UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Global and Multi-Objective Optimisation

Lecturer:
Prof. Luca Manzoni

Author:
Andrea Spinelli

July 23, 2025

github.com/Spina02 andreaspinelli2002@gmail.com

Preface

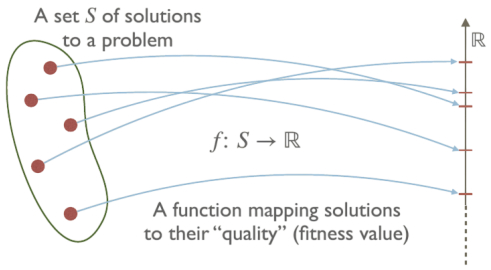
Contents

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.

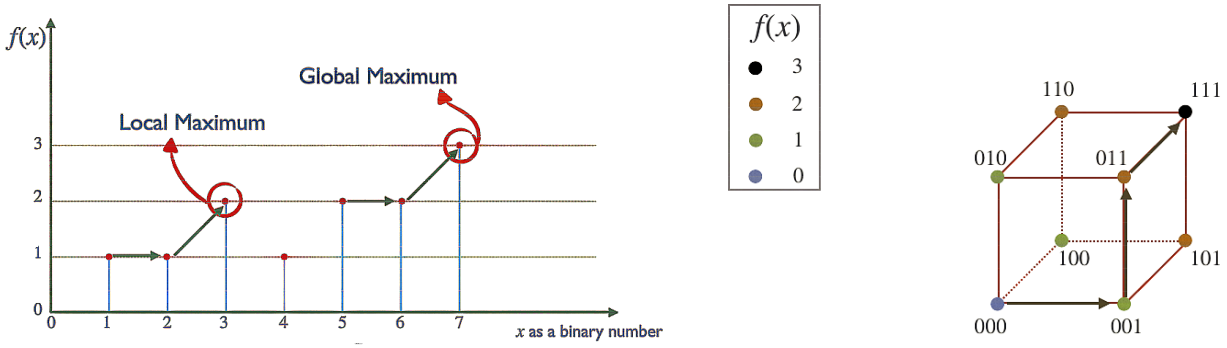


Figure 1.1: With a poor neighbourhood (left), hill climbing can get trapped in a local optimum. A richer neighbourhood (right) may eliminate local traps.

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

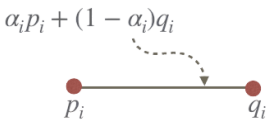
Traditional binary GAs can encode floating-point numbers using 32 or 64 binary genes, where different bit positions have varying impacts on the final value. However, real-valued GAs take a more direct approach by using floating-point genes directly and adapting the genetic operators accordingly.

Crossover

Real-valued GAs employ two main specialized crossover operators:

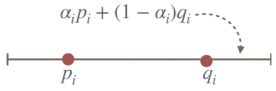
- **Intermediate recombination:** Creates offspring by taking weighted averages of the parents' genes. Given parents x_1 and x_2:

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



- **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

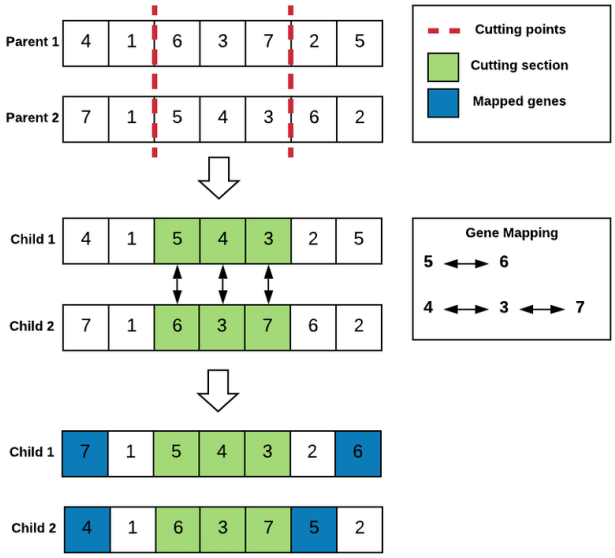


Figure 2.2: Partially Mapped Crossover [3].

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.

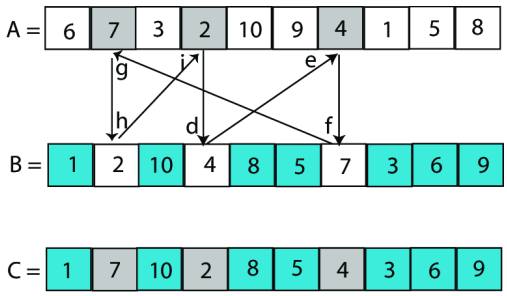


Figure 2.3: Cycle Crossover. [3]

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.

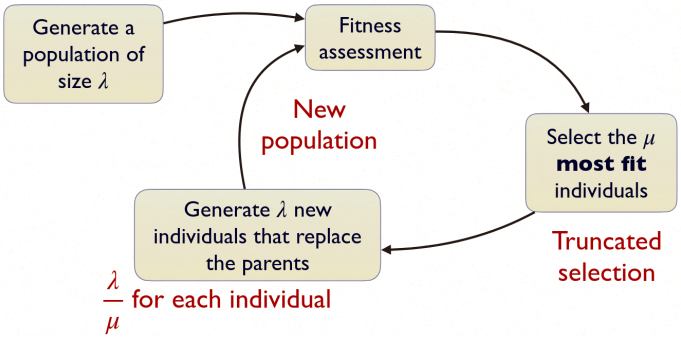


Figure 3.1: The lifecycle of a $(\mu, \lambda) - ES$

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.

