UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Global and Multi-Objective Optimisation

Lecturers:
Prof. Luca Manzoni

Author:
Andrea Spinelli

July 2, 2025

github.com/Spina02 andreaspinelli2002@gmail.com

Preface

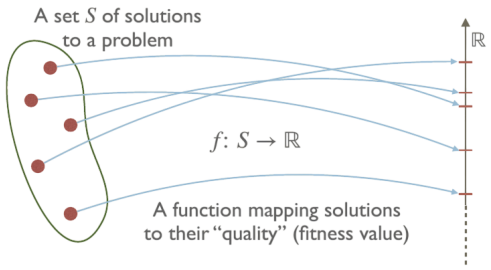
Contents

1 Introduction

1.1 Problem formulation

Given a set S of candidate solutions to an optimization problem, we seek a mapping $f : S \to \mathbb{R}$ which assigns to each solution $x \in S$ a fitness value $f(x)$. Our goal is to find:

$$\arg\max_{x \in S} f(x) \quad \text{or} \quad \arg\min_{x \in S} f(x).$$



In many practical settings it is not possible to solve this problem analytically: the search space S may be exponentially large, the function f may be a "black-box" (we have few assumptions on its smoothness or structure), and an exhaustive enumeration of all solutions can be infeasible. In such cases, we aim instead for heuristics that return solutions of acceptable quality in reasonable time.

1.1.1 A simple illustrative example: OneMax

As a motivating example, let $S = \{0,1\}^n$ and define:

$$f(x) = \text{the number of ones in } x.$$

Clearly the global maximiser is the string 1^n, with fitness n. Even this trivial problem becomes intractable for large n if approached by brute-force enumeration.

Random search

A simplest stochastic approach is random search: pick an initial $b \in S$, then repeatedly sample:

$$x \sim \text{Uniform}(S),$$

and if $f(x) \geq f(b)$ replace b by x. Terminate when a budget of evaluations is exhausted. In the worst case this explores a constant fraction of S, which is equivalent to an exhaustive search in some enumeration order, and so scales poorly in practice.

Tip: ***Random search***

Even if repeated samples are avoided, random search still requires sampling a significant fraction of the space, so it is generally unfeasible for high-dimensional or combinatorial domains.

Hill climbing

Hill climbing maintains a single incumbent solution b. At each iteration we choose a neighbour x of b (according to some neighbourhood structure) and replace b with x if $f(x) \geq f(b)$. The process stops when no improving neighbour can be found or a fixed evaluation budget is reached.

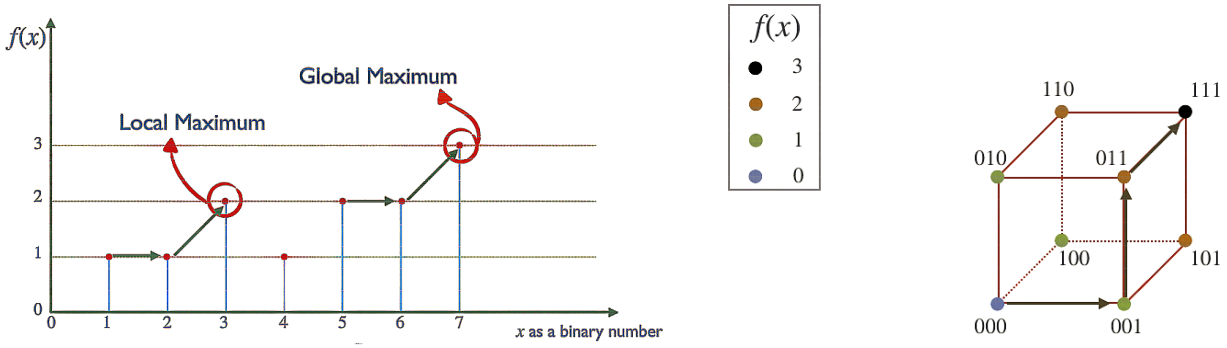


Figure 1.1: With a poor neighbourhood (left), hill climbing can get trapped in a local optimum. A richer neighbourhood (right) may eliminate local traps.

The effectiveness of hill climbing depends critically on the choice of neighbourhood. For `OneMax`, using the Hamming-1 neighbourhood (flip one bit at a time) guarantees reachability of the global optimum but may still require many steps; using only ± 1 on the integer-interpreted string can make the problem insoluble.

Simulated annealing

Simulated annealing augments hill climbing with occasional downhill moves to escape local optima. Starting from $b \in S$ and a "temperature" T, at each step we pick a neighbour x. If $f(x) \geq f(b)$ we accept it, otherwise we accept it with probability

$$\exp\big((f(x) - f(b))/T\big).$$

We then decrease T according to a cooling schedule. Proper tuning of the schedule trades off exploration against exploitation.

Tip: ***Simulated annealing***

Allowing uphill moves with probability depending on the temperature and fitness gap helps avoid entrapment in local maxima. The cooling schedule is crucial for performance.

Multiple restarts and population-based search

Both hill climbing and simulated annealing can be repeated from fresh random starts to reduce the chance of permanent stagnation. A more powerful paradigm uses a whole population of candidate solutions that "interact" (e.g. by recombination), leading naturally to evolutionary algorithms.

2 Genetic Algorithms

2.1 Introduction

Genetic Algorithms (GAs) are a class of stochastic optimization methods inspired by the principles of natural selection and genetics, first formalized by John Holland in the 1970s [1]. At their core, GAs maintain a population of candidate solutions, called ***individuals***, which are evolved over multiple generations to approximate an optimal or sufficiently good solution to a problem.

Each individual in the population is typically represented as a fixed-length string (often a binary vector) termed the ***genotype***. This genotype encodes a possible solution, whose ***phenotype*** is the actual candidate in the problem space, obtained by decoding the genotype. The quality of each individual is measured by a ***fitness function*** f, which assigns a scalar value indicating how well the individual solves the problem at hand.

The standard evolutionary cycle in a GA consists of the following steps:

1. **Selection:** Individuals are chosen probabilistically from the population based on their fitness. Fitter individuals have a higher probability of being selected as parents, implementing a form of artificial natural selection that drives evolution toward better solutions.
2. **Crossover:** Pairs of selected parents exchange genetic material to produce offspring, recombining their genotypes in various ways. This operator enables the algorithm to combine beneficial traits from different solutions and explore the search space effectively.
3. **Mutation:** Random, typically small, modifications are introduced into the offspring's genotypes to preserve genetic diversity and explore new regions of the search space. This operator helps prevent premature convergence and allows the discovery of novel solutions.
4. **Replacement:** The new generation replaces the old one, possibly retaining a fraction of the best solutions (elitism). This ensures the population maintains its best-found solutions while allowing for continuous improvement through evolution.

This process iterates for a predefined number of generations or until a stopping criterion is met. A summary of the cycle is illustrated in Figure 2.1.

Figure 2.1: Left: Diagram showing the main components and their interactions in the evolutionary process. Right: Example of genetic operators (selection, crossover, mutation) acting on binary strings.

2.2 Core Components

Representation

The **genotype** is the encoded representation of a solution (commonly a binary string or vector over a finite alphabet) on which the genetic operators (crossover and mutation) act. The **phenotype** is the decoded, actual candidate solution evaluated by the fitness function. Selection operates at the phenotypic level, favoring those solutions that yield higher fitness.

Observation: ***Genotype vs. Phenotype***

The distinction between the two allow GAs to operate flexibly: operators modify representations (genotypes) while selection is based on problem-specific performance (phenotypes).

Key Parameters

The performance and behavior of a genetic algorithm depend on several critical parameters:

- **Population size** N: Number of individuals maintained at each generation (typically 100-200). Larger populations increase diversity and exploration but require more computational resources.
- **Number of generations** G: How many iterations the algorithm will perform (often determined empirically). This affects the total computational budget and exploration time.
- **Selection method**: The algorithm used to select parents (tournament, roulette wheel, ...). Different methods vary the selection pressures that affect diversity and convergence speed.
- **Crossover operator**: The method for recombining genotypes. Should be chosen based on problem representation and known/assumed relationships between genes.
- **Crossover probability** p_{cross}: Probability with which crossover is applied. Higher values (typically 0.6-0.9) promote more exploration through recombination.
- **Mutation operator**: The method for introducing random changes. Must be appropriate for the chosen representation while maintaining solution feasibility.
- **Mutation probability** p_{mut}: Probability of mutating each gene. Usually set to $1/n$ for length-n genotypes to maintain a balance between exploration and stability.
- **Elitism** e: Number or percentage of best individuals preserved into the next generation. Small values (1-5%) help maintain good solutions while allowing population turnover.

2.2.1 Selection Methods and Genetic Operators

Selection

Several strategies exist for parent selection, each with different characteristics in terms of selection pressure and diversity preservation:

- ***Roulette Wheel Selection:***
  Each individual's probability of being selected is proportional to its fitness $f(x)$:

$$P_{x,P} = \frac{f(x)}{\sum_{y \in P} f(y)}$$

  This method is simple to implement, but can reduce diversity if a single individual dominates.
- ***Ranked Selection:***
  Individuals are ranked by fitness. Selection probabilities depend only on rank, not raw fitness, which helps control selection pressure and maintain diversity.

