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

June 6, 2025

`github.com/Spina02` `andreaspinelli2002@gmail.com`

Preface

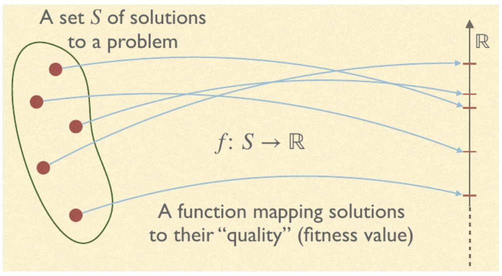
Contents

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:

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


A set S of solutions to a problem
$\mathbb{R}$
$f: S \to \mathbb{R}$
A function mapping solutions to their "quality" (fitness value)

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.

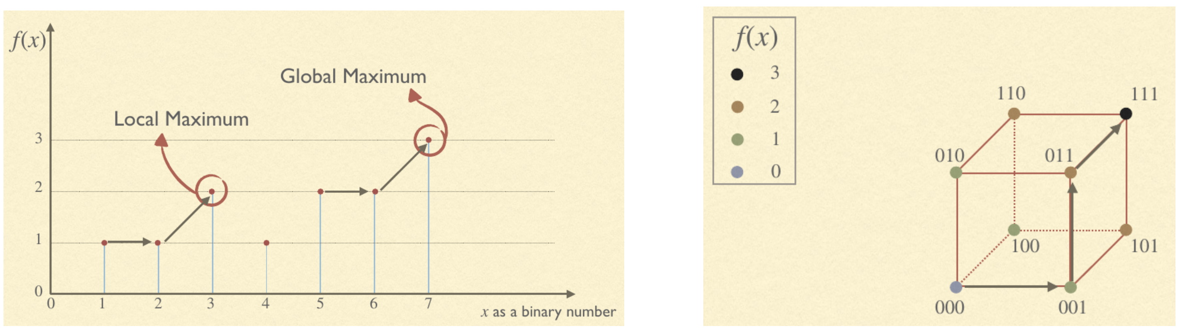


Figure 1.1: With a poor neighbourhood (left), hill climbing can get trapped in a local optimum. A richer neighbourhood (right) may eliminate local traps.

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.

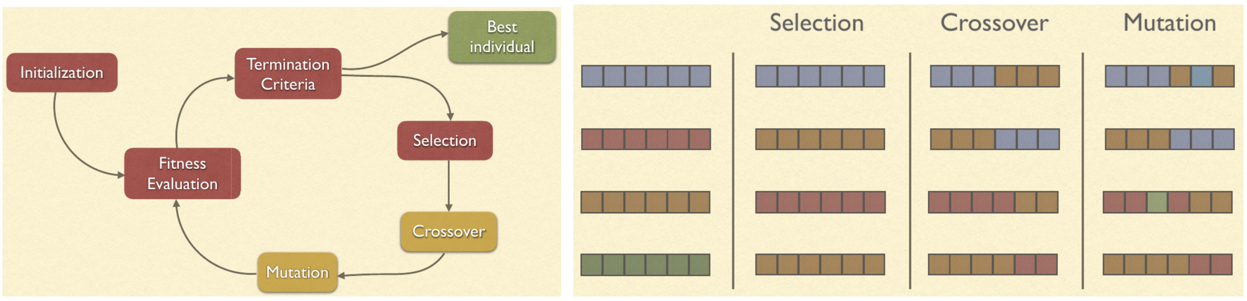


Figure 2.1: Left: Diagram showing the main components and their interactions in the evolutionary process. Right: Example of genetic operators (selection, crossover, mutation) acting on binary strings.

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:

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

  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

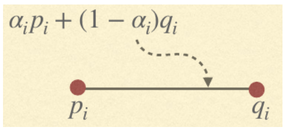
Traditional binary GAs can encode floating-point numbers using 32 or 64 binary genes, where different bit positions have varying impacts on the final value. However, real-valued GAs take a more direct approach by using floating-point genes directly and adapting the genetic operators accordingly.

Crossover

Real-valued GAs employ two main specialized crossover operators:

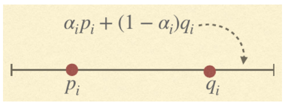
- **Intermediate recombination:** Creates offspring by taking weighted averages of the parents' genes. Given parents x_1 and x_2:

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



- **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

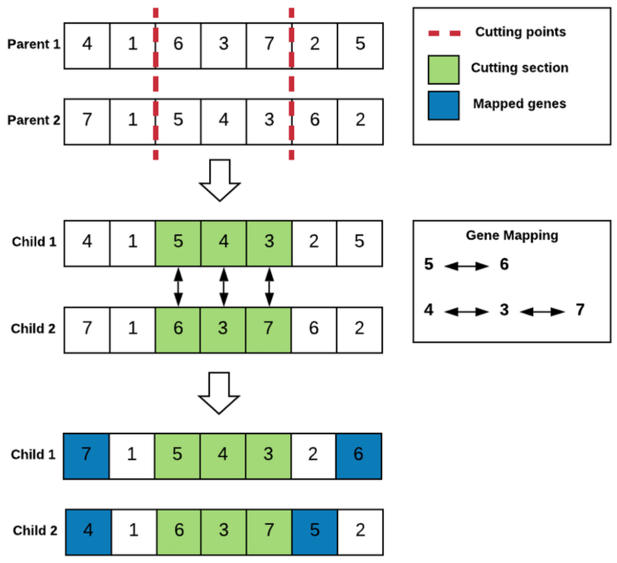


Figure 2.2: Partially Mapped Crossover [2].