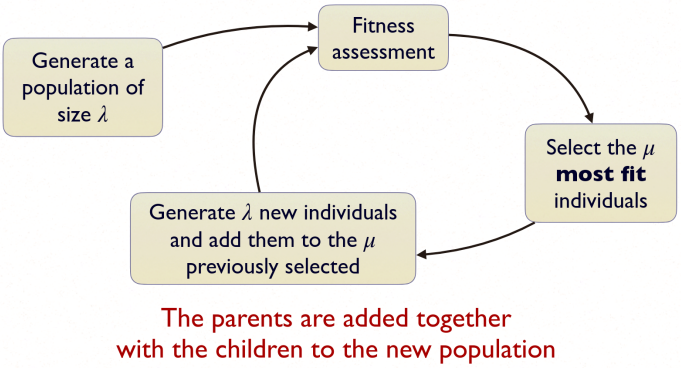
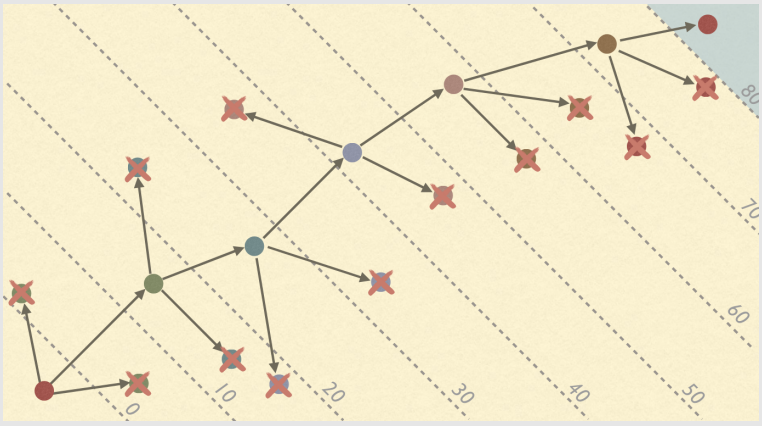


Figure 3.2: The lifecycle of a $(\mu + \lambda) - ES$.

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.

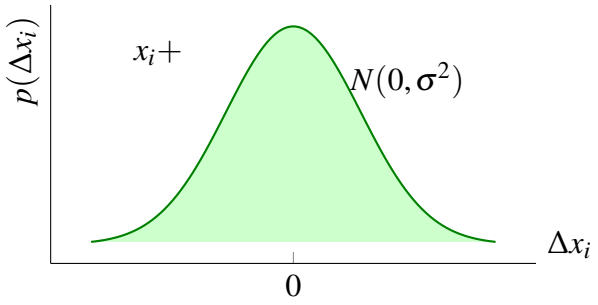


Figure 3.3: Gaussian mutation adds a random value drawn from a normal distribution to x_i.

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.

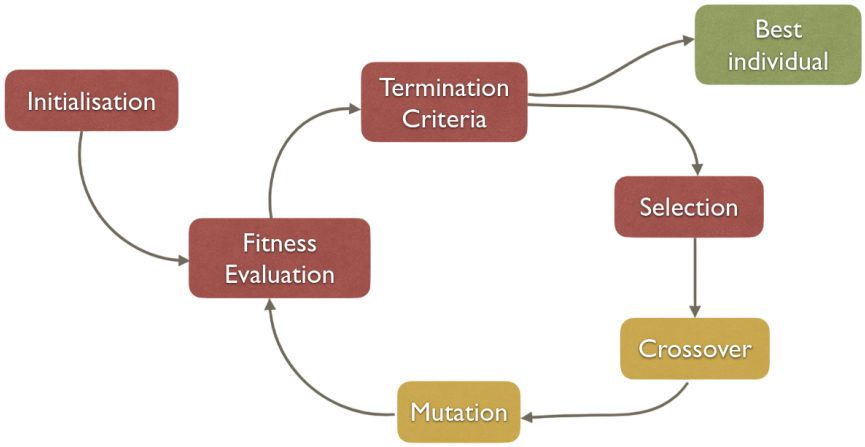


Figure 4.1: The general evolutionary cycle in Genetic Programming, which closely mirrors the cycle of a standard GA but operates on programs instead of fixed-length genotypes.

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.

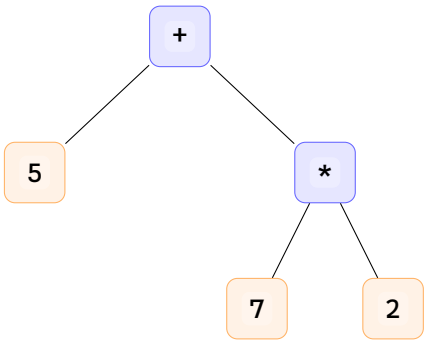


Figure 4.2: The expression $5+(7\times 2)$ as a syntax tree. Internal nodes represent functions or operators (`+` , `*`), while leaf nodes are terminals (5, 7, 2).

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.

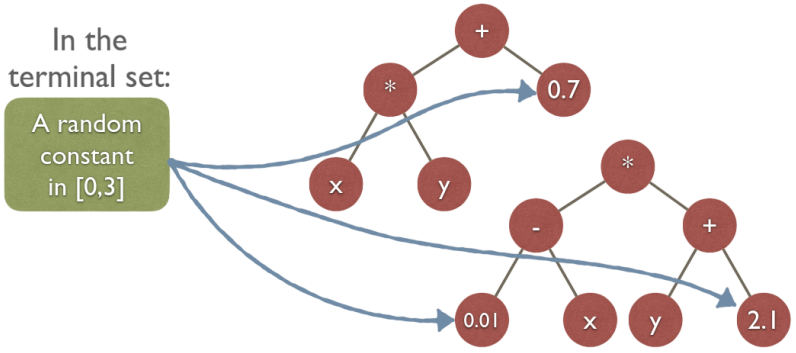


Figure 4.3: The *Ephemeral* terminal generates a random constant in $[0, 3]$. Each instance holds a different random value.