- ***Tournament Selection:***
  t individuals are sampled (with replacement) from the population, and the fittest among them is selected. The tournament size t directly tunes *selection pressure*: higher t increases the probability that the best individuals are chosen.

Tip: ***Selection Pressure***

Tournament selection is widely used due to its simplicity and ease of adjusting selection pressure by changing the tournament size.

Crossover

Crossover operators create new individuals by combining genetic material from two parents:

- ***One-Point Crossover:***
  A single crossover point k is randomly chosen between genes. The offspring inherit genes from parent A up to position k, and from parent B beyond k. This preserves contiguous gene sequences that may represent important building blocks:

  Parent A: `[1 1 1 | 1 1 1]`
  Parent B: `[0 0 0 | 0 0 0]`
  Offspring: `[1 1 1 | 0 0 0]`

- ***Multi-Point Crossover:***
  Multiple crossover points are chosen, and genetic material is alternately swapped between parents. This allows more flexible recombination while still preserving some gene linkage:

  Parent A: `[1 1 | 1 1 | 1 1]`
  Parent B: `[0 0 | 0 0 | 0 0]`
  Offspring: `[1 1 | 0 0 | 1 1]`

- ***Uniform Crossover:***
  For each gene position, the gene is swapped between parents with a fixed probability (commonly $1/2$). This provides maximum mixing potential and is especially useful when there is little/no linkage between adjacent genes:

  Parent A: `[1 1 1 1 1 1]`
  Parent B: `[0 0 0 0 0 0]`
  Offspring: `[1 0 1 0 0 1]`

The choice of crossover operator and its probability p_{cross} influences the balance between ***exploration*** and ***exploitation*** in the search process.

Tip: ***Crossover Design***

Select the crossover operator based on the structure of the problem representation. For representations where tightly coupled genes are adjacent, one-point crossover is often effective. For others, uniform crossover can better promote exploration.

Mutation

Mutation introduces random changes to individuals, preserving genetic diversity and enabling the exploration of new areas in the search space. The most common operator for binary representations is the ***bit-flip mutation***: for each gene, flip its value with probability p_{mut} (typically $1/n$ for length-n genotypes, so that on average one bit per individual mutates per generation).

2.2.2 Common Variants

Several variations of the basic GA have been developed:

- **Elitism**
  Strategies that preserve the best individual(s) unchanged into the next generation, guaranteeing solution quality does not degrade. Variants include retaining the single best solution, top k solutions, or best $p\%$.
- **Steady-State GA**
  Instead of replacing the entire population each generation, only a subset of individuals is replaced. The choice of which individuals to replace impacts algorithm dynamics.
- **Hybrid (Memetic) Algorithms**
  These incorporate local search techniques to further improve individuals after genetic operations. Also called *Lamarckian algorithms* or *Baldwin effect algorithms*, they require tuning of local search frequency and intensity.

2.3 Representation

While we have focused on binary representations so far, genetic algorithms can be generalized to work with symbols from any finite alphabet Σ instead of just binary values. When using a larger alphabet, mutation needs to be adapted - rather than simply flipping bits, it selects uniformly between the $|\Sigma| - 1$ alternative symbols. For ordered alphabets like $\{0, 1, 2, 3\}$, mutation can also be implemented to increment or decrement values, maintaining the ordering relationship.

2.3.1 Real-Valued GA

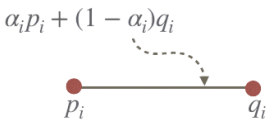
Traditional binary GAs can encode floating-point numbers using 32 or 64 binary genes, where different bit positions have varying impacts on the final value. However, real-valued GAs take a more direct approach by using floating-point genes directly and adapting the genetic operators accordingly.

Crossover

Real-valued GAs employ two main specialized crossover operators:

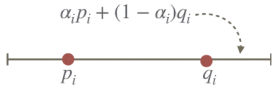
- **Intermediate recombination:** Creates offspring by taking weighted averages of the parents' genes. Given parents x_1 and x_2:

$$y_i = \alpha_i p_i + (1 - \alpha_i) q_i, \quad \alpha_i \sim \text{Uniform}(0, 1)$$



- **Line recombination:** Extends intermediate recombination by allowing exploration beyond parents:

$$y_i = \alpha_i p_i + (1 - \alpha_i) q_i, \quad \alpha_i \sim \text{Uniform}(-k, 1 + k)$$



Mutation

For real-valued representations, mutation operates by adding small perturbations to each coordinate.

$$p \leftarrow p + \varepsilon$$

These perturbations ε can be drawn from either a *uniform distribution* within specified bounds or a *Gaussian distribution* centered at the current value. The choice between these distributions affects how mutation explores the search space.

2.3.2 Permutation-Based GA

Many optimization problems involve finding optimal permutations of elements $\{1,\ldots,n\}$. These problems require specialized genetic operators that preserve the permutation constraints. While mutation typically operates by simply swapping two positions, crossover requires more sophisticated approaches.

Partially Mapped Crossover (PMX)

PMX is a sophisticated crossover operator that preserves permutation validity through a four-step process. It ensures that offspring maintain valid permutations by carefully handling element mappings and conflict resolution:

1. Two random crossover points are selected in both parent permutations, defining a matching segment
2. A mapping is constructed between corresponding elements in the segments, recording which values swap positions
3. The segments are directly exchanged between parents to create initial offspring
4. Remaining positions outside the segments are filled by:
   - Copying values from the original parent if no conflicts exist
   - For conflicts, following the mapping chain until finding a non-conflicting value

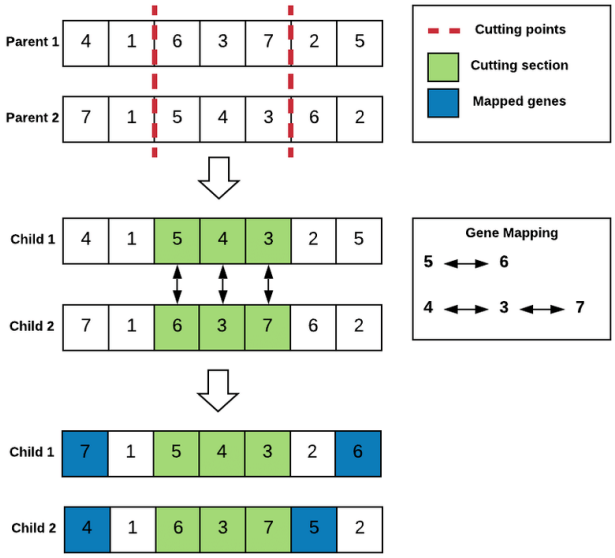


Figure 2.2: Partially Mapped Crossover [2].

This process guarantees that each element appears exactly once in the final offspring, maintaining permutation validity while allowing meaningful genetic exchange between parents.

Cycle Crossover

Cycle crossover provides an alternative approach that preserves absolute positions from the parents. It operates by identifying and preserving cycles in the permutations: starting at a position i, it copies the value from the first parent, then finds that same value in the second parent, continuing until a cycle is completed. The remaining values are then copied from the second parent.

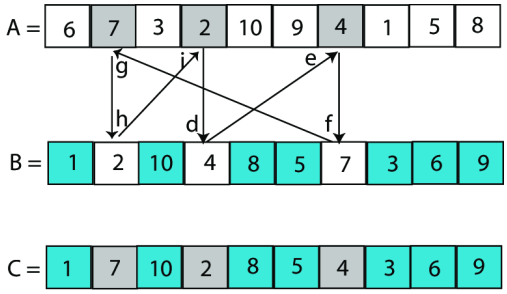


Figure 2.3: Cycle Crossover. [2]

2.3.3 Graph Representation

Graphs are ubiquitous structures in computer science, appearing in applications ranging from social networks and transportation systems to neural networks and circuit design. When applying genetic algorithms to graph-based problems, the graph representation must be carefully chosen. Graphs can be encoded in two fundamental ways:

- **Direct encoding:** The graph structure is explicitly represented through its vertices and edges, providing straightforward access to the graph's components but potentially requiring significant memory for large graphs
- **Indirect encoding:** Instead of representing the graph directly, this approach encodes a constructive process or set of rules that, when executed, builds the desired graph structure. This can be more compact and enable the emergence of patterns, though interpretation requires additional computation

Direct Encoding

Direct encoding methods explicitly represent the graph structure, offering intuitive but potentially memory-intensive representations:

- **Adjacency Matrix**
  A matrix A where entry a_{ij} represents the edge between vertices i and j. For weighted graphs, entries can be numerical values indicating edge weights, while for unweighted graphs, binary values (0/1) indicate edge presence:

$$A = \begin{bmatrix} 0.4 & \text{no edge} & 0.6 & -5.3 \\ \text{no edge} & 5.6 & 0.1 & 0.2 \\ 2.4 & 0.8 & 4.1 & 8.3 \\ -0.2 & \text{no edge} & 0.5 & \text{no edge} \end{bmatrix}$$

  Where "no edge" denotes absence of connection. This representation allows for efficient edge lookup ($O(1)$) but requires $O(|V|^2)$ space complexity.
- **Edge List**
  A more compact representation that explicitly maintains sets of vertices V and edges E. Particularly efficient for sparse graphs where $|E| \ll |V|^2$.
  For example, given vertices:

$$V = \{a, b, c, d, e\}$$

  we might have edges

$$E = \{(a,b), (a,c), (d,a), (e,e)\}$$

  For edge list representations, several specialized ***mutation*** operators are employed, each with carefully tuned probabilities to maintain graph validity:
  - Add a new edge between existing vertices (preserving graph constraints like maximum degree)
  - Remove an existing edge (maintaining connectivity if required)
  - Add a new vertex (with appropriate edge connections)
  - Remove a vertex and all its associated edges (ensuring graph remains valid)

Tip: ***Graph Crossover***

Graph representations pose **significant challenges for crossover operations** due to the difficulty in preserving graph properties and structure. In practice, it is often more effective to rely solely on mutation rather than attempting to implement complex crossover schemes.

