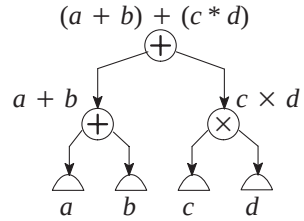
GENETIC PROGRAMMING (GP)

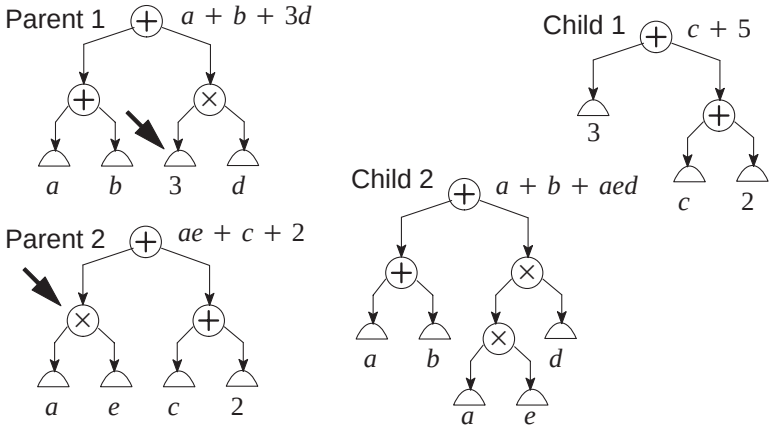
* GP is an evolutionary computation method proposed by John Koza around 1990.
* In genetic algorithms (GAs), a feasible solution is encoded as a chromosome which is interpreted and assigned a fitness.
* In GP, the genetic manipulation is performed on executable programs rather than strings. The fitness of a program is derived from the answers produced when the program is applied to test data.
* The ideal situation for the application of GP:
  + One needs a program that operates in some situation;
  + The situation is characterized by typical (or an exhaustive set of) inputs and associated outputs;
  + GP yields the program. It then can be used in operational mode. No programmer is necessary to write the program.

PROGRAM REPRESENTATION

* The programs in GP should be represented in such a way that manipulations such as *crossover* and *mutation* are possible.
* One cannot easily use text representations for conventional programming languages or assembly code; manipulations that preserve syntax and executability would become very complex.
* *Trees* are a much more suitable representation. *Subtrees* correspond to subprograms. Crossover can e.g. be based on exchanging subtrees between parents. Example:

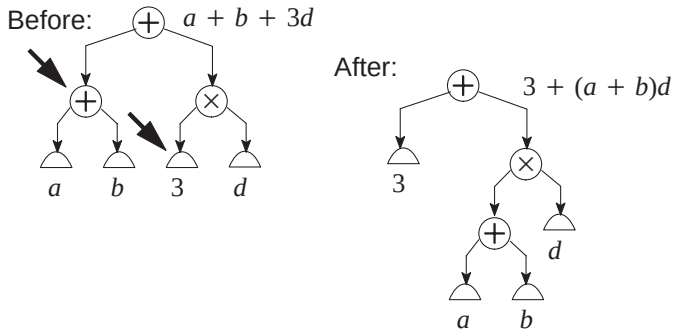


CROSSOVER IN TREES



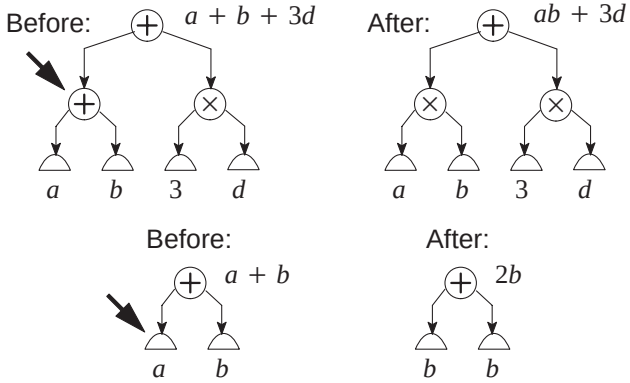
PERMUTATION

Permutation is somewhat comparable to *inversion* in GAs. It interchanges subtrees within the same tree.



MUTATION

Mutation randomly changes a terminal or nonterminal node in the tree.