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.

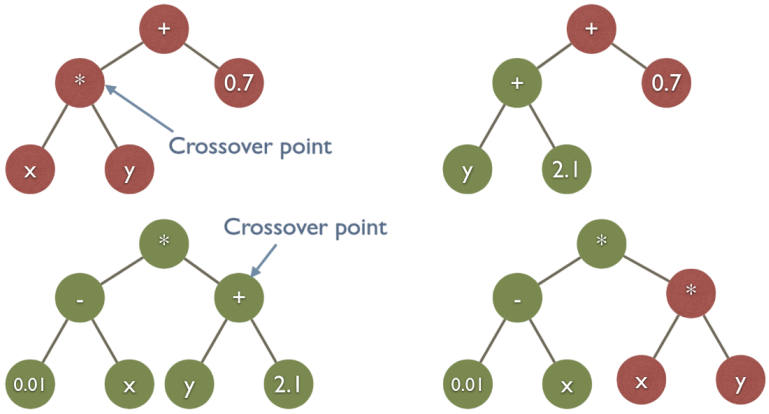


Figure 4.4: Subtree Crossover.

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.

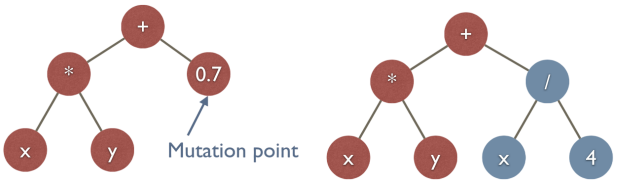
Observation: ***Non-Homologous Crossover***

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

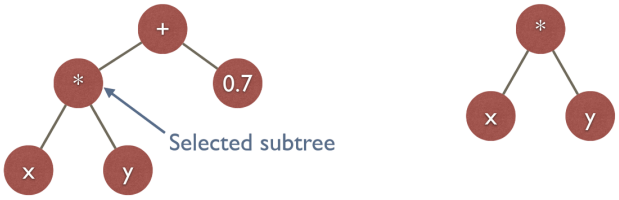
In GP, mutation involves making a small, random change to a single parent tree. There are several common mutation operators, each with a different effect:

- ***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.

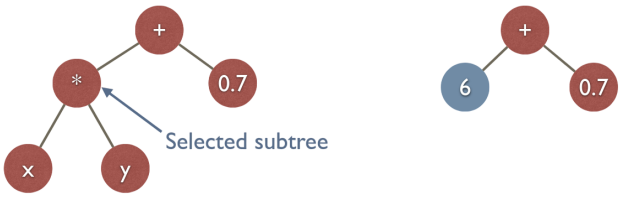


- ***Point Mutation:*** A random node in the tree is replaced by another primitive from the set. To maintain validity, a function can only be replaced by a function of the same arity, and a terminal can only be replaced by another terminal. This is a much smaller, more targeted change than subtree mutation.

- ***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.



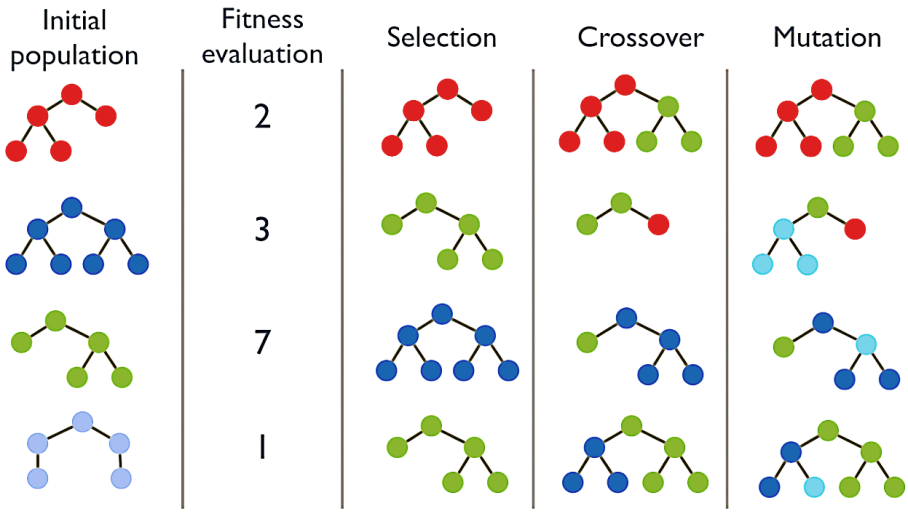
- ***Shrink Mutation:*** A randomly selected subtree is replaced by a single, randomly chosen terminal. This is another effective operator for reducing program size.



- ***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.

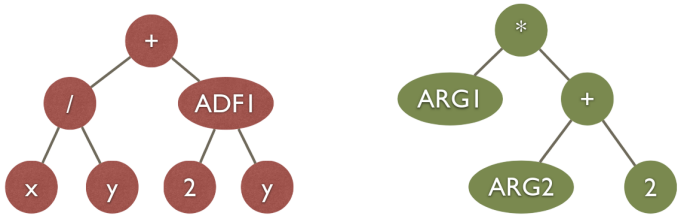


Figure 4.5: Illustration of Automatically Defined Functions (ADFs).

