} \cos(t-r)y(r)dr = 0$.
  – In one run, we found the solution to this integral equation, namely $y(t) = 1 - 2te^{-t}$.
  – The process of integration creates a variable (r), which is similar to the indexing variable of an iterative loop.
- Sequence Induction
  – The ability to correctly perform induction is widely viewed as an important component of human intelligence.

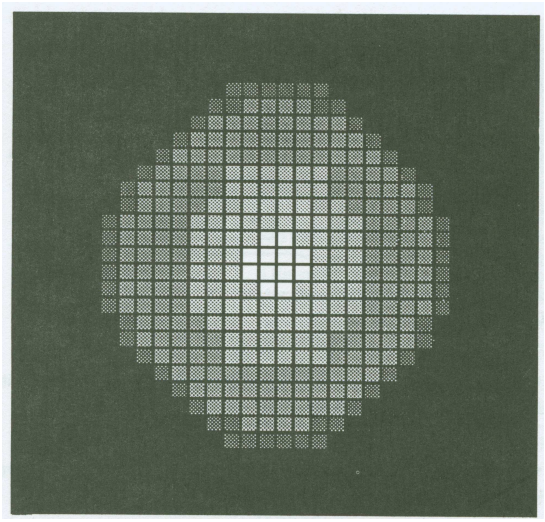

Figure 24: Target image for problem of programmatic image compression.

  - Sequence induction involves discovering a mathematical expression that can generate any arbitrary element in an infinite sequence
    $S = S_0, S_1, S_2, ..., S_j, ...$
    after seeing only a relatively small finite number of specific examples of the values of the unknown sequence,
  - Sequence induction is a special case of symbolic regression, namely the case where the domain of the independent variable x consists of the non-negative integers.
  - Of course, there is no one correct answer to a problem of sequence induction, there being an infinity of sequences that agree with any finite number of specific examples of the unknown sequence.
  - Suppose one is given the first 20 values of the following simple nonrecursive sequence of integers:
    S = 1, 15, 129, 547, 1593, 3711, 7465, 13539, 22737, 36983, 54321, 78915, 111049, 152127, 203673, 267331, 344865, 438159, 549217, 680163,.....
  - Terminal set = {`J, R`},
    where `J` is the index position (independent variable). The ephemeral random constant `R` ranging over the small integers 0, 1, 2, and 3.
  - Function set = {`+, -,` $\times$}
  - The fitness cases for this problem consists of the first 20 elements of the given sequence. The raw fitness is the sum of the absolute value of the difference between the value produced by the S-expression for sequence position `J` and the actual value of the sequence for position `J`.
  - The unknown mathematical expression we are seeking is $5J^4+4J^3+3J^2+2J+1$.
  - The 100%-correct individual emerged on generation 43.

- Programmatic Image Compression
  - A spectacular variety of color images can be produced by hand-selecting interesting images from a large number of randomly produced LISP S-expressions displayed by an interactive workstation.
  - Here, the objective is to use symbolic regression to discover a computer program that exactly or approximately represents the given color image.

- Figure 24 is a black-and-white diagram representing an image consisting of concentric ellipses in a spectrum of different colors. In this figure, there are 30 pixels in the horizontal direction and 30 pixels in the vertical direction, for a total of 900 pixels.
- The center of the color image is considered as the origin (0,0), the upper left corner is (-1.0,1.0),..., etc. The color of each pixel is one of 128 different shade from a red, green, and blue (RGB) color spectrum. In this color system, each floating-point number in the interval [-1.0,1.0] corresponding to one of the 128 shade. For example, -1.0 represents 100% red; 0.0 represents 100% green; 1.0 represents 100% blue.
- In this figure, the origin is colored red and the concentric ellipses surrounding it are various shades of red blended with green.
- The pattern in this figure is produced by the expression $3x^2 + 2y^2 - 0.85$.

Topic 5: Evolution of Emergent Behavior

1. Introduction

   - The repetitive application of seemingly simple rules can lead to complex overall behavior. Such emergent behavior arises in cellular automata, dynamical systems, fractals and chaos, and throughout nature. Emergent behavior is one of the main themes of research in artificial life.
   - In one avenue of work in emergent behavior, researchers try to conceive and then write sets of simple rules that produce complex overall behavior similar to that observed in nature.
   - The fact that it is possible to conceive and write such sets of handwritten rules is an argument in favor of the possibility that the complex overall behavior observed in nature may be produced by similar sets of relatively simple governing rules.
   - If it is true that complex overall behavior can be produced from sets of relatively simple rules, it should be possible to evolve such sets of rules by means of an artificial process such as genetic programming. If such artificial evolution proves to be possible, then there is at least an argument in favor of the possibility that the evolutionary process in nature might have produced the complex overall behavior observed in nature.

2. Central Place Food Foraging Behavior

   - Here, the goal is to genetically breed a **common computer program** that, when simultaneously executed by all the individuals in a group of independent agents (e.g., social insects such as ants or independently acting robots), causes the emergence of beneficial and interesting higher-level collective behavior.
   - Problem Properties:
     - Ants do not communicate with one another directly. The central place foraging problem is not solved by a coherent and synchronized set of commands being broadcast to individual ants from a central authority. Instead, each ant follows a common set of internal rules on a distributed, asynchronous, and local basis.
     - If the environment seen by an individual ant makes one of its internal rules applicable, then ant takes the appropriate action. The internal rules are prioritized so as to resolve potential conflict. Each ant is in direct communication with its environment. The ants communicate with one another in a very indirect way via the environment (i.e., they sense the presence or absence of pheromones and food).
   - Environment and Ant Characteristics:
     - There are two concentrated piles of food. A total of 144 pellets of food are piled eight deep in two 3×3 piles. The domain of action is a 32×32 grid. Also, see Figure 25.
     - In deference to animal right groups, the grid is toroidal, so that if an ant wanders off the edge it reappears on the opposite edge.
     - The two piles of food are some distance from the nest of the colony in locations that cannot be reached by merely walking in a straight line from the nest.

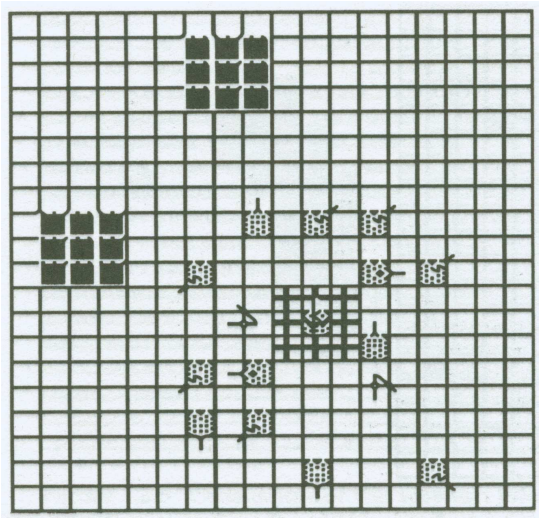

Figure 25: Begin of the random search for food as the 20 ants leave the nest.

- There are 20 ants in the colony. The state of each ant consists of its position on the grid, the direction it is facing (out of eight possible directions), and an indicator as to whether it is currently carrying food. Each ant initially starts at the nest and faces in a random direction.
- **Each ant in the colony is governed by a common computer program associated with colony**.
- Defining nine operators which are sufficient to solve this problem:
  - `MOVE-RANDOM` randomly changes the direction in which an ant is facing and then moves the ant two steps in the new direction.
  - `MOVE-TO-NEST` moves the ant one step in the direction of the nest. This implements the gyroscopic ability of many species of ants to navigate back to their nest.
  - `PICK-UP` picks up food (if ant) at the current position of the ant if the ant is not already carrying food.
  - `DROP-PHEROMONE` drops pheromones at the current position of the ant (if the ant is carrying food). The pheromones immediately form 3×3 cloud around the drop point. The cloud decays over a period of time.
  - `IF-FOOD-HERE` is a two-argument conditional branching operator that executes its first argument if there is food at the ant's current position and that otherwise executes the second (else) argument.
  - `IF-CARRYING-FOOD` is similar two-argument conditional branching operator that tests whether the ant is currently carrying food.
  - `MOVE-TO-ADJACENT-FOOD-ELSE` is a one-argument conditional branching operator that allows the ant to test for immediately adjacent food and then move one step toward it. If food is present in more than one adjacent position, the ant moves to the position requiring the least change of direction. If no food is adjacent, the "else" clause of this operator is executed.
  - `MOVE-TO-ADJACENT-PHEROMONE-ELSE` is a conditional branching operator similar to `MOVE-TO-ADJACENT-FOOD-ELSE` except that is based on the adjacency of pheromones.

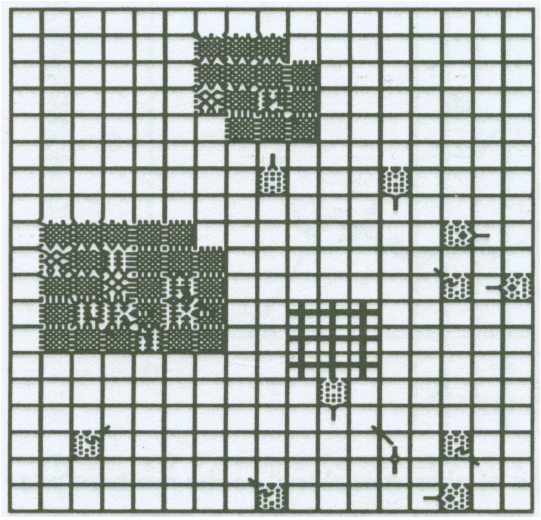

Figure 26: Initial contact with food and the beginning of formation of the pheromonal clouds around the two piles of food.

    - * `PROGN` is the LISP connective function that executes its arguments in sequence.
  - Each of the 20 ants in the colony executes the colony's common computer program at each time step. The action of one ant (e.g., picking up food, dropping pheromones) can, and does, alter the state of the system for the other ants.
  - Multiple ants are allowed to occupy the same square.
  - Terminal set = {`(MOVE-RANDOM)`, `(MOVE-TO-NEST)`, `(PICK-UP)`, `(DROP-PHEROMONE)`}.
  - Function Set = {`IF-FOOD-HERE`, `IF-CARRYING-FOOD`, `MOVE-TO-ADJACENT-FOOD-ELSE`, `MOVE-TO-ADJACENT-PHEROMONE-ELSE`, `PROGN`}.
  - The raw fitness of a computer program is measured by how many of the 144 food pellets are transported to the nest within the allowed time (i.e., 400 time steps and a maximum of 1000 operations for each computer program). When an ant arrives at the nest with food, the food is automatically dropped and counted.
- Simulation Result:
  - The 100%-fit program is a prioritized sequence of conditional behaviors that work together to solve the problem.
  - The collective behavior of the ants can be visualized over a series of phases. See Figure 26-29.
  - First, this program directs the ant to pick up any food it may encounter. If there is no food to pick up, the second priority established by this conditional sequence directs the ant to follow a previously established pheromonal trail.
  - If there is no food and no pheromonal trail, the third priority directs the ant to move at random.

3. Emergent Collecting Behavior

- Introduction
  - The stimulating work done by Deneubourh et al. (1986, 1991) on the emergent sorting and collecting behavior of independent agents is another illustration of

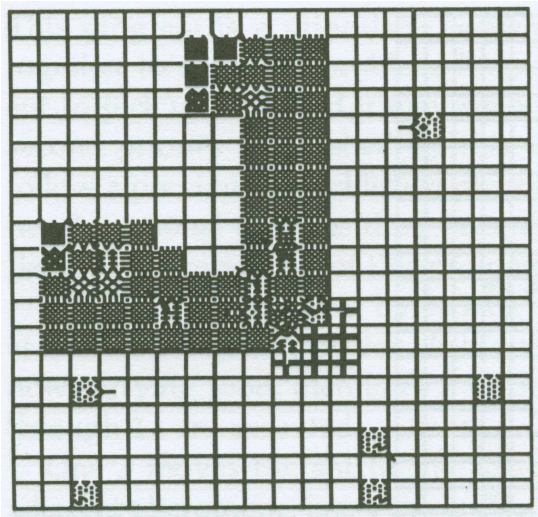

Figure 27: Two persistent pheromonal trails connecting the two piles of food with the nest.

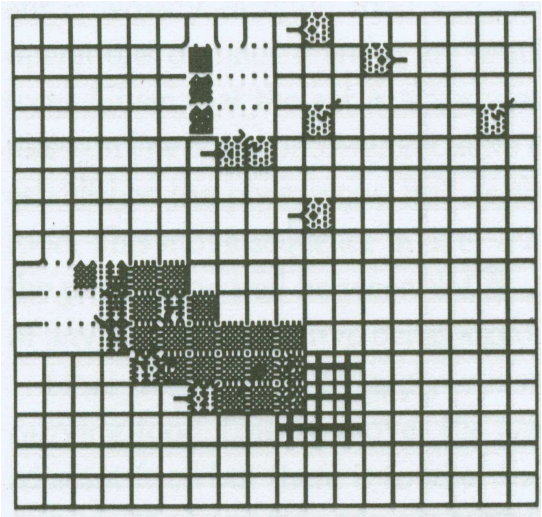

Figure 28: Premature disintegration of pheromonal trail to the northern pile.

how complex overall patterns of behavior can emerge from a relatively simple set of rules that control the action of a distributed set of agents acting in parallel.

- In this section, the goal is to evolve a computer capable of emergent collecting behavior.

- Environment and Ant Characteristics:
  - There are 25 food pellets and 20 independent agents.
  - Figure 30 shows the initial configure of food and independent agents on 25×25 toroidal grid.
  - The 25 food pellets, shown in grey, are initially isolated and dispersed in a regular rectangular pattern. Each agent, shown in black, starts at a random location and faces in a random direction, with a pointer showing the agent's initial facing direction.
  - **All the agents are governed by a common computer program**.
  - The `PICK-UP`, `IF-CARRYING-FOOD`, `IF-FOOD-HERE`, and `PROGN2` functions are as

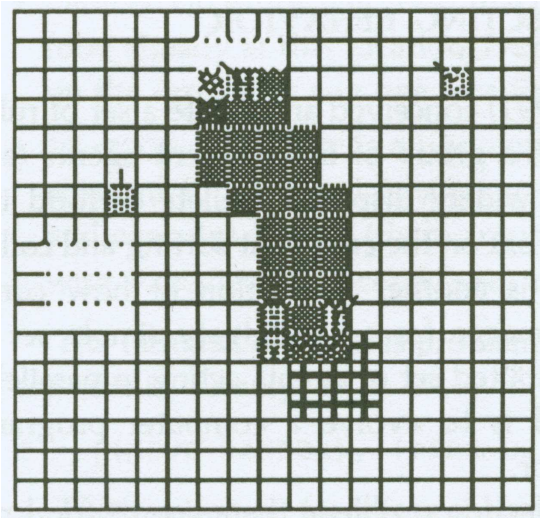

Figure 29: Exhaustion of the western pile and continued exploitation of the northern pile.

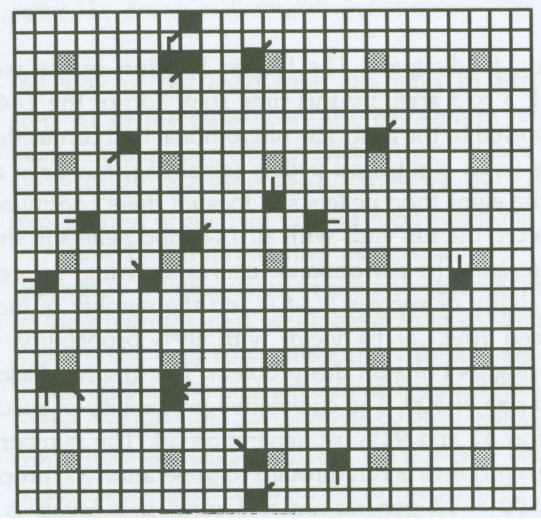

Figure 30: Initial configuration of 25 food pellets and 20 independent agents.

defined for the central place foraging problem . In addition, the following four functions are also used:

* `MOVE` moves the agent one step in the direction it is currently facing provided there is no agent already at that direction.
* `MOVE-RANDOM` randomly changes the direction in which an agent is facing and then executes the `MOVE` function twice.
* `DROP-FOOD` drops any food that the agent is carrying provided there is no food already at that location. During the run, only one pellet of food can be on the ground at any one location on the grid.
* `IF-FOOD-ADJACENT` is a two-argument function that searches the positions adjacent to the agent (changing the agent's facing direction as it searches) and executes its first (then) argument if any food is discovered and, otherwise, executes the second (else) argument.

– Terminal set = {`(MOVE-RANDOM), (PICK-UP), (MOVE), (DROP-FOOD)`}.

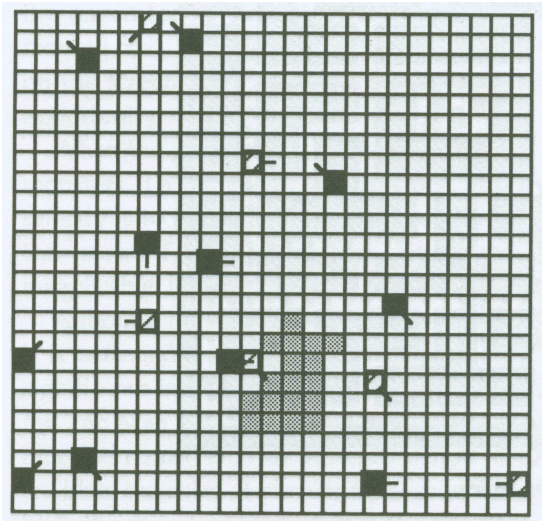

Figure 31: Epoch 2705 of best-of-generation individual from generation 34.

  - Function set = {`IF-FOOD-HERE, IF-CARRYING-FOOD, IF-FOOD-ADJACENT, PROGN2`}.
  - Goal:
    * The goal is to consolidate the food into one pile.
    * Therefore, raw fitness should measure compactness. In particular, raw fitness is the sum of the distances to each of the other 24 food pellets.
    * There are 300 distinct lines connecting each pair of food pellets. A smaller cumulative value for these 300 distances is obtained when the 25 food pellets are consolidated close together.
  - At most, 1200 evaluations, of the S-expression for each agent and 30000 individual operations are allowed. When an agent times out, any food being carried by the agent is, for purpose of computing the distances, considered to be at the location from which it was most recently picked up.
- Simulation Result:
  - The best-of-generation individual from generation 34 is highly effective in performing the task at hand. It has a fitness value of 1667, representing an average distance between food pellets of only about 2.8 units.
  - Let *epoch* be the average number of operations per agent. Figure 31 shows, at epoch 2705, that all the food not being carried has been dropped into the one reasonably compact island in the bottom center part of the grid. When execution ends, 100% of the food ends up as part of this single island. Figure 32 is the best-of-run program from generation 34.