Indirect Encoding

Indirect encoding takes a different approach by using **production rules** that generate graphs through an iterative expansion process. Rather than directly representing the graph structure, these rules define how to construct the graph step by step. Starting with a set of *non-terminal symbols*, the rules map each symbol to *sequences* or *matrices* that may contain both terminal and non-terminal symbols. This process continues recursively, until only terminal symbols remain.

Example: ***Hiroaki Kitano's graph generation system***

For example, Kitano's graph generation system uses production rules of the form:

$$S \to \begin{bmatrix} A & B \\ C & D \end{bmatrix},$$
$$A \to \begin{bmatrix} c & d \\ a & c \end{bmatrix}, \quad B \to \begin{bmatrix} a & a \\ a & e \end{bmatrix}, \quad C \to \begin{bmatrix} a & a \\ a & a \end{bmatrix}, \quad D \to \begin{bmatrix} a & a \\ a & b \end{bmatrix}$$
$$a \to \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad b \to \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad c \to \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad d \to \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}, \quad e \to \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Starting from the axiom S, we can iterate through the production rules:

$$S \longrightarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} \longrightarrow \begin{bmatrix} \begin{bmatrix} c & d \\ a & c \end{bmatrix} & \begin{bmatrix} a & a \\ a & e \end{bmatrix} \\ \begin{bmatrix} a & a \\ a & a \end{bmatrix} & \begin{bmatrix} a & a \\ a & b \end{bmatrix} \end{bmatrix} \longrightarrow \cdots$$

This process continues until we reach a final 8×8 binary matrix representing the adjacency matrix of a graph with 5 vertices (where some vertices may be isolated, meaning they have no connections to other vertices):

$$\begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix}$$

Production rules in indirect encoding systems can typically be represented as fixed-length vectors or strings, making them amenable to traditional genetic algorithm operators. For example, in Kitano's system, the first element represents the head (non-terminal symbol) and the remaining elements represent the body (the matrix elements). For instance, the rule $s \to \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ can be encoded as the vector $[S, A, B, C, D]$.

3 Evolution Strategies

3.1 Introduction

Evolution Strategies (ES) are a family of optimization algorithms developed in Germany during the 1960s. Initially conceived for experimental optimization of hydrodynamic shapes, ES has since evolved into a robust methodology for numerical optimization, particularly in continuous domains.

While sharing common roots with Genetic Algorithms (GAs) in the broader field of evolutionary computation, ES possesses distinct characteristics.

Observation: ***ES and GAs: Similarities and Key Differences***

Similarities:

- They maintain a ***population*** of candidate solutions.
- Offspring are generated primarily through ***mutation***, which introduces variation.
- A ***selection process*** determines which individuals survive and/or reproduce, driving the population towards better solutions.

Key Differences:

- ***No Crossover***: While modern ES can incorporate recombination, it was not a primary operator in early ES and is often considered secondary to mutation. GAs, conversely, typically emphasize crossover.
- ***Selection Mechanism***: ES commonly employs deterministic ***truncation selection***, where only the top-ranked individuals survive or become parents. GAs often use probabilistic selection methods like roulette wheel or tournament selection.
- ***Representation***: ES is predominantly designed for and applied to problems with ***real-valued (floating-point) individuals***, whereas GAs were initially developed with binary strings and have been adapted for other representations.
- ***Self-Adaptation***: A hallmark of advanced ES is the self-adaptation of strategy parameters (e.g., mutation strengths and directions), which are encoded within the individuals and evolve alongside the solutions themselves.

3.2 Parameters and Notation

Two key parameters define the size of the parent and offspring populations in ES:

- μ: The number of ***parent individuals*** selected in each generation to create offspring.
- λ: The number of ***offspring individuals*** generated in each generation.

It is generally required that $\lambda \geq \mu$.

Based on how parents and offspring are managed and selected, two main ES schemes are distinguished: the $(\mu, \lambda) - ES$ and the $(\mu + \lambda) - ES$.

The $(\mu, \lambda) - ES$ Scheme

In the $(\mu, \lambda) - ES$ (read "mu comma lambda ES"), the algorithm proceeds as follows:

1. Generate an initial population of λ offspring
2. Evaluate the fitness of all individuals in the current population.
3. Select the μ best-performing individuals from these λ *offspring only* (truncated selection) to become the parents of the next generation.
4. Generate λ new offspring by applying mutation (and/or recombination) to the μ parents.
5. Discard the previous generation's parents entirely (no parent survives to the next generation).
6. Return to step 2 and repeat until a termination criterion is met.

This scheme is inherently ***non-elitist*** because parents never survive into the next generation. As a result, the $(\mu, \lambda) - ES$ can more easily escape local optima, but it also risks losing the best-so-far solution if it is not re-discovered by an offspring.

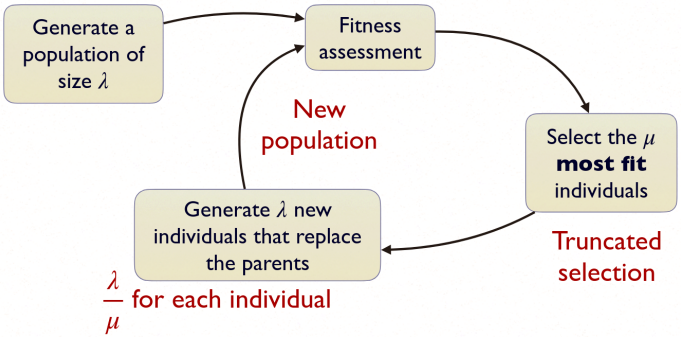


Figure 3.1: The lifecycle of a $(\mu, \lambda) - ES$

The $(\mu + \lambda) - ES$ Scheme

In the $(\mu + \lambda) - ES$ (read "mu plus lambda ES"), the algorithm proceeds as follows:

1. Generate an initial population of λ offspring.
2. Evaluate the fitness of all individuals in the current population.
3. Select the μ highest-fitness individuals to serve as the parents of the next generation.
4. Generate λ new offspring by applying mutation and/or recombination to the μ parents.
5. Form the next population by combining parents and offspring, resulting in $\mu + \lambda$ individuals.
6. Return to step 2 and repeat until a termination criterion is met.

This scheme is ***elitist*** because it always retains the top μ solutions from one generation to the next. Elitism tends to accelerate convergence toward high-quality solutions but can also lead to premature convergence if population diversity is lost too quickly.

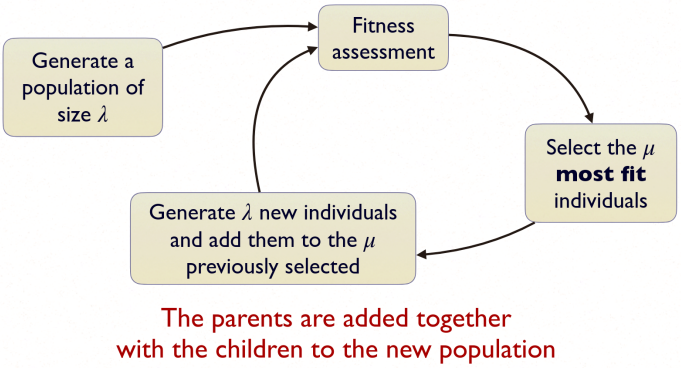
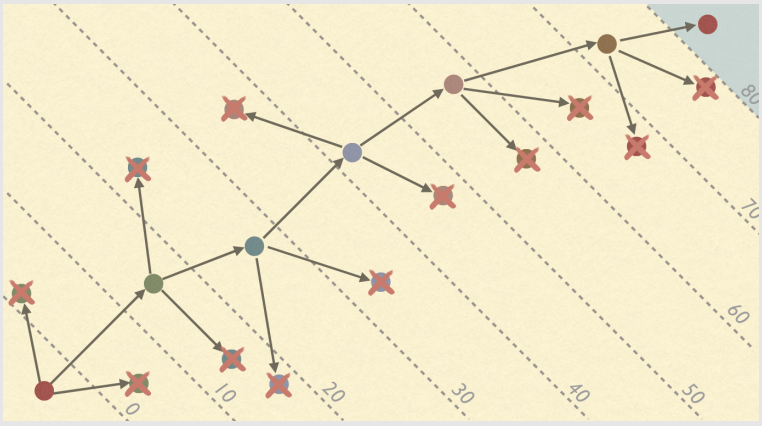


Figure 3.2: The lifecycle of a $(\mu + \lambda) - ES$.