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.

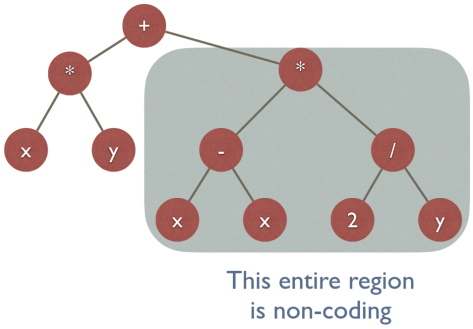


Figure 4.6: Illustration of bloat in GP.

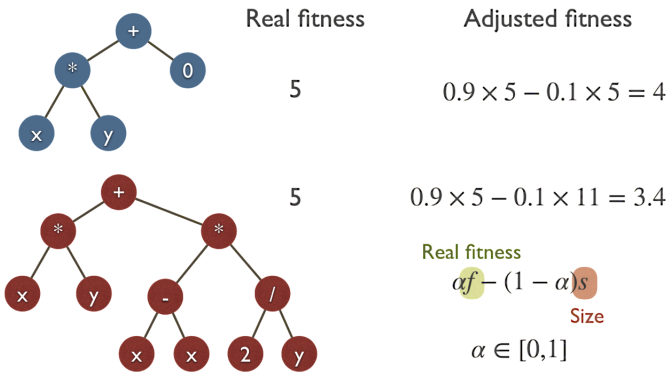
Controlling Bloat

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.

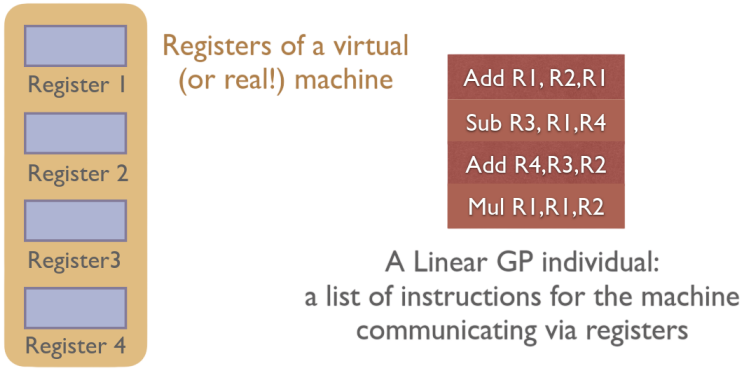


Figure 4.7: An example of a Linear GP individual.

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.

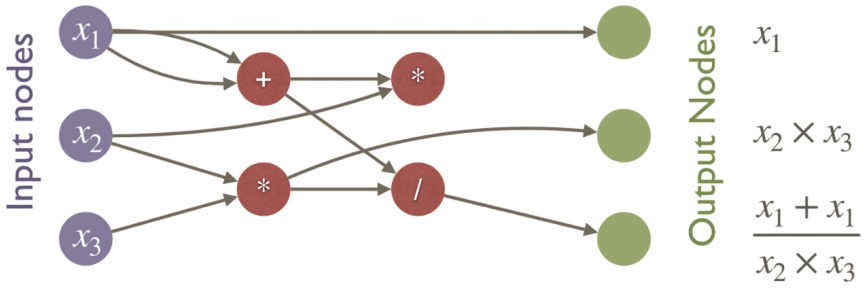


Figure 4.8: Genotype-to-phenotype mapping in CGP.

- 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.

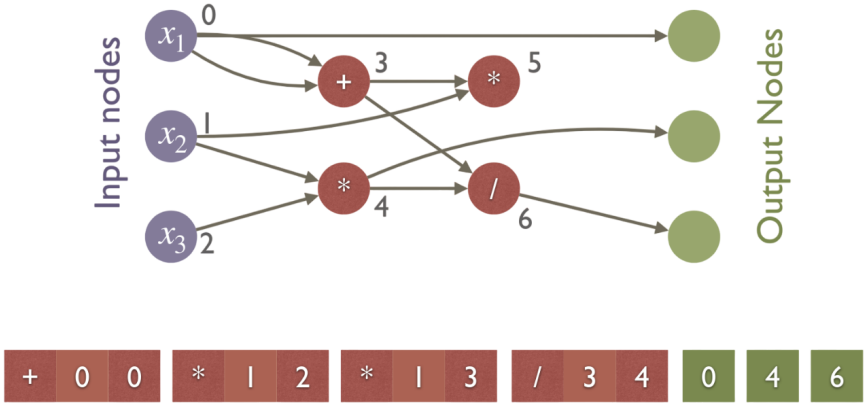


Figure 4.9: CGP individual and its genotype.

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.

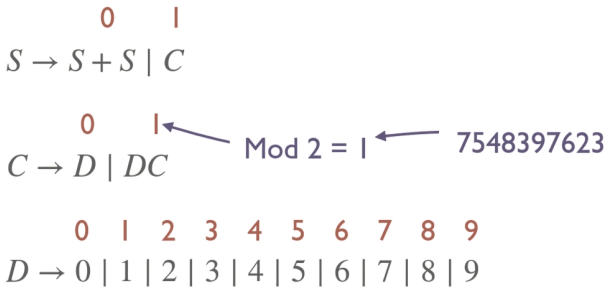


Figure 4.10: The core mapping mechanism in GE.
An integer codon is used to select a production rule via the modulo operator.

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.