4. Task Prioritization

- Introduction
  - The solution to a planning problem often involves establishing priorities among tasks with differing importance and urgency. Also, when special situations suddenly arise, a radically different arrangement of priorities may be required.
  - The familiar Pac Man and Mr. Pac Man video games present a planning problem in which task prioritization is a major characteristic.

```
(IF-FOOD-ADJACENT (IF-FOOD-ADJACENT (PROGN2 (IF-CARRYING-
FOOD (MOVE-RANDOM) (MOVE-RANDOM)) (PROGN2 (PROGN2
(IF-FOOD-ADJACENT (PICK-UP) (MOVE-RANDOM)) (PICK-UP))
(IF-FOOD-HERE (PROGN2 (PICK-UP) (IF-FOOD-ADJACENT
(MOVE-RANDOM) (PROGN2 (PICK-UP) (MOVE)))) (IF-FOOD-ADJACENT
(DROP-FOOD) (MOVE-RANDOM))))) (IF-FOOD-HERE (PROGN2
(PICK-UP) (MOVE)) (IF-CARRYING-FOOD (PROGN2 (DROP-FOOD)
(DROP-FOOD)) (IF-CARRYING-FOOD (PROGN2 (IF-FOOD-ADJACENT
(IF-FOOD-ADJACENT (PROGN2 (IF-CARRYING-FOOD (MOVE-RANDOM)
(MOVE-RANDOM)) (PROGN2 (PROGN2 (DROP-FOOD) (PICK-UP))
(IF-FOOD-HERE (PICK-UP) (IF-FOOD-ADJACENT (DROP-FOOD)
(MOVE-RANDOM))))) (IF-FOOD-HERE (IF-FOOD-ADJACENT (PICK-UP)
(MOVE-RANDOM)) (DROP-FOOD))) (PROGN2 (PICK-UP)
(MOVE-RANDOM))) (IF-FOOD-ADJACENT (DROP-FOOD)
(MOVE-RANDOM))) (MOVE-RANDOM))))) (PROGN2 (IF-FOOD-ADJACENT
(IF-FOOD-ADJACENT (MOVE) (IF-FOOD-ADJACENT (PROGN2
(IF-FOOD-ADJACENT (IF-FOOD-ADJACENT (IF-FOOD-HERE
(IF-FOOD-ADJACENT (MOVE-RANDOM) (MOVE)) (PICK-UP))
(IF-CARRYING-FOOD (PROGN2 (DROP-FOOD) (DROP-FOOD))
(DROP-FOOD))) (PROGN2 (MOVE-RANDOM) (MOVE-RANDOM)))
(IF-CARRYING-FOOD (PICK-UP) (IF-CARRYING-FOOD (PROGN2
(IF-CARRYING-FOOD (PICK-UP) (IF-FOOD-ADJACENT (MOVE-RANDOM)
(IF-FOOD-HERE (PICK-UP) (IF-FOOD-ADJACENT (DROP-FOOD)
(MOVE-RANDOM))))) (PICK-UP)) (DROP-FOOD)))) (PROGN2 (PICK-UP)
(PICK-UP)))) (PROGN2 (PICK-UP) (IF-FOOD-ADJACENT (MOVE)
(MOVE-RANDOM)))) (PROGN2 (IF-FOOD-ADJACENT (MOVE) (PICK-UP))
(PROGN2 (PICK-UP) (MOVE)))))
```

Figure 32: The best-of-run program from generation 34.

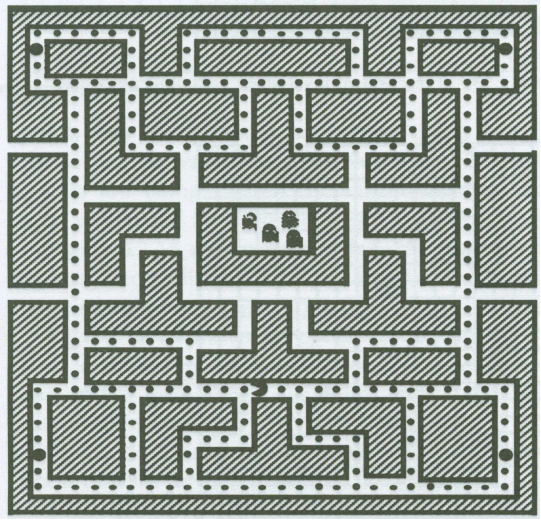

Figure 33: Pac Man screen.

- Environment:
  - Figure 33 shows the 31×28 toroidal grid on which the game is played. The majority of the squares are filled in, thus limiting the movement of the Pac Man to a maze of narrow corridors. There are two tunnels connecting the left side of the screen to the right side (and vice versa).
  - The Pac Man begins at position (13, 23) of the screen (in a coordinate system where all rows and columns are positively numbered and where the origin is in the upper left corner).
  - The four monsters (colored red, green, brown, and blue) begin the game in the 3×4 den in the center of the screen. As the game progresses, the monsters emerge at various times from their den at its doorway at position (13, 11).
  - The goal of the game is to maximize "points". Many, but not all, of the squares along the corridors contain small dots (i.e., food pellets, which are worth 10 points when encountered for the first time and eaten) by the Pac Man. Four of the squares contain "energizer" that are worth 50 points when encountered for the first time by the Pac Man.
  - Shortly after the game begins at time step 0, the monsters start emerging, one at a time, from their den. Any of the four monsters will eat the Pac Man if it catches him.
  - The monsters each have a rather limited span of attention. Out of every 25 time steps, they spend 20 time steps moving with the deliberate strategy of chasing the Pac Man whenever they see it. For five time steps out of every 25, the monsters abruptly change direction and shoot down some new corridor at random. The unpredictability of the four monsters magnifies their threat to the Pac Man.
  - A valuable piece of moving fruit appears at one of the entrances to the upper tunnel, namely at either position (0, 8) or position (28, 8) at time steps 25 and 125. The moving fruit moves unevasively and sluggishly around the screen for 75 time steps and then disappears.
  - If the Pac Man catch a moving fruit, he collects 200 points. Thus, while the moving fruit is present on the screen, the Pac Man's priorities shit toward capturing

this target of opportunity.

  - When the game starts, the highest priority of the Pac Man is to evade the monsters. To the extent that this first priority is being achieved, his second priority is to pursue and capture the moving fruit, if it is currently present on the screen. To the extent that this second priority is being achieved or is inapplicable, his third priority is to eat the dots.
  - Whenever the Pac Man eats one of the energizers, all four monsters immediately turn blue and remain blue for a latency period of 40 time steps. When the monsters are blue, the roles of pursuer and evader are reversed. The payoff for eating any one blue monster during the latency period caused by one energizer is a hefty 200 points. The payoff for eating additional monsters increases exponentially; eating a second one is worth 400, a third 800, and a fourth 1600 points.
  - Thus, when the monster are blue, the Pac Man's task and priorities change radically. His highest priority during the latency period is to chase the monsters. Eating an energizer during the latency period is usually a bad idea because it destroys a later opportunity to score a much larger number of points.
  - A good tactic for the Pac Man is to actively attract the attention of several monsters and then eat the energizer when the monsters are close enough for him to catch during the relatively brief blue latency period. Of course, it is inherently very dangerous to the Pac Man to have several monsters closely chasing him prior to their being turned blue.

- The human player normally controls the motion of the yellow Pac Man icon using human intelligence. In addition, the typical human player uses global knowledge of the grid to plan his play. When viewed globally, this game is a complex combination of combinatorial optimization and distance minimization (i.e., a form of the travelling-salesman problem), maze following, risk assessment, planning, and task prioritization.
- Here, we do not attempt to find a strategy that incorporates all these complex aspects of the game. Instead, we define the functions for this problem so as to focus on the game that emphasizes task prioritization.
- There are 15 primitive operators for this problem, and they can be divided into six distinct groups
  - Two of the primitive operators are conditional branching operators:
    - `IFB` (If Blue) is a two-argument conditional branching operator that executes its first argument if the monsters are currently blue and otherwise executes the second argument.
    - `IFLTE` (If-Less-Than-or-Equal) is a four-argument conditional comparative operator that executes its third argument if its first argument is less than or equal to its second argument and otherwise executes the fourth argument.
  - Three of the primitive operators relate to the nearest uneaten energizer:
    - `APPILL` (Advance-to-Pill) advances the Pac Man along the shortest route to the nearest uneaten energizer. In the event of a tie between routes, this function (and all other such functions) makes an arbitrary decision to resolve the conflict. This function (and all other such functions for which a return value is not specified in its description) returns the facing direction encoded as a modulo 4 number (with 0 being north, 1 being east, etc.).

    - * `RPILL` (Retreat-from-Pill) causes the Pac Man to retreat from the nearest uneaten energizer. That is, the Pac Man moves in a direction as close to topologically opposite as possible from the direction of the shortest route to the nearest energizer.
    - * `DISPILL` (Distance-to-Pill) returns the shortest distance, measured along paths of the maze, to the nearest uneaten energizer.
  - Three of the primitive operators relate to the monsters (called "Monster A") that is currently nearest as measured along paths of the maze (excluding the ghost of any monster that has been eaten by the Pac Man and whose eyes are returning to the monster den):
    - * `AGA` (Advance-to-Monster-A) advances the Pac Man along the shortest route to the nearest monster.
    - * `RGA` (Retreat-from-Monster-A) causes the Pac Man to retreat from the nearest monster in a manner equivalent to retreating from the energizer described above.
    - * `DISGA` (Distance-to-Monster-A) returns the shortest distance, to the nearest monster.
  - Three additional functions relate to the second nearest monster (called "Monster B"), and are defined in the same manner as above:
    - * `AGB`
    - * `RGB`
    - * `DISGB`
  - Two primitive operators relate to uneaten dots:
    - * `AFOOD` (Advance-to-Food) advances the Pac Man along the shortest route to the nearest uneaten dot.
    - * `DISU` (Distance-to-Uneaten-Dot) returns the shortest distance to nearest uneaten dot.
  - Two functions relate to the moving fruit (if any is present on the screen at the same time):
    - * `AFRUIT` (Advance-to-Fruit) advances the Pac Man along the shortest route to the moving fruit (if any is present on the screen at the time).
    - * `DISF` (Distance-to-Fruit) returns the shortest distance to the moving fruit. If no moving fruit is present on the screen, this function (and all other functions that may, at any time, try to measure the distance to a currently nonexistent object) returns a large number.
- Terminal set = {`(APILL)`, `(RPILL)`, `(DISPILL)`, `(AGA)`, `(RGA)`, `(DISGA)`, `(RGB)`, `(AGB)`, `(DISGB)`, `(AFOOD)`, `(DISU)`, `(AFRUIT)`, `(DISF)`}.
- Function set = {`IFB`, `IFLTE`}.
- Raw fitness is the number of points that the Pac Man scores before he is eaten by a monster or at the moment when all of the 22 food pellets have been eaten.
- The maximum value of raw fitness is 2220 for the 222 food pellet, 4000 for the two pieces of moving fruits, and 12000 for capturing the monsters while they are blue, for a grand total of 18220.
- Although we provided a facility for measuring the distance to the nearest monster and the second-nearest monster, we did not provide such a facility for the other two

```
(IFB (IFB (IFLTE (AFRUIT) (AFRUIT) (IFB (IFB (IFLTE (IFLTE (AGA)
(DISGA) (IFB (IFLTE (DISF) (AGA) (DISPILL) (IFLTE (DISU) (AGA)
(AGA) (IFLTE (AFRUIT) (DISU) (AFRUIT) (DISGA)))) (IFLTE (AFRUIT)
(RGA) (IFB (DISGA) 0) (DISGA))) (DISPILL)) (IFB (IFB (AGA)
(IFLTE (IFLTE (IFLTE (AFRUIT) (AFOOD) (DISGA) (DISGA)) (AFRUIT)
0 (IFB (AGA) 0)) (DISPILL) (IFLTE (AFRUIT) (DISPILL) (RGA) (DISF))
(AFRUIT))) 0) (AGA) (RGA)) (AFRUIT)) (IFLTE (IFLTE (RGA) (AFRUIT)
(AFOOD) (AFOOD)) (IFB (DISPILL) (IFLTE (RGA) (APILL) (AFOOD)
(DISU))) (IFLTE (IFLTE (RGA) (AFRUIT) (AFOOD) (RPILL)) (IFB
(AGA) (DISGB)) (IFB (AFOOD) 2) (IFB (DISGB) (AFOOD))) (IFB
(DISPILL) (AFOOD)))) (RPILL)) (IFB (DISGB) (IFLTE (DISU) 0
(AFOOD) (AGA)))) (IFB (DISU) (IFLTE (DISU) (DISU) (IFLTE (IFLTE
(AFRUIT) (AFOOD) (DISPILL) (DISGA)) (AFRUIT) 0 (IFB (AGA) 0))
(RGB))))
```

Figure 34: The best-of-run program from generation 35 for the Pac Man problem.

monsters. Because of this imitation in our programming of the game, it is probably not possible to score anywhere near the potential 18220, since attainment of the maximum score requires simultaneously maneuvering all four monsters into close proximity to an energizer and to the Pac Man and then eating all four monsters while they are blue.

- Simulation Result:
  - In all, in generation 35 of one run, the best program for the Pac Man scored 2220 points for the 222 food pellets, 200 for the four energizers, 4000 for the two pieces of moving fruits, and 3000 for the monsters, for a total of 9420 points. Thus, this program is not optimal. Figure 34 is the best-of-run program from generation 35.

Topic 6: Evolution of Subsumption

1. Introduction

   - The conventional approach to building control systems for autonomous mobile robots is to decompose the overall problem into a series of functional units that perform functions such as perception, modeling, planning, task execution, and motor control.
   - A central control system then executes each functional unit in this decomposition and passes the results on to the next functional unit in an orderly, closely coupled, and synchronized manner.
   - An alternative to the conventional centrally controlled way of building control systems for autonomous mobile robots is to decompose the problem into a set of asynchronous task-achieving behaviors.
   - In this alternative approach, called *subsumption architecture*, the overall control of the robot is achieved by the collective effective of the asynchronous local interactions of the relatively primitive task-achieving behaviors, all communicating directly with the world and among themselves.
   - In the subsumption architecture, each task-achieving behavior typically performs some low-level function. For example, the task-achieving behaviors for an autonomous mobile robot might include wandering, exploring, identifying, avoiding objects, building maps, panning changes to the world, monitoring changes to the world, and reasoning about the behavior of objects.
   - In the subsumption architecture, various subsets of the task-achieving behaviors typically exhibit some partial competence in solving a simpler version of the overall problem. For example, Figure 35 shows most of the major features of the subsumption architecture.
   - All three task-achieving behaviors are in direct communication with the robot's environment. If the current environment satisfies the applicability predicate of a particular behavior, the gate allows the behavioral action to feed out onto the output line of that behavior.
   - Since the task-achieving behaviors of the subsumption architecture operate independently, there is a need to resolve conflicts among behaviors. The right part of Figure 35 shows a hierarchical arrangement of suppressor nodes used to resolve potential conflicts among the outputs of the three task-achieving behaviors.
   - The question arises as to whether it is possible to evolve a subsumption architecture to solve problems. This evolution would involve finding
     - the appropriate behaviors, including the appropriate applicability predicate and the appropriate behavioral actions for each, and
     - the appropriate conflict resolution hierarchy.

2. Wall-Following Robot

   - Mataric's Research Work:
     - Mataric (1990) has implemented the subsumption architecture by conceiving and writing a set of four LISP computer programs for performing four task-achieving behaviors which together enable an autonomous mobile robot called TOTO to follow the walls in an irregular room.

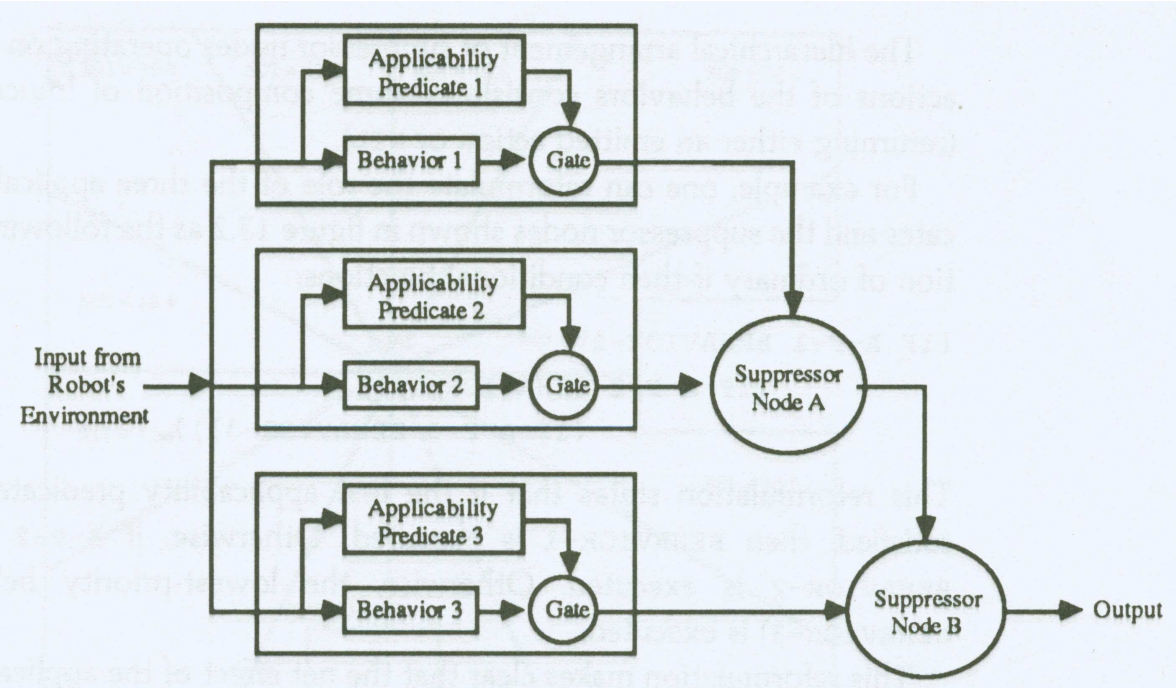

Figure 35: Subsumption architecture with three task-achieving behaviors.

- Environment
  - Figure 36 shows a robot at point (12, 16) near the center of an irregularly shaped room. The north (top) wall and the west (left) wall are each 27.6 feet long.
  - The robot has 12 sonar sensors, which report the distance to the nearest wall as a floating-point number in feet. These sonar sensors (each covering a 30° sector) together provide 360° coverage around the robot.
  - A protrusion from the wall may be indicated by an irregular in the sequence of consecutive sonar measurements.
  - In addition to the 12 sonar sensors, the robot has a sensor called `STOPPED` to determine if the robot has reached a wall and is stopped against it. This is, the input to the robot consists of 12 floating-point numbers from the sonar sensors and one Boolean input.
  - Mataric's robot was capable of executing five primitive motor functions:
    - moving forward by a constant distance,
    - moving backward by a constant distance (which was 133% of the forward distance),
    - turning right by 30°,
    - turning left by 30°, and
    - stopping.
  - In addition, three constant parameters are associated with the problem.
    - `EDG`: the edging distance representing the preferred distance between the robot and the wall was 2.3 feet,
    - `MSD`: the minimum safe distance between the robot and the wall was 2.0 feet,

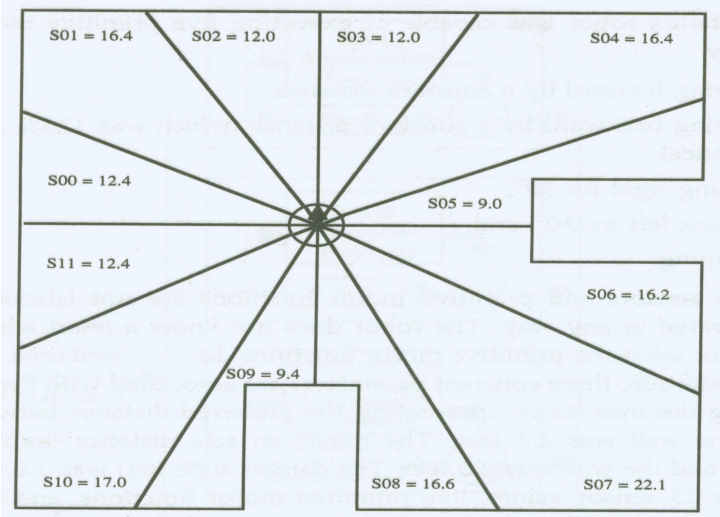

Figure 36: Robot with 12 sonar sensors located near middle of an irregular room.