Example: *Illustrative (1,3)-ES*

Consider a simple $(1,3)$-ES, a specific type of (μ,λ)-ES where $\mu = 1$ and $\lambda = 3$.

1. Start with one parent individual.
2. This single parent generates three offspring (e.g., by applying mutation three independent times).
3. The fitness of these three offspring is evaluated.
4. The single best offspring out of the three becomes the sole parent for the next generation. The original parent and the other two offspring are discarded.



This process continues over several generations, with each generation showing a parent producing multiple offspring, some being discarded, and one being selected to continue. Generally, this leads to an increase in fitness over time as the population evolves toward better solutions.

3.3 Mutation in Evolution Strategies

Mutation serves as the fundamental mechanism for generating diversity and enabling exploration in Evolution Strategies. While recombination can introduce additional variation, mutation remains the primary driver of evolutionary change, particularly in simpler ES variants that rely solely on mutation for population diversity. The effectiveness of an ES algorithm heavily depends on the design and implementation of its mutation operators.

Properties of a Good Mutation Operator

A well-designed mutation operator should ideally exhibit the following properties:

- ***Reachability:***
  Any point in the search space should be reachable from any other point in a finite number of mutation steps. This ensures the algorithm is not, in principle, confined to a subspace.
- ***Unbiasedness:***
  The mutation operator itself should not introduce any bias towards particular regions of the search space based on fitness values. The guidance towards promising regions is the role of selection.
- ***Scalability (Adaptability):***
  The ”strength” or step size of the mutation should be adaptable to the characteristics of the fitness landscape. For instance, larger steps might be beneficial early in the search (exploration), while smaller steps are preferred later for fine-tuning (exploitation).

Mutation for Different Representations

- ***Binary Values:*** For individuals represented as binary strings, mutation typically involves ***bit-flips***, similar to GAs, where each bit has a probability of being inverted.
- ***Real Values:*** For individuals represented as vectors of real numbers $\mathbf{x} = (x_1, \ldots, x_n)$, mutation is commonly performed by adding random noise, often from a Gaussian distribution:

$$x_i' = x_i + N(0, \sigma_i^2)$$

Here, x_i is the i-th component of the individual, $N(0, \sigma_i^2)$ is a random number drawn from a Gaussian (normal) distribution with mean 0 and standard deviation σ_i (or variance σ_i^2). The σ_i values are called ***mutation strengths*** or ***step sizes*** and are critical parameters. A key question is how to determine appropriate values for these σ_i's.

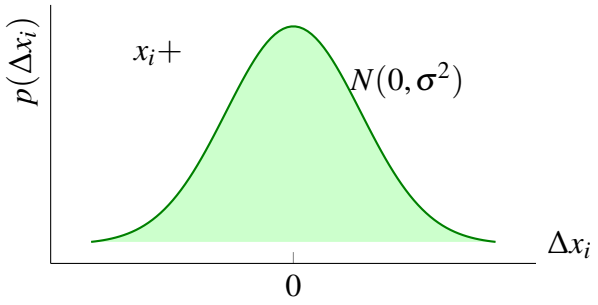


Figure 3.3: Gaussian mutation adds a random value drawn from a normal distribution to x_i.

Setting $\mu = 0$ for the Gaussian noise seems natural as it ensures mutation is unbiased in direction. However, selecting the variance is crucial and often addressed through self-adaptation.

Self-Adaptation of Mutation Parameters

A powerful feature of many ES variants is ***self-adaptation***, where the strategy parameters (like mutation step sizes σ_i) are not fixed but are themselves part of the individual's genotype and evolve alongside the solution variables.

- An individual can be represented as a pair $\langle \mathbf{x}, \mathbf{s} \rangle$, where $\mathbf{x}$ is the vector of solution variables and $\mathbf{s}$ is a vector of strategy parameters (e.g., σ_i values for each x_i).
- When an individual is mutated, its strategy parameters $\mathbf{s}$ are typically mutated first.
- Then, these mutated strategy parameters $\mathbf{s}'$ are used to mutate the solution variables $\mathbf{x}$ to get $\mathbf{x}'$.
- Selection acts on the fitness of $\mathbf{x}'$, indirectly selecting for effective strategy parameters as well.

This enables the ES to adapt its mutation strategy to the fitness landscape's local characteristics.

The One-Fifth (1/5) Success Rule

The ***1/5 success rule***, introduced by Ingo Rechenberg in the 1970s, is an early and influential heuristic for adapting a single, global mutation step size σ in a simple ES. The rule aims to maintain a certain rate of successful mutations (those that lead to fitter offspring).

Let p_s be the success rate (mutations producing offspring with fitness $\geq$ parent) over a fixed period. The rule states:

- If $p_s > 1/5$: The success rate is too high, thus the step size σ might be too small. Increase σ.
- If $p_s < 1/5$: The success rate is too low, thus the step size σ might be too large. Decrease σ.
- If $p_s = 1/5$: The step size is considered optimal; leave σ unchanged.

Operationally, this is often implemented by using two parameters: k and c. k is the number of generations between updates of σ and c is a constant, typically $0.817 < c < 1$ (e.g., $c \approx 0.85$).

- If $p_s > 1/5$, then set $\sigma \leftarrow \sigma / c$. (Since $c < 1$, $1/c > 1$, so σ increases).
- If $p_s < 1/5$, then set $\sigma \leftarrow \sigma \cdot c$. ($\sigma$ decreases).
- Otherwise (if $p_s \approx 1/5$), σ remains unchanged.

Tip: ***Rationale of the 1/5 Success Rule***

The 1/5 success rule provides a simple mechanism for controlling the balance between exploration and exploitation. A high success rate ($> 1/5$) implies that the algorithm is making progress easily, possibly with steps that are too small; increasing the step size encourages broader exploration. A low success rate ($< 1/5$) suggests that many mutations are detrimental, possibly because the step size is too large; decreasing it allows for finer exploitation of the current region. The value 1/5 was derived empirically and theoretically for specific model problems (e.g., the sphere model and corridor model).

3.4 Evolution Strategies with Recombination

In ES, while mutation serves as the main search mechanism, ***recombination*** (crossover) can be added as a complementary operator, proving particularly valuable in advanced ES implementations. This process typically combines ρ parent solutions to generate offspring. The extended notation is:

- $(\mu/\rho, \lambda) - ES$: μ parents are chosen, and from these, groups of ρ parents are used for recombination to produce λ offspring. The next generation is selected from these λ offspring.
- $(\mu/\rho + \lambda) - ES$: It follows the same parent selection and recombination process, but the next generation is selected from both the μ current parents and the λ offspring combined.

For real-valued representations, two main recombination types are used:

- **Discrete Recombination**: Each component j of offspring $\mathbf{x}'$ inherits its value from a randomly selected parent:
$$x'_j = x_{p_j,j}, \quad p_j \in \{1, \ldots, \rho\}$$
Strategy parameters $\mathbf{s}$ can be recombined similarly.
- **Intermediate Recombination**: Each component j is the average of corresponding parent values:
$$x'_j = \frac{1}{\rho} \sum_{i=1}^{\rho} x_{i,j}$$
Common choices are $\rho = 2$ (midpoint) or $\rho = \mu$ (all parents contribute).

Example: ***Recombination in Practice***

Suppose we use $\rho = 2$ parents for recombination.

- ***Discrete Recombination:*** For an offspring $\mathbf{x}' = (x'_1, \ldots, x'_n)$ from parents $\mathbf{p}_1$ and $\mathbf{p}_2$: For each $j \in \{1, \ldots, n\}$, x'_j is randomly chosen to be either $p_{1,j}$ or $p_{2,j}$.
- ***Intermediate Recombination:*** For an offspring $\mathbf{x}'$ from parents $\mathbf{p}_1$ and $\mathbf{p}_2$: For each $j \in \{1, \ldots, n\}$, $x'_j = (p_{1,j} + p_{2,j})/2$.

Recombination can be either ***global*** (using one set of ρ parents for all components of an offspring) or ***local/component-wise*** (using different parent sets for each component), with global recombination being more common in practice.