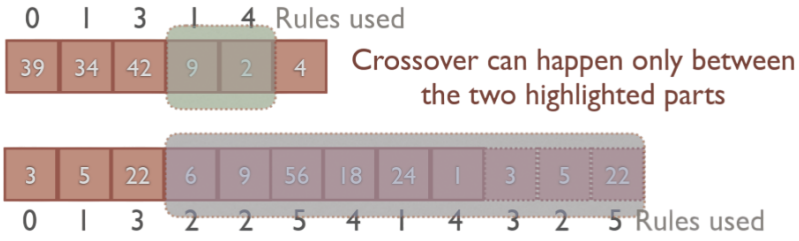


Figure 4.11: Homologous crossover in GE.

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.

5 Other Bio-Inspired Metaheuristics

Beyond Genetic Algorithms, Evolution Strategies, and Genetic Programming, the field of bio-inspired computation offers a rich landscape of other powerful metaheuristics. This chapter introduces three prominent algorithms: Differential Evolution, a simple yet powerful method for real-valued optimization; Particle Swarm Optimization, which models the collective intelligence of social swarms; and Ant Colony Optimization, which mimics the foraging behavior of ants via indirect communication.

5.1 Differential Evolution (DE)

Differential Evolution (DE), introduced by Storn and Price in 1997, is a population-based stochastic optimization algorithm designed primarily for real-valued optimization problems in $\mathbb{R}^m$. While it follows a general evolutionary pattern, DE distinguishes itself from canonical Genetic Algorithms in two fundamental ways:

1. **Generation of New Solutions**: DE creates new candidate solutions by combining existing ones in a unique way, using vector differences to guide the search direction and step size. This ”differential mutation” is the hallmark of the algorithm.
2. **Selection Process**: The selection mechanism is typically a simple one-to-one competition, where a newly created ”trial” vector replaces its parent in the next generation only if it has better or equal fitness.

These characteristics make DE a remarkably simple, robust, and effective algorithm for continuous optimization tasks.

5.1.1 The DE/rand/1 Algorithm

The most fundamental DE variant is known as `DE/rand/1`. The algorithm iterates through each individual in the population (the ”target vector”) and applies differential mutation, crossover, and selection to generate a corresponding individual for the next generation’s population.

Differential Mutation

For each target vector $\mathbf{x}_i$ in the current population, a ”donor” vector $\mathbf{v}_i$ is created. This is the core of DE’s search mechanism. The process is as follows:

1. Select three other vectors (a, b, and c) at random from the current population, ensuring they are distinct from each other and from the target vector x_i.
2. Compute the ***donor vector*** v_i using the formula:

$$v_i = a + F \cdot (b - c)$$

where $F \in [0,2]$ is a crucial hyperparameter called ***mutation factor*** or ***differential weight***. It scales the difference vector $(b-c)$, controlling the amplification of the differential variation.

Observation: *The Role of the Difference Vector*

The vector difference $(b - c)$ represents a random direction and magnitude derived from the population's current diversity. By adding this scaled difference to another vector a, DE creates a new trial point, effectively "perturbing" an existing solution with a vector that reflects the population's current spatial distribution.

Binomial Crossover

After creating the donor vector $\mathbf{v}_i$, it is combined with the original target vector $\mathbf{x}_i$ to create a ***trial vector*** $\mathbf{u}_i$. This step, analogous to crossover in GAs, introduces genetic material from the parent into the mutated vector. In DE, this is typically done via ***binomial crossover***: For each coordinate $j \in \{1, \ldots, m\}$ of the trial vector, a new value is chosen:

$$u_{i,j} = \begin{cases} v_{i,j} & \text{if } \text{rnd}_{i,j} \leq p_{CR} \text{ or } j = j_{rand} \\ x_{i,j} & \text{otherwise} \end{cases}$$

Here, $p_{CR} \in [0, 1]$ is the ***crossover probability***, $\text{rnd}_{i,j}$ is a uniform random number in $[0, 1]$ drawn for each component, and j_{rand} is a randomly chosen index from $\{1, \ldots, m\}$. The $j = j_{rand}$ condition ensures that the trial vector $\mathbf{u}_i$ receives at least one component from the donor vector $\mathbf{v}_i$.

Selection

The final step is a one-to-one competition between the trial vector $\mathbf{u}_i$ and its parent target vector $\mathbf{x}_i$. The winner, which is the vector with the better fitness, survives to become part of the population for the next generation. For a minimization problem:

$$\mathbf{x}_i^{\text{next gen}} = \begin{cases} \mathbf{u}_i & \text{if } f(\mathbf{u}_i) \leq f(\mathbf{x}_i) \\ \mathbf{x}_i & \text{otherwise} \end{cases}$$

This entire process (mutation, crossover, selection) is repeated for every individual in the population to form the next generation.