  - · DZ: the danger zone was 1.0 feet.
  - ∗ Mataric four LISP programs (STROLL, AVOID, ALIGN, and CORRECT) correspond to task-achieving behaviors which she conceived and wrote. These programs contains 25 different atoms and 14 different functions. These 14 functions consisted of the five primitive motor functions and nine additional LISP functions (IF, AND, NOT, COND, >, >=, =, <=, and <).

- Genetic Programming:
  - In starting to apply GP to this problem, we started with Mataric's set of terminals and primitive functions.
  - Learning algorithms in general require repeated experimentation in order to accumulate the information used by the algorithm to solve the problem. Learning from this problem necessarily requires the robot to perform the overall task a certain number of times in order to accumulate the information required by the learning algorithms. Simulation of the robot provides the practical means to accumulate this information.
  - Since our simulated robot was not intended to ever stop and could not, in any event, be damaged by running into a wall, we had no use for the STOP function, the STOPPED sensor, or the related danger zone parameter DZ. In our simulations, we let our robot push up against the wall, and, if no change of state occurred after an additional time step, the robot was viewed as having stopped, thereby ending that particular simulation.
  - SS (shortest sonar) is the minimum of all 12 sonar distances S00, S01,..., S11.
  - The conditional comparative operator IFLTE (If-Less-Than-Or-Equal) can perform all the functions Mataric achieved using COND, AND, NOT, IF, >, >=, =, <=,, and <. It takes four arguments.
  - Function set = {IFLTN, PROGN2}.
  - Terminal set = {S00,S01,...,S11,MSD,EDG,SS,(MF),(MB),(TR),(TL)}.
    Question: is there any problem when we employ this terminal set?
  - It is necessary that these functions return some numerical value. We decided that each of the four moving and turning functions would return the minimum of the

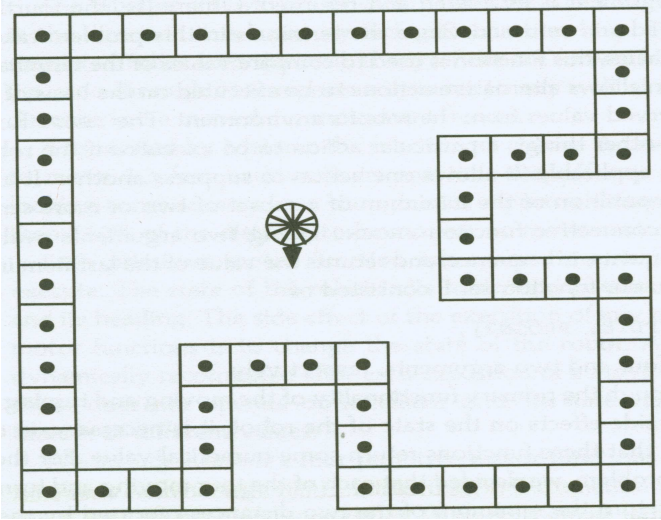

Figure 37: Irregular room with 56 square tiles.

two distances reported by the two sensors that look in the direction of forward movement (i.e., `S02` representing the 11:30 direction and `S03` representing 12:30 direction).

- Raw fitness can be the portion of the perimeter traversed by the robot within some reasonable amount of time.
- Noting that Mactaric's edging distance was 2.3 feet, we visualized placing contiguous 2.3-foot-square tiles along the perimeter of the room. See Figure 37.

- It is usually necessary to establish a reasonable maximum amount of time for the process.
  - If the 56 tiles were arranged in a straight line, traversing 56 tiles (129 feet) would require 129 time steps if the robot were traveling forward (when its speed is 1.0 foot per second) but only 99 time steps if the robot were traveling backward (when its speed is 1.3 feet per second).
  - At the beginning, the robot must get from its starting point in the middle of the room to some edge of the room. This adds modest number of additional steps, depending on the robot's starting point.
  - A minimum of 39 turns of 30° (each taking one time step) are required to make the 13 turns of 90° necessary to travel along the entire periphery of the irregular room.
  - Some backward and forward adjustments are required to align the robot to the wall after each turn.
- These four considerations suggest that the minimum number of time steps required by a wall-following robot for this problem would be between about 150 and 180 time steps *plus* a few additional steps for the experimentation associated with turning and post-turning alignment.
- We defined raw fitness to be the number of tiles that are touched by the robot within 400 time steps.. If a robot travels onto each of these 56 tiles within 400 time steps, we will regard it as successful in following the walls.
- Figure 38 summarizes the key features of the wall following problem.

Objective:	Find a computer program which, when executed by a mobile robot moves along the periphery of an irregular room.
Terminal set:	The distances to the nearest wall in each of 12 directions as reported by 12 sonar senses, two constants MSD and EDG, one derived value SS, Move Forward (MF), Move Backwards (MB), Turn Right (TR), and Turn Left (TL).
Function set:	If-Less-Than-Or-Equal (IFLTE), and the connective PROGN2.
Fitness cases:	One fitness case.
Raw fitness:	Number of 2.3 foot square tiles along the periphery of the room that are touched by the robot within an allotted time of 400 time steps.
Standardized fitness:	Total number of tiles (56) minus raw fitness.
Hits:	Equals raw fitness for this problem.
Wrapper:	None.
Parameters:	$M = 1{,}000$ (with over-selection). $G = 51$ (see text).
Success predicate:	An S-expression scores 56 hits.

Figure 38: Tableau for example 1 of the wall-following problem.

- Example 1
  - Figure 39 shows the wall-following trajectory of the robot while executing the best-of-run program from generation 57. This program starts by briefly moving at random in the middle of the room. However, as soon as it reaches the wall, it moves along it. It touches 100% of the 56 tiles along the periphery of the room. This program takes 395 time steps.
- Example 2
  - In the above discussion of the wall-following problem, the solution was facilitated by the presence of the sensors SS in the terminal set and the fact that the terminals (MF), (MB), (TL), and (TR) returned a numerical value equal to the minimum of the two designated sensors.
  - The wall-following problem can also be solved without the terminal SS being in the terminal set and with the four terminals each returning a constant value of 0. We call these four new terminals (MF0), (MB0), (TL0), and (TR0).
  - Terminal T_0 = {S00,S01,...,S11,MSD,EDG,(MF0),(MB0),(TR0),(TL0)}.
  - Figure 40 shows the trajectory of the robot executing the best-of-run program from generation 102. This program causes the robot to move backward as it follows the wall.

3. Box Moving Robot

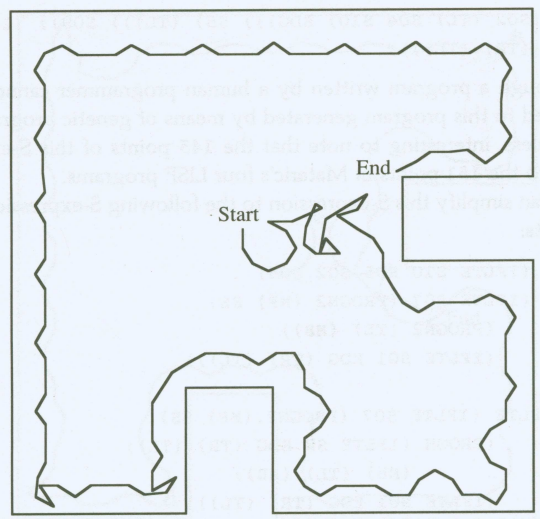

Figure 39: Wall-following trajectory of a best-of-run program from generation 57 for example 1.

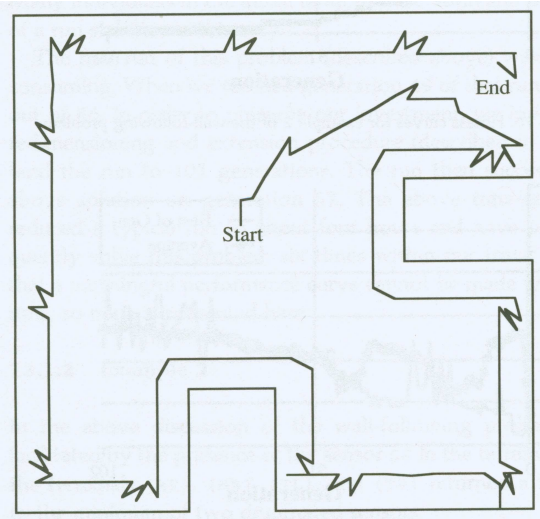

Figure 40: Trajectory of backward-moving best-of-run program from generation 102 for example 2 of the wall-following problem.

- In the box moving problem, an autonomous mobile robot must find a box located in the middle of an irregularly shaped room and move it to the edge of the room.
- After the robot finds the box, it can move the box by pushing against it. However, this subtask may prove difficult. If the robot applies force at a point other than the midpoint of the edge of the box or at an angle other than perpendicular to the edge, the box will start to rotate. The robot will then lose contact with the box and will probably then fail to push the box to the wall in a reasonable amount of time.
- We proceed with the box moving problem by using the same irregular room, the same robot, the same primitive moving and turning functions, and the same 12 sonar sensors as we used in the wall-following problem.
- We used a 2.5-foot-wide box in the room with 27.3-foot west and north walls. If the robot applies its force orthogonally to the midpoint of an edge of the box, it will move the box about 0.33 foot per time step. For this problem, the sonar sensors now detect the nearest object (whether wall or box).
- Example 1
  - The terminal set T is the same for the box moving problem as for the wall-following problem, except that the constants `MSD` and `EDG` (which are irrelevant to box moving) and the function `MB` (which was extraneous) were deleted.
  - Terminal set T = {`S00, S01,..., S11, SS, (MF), (TR), (TL)`}.
  - Function set F = {`IFBMP, IFSTK, IFLTE, PROGN2`}.
  - The function `IFBMP` (if bumped) and `IFSTK` (if stuck) are based on the `BUMP` detector and the `STUCK` detector. Each of these functions evaluates its first argument if the condition being detected applied, but otherwise evaluates its second argument.
  - We started the robot at four different starting positions in the room.
  - The fitness measure for this problem is the sum of the distances, taken over the four fitness cases, between the wall and the point on the box that is closest to the nearest wall at the time of termination of the fitness case.
  - A fitness case terminates upon execution of 350 time steps or when any part of the box touches a wall. If the box remains at its starting position for all four fitness cases, the raw fitness is 26.5 feet. If the box ends up touching a wall prior to timing out for all four fitness cases, the raw fitness is 0.
  - Figure 41 shows the irregular room, the starting positions of the box, and the starting position of the robot for the particular fitness case in which the robot starts in the southeast part of the room.
  - The raw fitness of a majority of programs from generation 0 is 26.5 since they cause the robot to stand still, to wander around aimlessly without ever finding the box, or, in the case of the individual program shown in the figure, to move toward the box without reaching it. Figure 42 shows the trajectory of the best-of-generation program from generation 0 when the robot starts in southeast corner of the room. This program containing 213 points finds the box and moves it a short distance for one of the four fitness cases, thereby scoring a raw fitness of 24.5.
  - By generation 45 of the run, the best-of-generation program was successful, for all four fitness cases, in finding the box and pushing it to the wall. Figure 43 shows the trajectory of the robot and the box for the best-of-run program from

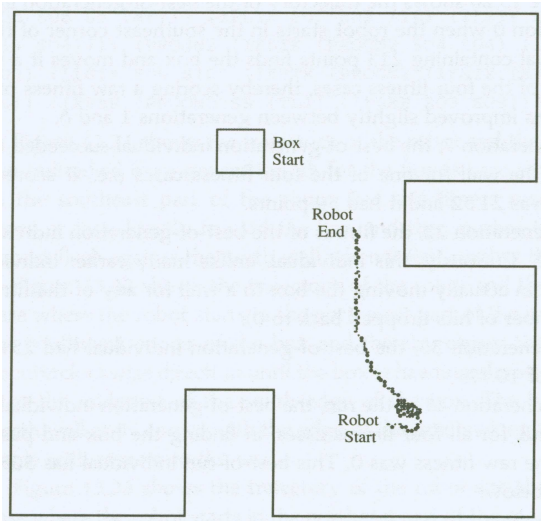

Figure 41: Trajectory of robot from generation 0 of the box moving problem that moves, but fails to reach the box for one fitness case.

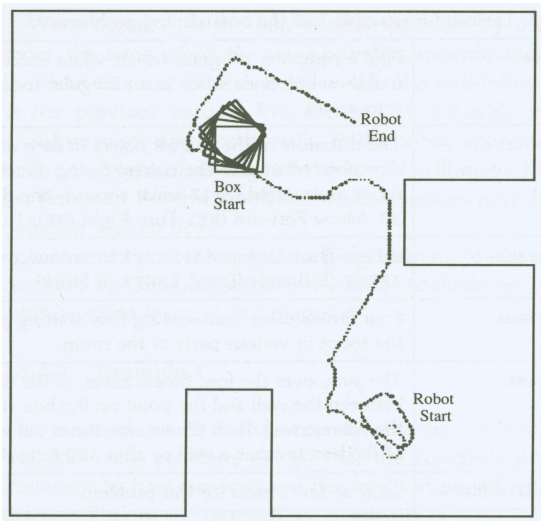

Figure 42: Trajectory of the best-of-generation program for generation 0 of the box moving problem for one fitness case.

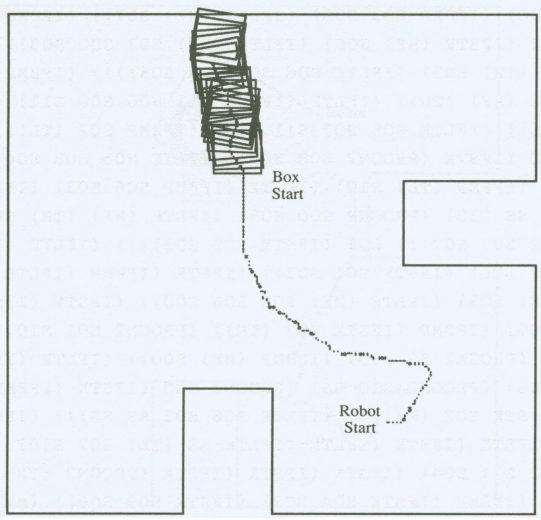

Figure 43: Trajectory of the best-of-generation program for generation 45 of the box moving problem with the robot starting in the southeast part of the room.

generation 45 for the fitness case where the robot starts in the southeast part of the room.

- Example 2
  - In the above discussion of the box moving problem, the solution was facilitated by the presence of the sensor `SS` in the terminal set and the fact that the functions `MF, TL,` and `TR` returned a numerical value equal to the minimum of several designated sensors.
  - This problem can also be solved without the terminal `SS` being in the terminal set and with the three function each returning a constant value of 0. We call these three new functions `MF0, TL0,` and `TR0`.
  - Function set F = {`IFBMP, IFSTK, IFLTE, PROGN2`}.
  - Terminal set T = {`S00, S01,..., S11, (MF0), (TR0), (TL0)`}.
  - We raised the population size from 500 to 2000 for example 2 because we thought the problem might be more difficult to solve in this form.
  - There is a simultaneous emergence of three 100%-correct programs scoring four hits on generation 20.

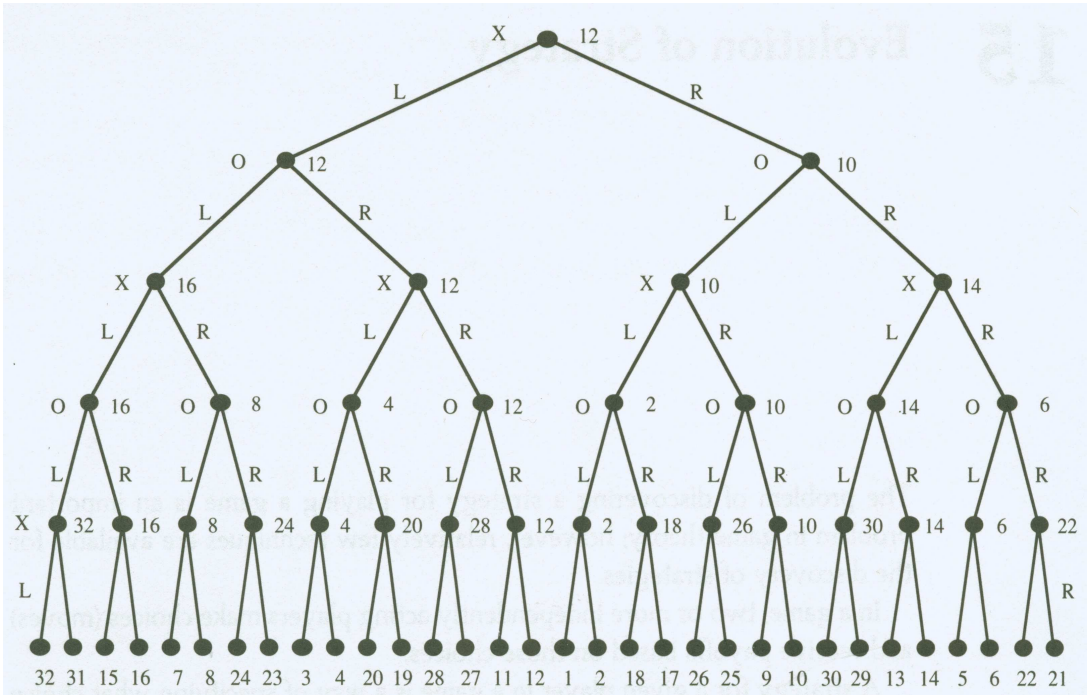

Figure 44: Game tree for 32-outcome discrete game.

Topic 7: Evolution of Strategy

1. Introduction

   - The problem of discovering a strategy for playing a game is an important problem in game theory, however, relatively few techniques are available for the discovery of strategies.
   - In a game, two or more independently acting players make choices (moves) and receive payoffs based on those choices.
   - A strategy for a given player in a game is a way of specifying what choice the player is to make at every point in the game from the set of allowable choices at that point, given all the information that is available to the player at that point.
   - The problem of discovering a strategy for playing a game can be viewed as requiring the discovery of a computer program. Depending on the game, the desired computer program takes as its input(s) either the entire history of past moves in the game or the current state of the game. The desired computer program then produces the next move as its output.

2. A Discrete Game

   - Environment
     - Consider the discrete 32-outcome game whose game tree is presented in extensive for in Figure 44.
     - This game is a two-person, competitive, zero-sum game in which the players make alternating moves. On each move, a player can choose to go `L` (left) or `R` (right).

- Each internal point of this tree is labeled with the player who must move. Each line is labeled with the choice (either L or R) made by the moving player. Each endpoint of the tree is labeled with the payoff (to player X).
- After player X has made three moves and player O has made two moves, player X receives (and player O pays out) the particular payoff shown at the particular endpoint of the game tree.