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.

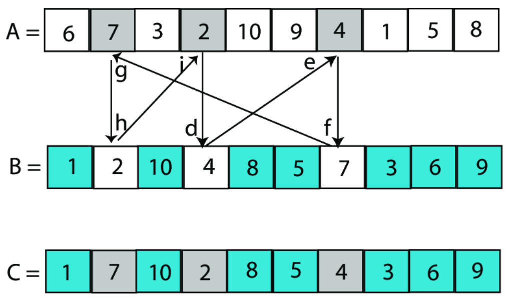


Figure 2.3: Cycle Crossover. [2]

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:

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

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

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

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):

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

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 $S \rightarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ 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.

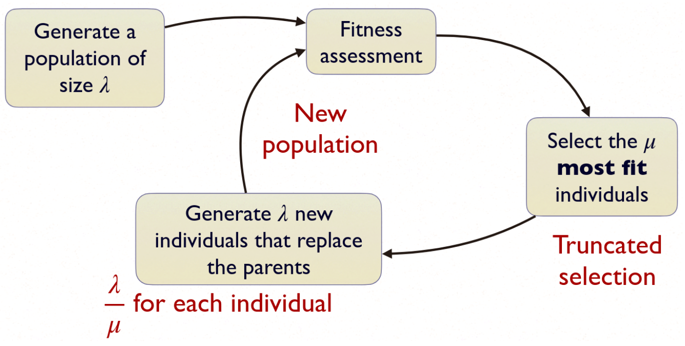

Generate a population of size λ
Fitness assessment
New population
Select the μ **most fit** individuals
Generate λ new individuals that replace the parents
Truncated selection
$\frac{\lambda}{\mu}$ for each individual

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.

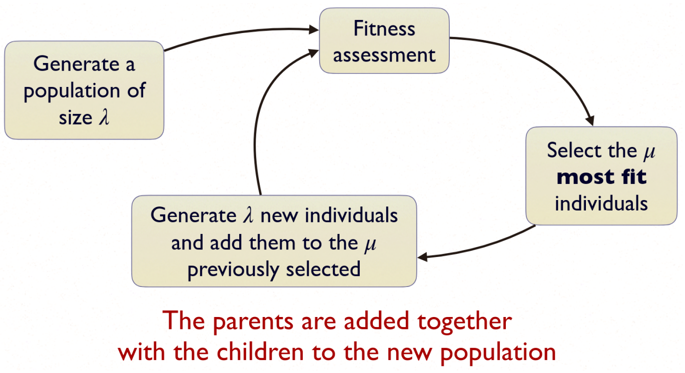
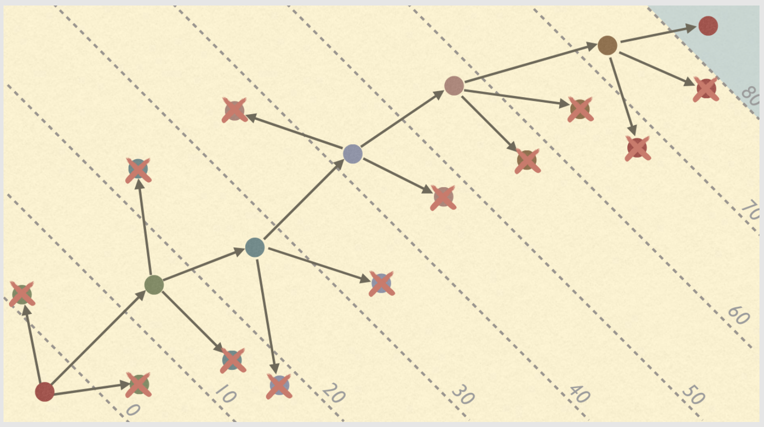

Generate a population of size λ
Fitness assessment
Select the μ **most fit** individuals
Generate λ new individuals and add them to the μ previously selected
The parents are added together with the children to the new population

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.

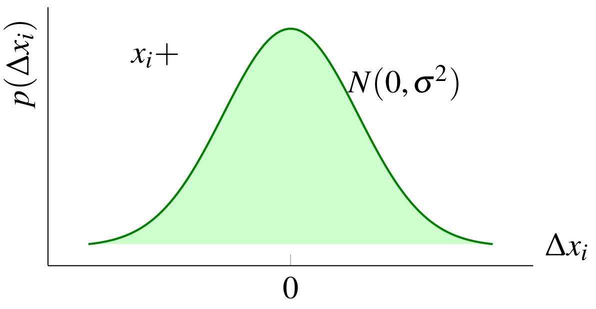


Figure 3.3: Gaussian mutation adds a random value drawn from a normal distribution to x_i.

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:
$$x'_j = x_{p_j, j}, \quad p_j \in \{1, \dots, \rho\}$$
Strategy parameters $\mathbf{s}$ can be recombined similarly.
- **Intermediate Recombination**: Each component j is the average of corresponding parent values:
$$x'_j = \frac{1}{\rho} \sum_{i=1}^{\rho} x_{i,j}$$
This creates offspring between parents. 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 $\mathbf{x}' = (x'_1, \dots, x'_n)$ from parents $\mathbf{p}_1$ and $\mathbf{p}_2$: For each $j \in \{1, \dots, n\}$, 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 $j \in \{1, \dots, n\}$, $x'_j = (p_{1,j} + p_{2,j})/2$.

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.

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.