4 Genetic Programming

4.1 Introduction

Genetic Programming (GP) is a specialized domain within evolutionary computation where the individuals of the population are not fixed-length strings representing parameters, but executable computer programs. Popularized by John Koza in the late 1980s, GP aims to stochastically evolve populations of programs to find one that performs a user-defined task satisfactorily.

While the high-level evolutionary cycle of GP is analogous to that of a Genetic Algorithm (involving initialization, fitness evaluation, selection, and variation through crossover and mutation) the core of GP's uniqueness lies in its representation. Instead of evolving solutions *to* a problem, GP *evolves the problem-solving logic itself*.

This shift in representation from data structures to programs opens up a vast application space, from symbolic regression and controller design to automatic programming and machine learning model discovery.

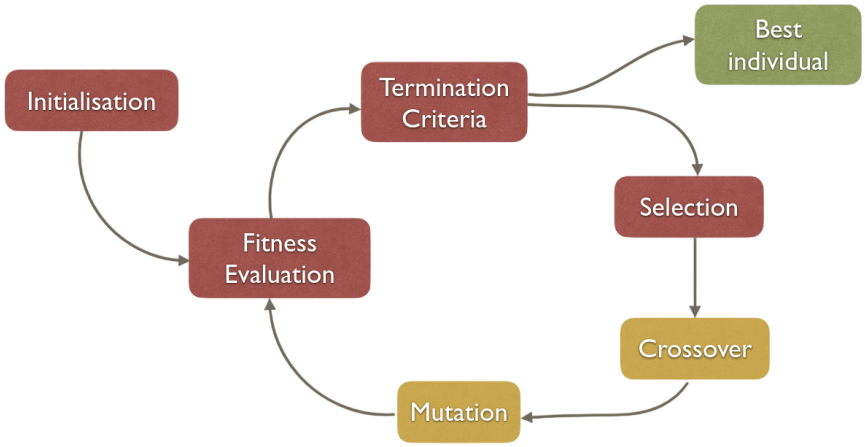


Figure 4.1: The general evolutionary cycle in Genetic Programming, which closely mirrors the cycle of a standard GA but operates on programs instead of fixed-length genotypes.

The fundamental challenge of GP's power, is finding a program representation that can be effectively manipulated by genetic operators to create syntactically valid and semantically useful offspring.

4.2 Program Representation

Representing programs as linear strings, similar to GAs, is problematic. Applying standard crossover to raw source code strings often results in syntactically invalid offspring, as the operator has no awareness of the underlying program structure. This is because the crossover point is likely to break syntax, resulting in non-compilable or nonsensical offspring. To overcome this problem, GP typically represents programs as ***syntax trees*** (or expression trees).

4.2.1 Tree-Based Representation

In the tree-based representation, each program is structured as a rooted tree: **internal nodes** correspond to functions or operators, while the **leaf nodes** are terminals, which can be variables or constants. This tree-based structure offers several key advantages for genetic programming:

- **Modularity:** Any subtree forms a valid, self-contained sub-program or expression, enabling flexible manipulation and reuse of code fragments.
- **Closure and Validity:** Genetic operators can operate on entire subtrees, ensuring that offspring remain syntactically valid and executable.
- **Interpretability:** The hierarchical structure mirrors the way mathematical and logical expressions are naturally composed, making evolved programs easier to visualize and understand.
- **Fitness Evaluation:** The output of a program can be computed by recursively evaluating the tree from the leaves up to the root, allowing for efficient and systematic fitness assessment.

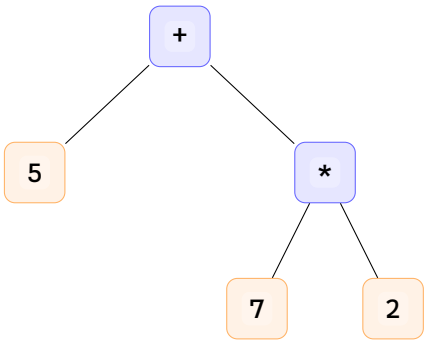


Figure 4.2: The expression $5+(7\times 2)$ as a syntax tree. Internal nodes represent functions or operators (`+` , `*`), while leaf nodes are terminals (5, 7, 2).

Terminals and Functions

A Genetic Programming (GP) system is built from two fundamental sets of building blocks, which together define the "language" in which solutions can be expressed:

- ***Terminal Set*** ($\mathcal{T}$): contains all possible leaf nodes of the tree, elements that do not take any arguments. Terminals are the basic inputs and constants that the evolved programs can use. Typical examples include:
  - **Input Variables:** represent the problem-specific inputs, such as $x_0, x_1, \ldots, x_n$.
  - **Constants:** fixed values such as numbers, or logical values (`true` , `false`).
- ***Function Set*** ($\mathcal{F}$): This set contains all possible internal nodes, elements that take one or more arguments (their *arity*), corresponding to their child nodes in the tree. The function set determines the kinds of computations and logic the GP system can evolve. Examples include:
  - **Arithmetic Operators:** `+` , `-` , `*` , `/` (often with protected division to avoid errors).
  - **Trigonometric Functions:** `sin` , `cos` , `tan` , etc.
  - **Boolean Operators:** `AND` , `OR` , `NOT` , `XOR` , etc., for logical or rule-based problems.
  - **Conditional Constructs:** `IF-ELSE` , `IF-GREATER` , etc., enabling programs to make decisions.
  - **Other Domain-Specific Functions:** functions as `abs` , `log` , `exp` , or user-defined primitives.

The combination of $\mathcal{T}$ and $\mathcal{F}$ forms the ***primitive set***, which specifies the fundamental components available to the GP system for constructing solutions.

Tip: *Primitive Set*

Choosing the primitive set is important: it should be ***expressive enough*** to represent solutions, but ***not so large or complex*** that it makes the search inefficient or the evolved programs unnecessarily complicated.

Properties of the Primitive Set

The choice of primitives is critical and must satisfy two important properties for the GP system to function correctly.

- **Closure**
  The ***closure*** property requires that any function in $\mathcal{F}$ must be able to accept any value returned by another function or any terminal from $\mathcal{T}$. This ensures that any tree constructed from the primitives is valid. It has two main aspects:
  - **Type Consistency:** All functions, operators, and terminals must operate on and return the same data type (or be part of a strongly-typed GP system that can handle multiple types). For example, if all functions expect numbers, a terminal returning `true` would violate closure.
  - **Evaluation Safety:** Functions prone to runtime errors must be ”protected.” For example, `safe_div(a, b)` returns 1 if $b = 0$, avoiding division-by-zero during evaluation.
- **Sufficiency**
  The ***sufficiency*** property requires that the primitive set must be capable of representing a solution to the problem. For instance, if the target function is an exponential, but the primitive set only contains polynomials and variables, it may be impossible to find an exact solution. While perfect sufficiency is often not knowable in advance, the set should be chosen to be rich enough to allow for good approximations.

4.3 Initialisation, Crossover and Mutation

4.3.1 Population Initialisation

Unlike the straightforward random initialisation of binary strings in a GA, generating a population of valid, random program trees in GP requires a more structured approach. The aim is to produce a diverse set of trees with varying sizes and shapes. The most widely used method for this is ***Ramped Half-and-Half***.

Ramped Half-and-Half

This method combines two fundamental tree-generation strategies:

- **The Grow Method**: nodes are randomly selected from the entire primitive set ($\mathcal{F} \cup \mathcal{T}$) as the tree is built, up to a specified maximum depth. Once a branch reaches this maximum depth, only terminals from $\mathcal{T}$ can be chosen. This approach tends to produce ***asymmetric trees*** of various shapes and sizes, since a branch may terminate before reaching the maximum depth if a terminal is selected at an internal node.
- **The Full Method**: nodes are chosen exclusively from the function set $\mathcal{F}$ until the maximum depth is reached. At this depth, all leaf nodes are filled with terminals from $\mathcal{T}$. This results in ”bushy” ***full trees*** where all terminals are located at the same depth.

To ensure diversity in the initial population, the **Ramped Half-and-Half** method works as follows:

1. A range of maximum depths is defined (e.g., from 2 to 6).
2. The population is divided into groups, each assigned a maximum depth from this range.
3. For each depth group, half of the individuals are generated using the *grow* method and the other half using the *full* method.

This approach guarantees that the initial population contains a broad variety of program structures and sizes, which is essential for effective evolutionary search.

Ephemeral Constants

When a problem requires numeric constants, they can be pre-defined in the terminal set (e.g., $\mathcal{T} = \{x, y, 1, \pi\}$). However, a more flexible approach is the use of ***ephemeral constants***. An ephemeral constant is a special terminal that, when selected, generates a random constant within a predefined range (e.g., a random float in $[-1, 1]$). Each time this terminal appears in a tree, it holds a different, randomly generated value.