- Strategy
  - Since this 32-outcome discrete game is a game of complete information, each player has access to complete information about his opponent's previous moves and his own previous moves.
  - This information is contained in four variables XM1 (X's move 1), OM1 (O's move 1), XM2 (X's move 2), OM2 (O's move 2).
  - These variables each assume one of three possible values: L (left), R (right), or U (undefined). A variable is undefined (U) prior to the time when the move to which it refers has been made. Thus, at the beginning of the game, all four variables are undefined.
  - The particular variables that are defined and undefined at a particular time can be used to indicate the point to which play has progressed in the game. For example, if both players have moved once, XM1 and OM1 are both defined (each being either L or R) but the other two variables (XM2 and OM2) are still undefined (i.e., have the value U).
  - A strategy for a particular player in a game specifies which choice that player is to make for every possible situation that may arise for that player.
  - For this particular game, a strategy for player X must specify his first move if the game is just beginning. Second, a strategy for player X must specify his second move if player O has already made exactly one move. Third, a strategy for player X must specify his third move if player O has already made exactly two moves.
  - A strategy is a computer program whose inputs are the relevant historical variables (XM1, OM1, XM2, OM2) and whose output is a move (L or R) for the player involved.
  - The testing functions CXM1, COM1, CXM2, and COM2 provide the ability to test each historical variables (XM1, OM1, XM2, OM2) that is relevant to deciding a player's move.
  - Each of these functions is a specialized form of the CASE function in LISP, each takes three arguments. For example, the function CXM1, it evaluates its first argument if XM1 is undefined; it evaluates its second argument if XM1 is L (left); and it evaluates its third argument if XM1 is R (right).
  - Terminal set T = {L, R}.
  - Function set F = {CXM1, COM1, CXM2, COM2}.
  - The raw fitness of a particular strategy for a player is the sum of the payoffs received when that strategy is played against all possible sequence of combinations of moves by the opposing player.
  - Thus, when we compute the fitness of an X strategy, we must test the X strategy against all four possible combinations of O moves, namely O choosing L or R for moves 1 and 2. Similarly, when we compute the fitness of an O strategy, we must

test it against all eight possible combinations of `X` moves, namely `X` choosing `L` or `R` for moves 1, 2, and 3.

- Simulation Result
  - In one run, the best-of-generation game playing strategy for player `X` in generation 6 had raw fitness of 88 and scored four hits:
    `(COM2 (COM1 (COM1 L (CXM2 R (COM2 L L L) (CXM1 L R L)) (CXM1 L L R)) L R) L (COM1 L R R)).`
    This strategy for player `X` simplifies to
    `(COM2 (COM1 L L R) L R)`
  - In one run, the best-of-generation strategy for player `O` in generation 9 had a raw fitness of 52 and scored eight hits and was, in fact, the minimax strategy for player `O`:
    `(CXM2 (CXM1 L (COM1 R L L) L) (COM1 R L (CXM2 L L R))`
    `(COM1 L R (CXM2 R (COM1 L L R) (COM1 R L R))))).`

3. A Differential Pursuer-Evader Game

- In the game of simple pursuit described in Isaacs' *Differential Games* (1965), the goal is to find a minimax strategy for one player when playing against a minimax opponent.
- Environment
  - This differential pursuer-evader game is a two person, competitive, zero sum, simultaneous-move, complete-information game in which a fast pursuing player P is trying to capture a slower evading player E.
  - The choice available to a player at a given moment consists of choosing a direction (angle) in which to travel.
  - In this game, the players may travel anywhere in a plane and both players may instantaneously change direction without restriction. Each player travels at a constant speed, and the pursuing player's speed w_p (1.0) is greater than the evading player's speed w_e (0.67).
  - The state variables of the game are x_p, y_p, x_e, and y_e, representing the coordinate positions (x_p, y_p) and (x_e, y_e) of the pursuer P and evader E in the plane.
  - Figure 45 shows the purser and the evader. At each time step, both players know the positions (state variables) of both players. The choice for each player is to select a value of his control variable (i.e., the angular direction in which to travel).
  - The pursuer's control variable is the angle ϕ (from 0 to 2π radians), and the evader's control variable is the angle ψ
  - The analysis of this game can be simplified by reducing the number of state variables from four to two. This state reduction is accomplished by simply viewing the pursuer P as being at the origin point (0, 0) of a new coordinate system at all times and then viewing the evader E as being at position (x, y) in this new coordinate system.
  - The state-transition equations for the evader E are

$$x(t+1) = x(t) + w_e \cos\psi - w_p \cos\phi,$$
$$y(t+1) = y(t) + w_e \sin\psi - w_p \sin\phi.$$

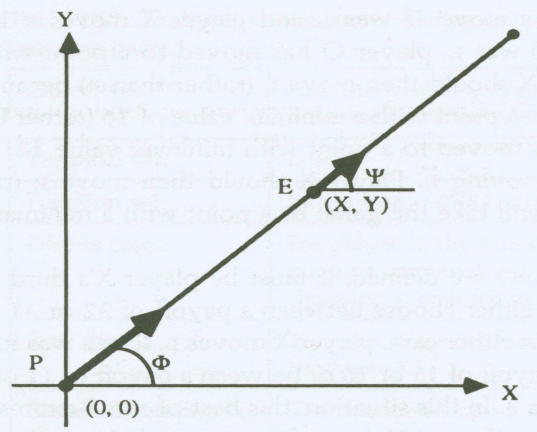

Figure 45: Purser P and Evader E.

- We use a set of 20 fitness cases consisting of random initial condition positions (x_i, y_i) for the evader. Each initial condition value of x_i and y_i lies between -5.0 and +5.0
- The payoff for a given player is measure by time. The payoff for the player P is the total time it takes to capture the evader E over all the initial condition cases. The pursuer tries to minimize the time to capture.
- The payoff for the evader is the total time of survival for E. The evader tries to maximize this time of survival.
- A maximum allowed time of 100 time steps is established so that if a particular pursuer strategy has not made the capture within that amount of time, that maximum time becomes the payoff for that particular fitness case and that particular strategy.
- For this game, the best strategy for the pursuer P at any give time step is to chase the evader E in the direction of the straight line currently connecting the pursuer to the evader. And, the best strategy for the evader E is to race away from the pursuer in the direction of the straight line connecting the pursuer to the to the evader.
- We start by evolving a minimax pursuer. In doing so, each program the population of pursuing programs is tested against one minimax evader. The optimal evader travels with the established constant speed w_e in the angular direction specified by two argument Arctangent function.
- We later, separately, evolve a minimax evader. Each program in the population of evading programs is tested against the minimax pursuer.
- Terminal set T = {`X, Y, R`}.
  where `X` and `Y` represent the position of the evader E in the plane in a reduced coordinate system. `R` is the ephemeral floating-point constant ranging between -1.000 and +1.000.
- Function set F = {`+, -, ×, %, exp, IFLTZ`}.
  where `IFLTZ` means "if less than zero".
- Each computer program will evaluate to a number that provides *the new direction of motion*, in radians, for the pursuer.
- The best-of-run program from generation 48 closely matches the desired Arctangent behavior. A near-optimal evader has been similarly evolved using an

optimal pursuer.

4. Co-Evolution

   - Introduction
     - In the previous topic, we genetically bred the strategy for one player in a game by testing each program in the evolving population of strategies against the minimax strategy for the opposing player or against an exhaustive set of combinations of choices by the opposing player.
     - However, in game theory and in practice, one almost never has a *prior* access to a minimax strategy for the opposing player or the ability to perform an exhaustive test.
     - Since exhaustive testing is practical only for very small games, one faces a situation where genetically breeding a minimax strategy for one player requires already having the minimax strategy for the other players.
     - The environment is actually a composite consisting of both the physical environment and other independently acting biological populations of individuals which are simultaneously actively adapting to their environment.
   - Co-Evolution of a Game-Playing Strategy
     - In co-evolution, there are two (or more) populations of individuals. The environment for the first population consists of the second population. And, conversely, the environment for the second population consists of the first population.
     - This process is carried out by testing the performance of each individual in the first population against each individual from the second population. The average performance observed is called the *relative fitness* of that individual, because it represents the performance of that individual relative to the environment consisting of the entire second population.
     - Even though both initial populations are highly unfit, over a period of time, both populations of individuals will tend to co-evolve and to rise to higher levels of performance as measured in terms of absolute fitness.
     - Both populations do this without the aid of any externally supplied measure of absolute fitness serving as the environment. Co-evolution is a self-organizing, mutually bootstrapping process that is driven only by relative fitness.
     - In co-evolution, the relative fitness of a particular strategy in a particular population is the average of the payoffs that the strategy receives when it is played against fitness cases consisting of each strategy in the opposing population of strategies.
   - Simulation Result
     - The best-of-generation individual `O` strategy from generation 1 is a minimax strategy for player `O`. Between generations 2 and 14, the number of individuals in the `O` population equivalent to the minimax `O` strategy was 2, 7, 17, 28, 35, 40, 50, 64, 73, 83, 93, 98, and 107, respectively.
     - The best-of-generation individual `X` strategy from generation 14 is a minimax strategy for player `X`. Between generations 15 and 29, the number of individuals in the `X` population equivalent to the minimax `X` strategy was 3, 4, 8, 11, 10, 9, 13, 21, 24, 29, 43, 32, 52, 48, and 50, respectively.

– By generation 38, the minimax strategies for both players were becoming dominant.
– In summary, we have seen the discovery, via co-evolution, of the minimax strategies for both players in the 32-outcome discrete game. This mutually bootstrapping process found the minimax strategies for both players without using knowledge of the minimax strategy for either player.

Topic 8: Evolution of Classification

1. Introduction

   - Learning patterns that discriminate among problem solving choices is one approach to problem solving. The patterns that are learned may be expressed in many ways, including decision trees, sets of production rules, formal grammars, or mathematical equations.

2. Decision-Tree Induction

   - Introduction
     - One approach to classification and pattern recognition involves the construction of a decision tree.
     - ID3 is a hierarchical classification system for inducing a decision tree from a finite number of examples or training cases. Its goal is to assign each object in a universe of objects to a class. Each object in the universe is described in terms of attributes.
     - The ID3 system is presented with a set of training cases. A training case consists of the attributes of a particular object and the class to which it belongs. ID3 then generates a decision tree that can correctly reclassify any particular previously seen object, and, more important, ID3 can very often generalize and correctly classify a new object into its correct class.
     - The external points (leaves) of the decision tree produced by ID3 are labeled with the class names. Each internal point of the decision tree is labeled with an attribute test. One branch radiates downward from each such internal point of the tree for each possible outcome of the attribute test.
     - Quinlan (1986) presented a simple illustrative example of the ID3 hierarchical classification system for inducing a decision. In this example, the set of training cases consists of 14 objects representing characteristics of Saturday mornings. Each object in Quinlan's illustrative example belongs to one of the two classes, namely positive (class 1) and negative (class 0).
     - Each object has four attributes, namely, `temperature`, `humidity`, `outlook`, and `windy`.
       * `temperature`: `hot`, `mild`, `cool`.
       * `humidity`: `high`, `normal`.
       * `outlook`: `sunny`, `overcast`, `raining`.
       * `windy`: `true`, `false`.
     - Figure 46 shows Quinlan's example of a decision tree that successfully classifies the 14 training cases.
   - Genetic programming:
     - Terminal set T = {0, 1}.
     - Function set F = {`TEMP, HUM, OUT, WIND`}.
     - Each of these four attribute-testing functions operates much like the `CASE` statement in LISP. We obtain the function set by converting each of Quinlan's attributes into an attribute-testing function. Therefore, there are as many functions in our function set as there are attributes.

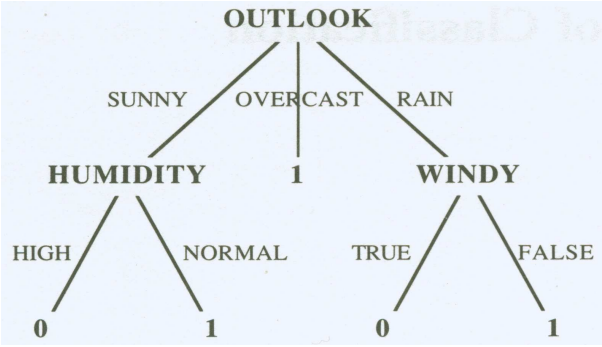

Figure 46: Decision tree for classifying Saturday morning into either class 0 or class 1.

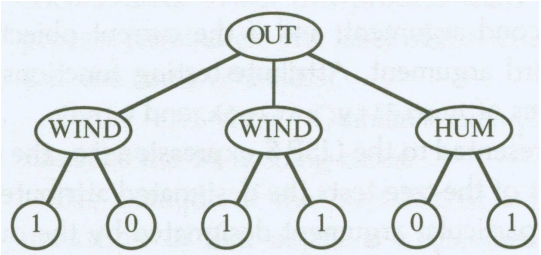

Figure 47: Genetically discovered S-expression found on generation 8 for classifying Saturday mornings.

- The raw fitness of a program is the number of training cases for which the program returns the correct class, and it ranges from 0 to 14.
- In one run, the following program emerged on generation 8 scores the maximal value of raw fitness, namely 14.
  `(OUT (WIND 1 0) (WIND 1 1) (HUM 0 1))`
- Since `(WIND 1 1)` is equivalent to just the constant atom 1, this S-expression is equivalent to the decision tree shown in Figure 47.

3. Grammar Induction

- The problem of grammar induction involves recognizing patterns existing in a sequence of data. This section demonstrates how to genetically breed a computer program that is capable of determining whether a given sequence of symbols is one of the sentences produced by some unknown grammar.
- Instead, we present this problem in terms of a biological metaphor, namely the problem of exon identification arising from the analysis of the DNA constituting the genome of a biological organism.
- For nucleiotide bases (i.e., A, C, G, and T) appear along the length of the DNA molecule.
- This section presents an example involving a simple grammar where we genetically discover an S-expression that is capable of identifying the subsequences of length 5 that belong to the language from a long sample sequence of length 1000 where the exons (i.e., subsequence to be accepted as belonging to the language) will occupy about 10% of the sequence. The alphabet will consist of the four letters, A, C, G, and T.

- An S-expression will identify exons by returning a `True` for each position of the given sequence which it believes to be an exon and by returning a `NIL` for each position which it believes to be an intron.
- Processing proceeds from the beginning (i.e., left) of the given sequence of bases. An S-expression may examine a variable number of positions to the right of the current position under examination.
- If an S-expression identifies a subsequence of the base sequence as an exon, it will return a `T` for all the positions between the current position and the rightmost position it examined.
- The current position is then advanced to the first position of the sequence that has not yet been examined. However, if an S-expression does not identify the subsequence as an exon, it returns a `NIL` for the current position only, and then the position under examination is advanced to the right by one position.
- There are four conditional operations for this problem:
  - `AIF` return `T` (`True`) if the current position of the sequence is adenine (A) and `NIL` otherwise.
  - `CIF` return `T` (`True`) if the current position of the sequence is cytosine (C) and `NIL` otherwise.
  - `GIF` return `T` (`True`) if the current position of the sequence is guanine (G) and `NIL` otherwise.
  - `TIF` return `T` (`True`) if the current position of the sequence is thymine (T) and `NIL` otherwise.
- `MHG` advances the position of the sequence under examination to the right by one position and returns `T`.
- Terminal set T = {`(AIF), (CIF), (GIF), (TIF), (MHG)`}.
- Function set F = {`AND, OR, NOT`}.
- The raw fitness is the sum, taken over the 1000 fitness cases, of the Hamming distances (0 or 1) between the value returned by the S-expression and the correct classification of that position.
- For this problem, an exon will always be of length 5 and begin with two leading A's and end with two trailing T's. The middle symbol of the sequence can be any letter.
- On generation 35, the best-of-generation program scored 1000 out of a possible 1000. It is a 100% correct solution to the problem.

4. Intertwined Spirals

- The goal is to distinguish two intertwined spirals. In Figure 48, two spirals coil around the origin three times in the x-y plane. The x-y coordinates of 97 points from each spiral are given. The problem involves learning to classify each points as to which spiral it belongs.
- Figure 48 shows the 97 points of the first spiral (indicated by squares, class +1) and the 97 points of the second spiral (indicated by circles, class -1).
- The terminal set for this problem consists of the x and y position of the given points and the ephemeral random floating-point constant `R` ranging between -1.000 and +1.000.

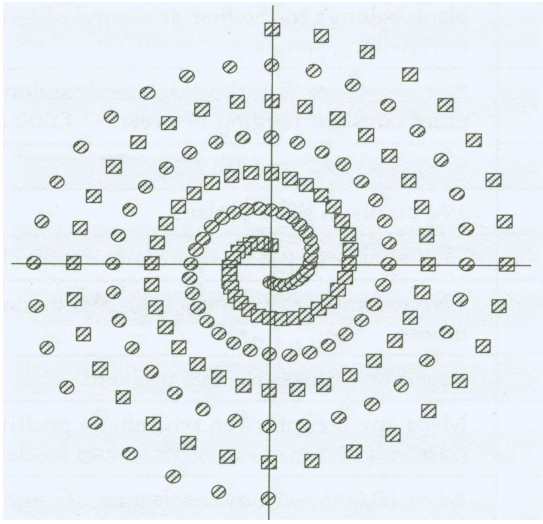

Figure 48: 194 points lying on two intertwined spirals.

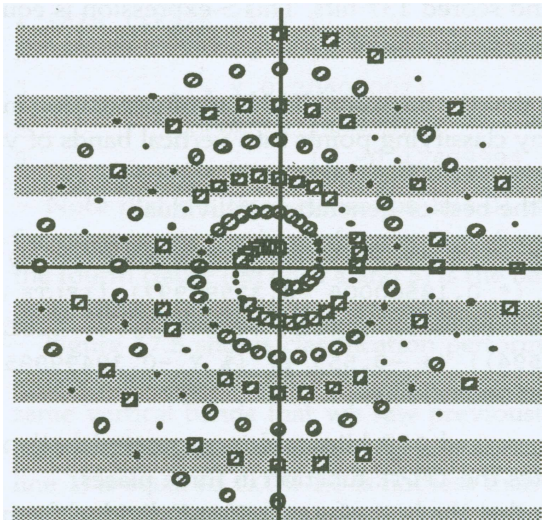

Figure 49: Classification performed by best-of-generation program form generation 36.

- Terminal set T = {`X, Y, R`}.
- Function set F = {`+, -,` $\times$`, %, IFLTE,` sin, cos}.
- The raw fitness is the number of points (0 to 194) that are correctly classified.
- Since the S-expressions in the population are compositions of functions operating on floating-point numbers and since the S-expressions in this problem must produce a binary output (+1 or -1) to designate the class, a wrapper (output interface) is required. **This wrapper maps any positive value to class +1 and maps any other value to class -1**.
- Figure 49 shows the classification performed by this best-of-run program. 100% of the points in this figure are correctly classified.

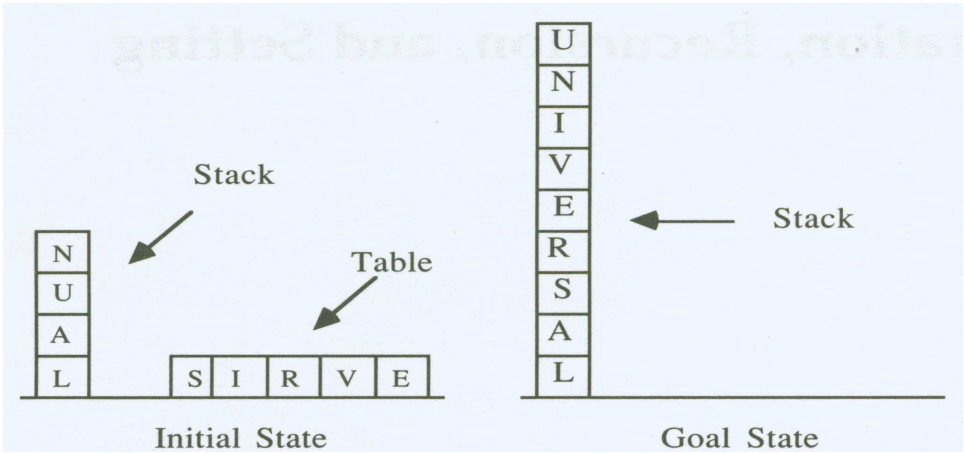

Figure 50: One possible initial state of the `STACK` and `TABLE` is shown at left. The goal state is shown at right.