Graphical Representation

In DE, creating a trial vector can be visualized as vector arithmetic in the search space:

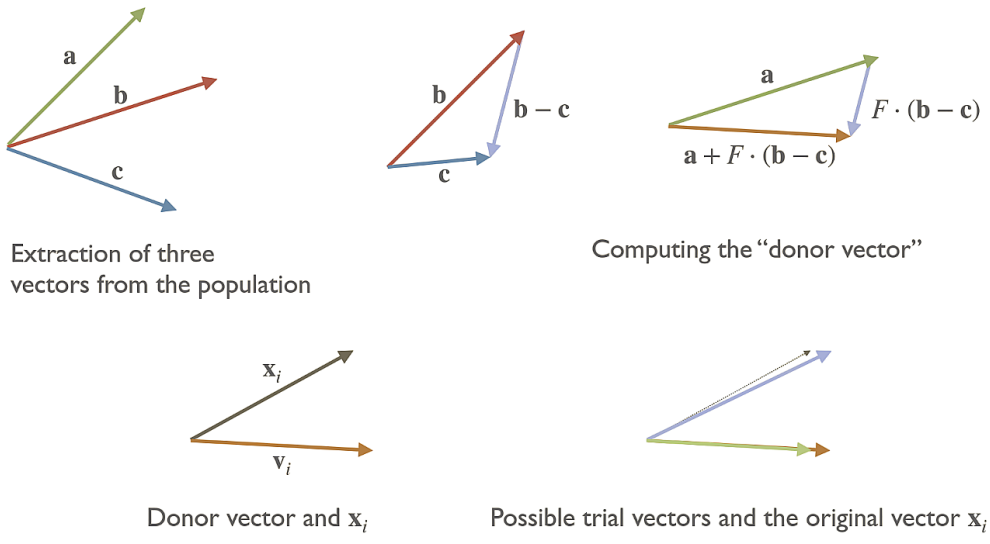


Figure 5.1: A graphical representation of the DE operators.

5.1.2 DE Variants and Taxonomy

Taxonomy of DE Strategies

The `DE/rand/1` scheme is just one of many possible DE strategies. A standard taxonomy, `DE/x/y`, is used to describe them:

- `x` specifies how the base vector for the mutation is chosen (e.g., `rand` for a random vector, `best` for the best individual in the population).
- `y` is the number of difference vectors used.

The most common variants alter the differential mutation formula to change the balance between exploration and exploitation.

Advanced Concept: *Common DE Mutation Strategies*

Here are some popular mutation strategies beyond the basic `DE/rand/1`:

- **DE/best/1**: Uses the best individual found so far, $\mathbf{x}_{\text{best}}$, as the base vector. This increases exploitation and convergence speed, but may lead to premature convergence.

$$v_i = \mathbf{x}_{\text{best}} + F \cdot (\mathbf{b} - \mathbf{c})$$

- **DE/current-to-best/1**: A hybrid strategy that includes both the current individual and the best individual, providing a good balance.

$$v_i = \mathbf{x}_i + F \cdot (\mathbf{x}_{\text{best}} - \mathbf{x}_i) + F \cdot (\mathbf{b} - \mathbf{c})$$

- **DE/rand/2**: Uses two different difference vectors, increasing the potential for exploration.

$$v_i = \mathbf{a} + F \cdot (\mathbf{b} - \mathbf{c}) + F \cdot (\mathbf{d} - \mathbf{e})$$

- **DE/rand-to-best/1**: Perturbs a random vector with a vector pointing towards the best solution.

$$v_i = \mathbf{a} + F \cdot (\mathbf{x}_{\text{best}} - \mathbf{a}) + F \cdot (\mathbf{b} - \mathbf{c})$$

Adaptive Differential Evolution (JADE)

JADE (Adaptive DE with Optional External Archive) is a powerful adaptive variant of Differential Evolution, introduced in 2009 by Zhang and Sanderson [4]. JADE enhances the standard DE algorithm with several key innovations:

- **New Mutation Strategy:** JADE introduces the `DE/current-to-pbest` mutation scheme, which improves the balance between exploration and exploitation by guiding individuals toward the best solutions found so far.
- **External Archive:** An external archive of sub-optimal (recently replaced) solutions is maintained. This archive is used during mutation to increase population diversity and help escape local optima.
- **Dynamic Hyper-parameter Update:** JADE adaptively updates its control parameters (F and p_{CR}) during the run, allowing the algorithm to self-tune and respond to the problem landscape.

5.2 Particle Swarm Optimization (PSO)

5.2.1 Introduction

Particle Swarm Optimization (PSO) is a population-based metaheuristic belonging to the family of ***swarm intelligence*** algorithms. Swarm intelligence is inspired by the collective behavior of decentralized, self-organized systems, such as flocks of birds, schools of fish, or colonies of ants. The core idea is that complex, global, and intelligent behavior can emerge from the interactions of many simple agents, each with very limited capabilities and no centralized control. These agents follow simple rules, and their collective action solves complex problems. In PSO, the individuals are called ***particles*** and the population is called a ***swarm***.

5.2.2 The PSO Algorithm

PSO models a swarm moving through an m-dimensional problem search space. Each particle represents a candidate solution and has two main properties:

- a ***position*** $\mathbf{x}_i(t)$, which is the point in the search space the particle currently occupies.
- a ***velocity*** $\mathbf{v}_i(t)$, which determines its direction and speed of movement.

At each discrete time step t, a particle's position is updated based on its velocity:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1)$$

The Velocity Update Rule

The velocity update is the heart of the PSO algorithm. It models the social and cognitive forces that guide a particle's movement. The new velocity $\mathbf{v}_i(t+1)$ is a combination of three components:

1. ***Inertia:*** The tendency of the particle to continue moving in its current direction.
2. ***Cognitive Attraction:*** The particle's memory of its own personal best position found so far, denoted b_i. This represents the particle's individual experience.
3. ***Social Attraction:*** The attraction towards the global best position found so far by any particle in the entire swarm, denoted g. This represents the collective experience of the group.