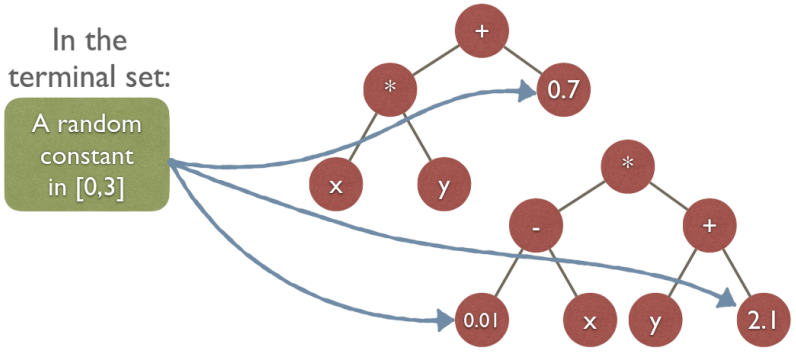


Figure 4.3: The *Ephemeral* terminal generates a random constant in $[0, 3]$. Each instance holds a different random value.

4.3.2 Crossover

The standard crossover operator in GP is **subtree crossover**. It operates as follows:

1. Two parent trees are selected from the population.
2. A random node (the crossover point) is chosen in each parent tree.
3. Swap the selected subtrees between parents to create one or two offspring.

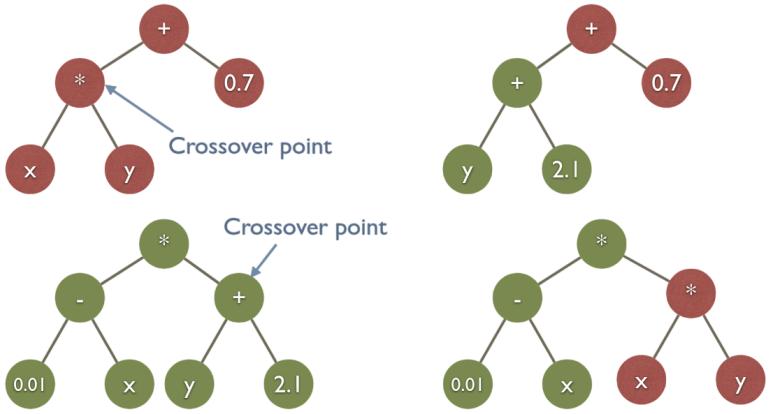


Figure 4.4: Subtree Crossover.

This operator is simple yet powerful, as it combines potentially useful, self-contained building blocks (subtrees) from different parents. A common constraint is to enforce a maximum depth on offspring to prevent them from growing excessively large after crossover.

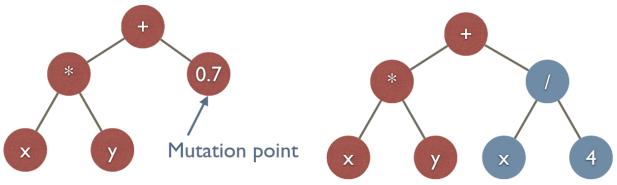
Observation: ***Non-Homologous Crossover***

Unlike one-point or uniform crossover in GAs, which align genes by position (***homologous***), subtree crossover is ***non-homologous***. The exchanged subtrees can come from different positions and depths and have different sizes and shapes. This allows for a more flexible and dynamic exchange of genetic material.

4.3.3 Mutation

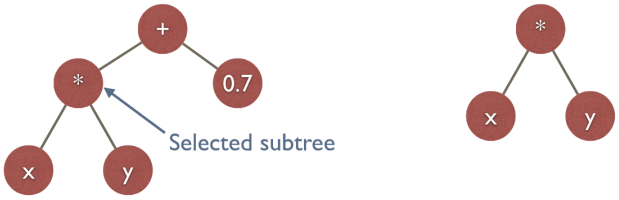
In GP, mutation involves making a small, random change to a single parent tree. There are several common mutation operators, each with a different effect:

- ***Subtree Mutation:*** A random crossover point is selected in the tree, and the entire subtree rooted at that point is replaced by a new, randomly generated tree (often using the `grow` method). This is a major mutation, capable of drastically changing the program's behavior.

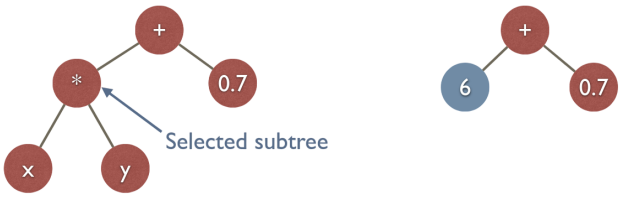


- ***Point Mutation:*** A random node in the tree is replaced by another primitive from the set. To maintain validity, a function can only be replaced by a function of the same arity, and a terminal can only be replaced by another terminal. This is a much smaller, more targeted change than subtree mutation.

- ***Hoist Mutation:*** A random subtree is selected from within the tree, and this subtree becomes the new, complete tree. This is a size-reducing operator, as it replaces the entire tree with one of its smaller parts.



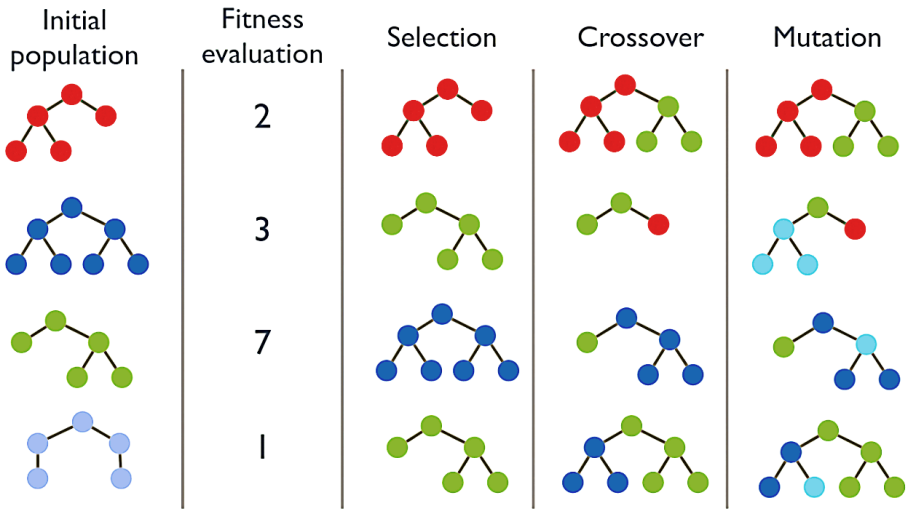
- ***Shrink Mutation:*** A randomly selected subtree is replaced by a single, randomly chosen terminal. This is another effective operator for reducing program size.



- ***Permutation Mutation:*** The arguments of a selected function are permuted. For a binary operator, this would mean swapping its two child subtrees. This is most effective for non-commutative functions.

4.3.4 The Genetic Operators in Action

Let's provides a visual overview of the main steps in a typical Genetic Programming (GP) evolutionary cycle. The **??** shows how an initial population of program trees is evaluated for fitness, followed by the selection of individuals based on their fitness values. The selected individuals then undergo genetic operations such as crossover, where subtrees are exchanged to create new offspring, and mutation, where random changes are introduced to maintain diversity. This process is repeated over successive generations to evolve better solutions.



4.4 Code Re-use: Automatically Defined Functions (ADF)

A fundamental limitation of standard Genetic Programming is that, when a useful code fragment (i.e. a subtree) is required in multiple locations within a program, the evolutionary process must independently evolve this fragment each time it is needed. To address this inefficiency and to promote modularity and code reuse, the concept of ***Automatically Defined Functions (ADFs)*** was introduced, drawing inspiration from *subroutines* in conventional programming languages.

With ADFs, an individual is no longer a single tree but a ***forest*** of trees, comprising:

- A primary **result-producing branch**, which serves as the main program tree.
- One or more **function-defining branches**, each corresponding to a distinct ADF.

Each ADF has a unique identifier (e.g., 'ADF1', 'ADF2') and a fixed arity, with arguments denoted (e.g., 'ARG1', 'ARG2'). These ADFs may be invoked as functions within the main program tree or within other ADFs, thereby enabling the construction of hierarchical and nested function calls.

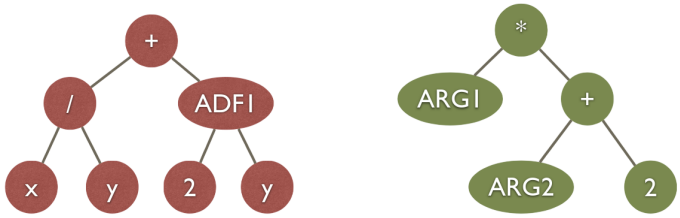


Figure 4.5: Illustration of Automatically Defined Functions (ADFs).

When employing ADFs, genetic operators are applied to all trees within the forest. For instance, crossover may occur between the main program trees of two parent individuals, between their respective ADF trees, or between an ADF tree and a main program tree. This co-evolutionary mechanism enables GP to autonomously discover and exploit useful, reusable submodules.

4.5 Bloat and Parsimony Pressure

Bloat

A common and challenging problem in GP is ***bloat***: the tendency for programs to grow in size over generations without a corresponding improvement in fitness. This happens because larger trees can often have more neutral crossover or mutation points, giving them a slight selective advantage.