THE LISP LANGUAGE (1)

* One of the oldest programming languages, invented by John McCarthy in the late fifties. It has been adapted throughout the years and is still especially popular in artificial intelligence.
* LISP was chosen by Koza for his first GP experiments. It has some convenient features that make it attractive for GP. Many modern GP implementations no longer use LISP (but e.g. C++).
* LISP = LISt Processing. *Lists* are the main data structure. List elements can be *atoms* (atomic symbols) or other lists. Examples:

```
+ (A B C)
+ (A (B C) (D (E (F))))
```


THE LISP LANGUAGE (2)

* A LISP program is itself a list. Functions are expressed using a prefix notation, where the function name corresponds to the first element in the list. Examples:

  `+ (+ B C)` corresponds to $B + C$.

  `+ (* B (+ E (* A C)))` corresponds to $B(E + AC)$.
* As lists are the most important data structure, LISP has many built-in functions for the manipulation of lists. If the individuals for GP are represented as lists in LISP, their manipulation with crossover, mutation, etc. become easy to implement.
* The evaluation of an individual is also easy. One simply supplies the list representing the program to the function `eval`. Assume that `B` and `C` are inputs with values 2 and 5; then:

  `(eval '(+ B C))` $\equiv 7$.

THE EVOLUTION ALGORITHM

* In GP the same evolutionary principles are used as in GAs.
* There is a *population* of individual programs. The initial population is often chosen randomly.
* The *fitness* is often *normalized* and used in a selection mechanism.
* Various *selection mechanisms* (roulette wheel, tournament, etc.) are used. Sometimes, when working with large population sizes (larger than 1000), *overselection* is used: the selection is more than proportional to fitness.
* Various *replacement schemes* are used.

THE GP INSTANCE

An instance of the GP problem is defined by:

* A problem-dependent set of functions that can be used as nonterminal nodes. Example: $\{+, \times, /, -, \sin, \cos\}$.
* A problem-dependent set of terminal nodes. This set should contain all input variables of the problem and constants. Constants can be specified by an interval from which a random choice is made when individuals for the initial population are selected.
* A data set of typical inputs (or a set of all possible inputs).
* A function that computes fitness based on a program's performance for the input data set.

THE CLOSURE PROBLEM

* In order to make it possible to freely interchange nodes in the tree, all functions datatypes in the program should be the same, e.g. integers.
* This requires a careful design of the function set as well as a conversion of inputs to the chosen datatype.
* It is undesirable that program evaluation halts due to exceptions such as "divide by zero": the divide function to be provided should check for a zero divider and return an appropriate numeric value.

CONDITIONAL CONSTRUCTS

* The trees shown until now represented simple combinational functions.
* Sometimes it is useful to be able to have conditional computations. This could be achieved by a three-argument function `if`, the first argument being the condition, the second the `then` branch and the third the `else` branch. Example:

```
(if (> a 0) (- b c) (- c b))
```

* Note: supposing that the program uses the integer datatype, one has to agree that Booleans are also encoded as integers.

EXAMPLE: SYMBOLIC REGRESSION

* *Problem description:* one knows that some set of input-output pairs have been generated by a mathematical function, e.g. a polynomial of unknown order; try to find this function.
* Problem instance reported by Koza [1] concerns the function $x^4 + x^3 + x^2 + x$ in the interval $[-1, 1]$, characterized by 20 data points in this interval. A GP run was set up with `X` as the only terminal nodes and a function set consisting of "protected" versions of `+`, `-`, `*`, `/`, `SIN`, `COS`, `EXP`, and `LOG` (protected means that division by zero, the logarithm of a negative number, etc. do not cause exceptions).
* In an experiment with a population size of 500 the following correct solution was found at generation 34:

```
(+ X (* (+ X (* (* (+ X (- (COS (- X X)) (- X X)))
                   X) X)) X)).
```

EXAMPLE: PAC-MAN (1)

* *Problem description:* perform one move in the well-known game of Pac-Man, a computer game where the “Pac-Man” can walk through a two-dimensional maze; it should eat as many food items deposited in the maze while avoiding to be eaten by “monsters”.
* The program to be evolved is called repetitively until it either eats all the food or it is eaten by a monster. The score collected until the end is a measure for the fitness.
* In a straightforward approach, the complete state of the maze would be an input to the program and the program would have to figure out how to compute distances, estimate danger, etc. This becomes too complex.

EXAMPLE: PAC-MAN (2)

* Instead, the problem was solved by special-purpose functions and terminals.
  + Function example: `if-less-than-or-equal`, a four-argument function that compares its first and second arguments and then either evaluates its third or its fourth argument.
  + Terminal example: `distance-to-food`, an input variable that is updated after each move.
  + Other terminal example: `advance-to-food`; the evaluation of this input variable causes the Pac-Man to move towards the food; its numeric value is less relevant.