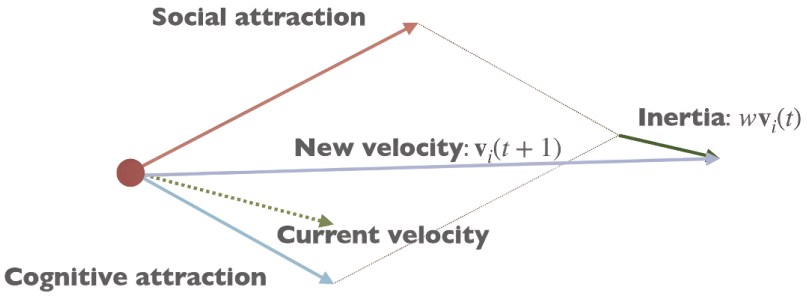


Figure 5.2: The three components influencing a particle's new velocity.

The full velocity update formula combines these influences ($\odot$ denotes the element-wise product):

$$\mathbf{v}_i(t+1) = w \cdot \mathbf{v}_i(t) + c_c \cdot \mathbf{r}_1 \odot (\mathbf{b}_i - \mathbf{x}_i(t)) + c_s \cdot \mathbf{r}_2 \odot (\mathbf{g} - \mathbf{x}_i(t))$$

Where:

- w is the ***inertia weight***, balancing global and local exploration.
- c_c and c_s are the ***cognitive*** and ***social factors***, controlling attraction to personal and global best.
- $\mathbf{r}_1, \mathbf{r}_2$ are vectors of random numbers in $[0,1]^m$ that introduce stochasticity.

Tip: ***PSO Parameters***

The algorithm's behavior is sensitive to its parameters. Commonly used default values have been established through empirical studies:

- **Cognitive and Social Factors**: c_c and c_s are often both set to approximately 1.49445.
- **Inertia Weight**: w is often decreased linearly during the run, for example from 0.9 to 0.4. A higher initial value encourages global exploration, while a lower final value allows for fine-tuning around the best-found solutions.

Boundary Conditions

When a particle moves beyond the limits of the search space, it is necessary to apply a boundary handling strategy to determine how its position and velocity should be managed.

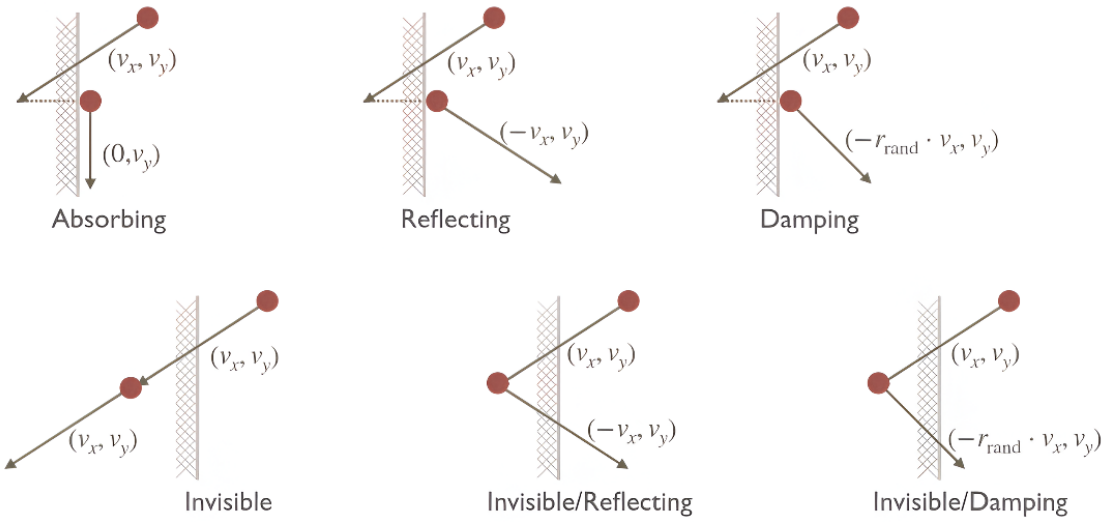


Figure 5.3: Boundary handling strategies.

There are several widely used approaches for dealing with particles that cross the boundaries:

- **Absorbing:** The particle is halted at the boundary, and its velocity component perpendicular to the boundary is set to zero.
- **Reflecting:** The particle rebounds off the boundary, reversing the relevant component of its velocity.
- **Damping:** Similar to reflecting, but the velocity is also multiplied by a damping factor r_{damp}, which reduces its magnitude.
- **Invisible:** The particle is permitted to move outside the boundary, but its position is not evaluated or updated until it returns to the feasible region.
- **Invisible/Reflecting:** The particle remains invisible while outside the boundary, but if it tries to re-enter, it is reflected back.
- **Invisible/Damping:** The particle is invisible outside the boundary, and when it re-enters, its velocity is damped.

The choice of boundary handling strategy can have a substantial impact on how the swarm explores the search space and converges to solutions.

5.3 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) is another major paradigm of swarm intelligence, primarily used to solve combinatorial optimization problems on graphs (e.g., the Traveling Salesperson Problem). The core inspiration for ACO is the foraging behavior of ants, which exhibit complex collective problem-solving capabilities despite being individually simple.

This behavior is coordinated through ***stigmergy***, a mechanism of indirect communication where individuals interact by modifying their local environment. Ants deposit a chemical substance called a **pheromone** as they travel. Other ants can sense this pheromone and are more likely to follow paths with higher concentrations. Because pheromone evaporates over time, shorter paths, which can be traversed more frequently in a given time period, accumulate stronger pheromone trails. This creates a feedback loop that guides the colony to find the shortest paths between nest and food.