One major symptom of bloat is the emergence of large **non-coding regions**, also known as ***introns***. These are sections of code that have no effect on the final output (e.g., a subtree like `(x - x)`, which always evaluates to 0, or an `IF` statement that is never taken).

Bloat is problematic because it significantly increases the computational cost of fitness evaluations, slowing down the evolutionary process.

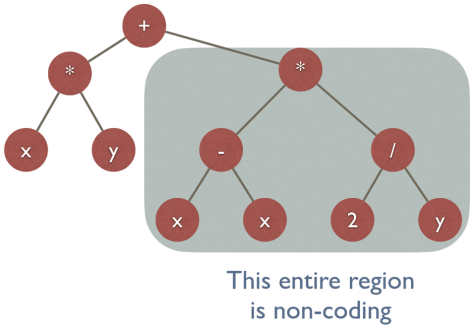


Figure 4.6: Illustration of bloat in GP.

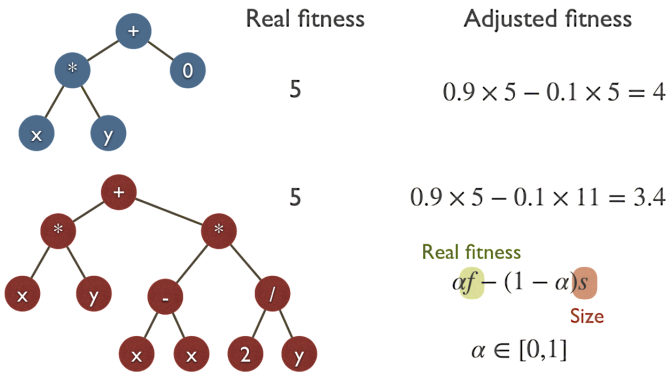
Controlling Bloat

Several methods are used to combat bloat:

1. **Strict Size/Depth Limits:** The most direct method is to impose a hard maximum limit on tree depth or the number of nodes. Offspring that exceed this limit after are discarded.
2. **Size-Reducing Operators:** Including mutation operators like `Hoist` and `Shrink` in the evolutionary process can help counteract growth.
3. **Parsimony Pressure:** This widely used technique modifies the fitness function to penalize larger trees, creating selection pressure for both correctness and simplicity. A common approach is to use a weighted combination of fitness and size:

$$f_{adj}(p) = \alpha f(p) - (1-\alpha)\,\text{size}(p)$$

Here, $f(p)$ is the original fitness of program p, $\text{size}(p)$ is its size (number of nodes), and $\alpha \in [0,1]$ is a parameter that balances the importance of fitness versus parsimony. When α is close to 1, fitness dominates; when it is smaller, parsimony is emphasized.



Tip: ***Parsimony Pressure Tuning***

The parameter α must be chosen carefully. If it is too close to 1, bloat will not be controlled. If it is too small, evolution may be stifled by penalizing necessary complexity, resulting in trivial, low-fitness solutions. Its value is often tuned empirically to balance the two objectives.

4.6 Alternative Program Representations

While tree-based representation is the canonical form of Genetic Programming, it is not the only one. Several alternative representations have been developed to address some of the challenges of tree-based GP or to better suit specific problem domains, such as evolving imperative programs or designing digital circuits. These methods often replace the explicit tree structure with a linear genotype that is either executed directly or used to generate a more complex phenotype.

4.6.1 Linear Genetic Programming (LGP)

Linear Genetic Programming (LGP) departs from the LISP-like expression trees and instead represents programs as a linear sequence of instructions, much like a program in assembly language.

Representation and Execution

In LGP, an individual is a variable-length sequence of simple instructions that operate on a set of registers. The program is executed by interpreting these instructions sequentially, and the final result is typically read from a designated output register after execution.

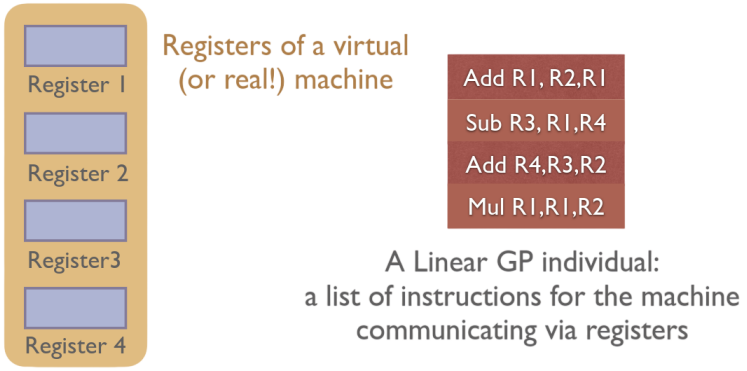


Figure 4.7: An example of a Linear GP individual.

Observation: *LGP vs. Genetic Algorithms*

At first glance, LGP's linear genotype seems very similar to that of a standard GA. However, there is a crucial distinction:

- In a GA, the genotype *is* the solution.
- In LGP, the genotype is a *program* that must be executed to produce a result. The fitness is determined by the outcome of this execution, not by the instruction sequence itself. This introduces a complex genotype-to-phenotype mapping.

Genetic Operators in LGP

Since the representation is linear, LGP can adapt standard GA operators.

- ***Crossover:*** A common approach is a two-point crossover. Since individuals can have different lengths, the crossover points in each parent are chosen independently. A segment of instructions is then swapped between the two parents.
- ***Mutation:*** Mutation can take several forms, such as changing an instruction's operator (e.g., `ADD` to `SUB`), altering a register pointer, or modifying a constant value. Macro-mutations, like inserting or deleting an entire instruction, are also used to change the program's length.

4.6.2 Cartesian Genetic Programming (CGP)

Cartesian Genetic Programming (CGP) is a form of genetic programming introduced by Julian F. Miller. In CGP, candidate programs are represented as **directed acyclic graphs** (DAGs). This representation is particularly well-suited for tasks involving multiple inputs and outputs. Although the phenotype (the expressed program) is a graph, the underlying genotype is a linear, fixed-length string of integers, similar in spirit to the representation used in LGP.

Representation and Encoding

A CGP individual consists of a two-dimensional grid of computational nodes, typically organized into a specified number of rows and columns. Each node in the grid implements a function selected from a predefined function set.

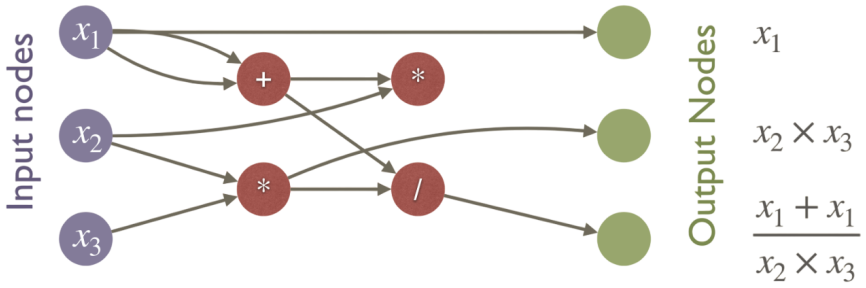


Figure 4.8: Genotype-to-phenotype mapping in CGP.

- The **genotype** is a fixed-length sequence of integers. It encodes the following information:
  1. For each node in the grid:
     - The function to be computed by the node (identified by an index into the function set).
     - The source of each input to the node, specified as the address (index) of either a program input or an output from a preceding node in the grid.
  2. For each program output:
     - The address (index) of the node whose output is to be used as the program's final output.
- Each node is typically encoded by a fixed group of integers (a *gene*), for example: `[function_id, input1_addr, input2_addr]`.
- The complete genotype is formed by concatenating the genes for all nodes, followed by the output node addresses.

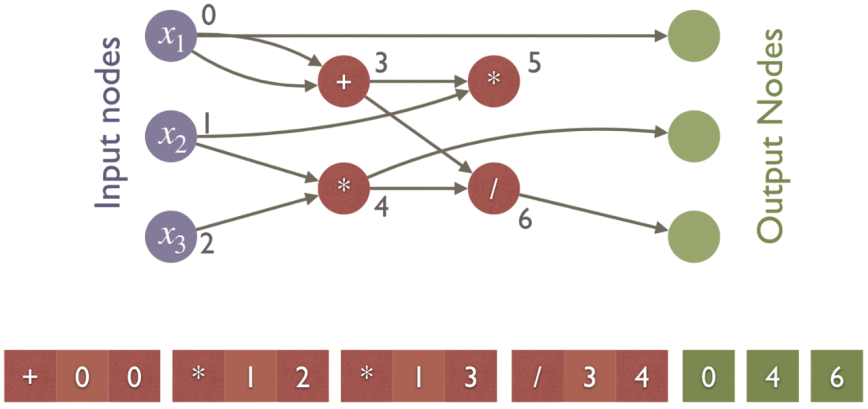


Figure 4.9: CGP individual and its genotype.