Topic 9: Iteration, Recursion, and Setting

1. Block Stacking

   - Introduction
     - Planning in artificial intelligence and robotics requires finding a plan that receives information from sensors about the state of the various objects in a system and then uses that information to select actions to change the state of the objects in that system toward some desired state.
     - The block stacking problem is a planning problem requiring the rearrangement of uniquely labeled blocks into a specified order on a single target tower.
     - In the version of the problem involving nine blocks, the blocks are labeled with the nine different letters of FRUITCAKE or UNIVERSAL. The goal is to automatically generate a plan that solves this problem.
   - Environment
     - The `STACK` is the ordered set of blocks that are currently in the target tower (where the order is important). The `TABLE` is the set of blocks that are currently not in the target tower (where the order is not important and where they can be randomly accessed).
     - The initial configuration consists of certain blocks in the `STACK` and the remaining blocks on the `TABLE`.
     - The desired final configuration consists of all blocks being in the `STACK` in the desired order and no blocks remaining on the `TABLE`.
     - The left half of Figure 50 shows a possible initial state in which the four blocks NUAL are in the `STACK` and the five blocks, S, I, R, V, and E are on the `TABLE`. The right half of Figure 50 shows the nine blocks UNIVERSAL in the `STACK` in the desired order.
     - In the discussion here, we use the particular symbolic sensors and functions defined for this problem by Nilsson (1989). In particular, the following three sensors dynamically track the system:
       * The sensor `CS` dynamically specifies the top block of the `STACK`.

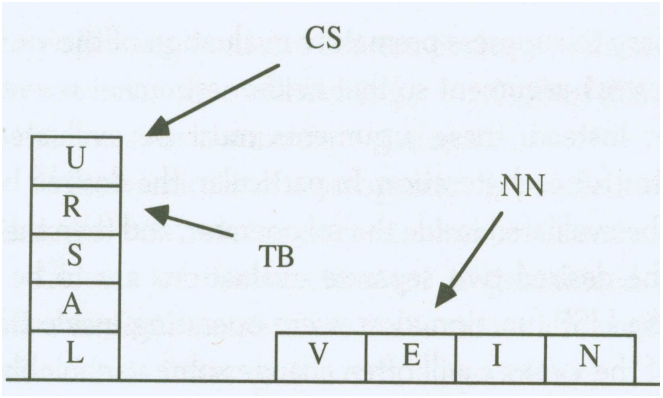

Figure 51: STACK with "URSAL" and TABLE with V, E, I, and N.

  - The sensor TB (top correct block) dynamically specifies the top block on the STACK such that it and all blocks below it are in the correct order.
  - The sensor NN (next needed) dynamically specifies the block immediately after TB in the goal universal (regardless of whether or not there are incorrect blocks in the STACK).
- For example, in Figure 51, the STACK consists of URSAL while the TABLE contains the four letters V, E, I, and N. The sensor CS is therefore U. The sensor TB is R, since RSAL are in the correct order on the STACK and U is out of place. The sensor NN is E, since E is the block that belongs on the top of RSAL.
- Terminal set = {TB, NN, CS}.
- Each of these terminals is a variable atom that may assume, as its value, one of the nine block labels or NIL.
- The function set should contain functions that might be needed to express the solution to the problem. The following function set combines two functions defined by Nilsson (MS and MT), an iterative function (DU), and two logical functions (NOT and EQ) needed to express predicates for controlling the iterative function.
- Function set = {MS, MT, DU, NOT, EQ}.
- Function MS: Move to the Stack.
  - It has one argument.
  - The S-expression (MS <X>) moves block <X> to the top of the STACK if <X> is on the TABLE.
  - It does nothing if <X> is already in the STACK, if the TABLE is empty, or if <X> itself is NIL.
  - Both MS and MT return NIL if they do nothing and T if they do something.
  - The real functionality of both MS and MT lies in their side effects on the STACK and TABLE, not in their return values.
- Function MT: Move to the Table.
  - It has one argument.
  - The S-expression (MT <X>) moves the top item of the STACK to the TABLE if the STACK contains <X> anywhere in the STACK.
  - It does nothing if <X> is already on the TABLE, if the STACK is empty, or if <X> is NIL.

– Function DU:
  * It has two arguments.
  * It appears in the form (DU <WORK> <PREDICATE>). It causes the <WORK> to be iteratively executed until the <PREDICATE> is satisfied (i.e., becomes non-NIL).
  * The iterative operator needed in this problem cannot be implemented directly as an ordinary LISP function. The reason is that, ordinarily, when LISP evaluates a function call, it first evaluates each of the arguments to the function and then passes the value to which the arguments have evaluated into the function.
  * If the iterative DU operator were implemented as an ordinary LISP function, the operator would merely evaluate the *value* returned by the <WORK>, as opposed to *doing* the <WORK> while iterating through the loop.
  * Thus, it is necessary to suppress premature evaluation of the <WORK> argument and the <PREDICATE> argument so that neither argument is evaluated outside the DU operator. Instead, these arguments must be evaluated dynamically inside the operator for each iterator.
  * In particular, the desired behavior is that the <WORK> first be evaluated inside the DU operator, and then the <PREDICATE> be evaluated. The desired two separate evaluations are to be performed, in sequence.
  * The execution of the <WORK> will often change some variable that will then be tested by <PREDICATE>. Indeed, that is often the purpose of a loop.
  * Thus, it is important to suppress premature evaluation of the NIL and <PREDICATE> arguments of the DU operator. The remedy is to implement the iterative DU operator with a macro definition in the manner in connection with conditional branching operators.
  * However, many S-expressions generated by genetic programming contain deep nestings of DU operators which are extremely time-consuming to execute. It is a practical necessity to place a limit on the number of iterations allowed by any one execution of a DU operator.
  * A similar limit must be placed on the total number of iterations allowed for all DU functions that may be encountered in the process of evaluating any one individual S-expression for any one fitness case.
  * Thus, the termination predicate of each DU operator is actually an implicit disjunction of the explicit predicate argument <PREDICATE> and two additional time-out predicates.
  * In particular, we decided that it is reasonable for this problem that DU operator times out if either more than 25 iterations are performed in evaluating a single DU operator, or more than 100 iterations are performed by DU operators in evaluating a particular individual S-expression for a particular fitness case.
  * If a DU operator times out, the side effects that have already been executed are not retracted.
  * Of course, if we could execute all the individual LISP S-expressions in parallel, so the infeasibility of one individual in the population does not bring the entire process to a halt, we would not need these limits.

    * The return value of the DU operator is a Boolean value that indicates whether the <PREDICATE> was successfully satisfied or whether the DU operator timed out.
    * The data type of the return value is the same as the type of the value of <WORK>. In particular, if the NIL evaluates to T (True) or NIL (False), this return value is a Boolean T or NIL.
    * On the other hand, if numeric-valued logics is being used and NIL evaluates to some positive number or some negative number, then this return value is either a numeric +1 (for T) or a numeric -1 (for NIL).
    * If the predicate of a DU operator is satisfied when the operator is first called, then the DU operator does no work at all and simply returns T.
  - In lieu of the millions of possible fitness cases, we constructed a structured sampling of fitness cases for measuring fitness consisting of the following 166 fitness cases:
    * the ten cases where the 0-9 blocks in the STACK are already in the correct order,
    * the eight cases where there is precisely one out-of-order block in the initial STACK on the top of whatever correctly ordered blocks happen to be in the initial STACK,
    * a structured random sampling of 148 additional cases with 0-8 correctly ordered blocks in the initial STACK and a random number (between 2 and 8) of out-of-order blocks on top of the correctly ordered blocks.
  - The raw fitness of a particular individual plan is the number of fitness cases for which the STACK contains nine blocks spelling "UNIVERSAL" after the plan is executed. Raw fitness ranges from 0 to 166.
  - Obviously, the construction of a sampling such as this must be done so that the process is not misled into producing solutions that correctly handle some unrepresentative subset of the entire problem but cannot correctly handle the entire problem.
- Example 1: Correctness
  - In generation 10, the best-of-generation individual achieved a perfect score (that is, the plan produced the desired final configuration of blocks in the STACK for all 166 fitness cases). This 100%-correct best-of-run plan is
    (EQ (DU (MT CS) (NOT CS)) (DU (MS NN) (NOT NN))).
    Also, see Figure 52
  - This best-of-run plan consists of two subplans, which are connected via the function EQ (an irrelevant operation that is merely serving as a connective).
  - The first subplan,
    (DU (MT CS) (NOT CS)),
    does the work of first moving CS (i.e., the top of the STACK) to the TABLE continuing until the predicate (NOT CS) becomes T (True). This predicate becomes true when the top of the STACK becomes NIL (i.e., the STACK becomes empty).
  - In other words, the first subplan involves completely disassembling the STACK onto the table.
  - The second subplan,
    (DU (MS NN) (NOT NN))),

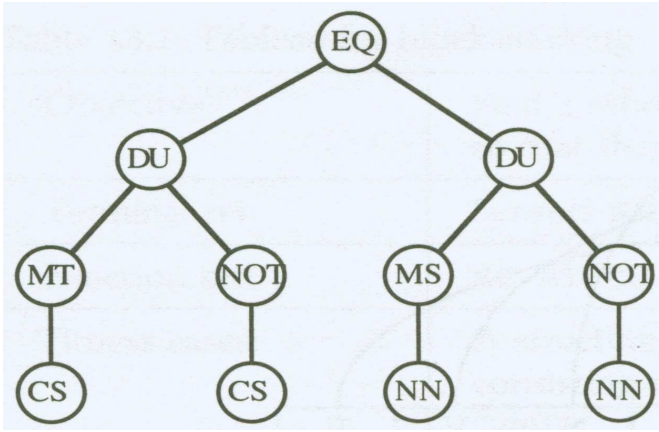

Figure 52: Correct best-of-run plan for example 1 of the block stacking problem.

does the work of iteratively moving the next need block `NN` to the `STACK` until there is no next needed block `NN`.

- Example 2: Correctness and Efficiency
  - The 100%-correct solution found in example 1 of the block stack problem is highly inefficient in that it removes all the blocks, one by one, from the `STACK` to the `TABLE` (even if they are already in the correct order on the `STACK`). This plan then moves the blocks, one by one, from the `TABLE` to the `STACK`. As a result, this plan uses 2319 block movements to handle the 166 cases.
  - The most efficient way to solve the block stacking problem while minimizing total block movements is to remove only the out-of-order blocks from the `STACK` and then to move the next needed blocks to the `STACK` from the `TABLE`. This approach uses only 1641 block movements to handle the 166 fitness cases.
  - In example 1, since nothing in the fitness measure gave any consideration whatsoever to efficiency as measured by the total number of block movements. The sole consideration was whether or not the plan correctly handled each of the 166 fitness cases (without timing out).
  - We can, however, simultaneously breed a population of plans for two attributes at one time. In particular, we can specifically breed a population of plans for both correctness and efficiency by using a combined fitness measure. Such a combined fitness measure might assign some of the weight to correctness and the remainder of the the weight to efficiency.
  - Somewhat arbitrarily, we assigned 75% and 25%, respectively, to these two factors.
    * First, consider correctness. If a plan correctly handled 100% (0%) of the 166 fitness cases, it would receive 100% (0%) of the 75 points associated with correctness in the combined fitness measure.
    * Second, consider efficiency. If a plan took 1641 block movements prior to termination (either natural termination or termination due to timing out), it would 100% of the 25 associated with efficiency in the combined fitness measure.
    * If a plan took between 0 and 1641 block movements to perform this work, the 25 points assigned to efficiency would be scaled linearly upward, so a plan

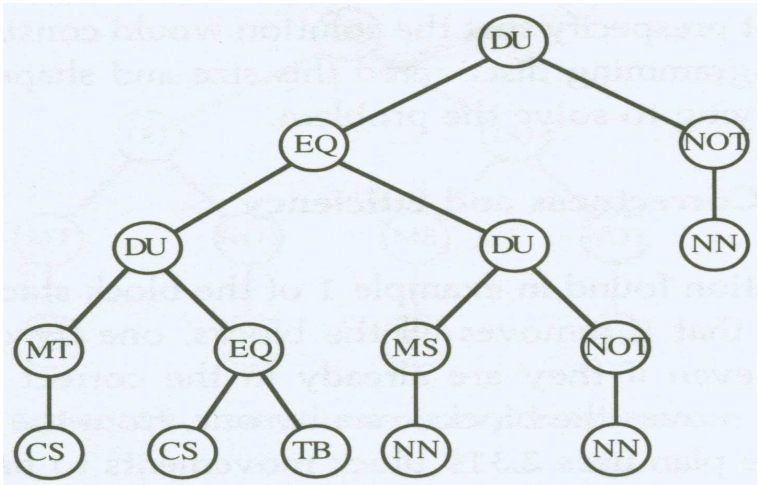

Figure 53: Efficient and correct plan for example 2 of the block stacking problem.

making no block movements would receive none of the 25 points.

* If the plan made between 1642 and 2319 block movements, the 25 points assigned to efficiency would be scaled linearly downward, so a plan making 2319 (or more) block movements would receive none of the 25 points.

– In the run described below, we applied this combined fitness measure starting at generation 0. However, it may be desirable to start a run using only the primary factor (i.e., correctness) in the fitness measure and then introduce the secondary factor into the fitness measure after some milestone has been achieved on the primary factor. (e.g, perhaps 90% correctness).

– In generation 11, the best-of-run plan, both 100% correct and 100% efficient, is found. It uses the minimum number (1641) of the block movements to correctly handle all 166 fitness cases. Figure 53 graphically depicts this best-of-run plan for example 2.

– It consists of 15 points and three iterative `DU` operators. The first subplan,
`(DU (MT CS) (EQ CS TB))`,
iteratively moves `CS` (the top block) of the `STACK` to the `TABLE` until the predicate `(EQ CS TB)` becomes satisfied. This predicate becomes satisfied when `CS` (the top of the stack) equals `TB` (the top correct block in the `STACK`).

– Figure 54 shows two out-of-order blocks (I and V) sitting on the top of the four correctly ordered blocks ("RSAL") on the `STACK`. The sensor `TB` is R. The two out-of-order blocks (I and V) are moved to the `TABLE` from the `STACK` until `R` becomes the top of the `STACK`. At that point, the top of the stack (`CS`) is R, which then equals `TB`.

– Then, the previously discovered second subplan iteratively moves the next needed blocks (`NN`) to the `STACK` until there is no longer any next block.

– The function `EQ` serves only as a connective between the two subplans. Notice also that the outermost `DU` function performs no function (but does no harm), since the predicate (`NOT NN`) is satisfied at the same time as the identical predicate of the second subplan.

• Example 3: Correctness, Efficiency, and Parsimony

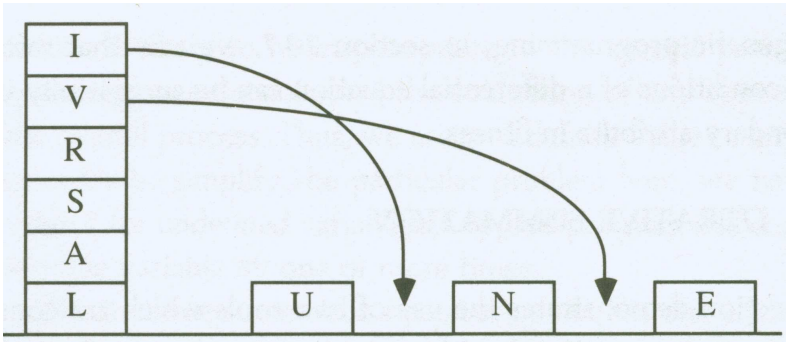

Figure 54: Subplan that moves two out-of-order blocks (I and V).

- The solution discovered in example 2 has 15 points in the tree corresponding to the S-expression. There could be as few as 12 points.
- By including parsimony in the fitness measure with 15% weight (in addition to correctness with 70% weight and efficiency 15% weight), we can breed for correctness, efficiency, and parsimony.
- In one run, we obtained the following 100%-correct and 100%-efficient S-expression consisting of the minimal number of points:
  `(EQ (DU (MT CS) (EQ CS TB)) (DU (MS NN) (NOT NN)))`.
- Parsimony and efficiency ar just two of many passible secondary attributes that can be considered in evaluating the fitness of the S-expressions generated with genetic programming.

2. Iterative Summation

- In this section, we will demonstrates the use of two tools which are commonly used in computer programming, namely
  - iteration and
  - assignment (i.e., setting).
- Assignment is an important tool in computer programming. It involves giving a name to the results of a calculation so as to provide a means for latter referring to the value of the calculation. Assignment can also be used to create an internal state within a computer program.
- Here, we introduce
  - `SIGMA` operator: it is based on the Σ notation for adding up an infinite series in mathematics.
  - `SET-SV` operator: it is a assignment operator which provides a means of assigning a value to a settable variable `SV`.
- Suppose the problem is to find the solution to $\frac{dy}{dx} - y = 0$, having an initial value $y_{\text{initial}} = 2.718$ for an initial value $x_{\text{initial}} = 1.0$.
- This problem is equivalent to computing e^x using the Taylor power-series expansion $\sum_{j=1}^{\infty} \frac{x^j}{j!}$.
- The assignment operator `SET-SV` has one argument and works in association with a settable global variable, `SV`. In particular, `SET-SE` sets `SV` to the value of its argument.

- The value to which the global variable SV is set is not affected by any subsequent event other than another SET-SV, including a change in the values of the terminals contained in the argument to the SET-SV operator.
- However, in order to avoid the undefined variables which cause the program to halt, in genetic programming, we assign a default value to any undefined variable. For simplicity, we made the default value 1 for undefined variables.
- The iterative summation operator SIGMA has one argument, called <WORK>. It evaluates its <WORK> argument repeatedly until a summand is encountered that is small (e.g., less than a fixed built-in threshold of 0.000001 in absolute value). The operator SIGMA then returns the value of its accumulated sum.
- The operator SIGMA is similar to DU in the following ways:
  - Time-out limits must be established for this operator.
  - An iteration variable J (starting with 1) is available inside this operator for possible incorporation into the <WORK> argument.
  - The arguments to this operator are evaluated only inside the SIGMA operation.
  - If the operator times out, any side effects of the <WORK> already performed remain.
- The index variable J is usually included in the terminal set for the problem so that when the argument <WORK> contains a reference to J, the SIGMA operator becomes a summation over the indexing variable. If SIGMA operators are nested, the indexing variable J takes on the value of the innermost SIGMA operator.
- There is no guarantee that the <WORK> of the SIGMA operator will ever become small. Therefore, it is a practical necessity, when working on a serial computer, to limit
  - the number of iterations allowed by any one execution of a SIGMA operator, and
  - the total number of iterations allowed for all SIGMA operators that may be evaluated in the process of executing any individual S-expression for any one fitness case.
- Here, these limits are 15 and 100 respectively. If either of these limits is exceeded, the SIGMA operator times out. When a SIGMA operator times out, it returns the sum accumulated up to that moment.
- Terminal set = {X, J, SV}.
  where X is the independent variable.
- Function set = {+, -, ×, %, SET-SV, SIGMA}.
- The raw fitness is defined in the same way as for differential equations problem. It is the sum, taken over the 200 fitness cases, of 75% of the absolute value assumed by the individual genetically produced function $f_j(x_i)$ at domain point x_i plus 25% of 200 times of the absolute value of the difference between $f_i(x_{\text{initial}})$ and the given value y_{initial}.
- In one run, the best-of-run individual S-expression was
  (SIGMA (SET-SV (× SV (% X J)))).
- This S-expression satisfies the differential equation and its initial conditions because it is computes the value of the power series for $e^x - 1$ for a given value x.
- It consists of a SIGMA operator that starts by setting the settable variable SV to the result of multiplying the value of SV (which is initially 1) by X and dividing by the iterative variable J associated with the SIGMA operator.