ANALYSIS (1)

* The GP technique is a heuristic technique: no guarantee exists that a solution will be found after some number of generations. In addition, the method is probabilistic and dependent on the initial population.
* An interesting question is: what is more efficient, to have a few runs with many generations or many runs with few generations?
* Call $Y(G_p, i)$ the probability that a run with population size G_p finds the required solution for the first time after i generations.
* The cumulative probability that a solution is found between generation 0 and i in a run with population size G_p is given by $P(G_p, i)$:

$$P(G_p, i) = \sum_{k=1}^{i} Y(G_p, k).$$

ANALYSIS (2)

* The probability z of finding a solution after r runs of at most i generations is:

$$z = 1 - \left[1 - P(G_p, i)\right]^r.$$

* One can experimentally determine the number of evaluations needed to achieve a fixed value for z, say $z = 0.99$. It turns out that there is a minimum somewhere in the middle of the ranges that r and i can assume.
* For low i, $P(G_p, i)$ is low and a high r is required resulting in a high number of evaluations. For high i, $P(G_p, i)$ starts growing very slowly which also means that many evaluations are needed.

AUTOMATICALLY DEFINED FUNCTIONS

* Any nontrivial computer program has hierarchy and consists of *subroutines* that are (often) called multiple times.
* GP can take advantage of subroutines. Suppose that some subtree that solves part of the problem would be useful another part of the problem as well, the effort to discover the subtree once more can be saved if there is a mechanism to treat the useful subtree as a subroutine.
* Such subroutines that are subject to evolution themselves are called *automatically defined functions* (ADFs) in the context of GP.

ADF REPRESENTATION

* Consider the LISP function definition mechanism consisting of the function name `defun`, a function name, an argument list and a function body. Example:

```
(defun dist (a b) (+ (* a a) (* b b)))
```

* Another useful LISP function is `progn`: it has an arbitrary number of arguments and evaluates its arguments sequentially.
* Then, individuals for GP with ADFs look like:

```
(progn (defun ADF0 <argument list> <function body>)
                    <main body>)
```

* `ADF0` should be part of the function set that can be called from the main body.
* The top-level structure of individuals are protected such that genetic operators only manipulate the function and main bodies.

MORE ADVANCED TOPICS

* When using ADFs, the user should indicate how many ADFs to use and how many arguments each ADF should have. One says that the GP has a *fixed architecture.*
* It is also possible to let GP itself determine its architecture. Then genetic operations should be defined that modify the architecture of individuals. These are called *architecture altering operations.*
* All programs considered for GP discussed so far did not have any internal memory. In a way similar as the extensions for ADFs, one can think of *memories* that become part of the architecture, and functions that can read and write that become part of the function set.
* *Loops* are also a control structure that can be incorporated in GP.

HARDWARE EVOLUTION

* GP deals with the automatic invention of software programs including software that create hardware and can e.g. be simulated by SPICE. The result of the simulation is a measure for the fitness.
* Nowadays, *field-programmable gate arrays* (FPGAs) with large complexity are becoming available. They consists of large numbers of simple electrical gates (look-up tables for constructing combinational logic such as AND and OR gates, flipflops and memories). The interconnection between the gates is controlled by a very long bit string that is downloaded on the FPGA.
* Configuring FPGAs and let them run becomes an interesting alternative for fitness evaluation.
* A completely different aspect of hardware evolution is *embryonics* that aims at simulating cellular growth in hardware (e.g. FPGAs).

FURTHER READING (1)

John Koza is the leading researcher in genetic programming. He has (co)authored three voluminous books on the topic. [1] explains the basics, [2] is mainly dedicated to “automatically defined functions” and [3] deals with “architecture altering operations” and has extensive attention for the “automated synthesis of analog electrical circuits”.

[1] Koza, J.R., Genetic Programming, On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, Massachusetts, (1992).

[2] Koza, J.R., Genetic Programming II, Automatic Discovery of Reusable Programs, MIT Press, Cambridge, Massachusetts, (1998).

[3] Koza, J.R., F.H. Bennett III, D. Andre and M.A. Keane, Genetic Programming III, Darwinian Invention and Problem Solving, Morgan Kaufmann, San Francisco, California, (1999).

FURTHER READING (2)

* Different aspects of hardware evolution including embryonics are discussed in [4].
* Results of research on using digital (!) FPGA technology for designing analog (!) circuits are presented in [5].

[4] Mange, D. and M. Tomassini (Eds.), Bio–Inspired Computing Machines, Towards Novel Computational Architectures, Presses Polytechniques et Universitaires Romandes, Lausanne, (1998).

[5] Thompson, A., Hardware Evolution, Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution, Springer, London, (1998).