Tip: *Non-Coding Genes in CGP*

A key feature of CGP is that not all nodes in the grid need to be part of the final, active graph that connects inputs to outputs. Nodes whose outputs are not used by any subsequent active node are effectively non-coding genes (introns). This neutrality is believed to be beneficial for the evolutionary search, allowing genetic changes to accumulate without immediate phenotypic effect.

Genetic Operators in CGP

Evolution in CGP is typically driven exclusively by **mutation**. A common strategy is a $(1+\lambda)$-ES, where a parent generates λ offspring via mutation, and the fittest individual among the parent and offspring survives. Crossover is rarely used. ***Point mutation*** is the primary operator: one or more integers in the genotype are randomly changed to another valid value. This can either change a node's function or re-wire its connections.

4.6.3 Grammatical Evolution (GE)

Grammatical Evolution (GE) offers a powerful bridge between the simplicity of a linear genome and the structural complexity of high-level languages. Instead of evolving program trees directly, GE evolves a sequence of integers which, guided by a formal grammar, deterministically generates a syntactically correct program. This approach separates the genotype (the integer string) from the phenotype (the final program).

Backus-Naur Form (BNF) Grammar

The core engine of GE is a context-free grammar, specified in ***Backus-Naur Form (BNF)***. Invented by John Backus and Peter Naur for the ALGOL language, a BNF grammar formally defines a language as a set of production rules. A grammar is a quadruple $(\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$:

- ***Terminals*** ($\mathcal{T}$)***:*** The set of final (non-expandable) symbols that form the language's strings.
- ***Non-Terminals*** ($\mathcal{N}$)***:*** The set of abstract symbols that can be expanded into other symbols.
- ***Production Rules*** ($\mathcal{P}$)***:*** A set of rules mapping a non-terminal to a sequence of terminals and/or non-terminals. The pipe symbol `|` is used to denote alternative expansions.
- ***Start Symbol*** ($\mathcal{S}$)***:*** The specific non-terminal where the expansion process begins.

Example: *BNF Grammar for Sum of Integers*

Consider the following grammar from the slides:

- $\mathcal{T} = \{+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\mathcal{N} = \{S, C, D\}$
- $\mathcal{S} = S$
- $\mathcal{P}$:
  $S \to S + S \mid C$
  $C \to D \mid DC$
  $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

This grammar can generate strings representing the sum of any positive integers (e.g., "42+8+6"). The expansion process, or ***derivation***, starts with S and repeatedly applies rules to expand non-terminals until only terminals remain. For example:

$$S \to S + S \to C + S \to DC + S \to \ldots \to 42 + 8 + 6$$

This derivation can also be viewed as the construction of a syntax tree, where non-terminals are internal nodes and terminals are the leaves.

The Genotype-to-Phenotype Mapping

GE uses a linear vector of integers, called **codons**, as its genotype. This genome guides the derivation process. To generate a program (phenotype), GE performs the following steps:

1. Start the derivation string with the axiom $\mathcal{S}$.
2. Read the next codon from the genome.
3. Identify the leftmost non-terminal in the current derivation string.
4. Count the number N_r of alternative production rules available for this non-terminal.
5. Select which rule to apply using the formula: rule_index $=$ codon $\pmod{N_r}$.
6. Replace the non-terminal with the chosen expansion.
7. Repeat from step 2 until only terminals are left in the string or a termination limit is reached.

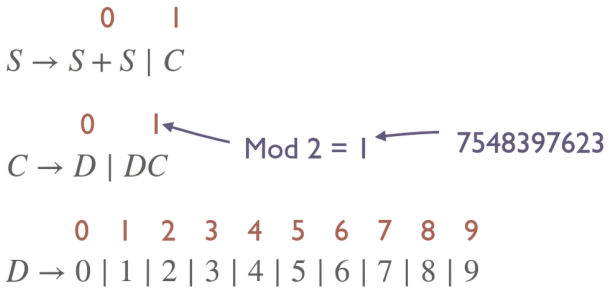


Figure 4.10: The core mapping mechanism in GE.
An integer codon is used to select a production rule via the modulo operator.

Example: ***Full Derivation Walkthrough***

Let's consider the following grammar:

$$\begin{array}{lcll} S & \to & (S\ \text{OP}\ S) \mid V \mid N & \text{(3 choices)} \\ OP & \to & + \mid \times \mid - \mid \div & \text{(4 choices)} \\ V & \to & x_1 \mid x_2 \mid x_3 & \text{(3 choices)} \\ N & \to & \text{-}DD \mid \text{-}D \mid D \mid DD & \text{(4 choices)} \\ D & \to & 0 \mid 1 \mid \ldots \mid 9 & \text{(10 choices)} \end{array}$$

and the genome:

$$[3, 5, 22, 6, 9, 2, 56, 18, 24, 1]$$

Mapping Process:

1. Start with axiom `S`.
2. Read codon **3**. For `S` (3 choices), rule is $3 \pmod 3 = 0$.
3. Read codon **5**. For `S` (3 choices), rule is $5 \pmod 3 = 2$.
4. Read codon **22**. For `N` (4 choices), rule is $22 \pmod 4 = 2$.
5. …

The process continues, always expanding the leftmost non-terminal using the next available codon. The final phenotype is the program $(5 \times x_2)$.

Note that the last three codons are not used in the derivation.

Full Derivation Chain:

$$\begin{array}{lll} \underline{S} & & \\ \to & (\underline{S}\ \text{OP}\ S) & [0] \\ \to & (\underline{N}\ \text{OP}\ S) & [2] \\ \to & (\underline{D}\ \text{OP}\ S) & [2] \\ \to & (5\ \underline{\text{OP}}\ S) & [6] \\ \to & (5 \times \underline{S}) & [1] \\ \to & (5 \times \underline{N}) & [1] \\ \to & (5 \times \underline{V}) & [1] \\ \to & (5 \times x_2) & \end{array}$$

Unique Properties and Challenges

The indirect genotype-phenotype mapping in GE leads to several distinctive characteristics:

- ***Variable-Length Genomes and Wrapping:***
  Genomes can have different lengths. If the derivation process exhausts all codons before the program is complete, GE can ”wrap around” and reuse codons from the beginning of the genome.
- ***Handling Incomplete Derivations:***
  To prevent infinite recursions (e.g. $S \rightarrow S + S \rightarrow S + S + S \rightarrow \ldots$), a maximum number of derivation steps is enforced. If a genome fails to produce a complete program within this limit, it is deemed invalid and assigned a very low fitness score.
- ***Degeneracy and Neutrality:***
  There is no one-to-one mapping between genotype and phenotype. Many different integer genomes can produce the exact same final program. This is known as degeneracy and is a source of neutral mutations (changes in the genotype that do not change the phenotype), which can be beneficial for traversing the fitness landscape.
- ***Epistasis and the Ripple Effect:***
  The function of a gene is highly dependent on the genes that came before it, as they determine the sequence of derivation steps. A single mutation early in the genome can cause a ”ripple effect,” completely changing how the rest of the codons are interpreted and leading to a radically different phenotype.

Genetic Operators for GE

Since the genotype is a linear vector of integers, GE can leverage standard GA operators.

- **Crossover:** One-point and two-point crossover are commonly used.
- **Mutation:** Randomly replaces a codon with a new integer.

More advanced, GE-specific crossover operators also exist. For example, **sensible crossover** is a one-point crossover that uses the actual effective length of each individual. Other operators include **ripple crossover**, and **homologous crossover**, which attempts to align two genomes based on their derivation history and perform crossover at points where their derivations begin to differ, aiming for a more meaningful exchange of information.

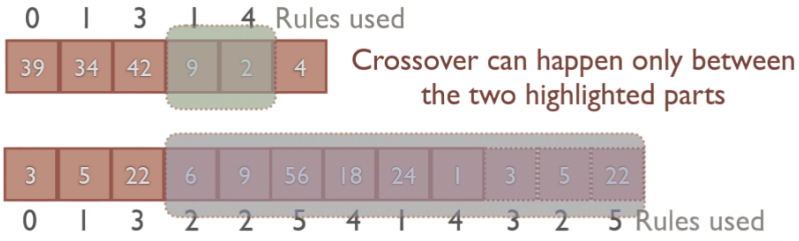


Figure 4.11: Homologous crossover in GE.

Tip: ***PonyGE2***

Grammatical evolution is available in the **PonyGE2** library:

`https://github.com/PonyGE/PonyGE2`

Tip: To install PonyGE2, you must *clone or download* the repository from GitHub.
You cannot use `pip` or Anaconda for installation.

Bibliography

[1] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, Apr. 1992. ISBN: 9780262275552. DOI: 10.7551/mitpress/1090.001.0001. URL: http://dx.doi.org/10.7551/mitpress/1090.001.0001.

[2] Maritzol Tenemaza et al. "Improving Itinerary Recommendations for Tourists Through Metaheuristic Algorithms: An Optimization Proposal". In: *IEEE Access* PP (Apr. 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.2990348.