- As this iterative process continues, the summands successively consists of the consecutive powers of `X` divided by the factorials of the iteration variable `J`. When the current value of the settable variable `SV` becomes small, the `SIGMA` operator terminates and returns its accumulated value.
- The approach has the shortcoming of requiring that a default value be assigned to the settable variable in the event that it is referenced before it is defined. This default assignment makes the value of the settable variable depend in an especially erratic way on its position within the program in relation to the location of the set operator, if any, in the program.
- In the next topic, we will address the above shortcoming of cascading variables and describe a way of assigning a value to a variable so that it can later be referenced and used without the possibility of having an undefined variable.

3. Recursive Sequence Induction

- Consider the first 20 elements of the unknown sequence are as follows:
  S=1, 1, 2, 3, 5, 8, 13, 21, 34, 55,..., 4181, 6765.
  This sequence is the well-known Fibonacci sequence which can be computed using the recursive expression $s_j = s_{j-1} + s_{j-2}$, where s_0 and s_1 are both 1.
- If we expect to be able to induce a mathematical expression for the Fibonacci sequence, we must provide a facility in the function set to allow an S-expression being evaluated for a particular index j to refer to the values the S-expression itself has previously computed for indices less than j.
- We therefore define the sequence referencing function `SRF` to allow an S-expression to call on the value that it itself has previously computed.
- In particular, if an S-expression containing `SRF` function is being evaluated for index position j, the subexpression `(SRF K D)` returns the value that the S-expression itself previously computed for sequence position `K` provided `K` is between 0 and $j - 1$; otherwise it returns the default value `D`.
- Note that the `SRF` function returns the value computed by the current S-expression for any index `K` earlier than the current index j, not the actual correct value of the Fibonacci sequence for the index `K`.
- Note also that the `SRF` permits recursion where each new element in the sequence is defined recursively in terms of one or more previous elements of the sequence.
- Terminal set = {`J, R`},
  where the ephemeral random constant `R` ranges over the small integers 0, 1, 2, and 3.
- Function set = {`+, -,` $\times$`, SRF`}.
- The fitness cases for this problem consist of the first 20 elements of the actual Fibonacci sequence.
- The raw fitness of an S-expression is the sum, taken over 20 fitness cases, of the absolute value of the difference between the sequence value produced by the S-expression and the actual value of the Fibonacci sequence.
- The best-of-generation individual for generation 22 had a raw fitness of 0, i.e., a 100%-correct S-expression which scored 20 hits:
  `(+ (SRF (- J 2) 0) (SRF (+ (+ (- J 2) 0) (SRF (- J J) 0)) (SRF (SRF 3 1) 1))).`

Topic 10: Evolution of Constrained Syntactic Structures

1. Introduction

   - In genetic programming, unrestricted S-expressions are sufficient to solve numerous problems provided one selects a function set and a terminal set satisfying the closure property. Nonetheless, some problems call for constrained syntactic structure for the S-expressions.
   - Three new considerations arise when one is implementing genetic programming with S-expressions that have such constrained syntactic structure.
     - The initial population of random individuals must be created so that every individual computer program in the population has the required syntactic structure.
     - When crossover (or any other genetic operation that modifies an individual) is performed, the required syntactic structure must be preserved. The required syntactic structure can be preserved by means of structure-preserving crossover.
     - The fitness measure must take the syntactic structure into account.

2. Symbolic Multiple Regression

   - The previous examples of symbolic regression involved one or more independent variables, but only one dependent variable. The problem of symbolic *multiple regression* illustrates the need to create random individuals that comply with a special set of syntactic rules of construction and to then perform structure-preserving crossover on those individuals.
   - Consider a symbolic regression problem with two dependent variables y_1 and y_2 and four independent variables x_1, x_2, x_3, and x_4.
   - Suppose we are given a set of 50 fitness cases in the form of 6-tuples of the form (x_{1i}, x_{2i}, x_{3i}, x_{4i}, y_{1i}, y_{2i}). Each x_{1i}, x_{2i}, x_{3i}, and x_{4i}, (for i between 1 and 50) lies in some interval of interest.
   - Suppose further that the unknown relationships between these two dependent variables and the four independent variables are
     $y_{1i} = x_{1i}x_{3i} - x_{2i}x_{4i}$
     and
     $y_{2i} = x_{2i}x_{3i} + x_{1i}x_{4i}$
   - In other words, the unknown relationship is vector multiplications (i.e., complex multiplication).
   - In order to solve this problem by means of genetic programming, we need a computer program that returns two values (y_1 and y_2), instead of merely the single return value. That is, the computer program should return an ordered set (vector) of two numbers, rather than a single number.
   - The `LIST` function with two arguments (called `LIST2` herein) can be used to create an ordered set of two numbers.
   - Terminal set T = {`X1, X2, X3, X4`}.
   - Function set F = {`+, -,` $\times$`, %, LIST2`}.
   - The syntactic rules of construction required for the multiple regression problem where a vector of two values must be returned are these:

- The root of the tree (i.e., the function just inside the leftmost parenthesis of the LISP S-expression) is the `LIST2` function.
- The root is the only place where the `LIST2` function will appear in a given S-expression.
- Below the root, the S-expression is an unrestricted composition of the available functions (other than `LIST2`) from the function set F and the available terminals from the terminal set T.

- Three additional changes are also required when a problem involves syntactic rules of construction.
  - Initial population of random individuals must be created using these syntactic rules of construction.
  - The choice of points in the crossover operation must be constrained so as to preserve the structure required by the problem.
    - The simplest way to perform the restraining process is to exclude the root of tree from being selected as the crossover point of either parent in the crossover operation. However, this approach lacks generality when the rules of construction are more complex.
    - The general way to perform the restraining process for structure-preserving crossover is to allow any point to be selected as the crossover point for the *first* parent. However, once the crossover point has been selected for the first parent, the selection of the crossover point in the *second* parent is then restricted to a point of the same type as the point just chosen from the first parent.
  - The fitness measure must take into account the fact that a vector of values is returned by each S-expression. Raw fitness is equal to the sum, taken over all 50 fitness cases, of the absolute values of the differences between the value of the first dependent variable returned by S-expression and the target value of the first dependent variable plus a similar sum, taken over all 50 fitness cases, for the second dependent variable.
- In one run, the best LISP S-expression in generation 31 was `(LIST2 (- (× X3 X1) (× X4 X2)) (+ (× X3 X2) (× X1 X4)))`.
- Figure 55 graphically depicts this best-of-run S-expression from generation 31. The two arguments to the `LIST2` function are the two desired functional relationships.

3. Design of a Two-Bit Adder Circuit

- The problem of designing a circuit with multiple outputs illustrates the need to create random individuals that comply with a special set of syntactic rules of construction and to then perform structure-preserving crossover on those individuals.
- In this problem, we want to evolve a circuit composed of `AND`, `OR`, and `NOT` elements with four binary inputs and three binary outputs.
- The input to the circuit consists of two two-bit numbers. The output consists of three-bit binary sum of the two two-bit inputs.
- The terminal set for this problem consists of the four input signals. That is, Terminal set T = {`A1, A0, D1, D0`}.

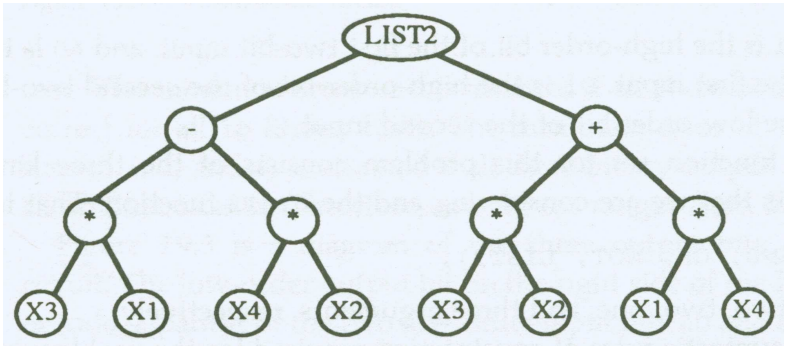

Figure 55: Best-of-run individual from generation 31 of the symbolic multiple regression problem for complex multiplication.

- Here `A1` (`A0`) is the high-order (low-order) bit of the first two-bit input. `D1` (`D0`) is the high-order (low-order) bit of the second two-bit input.
- The function set for this problem consists of the three kinds of circuit elements that we are considering and the `LIST3` function. That is,
  Function set F = {`AND, OR, NOT, LIST3`}.
- The syntactic rules of construction required for the problem of designing a circuit are similar to the previous example.
- The fitness cases for this problem consists of the 16 combinations of the values that may be assumed by the Boolean-valued terminals `A1`, `A0`, `D1`, and `D0`. The three binary output signals are weighted by consecutive powers of 2 (i.e., 1, 2, 4).
- Fitness is the sum, taken over 16 fitness cases, of the position-weighted differences (errors) between the three binary signals produced by the S-expression and the correct three binary signals for that fitness case.
- The minimum error of 0 is attained if all three binary output signals are correct for all 16 fitness cases. The maximum error of 112 (i.e., 16 times the sum of 1, 2, 4) is attained if all three binary output signal are wrong for all 16 fitness cases. Therefore, raw fitness ranges from 0 to 112.
- Figure 56 is a diagram of the three output bits of the two-bit adder circuit.
  - The low-order output bit on the right side of the figure is the exclusive-or (odd-2-parity) of the two low-order inputs `A0` and `D0`.
  - The middle output bit is the odd-3-parity of the two high-order input bits `A1` and `D1` and the output of the `AND` of the two low-order bits `A0` and `D0` (i.e., the carry bit from the low-order position).
  - The high-order output bit is the 3-Majority-On function of the two high-order input bits `A1` and `D1` and the output of the `AND` of the two low-order bits `A0` and `D0`.
- In one run, 100%-correct individual emerged as the best-of-run S-expression on generation 24.
- In an actual circuit-design problem, it might be appropriate to include the number of logical gates (a parsimony measure) in the fitness function for this problem, either from the beginning of the run or at a certain point later in the run.

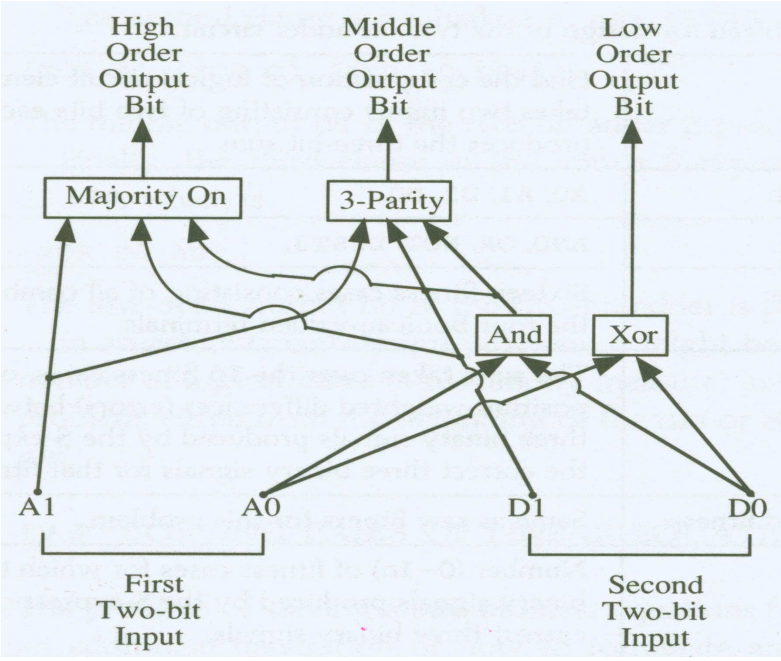

Figure 56: Two-bit adder circuit.

4. Solving Pairs of Linear Equations

- The problem of solving a pair of linear equations for *both* unknowns provides an additional illustration of how to formulate and solve a problem whose solution is a vector of values.
- The problem here is to find a vector-valued S-expression for solving a pair of consistent non-indeterminate linear equations, namely
  $a_{11}x_1 + a_{12}x_2 = b_1$
  and
  $a_{21}x_1 + a_{22}x_2 = b_2$,
  for both of its two unknown variables (x_1 and x_2).
- In other words, we are seeking a computer program that takes a_{11}, a_{12}, a_{21}, a_{21}, a_{22}, b_1, and b_2 as its inputs and produces x_1 and x_2 as its output. We assume that the coefficients of the equations were prenormalized so the determinant is 1.
- Terminal set T = {`A11, A12, A21, A22, B1, B2`}.
- Function set F = {`+, -,` $\times$`, %, LIST2`}.
- The syntactic rules of construction and the resulting two types of points for this problem are the same as for the multiple regression problem.
- The fitness cases are ten randomly created pairs of linear equations whose determinant is one. Each pair of equations is created by generating six random numbers between -10.000 and +10.000 and assigning the six random numbers to a_{11}, a_{12}, a_{21}, a_{21}, a_{22}, b_1, and b_2 for that pair of equations.
- If the determinant is nonzero, the values of a_{11}, a_{12}, a_{21}, a_{21}, a_{22}, b_1, and b_2 are divided by the square root of the determinant for that pair in order to yield a pair of equations with a determinant of 1.

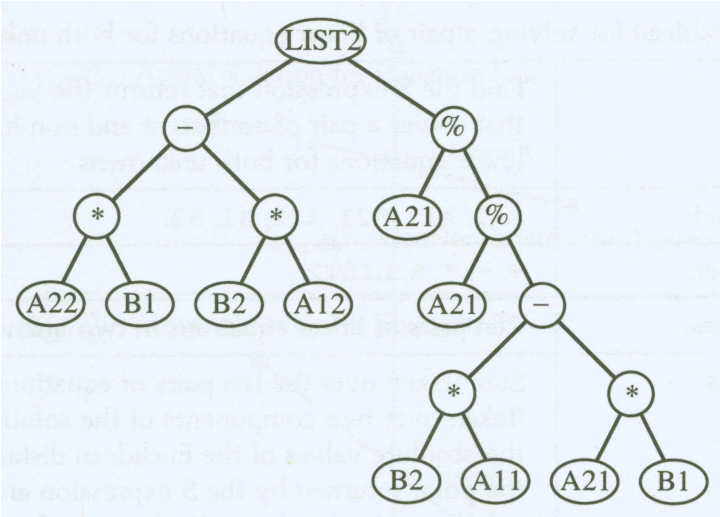

Figure 57: Best-of-run individual for the problem for solving a pair of linear equations for both unknowns.

- Raw fitness is measured by the erroneousness of the S-expression. The fitness measure is the sum, taken over all ten fitness cases, of the Euclidean distance in the plane between the genetically produced solution point (x^g_{1i}, x^g_{2i}) for equation pair i and the actual solution point (x^s_{1i}, x^s_{2i}) for equation pair i.
- In one run, the following S-expression emerged as the 100%-correct solution to the problem:
  `(LIST2 (- (× A22 B1) (× B2 A12)) (% A21 (% A21 (- (× B2 A11) (× A21 B1))))).`
- Figure 57 graphically depicts this 100%-correct best-of-run S-expression.

5. Finding a Global Optimum Point

- The conventional genetic algorithm operating on strings is frequently used to solve function-optimization problems requiring the finding of a global optimization point for a highly nonlinear multi-dimensional function.
- In such problems, the goal is to find a point in the multi-dimensional space over the various independent variables of the problem for which the function attains a minimal (or maximal) value.
- When conventional genetic algorithm operating on strings is applied to such a problem, each chromosome string typically represents a combination of values of the various independent variable.
- The result produced by the genetic algorithms is a single point from the search space. It is often the point where the function attains its global optimum value.
- Ordinarily, when we apply genetic programming to an optimization problem, the result typically comes in the form of an entire function, not just a single point. However, we can use genetic programming to find a single optimum point, rather than an entire function, by simply excluding all independent variables from the terminal set.
- Consider the problem of finding the global optimum (i.e., minimum) point for the five-dimensional function:
  $F_1 = (x_1, x_2, x_3, x_4, x_5) = (x_1 - \sqrt{1})^2 + (x_2 - \sqrt{2})^2 + (x_3 - \sqrt{3})^2 + (x_4 - \sqrt{4})^2 + (x_5 - \sqrt{5})^2$

- The desired global optimum value is 0, and it occurs at the point ($\sqrt{1}$,$\sqrt{2}$, $\sqrt{3}$,$\sqrt{4}$,$\sqrt{5}$) in the five-dimensional space.
- Terminal set T = {`R`}.
- Function set F = {`+, -, ×, %, LIST5`}.
- Raw fitness is the value of the function F_1 for the point $(x_1, x_2, x_3, x_4, x_5)$ in the five-dimensional space.
- In generation 45 of one run, the best-of-run S-expression containing 92 points scored 5 hits out of 5 and had a fitness of 0.00186. This S-expression is equivalent to `(LIST5 1.0007 1.414 1.732 2.0005 2.2364)`.

6. Solving Quadratic Equations

- The problem here involves complex numbers, and we could have proceed in the same way. Instead, we will use this problem to explicitly illustrate how we might *use some knowledge about the nature of the solution of the problem to decompose the problem* and create the syntactic rules of construction.
- The quadratic equation, $ax^2 + bx + c = 0$, can be solved for its two roots (which are often imaginary or complex numbers) by the familiar formula: $x = (1/2a)(-b \pm \sqrt{b^2 - 4ac})$.
- Terminal set T = {`A, B, C`}.
- Function set F = {`+, -, ×, %, SQRT, LIST2`}.
- `SQRT` is the square root function and returns either a floating-point value or a complex value, as appropriate. For example, $\sqrt{-4}$=`#C(0.0 2.0)`, where `#C(0.0 2.0)` represents $2i = 2\sqrt{-1}$.
- Each S-expression is a list of two components and represents the pair of complex numbers. The two complex numbers belonging to the pair are produced from the two components of a single S-expression by a wrapper (an output interface).
- The wrapper first takes the first component of the S-expression and *adds* the second component to produce the first number. Then the wrapper takes the first component of the S-expression and *subtracts* the second component to produce the second number.
- In other words, the wrapper performs the role of the ± sign and produces the two complex numbers of the desired complex pair from the S-expression.
- The random fitness cases for this problem consist of ten randomly generated quadratic equations. The coefficients `A`, `B`, and `C` being random numbers between -10 and +10 (and with the coefficient `A`$\neq$0 to avoid degeneracy).
- We envision the first and second components of a perfect solution to be $\frac{-b}{2a}$ and $\frac{\sqrt{b^2-4ac}}{2a}$ respectively.
- The fitness of a given S-expression is the sum, taken over the ten fitness cases, of the sum of two Euclidean distances representing errors.
- The Euclidean distance between two points in the complex plane is the square root of the sum of the squares of the difference between the real parts of the two points and the imaginary parts of the two points.
- In one run, the best-of-run S-expression emerged on generation 30 had a raw fitness of about 10^{-8} and scored 10 hits out of 10.

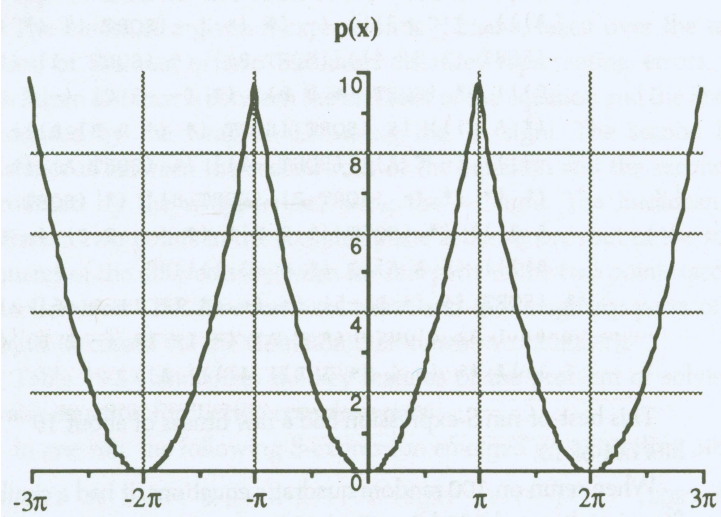

Figure 58: The function x^2 repeated over three intervals of size 2π.

- The first component of the S-expression is equivalent to
  `(% (- 0 B) (× 2 A))`.
- The second component is equivalent to
  `(% (SQRT (- (× (× 0.25 B) B) (× A C))) A)`.

7. Finding a Fourier Series

   - In this section, we find the Fourier series that represents a given periodic function.
   - Figure 58 shows the function x^2 over the interval [-π, +π] with the curve from that interval then repeated over one subsequent and one preceding 2π interval. The function is an even function in the sense that $f(x) = f(-x)$.
   - Finding a Fourier series, $a_0 + \sum_{j=1}^{\infty} a_j \cos\theta + b_j \sin\theta$, that represents a given function, such as x^2, requires finding both the Fourier coefficients and the appropriate number of terms to include in the series.
   - In other words, the size of the S-expression for the x^2 function is not known in advance and must be determined dynamically.
   - The first few terms of correct Fourier series for x^2 are the following:

$$x^2 = \frac{\pi^2}{3} - 4\cos x + \frac{4\cos 2x}{2^2} - \frac{4\cos 3x}{3^2} + \frac{4\cos 4x}{4^2} - .....$$

   - In this problem, we want to genetically breed individuals with an indefinite number of additive terms, each consisting of either the sine or the cosine of an integral multiple of the argument x.
   - We do not want to specify the number of such terms in advance, instead, we want the number of such terms to emerge as a result of the evolutionary process.
   - Defining function operators:
     - `&`:
       It is merely the ordinary arithmetic addition function with two arguments. However, for purposes of defining the syntactic structure required for this problem,

we give ordinary addition this special name in order to distinguish it from other uses of this same function in this problem.

- `XSIN`:
  - It is a two-argument function that can be defined in LISP as follows: `(defun XSIN (arg1 arg2) (× arg1 (sin (round (× arg2 X)))))`.
  - The first argument to `XSIN` is the coefficient (e.g., b_3), and the second argument is the harmonic. The second argument is rounded off to the nearest integer by the function in order to produce the desired harmonics at the fundamental frequencies.
  - Note that the variable `X` is not an explicit argument to this function; it is, instead, a global variable that acquires its value outside the function `XSIN` (i.e., in the loop that iterates through the fitness cases).
- `XCOS`:
  It is defined in a similar manner.

- The syntactic rules of construction required for the indefinitely size S-expression required for this problem are the following:
  - The root of the tree (i.e., the function just inside the leftmost parenthesis of the LISP S-expression) is the function `&` with two arguments.
  - The only thing allowed below an `&` function is either another `&` function or an `XSIN` or `XCOS` function.
  - The only thing allowed below an `XSIN` or `XCOS` function is either a floating-point random constant or an arithmetic function (`+, -, ×, %`).
- Therefore, there are three types of points:
  - a point containing an `&` function,
  - a point containing an `XSIN` or `XCOS` function, and
  - a point containing an arithmetic function or a random constant.
- The fitness cases for this problem consist of 200 pairs of (x_i,y_i) points, where x_i ranges randomly over the interval [-3π, +3π].
- The raw fitness is the sum, taken over the 200 fitness cases, of the square of the differences between the value returned by the S-expression and the actual value y_i of the given periodic function.
- Since the variable `X` is implicit, the terminal set for this problem consists only of ephemeral random float-point constant atom `R` which ranges from -10.000 to +10.000, namely
  Terminal set T = {`R`}.
- Function set F = {`&, XSIN, XCOS, +, -, ×, %`}.
- The correct Fourier series is

$$\begin{aligned} x^2 \;=\; +3.2899 \;&-\; 4.0000\cos x \;+\; 1.0000\cos 2x \\ &-\; 0.4444\cos 3x \;+\; 0.2500\cos 4x \\ &-\; 0.1600\cos 5x \;+\; 0.1111\cos 4x \\ &-\; 0.0816\cos 7x \;+\; \ldots\ldots \end{aligned}$$

- In generation 50, the best-of-generation individual has a raw fitness of 2.13 and scored 190 hits out of 200. This individual has 153 points and can be simplified to

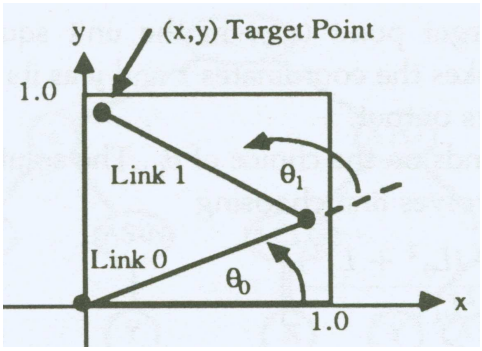

Figure 59: Robot arm consisting of link 0 at angle θ_0 from the x axis and link 1 at angle θ_1 from link 0 so that the endpoint of link 1 reaches the target point (x,y).

$$\begin{aligned} x^2 \;=\; +3.3200 \;&-\; 4.0120\cos x \;+\; 1.0109\cos 2x \\ &-\; 0.4402\cos 3x \;+\; 0.2418\cos 4x \\ &-\; 0.1564\cos 5x \;+\; 0.1409\cos 4x \\ &-\; 0.0743\cos 7x \end{aligned}$$

8. Inverse Kinematics

- In this section, the problem is to find a computer program to control the movement of a two-link robot arm from a resting position to a desired target location. That is, we are seeking the inverse kinematics for the robot arm.
- The solution to this problem consists of a vector of values, namely the two angles by which the two links of the robot arm should be moved. The second angle in the vector will explicitly depend on the choice of the first angle.
- In previous topic, we discussed the problem of iterative summation. We used the set operator `SET-SV` to assign the current value of a specified sub-S-expression to a settable variable `SV`, thus making that value available whenever the terminal `SV` subsequently appeared within the program.
- This approach has the shortcoming of requiring that some default value be assigned to the settable variable in the event that it is referenced before it is defined.
- In this section, we overcome this disadvantage by employing a cascade in which the result of an earlier step is used in a subsequent step.
- Figure 59 shows a robot arm with two links of length $L_0 = L_1 = 1$. The first angle, θ_0, is the angle by which link 0 (the first link) is to be rotated about the base of the robot arm (which is grounded at the origin). After link 0 has been rotated, the second angle, θ_1, is the angle by which link 1 is to be rotated about the endpoint of link 0.
- The forward kinematic equations specify the point (x,y) reached as a function of angles θ_0 and θ_1. In particular, these equations specify the point (x,y) reached by the endpoint of link 1 if link 0 is rotated about the base of the robot arm by angle θ_0 and if link 1 is rotated about the endpoint of link 0 by angle θ_1.
- The forward kinematic equations are
  $x = L_0\cos\theta_0 + L_1\cos(\theta_0+\theta_1)$
  and
  $y = L_0\sin\theta_0 + L_1\sin(\theta_0+\theta_1)$

- The goal is to find a computer which specifies how to rotate the two links so that the endpoint of the second link of the robot arm reaches a given target (x,y) in the unit square.
- The computer program being sough takes the coordinates x and y as its input and produces the angles θ_0 and θ_1 as its output.
- The choice of θ_1 depends on the choice of θ_0. The solution to this problem of inverse kinematics involves first choosing

$$\theta_0 = \texttt{ACOS}\left[\frac{(x^2+y^2)-(L_0^2+L_1^2)}{2L_0^2L_1^2}\right]$$

  and then choosing

$$\theta_1 = \texttt{ATG}(y,x) - \texttt{ATG}(L_1\sin\theta_0, L_0 + L_1\cos\theta_0)$$

  where `ATG` is the two-argument Arctangent function.
- This calculation involves a cascade in which the choice of one variable depends on a previous choice of another variable.
- In this problem, the result is an ordered set (i.e., vector) of two angles. The terminal set is different for the two components of the solution. When the first angle θ_0 is being chosen, the calculation depends on the coordinates x and y of the target points.
- That is, the terminal set for component 0 of the solution vector is
  Terminal set T_0 = {`X, Y, R`},
  where `R` is the ephemeral floating-point random constant between -1.000 and +1.000.
- The second angle depends directly on the choice already made for the first angle (as well as the coordinates x and y of the target point). Thus, when θ_1 is being chosen, the terminal set for component 1 of the solution vector is
  Terminal set T_1 = {`ANGLE-0, X, Y, R`},
  where `ANGLE-0` is θ_0.
- Thus, there are three types of points in the S-expressions for this problem:
  - the root,
  - points in the first component, and
  - points in the second component.
- Function set F = {`LIST2, +, -, ×, %, exp, ASIN, ACOS, ATG`}
- The rules of construction for individual S-expressions in the problem of inverse kinematics are these:
  - The root of the tree must be the `LIST2` function with two arguments.
  - The root is the only place where the `LIST2` function will appear in a given S-expression.
  - The first component of the vector created by the `LIST2` function is a composition of the arithmetic functions from function set F (other than `LIST2`) and terminals from the terminal set T_0.
  - The second component of the vector created by the `LIST2` function is a composition of the arithmetic functions from function set F (other than `LIST2`) and terminals from the terminal set T_1.

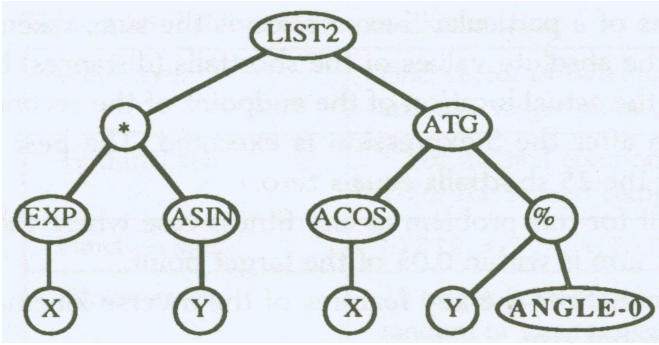

Figure 60: Random individual from generation 0 for the inverse kinematics problem.

- Figure 60 shows a possible initial random individual for this problem. Its first component is executed first. The result of its evaluation is then available, by the name `ANGEL-0`, when the second component of the vector is evaluated.
- Thus, we acquire the ability to give a name to an intermediate value (i.e., the first component) and to refer to that intermediate value by its name in the remainder of the computer program.
- The value of cascading variable is based on the value of the terminals in the first (leftmost) main branch of the program tree at the time of execution of that branch and does not reflect any subsequent events, such as changes in the value of those terminals.
- The fitness cases for this problem consists of 25 target points chosen inside the unit square. The 25 target points are the points of a regular 5×5 grid within the unit square.
- The raw fitness of a particular S-expression is the sum, taken over the 25 fitness cases, of the absolute values of the short falls (distances) between each target point and the actual location of the endpoint of the second link of the robot arm after the S-expression is executed.
- In one run, the best-of-generation individual from generation 154 scored 25 hits and contained 231 points. The average shortfall between the endpoint of the second link of the robot arm and the target point was 0.024.
- The cascading variable is guaranteed to be defined before it is referenced, because the main branches of the program tree are executed in a well-defined sequential order (i.e., left-to-right).
- More than one cascading variable could be defined if additional branches were added to the program tree. Moreover, cascading variables are not inherently limited to the final (rightmost) branch of the program tree. If desired, a cascading variable can appear in any branch of an S-expression that is evaluated after the branch in which it is defined. That is, a cascading variable can be defined in terms of one or more cascading variables.

9. Local Tracking of a Dynamical System

- One aspect of the study of dynamical systems and deterministic chaos involves finding the function that fits a given sample of data. Often the functions of interest are

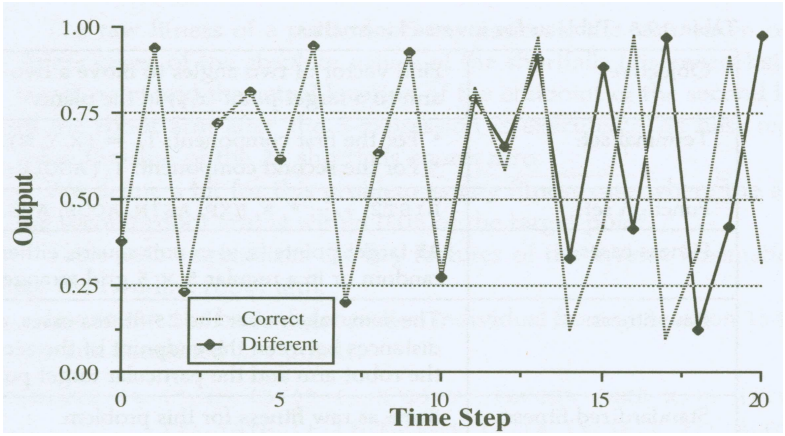

Figure 61: Times series for two logistic functions with slightly different initial conditions.

recursive in the sense that the sequence of values of the function over time depends on previous values of the function. Some dynamical systems exhibits deterministic chaos.

- This problem is similar to the inverse kinematics problem in that the solution consists of a vector of values and there is a cascade in which the first component of the vector is used in computing the second component of the vector.
- The nonlinear *logistic equation*, $x(t+1) = rx(t)[1-x(t)]$, where $0 < r \leq 4$ and $0 \leq x(t) \leq 1$, is one of the simplest models of a system displaying deterministic chaos.
- The sequence of values of $x(t)$ for the logistic equation in highly dependent on the initial value of x at time 0. A very small change in the initial value $x(0)$ can, after surprising few time steps, produce major differences in the value of this time series.
- After a large number of time steps, all transient behavior disappears for the logistic equation. All sequences, regardless of the initial condition, eventually settle down to 0.0 (i.e., the point 0.0 is a global attractor for the logistic equation) for a given r.
- Figure 61 shows two graphs of the logistic function where $r = 4$. For the first graph, the initial condition is $x(0) = 3/8$ (0.37500). For the second graph, the initial condition $x(0)$ is $3/8 + 2^{-16}$ (≈ 0.375015). Even though the two initial conditions differ by only 10^{-5}, the sequence of values of $x(t)$ is very different after only 15 time steps.
- Figure 62 shows the residual error between the two logistic functions.
- To perform this symbolic regression, we would include in the terminal set the value `PV` of the sequence at the previous time step.
- When the initial condition is not known, the problem of symbolic regression for a chaotic sequence is more difficult. In this event, the regression must find *both* the initial condition value $x(0)$ and the recursive equation for taking the value of the sequence for time step t and producing the value of the sequence for time step $t+1$.
- If a chaotic sequence is involved, we know that we cannot perform this symbolic regression over long sequences of time.
- We need to define an S-expression with a syntactic structure. In particular, each S-expression for this problem will be a vector consisting of two components.

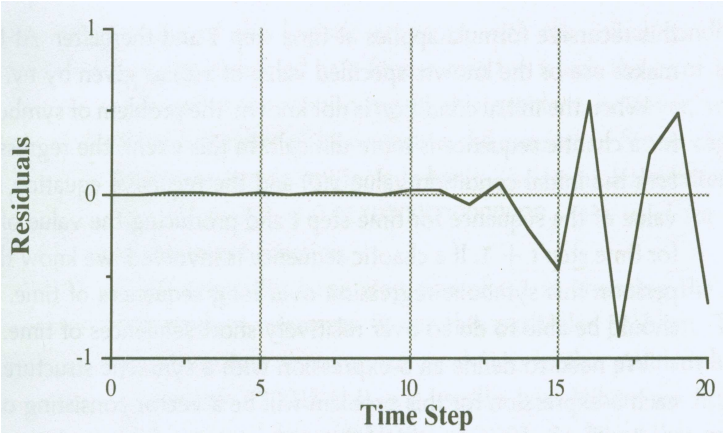

Figure 62: Difference, over time, between the two logistic functions with slightly different initial conditions.

  - The first component of the vector is a purely numerical component (not containing the terminal `PV`) expressing the initial condition value $x(0)$ for the sequence in terms of various arithmetic functions and various numerical constants.
  - The second component (which, in general, contains the terminal `PV`) is the equation that expresses the value of the sequence at time step t in terms of the value of the sequence at the previous time step as well as various arithmetic functions and various numerical constants.
- Function set F = {`+`, `-`, $\times$, `%`, `LIST2`}.
- The rules of construction for individual S-expressions in the problem of locally tracking a chaotic dynamical system are the following:
  - The root of the tree must be the `LIST2` function with two arguments.
  - The root is the only place where the `LIST2` function will appear in a given S-expression.
  - The first component of the vector created by `LIST2` is a composition of the functions from function set F (other than `LIST2`) and the ephemeral random floating-point constant atom `R` ranging between 0.0 and 1.0 in the terminal set T_0, namely
    T_0 = {`R`}.
  - The second component of the vector created by `LIST2` is a composition of the functions from function set F (other than `LIST2`) and terminals from the terminal set T_1, namely
    T_1 = {`PV`, `R`}.
- There are three types of points in the S-expressions for this problem:
  - the root,
  - points in the first component, and
  - points in the second component.
- A fitness case for this problem is one of the ten integral values of time between 0 and 9. Fitness is the sum, taken over the ten values of time, of the differences between the sequence value produced by the S-expression and the actual value of the logistic sequence with the initial condition value $x(0) = 3/8 (= 0.37500)$

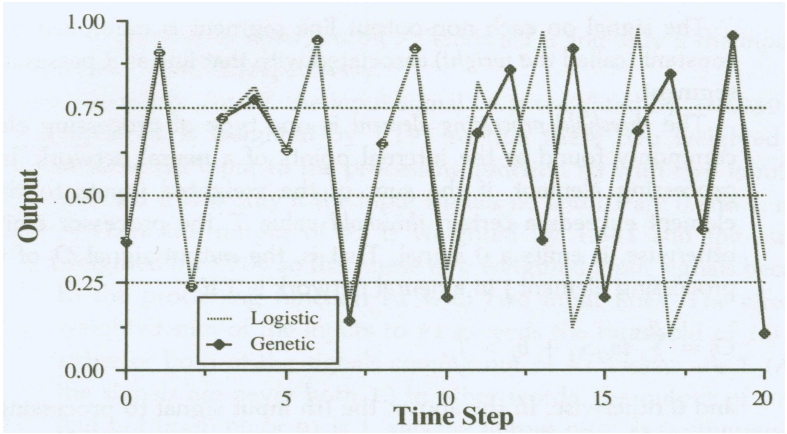

Figure 63: Time series for the correct logistic function and the best-of-run individual.

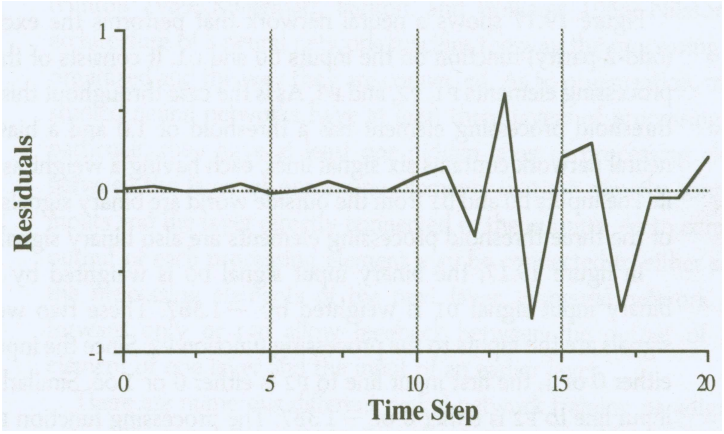

Figure 64: Difference, over time, between the correct logistic function and the best-of-run individual.

- In one run, the best-of-generation S-expression for generation 44 had 103 points and a raw fitness of 0.167. The first component of this S-expression is equal to 0.36288.
- Figure 63 is a graph of the values of the times series produced by this S-expression overlaid on to a graph of the values of the logistic function. Figure 64 shows the difference between the two graphs contained in Figure 63.
- Note that we did not obtain the exact value of the initial condition, nor did we obtain the exact functional relationship.

10. Designing a Neural Network

- Genetic programming approach I
  - In this section, we further illustrate S-expressions with a syntactic structure and structure-preserving crossover with a problem having an even more complex set of rules of construction. In particular, we show how to simultaneously discover both the weights and design the architecture of a neural network.
  - Figure 65 shows a neural network that performs the exclusive-or `XOR` (odd-2-parity) function on the inputs `D0` and `D1`. It consists of three threshold processing

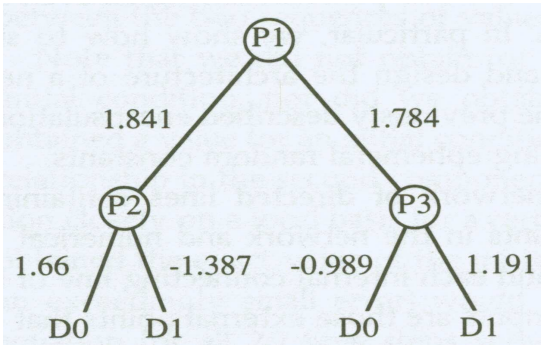

Figure 65: Neural network for exclusive-or (odd-2-parity) function.

elements `P1`, `P2`, and `P3`. (As is the case throughout this section, each threshold processing element has a threshold of 1.0 and a bias of 0.0).

- In Figure 65, the binary signal `D0` is weighted by 1.66 and the binary input signal `D1` is weighted by -1.387. These two weighted input signals are the inputs to the processing function `P2`.
- The processing function `P2` adds up its two weighted input lines and emits a 1 if the sum exceeds the threshold of 1.0 and emits a 0 otherwise.
- The effect is that the weighted sum of the inputs to `P1` exceeds the threshold of 1.0 if and only if either or both of the signals coming out of `P2` and `P3` are 1. In other words, the output of `P1` is 1 if either (but not both) `D0` or `D1` is 1, and the output of `P1` is 0 otherwise.
- Neural network training paradigms usually presuppose that the architecture of the neural network has already been determined. That is, they presuppose that selections have been made for the number of layers of processing elements, the number of processing elements in each layer, and the connectivity between the processing elements.
- Of course, the fact that a neural net is involved means that the functionality of the weighting function and the functionality of the processing element are also inherently selected.
- Terminal set T = {`D0, D1, R`},
  where `R` is the ephemeral random floating-point constant ranging between -2.000 and +2.000.
- Function set F = {`P2, P3, P4, W, +, -,` $\times$`, %`},
  taking two, three, four, two, two, two, two arguments respectively.
  - The function `P` is the processing function and appears in the function set with a varying number of arguments. We refer to these functions as `P2`, `P3`, and `P4`.
  - The function `W` is the weighting function used to give a weight to a signal going into a processing function. `W` is merely the multiplication function with two arguments.
    - The first (left) argument of the `W` function is always a numeric weight.
    - The second (right) argument is always a binary signal (coming from the outside world as an input to the neural network or coming out of a processing element within the network).

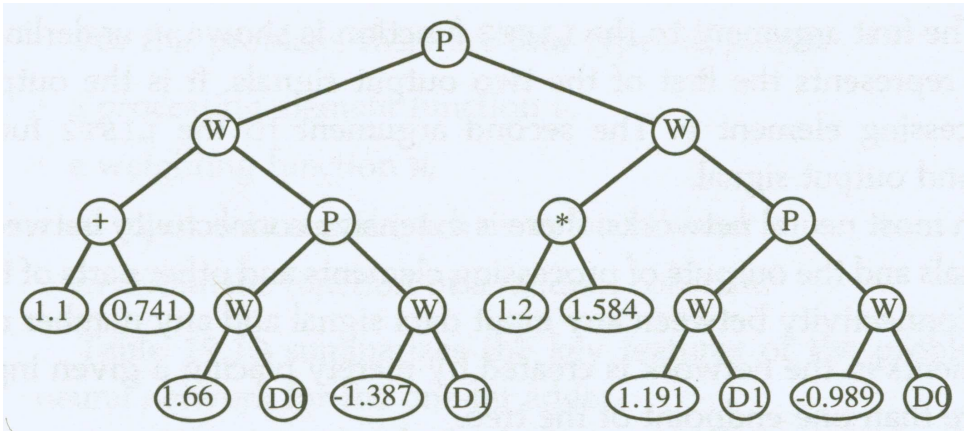

Figure 66: Tree for the S-expression that performs the exclusive-or (odd-2-parity) task.

    - The four basic arithmetic functions in the function set are used to create and modify the numeric constants (weights) of the neural network.
- Not all possible compositions of functions from the function set and terminals from the terminal set correspond to what we would reasonably call a neural network. Thus, the problem of designing neural networks requires rules of construction that specify what structures are allowable for this particular problem.
- There are four types of points in the S-expressions for this problem:
    - points with a processing function `P`,
    - points with a weighting function `W`,
    - points with input data signals (such as `D0` and `D1`), and
    - points with arithmetic functions or floating-point random constants.
- The rules of constructions for a neural network with one output signal are as follows:
    - The root of the tree must be a processing function `P`.
    - The only thing allowed at the level immediately below any processing function `P` is a weighting function `W`.
    - The only thing allowed below a weighting function `W` on the left is either a floating-point random constant or an arithmetic function.
    - The only thing allowed below a weighting function `W` on the right is either an input data signal (such as `D0` and `D1`) or the output of a `P` function.
    - The only thing allowed below an arithmetic function is either a floating-point random constant or an arithmetic function.
- These rules are applied recursively.
- Note that the external points of the tree are either input signals or floating-point random constant.
- Figure 66 graphically depicts the S-expression which represents the same neural network as shown in Figure 65.
- If there is more than one output signal from a neural network, the output signals are returned as a list. For example, if there are two output signals, they are returned via a `LIST2` function. That is, the function set F is enlarged to F = {`LIST2, P, W, +, -,` $\times$`, %`}.

  - The number of arguments to the function `LIST2` equals the number of output signals. That is the only time the `LIST2` function is used in a given S-expression. The rules of construction also require that the function at the level of the tree immediately below the `LIST2` function be a processing function `P`. Thereafter, the previously described rules of construction apply.

- Genetic programming approach II: encapsulation
  - In most neural networks, there is extensive connectivity between input data signals and the outputs of processing elements and other parts of the network.
  - Connectivity between any input data signal and any number of processing elements in the network is created by merely placing a given input signal on more than one endpoint of the tree.
  - It is also necessary to have connectivity between the output from a processing element function `P` in the network which feeds into more than one other processing element function. This connectivity cannot be obtained from ordinary LISP S-expression by means of any technique used so far.
  - The encapsulation operation provides a means for obtaining the desired connectivity. This operation identifies potentially useful subtree and gives them a name so that they can be referenced later in more than one place (thus connecting an output to more than one place).
  - We now show how to simultaneously do both the architectural design and the training of the neural network to perform the task of adding two one-bit inputs to produce two output bits.
  - The fitness cases for this task consist of the four cases representing the four combinations of binary input signals (i.e., `00, 01, 10,` and `11`) that could appear on `D1` and `D0`. The correct two output signals `01` and `02` are then associated with each of these four fitness cases.
  - The raw fitness measure is the sum of the binary differences, taken over the four fitness cases, between the first (lower-order) output signal from the neural network and the correct low-order bit from the one-bit adder function plus the sum of *twice* the binary differences, taken over the four fitness cases, between the second (high-order) output signal from the neural network and the correct high-order bit from the one-bit adder function.
  - That is, the errors are weighted by consecutive power of 2 (i.e., 1 and 2) according to the binary position. The minimum error is 0, and the maximum error is 12.
  - For this problem, there are four types of points:
    * a processing element function `P`,
    * a weighting function `W`,
    * an input data signal (such as `D0` or `D1`), and
    * an arithmetic function or a random constant.
  - Terminal set = {`D0, D1, R`}, where `R` $\in$ [-2.000, 2.000].
  - Function set F = {`LIST2, P, W, +, -,` $\times$`, %`}.
  - In one run, an individual, which was 100% correct in performing the one-bit adder task, emerged on generation 31. This individual was complex. The simplified form is shown in Figure 67.
  - The constant **5.25** and two calls on the encapsulated functions `E103` and `E111` are graphically depicted in Figure 68, 69, and 70 respectively.

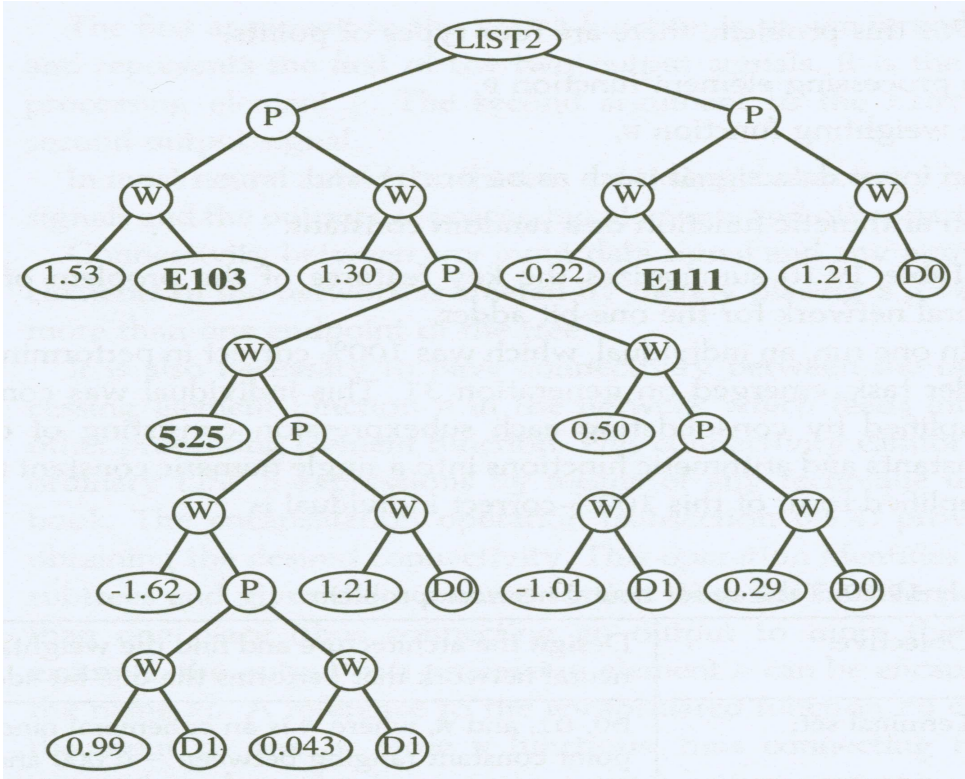

Figure 67: Genetically created neural network for the one-bit adder task.

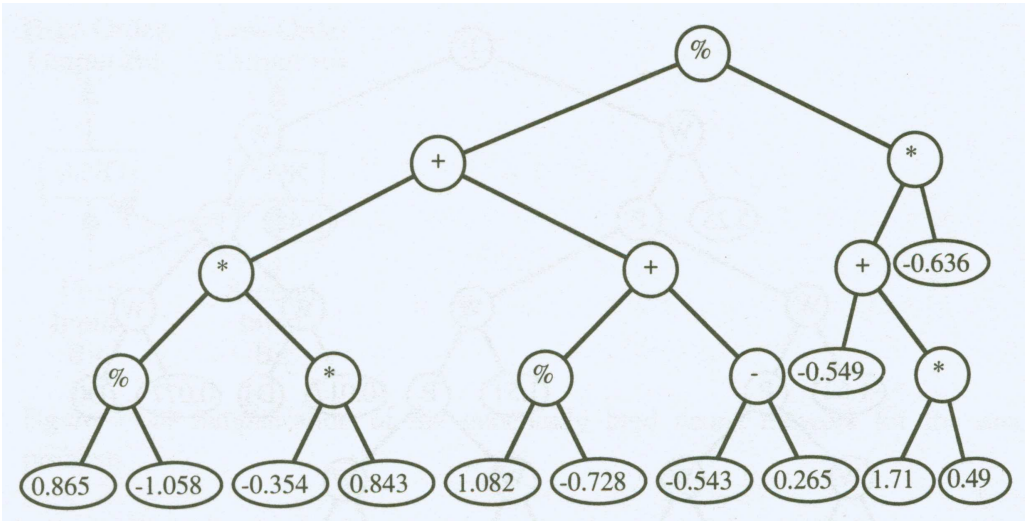

Figure 68: Subtree representing the genetically bred constant 5.25.

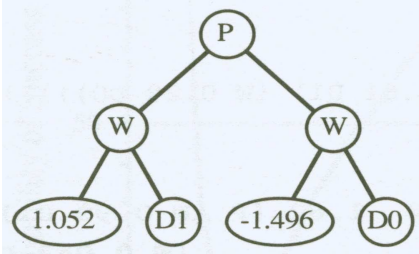

Figure 69: Encapsulated function `E103`.

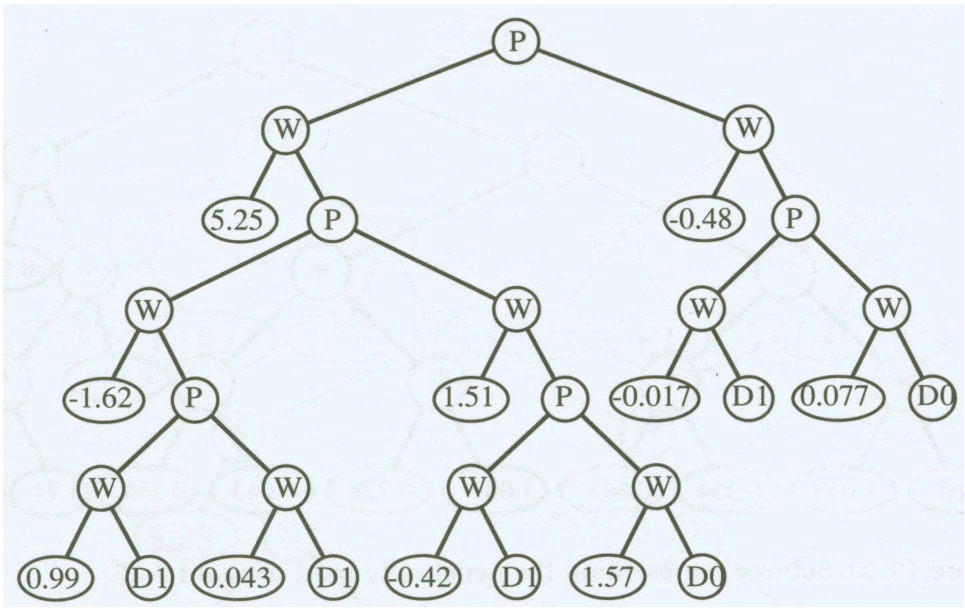

Figure 70: Encapsulated function `E111`.

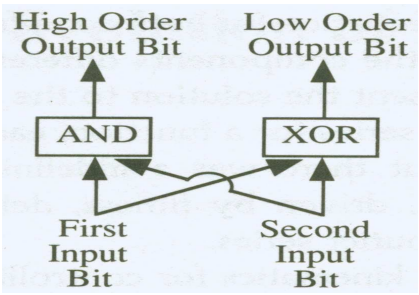

Figure 71: Simplification of the genetically bred neural network for the one-bit adder problem.

- The interpretation of the 100%-correct individual is as follows:
  - The first element of the `LIST2` is the low-order bit of the result. Upon examination, this first element is equivalent to
    `(OR (AND D1 (NOT D0)) (AND D0 (NOT D1)))`,
    which is equivalent to `(XOR D0 1)`, namely the odd-2-parity function of the two input bits `D0` and `D1`. This is the correct expression for the low-order bit of the result.
  - The second element of the `LIST2` is the high-order bit of the result. Upon examination, this second element is equivalent to
    `(AND D0 (NOT (OR (AND D0 (NOT D1)) (NOT D1)))`, which is equivalent to `(AND D0 D1)`. This is the correct expression for the high-order bit of the result.
- In other words, the 100%-correct individual can be simplified as shown in Figure 71.

References

[1] Banzhaf, W., P. Nordin, R. E. Keller and F. D. Francone (1998), *Genetic Programming – An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*, Morgan Kaufmann Publishers, Inc.

[2] Deneubourh, J. L., S. Aron, S. Goss, J. M. Pasteels, and G. Duerinck (1986), "Randon Behavior, Amplification Processes and Number of Participants: How They Contribute to The Foraging Properties of Ants," In Farmer, Doyne, Lapedes, Alan, Packard, Norman, and Wendroff, Burton (editors), *Evolution, Games, and Learning*. North-Holland.

[3] Deneubourh, J. L. S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chretien (1991), "The Dynamics of Collective Sorting Robot-Like Ants and Ant-Like Robots," In Meyer, Jean-Arcady, and Wilson, Stewart W. (editors), *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*. MIT Press.

[4] Isaacs, R. (1965), *Differential Games*. Wiley.

[5] Koza, J. R. (1992), *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press.

[6] Koza, J. (1994), *Genetic programming II: Automatic Discovery of Reusable Programs*. MIT Press.

[7] Mataric, M. J. (1990), "A Distributed Model for Mobile Robot Environment-Learning and Navigation," MIT Artificial Intelligence Labortory technical report AI-TR-1228.

[8] Nilsson, N. J. (1989), "Action Networks," In Tenenberg, J. et al. (editord), *Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems*. University of Rochester Computer Science Department techincal report 284.

[9] Quinlan, J. R. (1986), "Induction of Decision Trees," *Machine Learing*, 1 (1), pp. 81-106.

[10] von Neumann, J. (1987), "Probabilistic Logica and the Synthesis of Reliable Organisms from Unreliable Components," In Aspray, William, and Burks, Arthur (editors), *Papers of John von Neumann on Computing and COmputer Theory*. MIT Press.