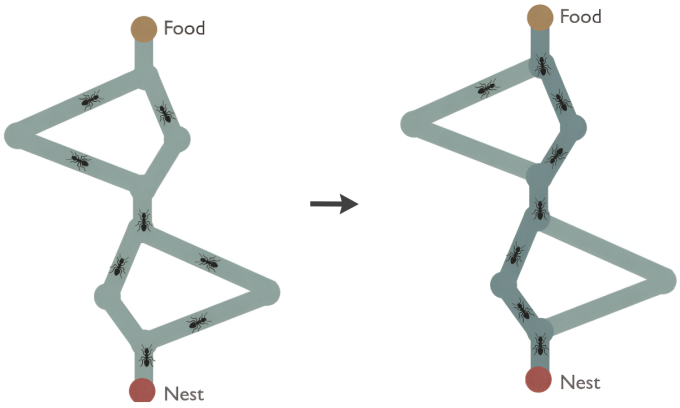


Figure 5.4: The Double Bridge Experiment, a classic illustration of stigmergy. Ants initially explore both paths randomly. Because the shorter path is traversed more quickly, its pheromone trail is reinforced faster than the longer path's trail, eventually attracting the entire colony.

5.3.1 The ACO Algorithm for the TSP

Let's illustrate ACO with its most famous application: the Traveling Salesperson Problem (TSP). The goal is to find the shortest Hamiltonian circuit in a complete graph of m cities.

The ACO algorithm follows a general cycle:

1. **Initialize** pheromone trails on the graph edges.
2. **Construct Solutions**: A colony of artificial ants construct tours (solutions) probabilistically.
3. **Update Pheromones**: Pheromone trails are updated based on quality of the solutions found.
4. **Repeat** until a termination criterion is met.

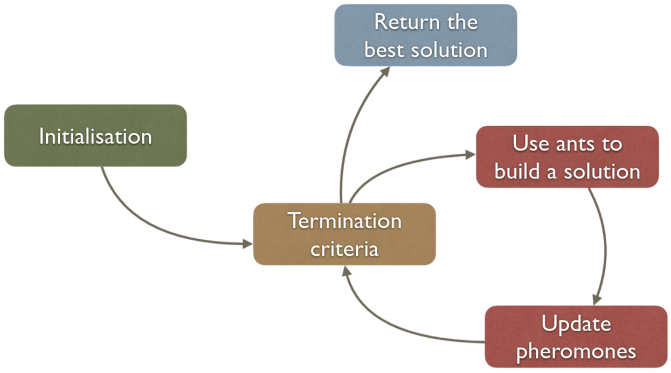


Figure 5.5: The ACO cycle.

Solution Construction

Each ant begins at a randomly selected city and incrementally constructs a tour by selecting the next city to visit from the set of unvisited cities, denoted N_i^k for ant k currently at city i. The selection of the next city $j \in N_i^k$ is governed by a probabilistic rule that incorporates two factors:

- ***Pheromone Trails*** (τ_{ij}): The current amount of pheromone on edge (i, j), representing the collective learning of the colony.
- ***Heuristic Information*** (η_{ij}): A problem-specific value, which for the TSP is typically $\eta_{ij} = 1/d_{ij}$, where d_{ij} is the distance between cities i and j.

The probability that ant k in city i moves to city j at time t is defined as:

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}(t)]^\alpha [\eta_{il}]^\beta}, \quad j \in N_i^k$$

where α and β are parameters that determine the relative importance of pheromone trails and heuristic information, respectively.

Pheromone Update

Once all ants have completed their tours, the pheromone levels on all edges are updated according to two mechanisms:

1. **Evaporation:** Each pheromone value is decreased by a factor $\rho \in [0, 1]$, known as the evaporation rate.
2. **Deposition:** Each ant k deposits an amount of pheromone $\Delta\tau_{ij}^k$ on every edge (i, j) it traversed in its tour. Typically, $\Delta\tau_{ij}^k = 1/L_k$ if ant k used edge (i, j) in its tour, where L_k is the total length of the tour constructed by ant k; otherwise, $\Delta\tau_{ij}^k = 0$.

The pheromone update rule for each edge (i, j) is given by:

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{n} \Delta\tau_{ij}^k$$

Edges that are part of better solutions, specifically those that contribute to shorter tours, receive larger amounts of pheromone during the update phase. Additionally, edges that are included in the tours of more ants accumulate even more pheromone over time. As a result, these edges become increasingly attractive to subsequent ants constructing their own solutions. The desirability of an edge, as reflected by its pheromone level, directly influences the probability that it will be selected in the future. This mechanism allows the algorithm to reinforce promising regions of the search space, gradually biasing the collective search toward optimal or near-optimal solutions.

6 Geometric Semantic GP

Traditional Genetic Programming operators, such as subtree crossover and mutation, operate on the *syntactic* structure of program trees. While effective, this approach provides little to no direct control over the *behavioral* or *semantic* change induced by the operators. An operator might cause a drastic change in program output or no change at all (in the case of introns), making the search process somewhat undirected.

Geometric Semantic Genetic Programming (GSGP) proposes a radical shift in perspective. It defines genetic operators that work directly on the semantics of the programs, treating them as points in a geometric space. By doing so, it allows for a fine-grained control over the search process, ensuring that offspring are behaviorally located "between" their parents (for crossover) or within a controlled distance of their parent (for mutation).

6.1 Geometric Operators on Metric Spaces

The foundation of GSGP lies in defining the space of solutions as a ***metric space***. A metric space is a set S equipped with a ***distance function*** (or metric) $d : S \times S \to \mathbb{R}^+$, which satisfies four key properties for any $x, y, z \in S$:

- **Identity:** $d(x,y) = 0 \iff x = y$ The distance is zero if and only if the points are identical.
- **Symmetry:** $d(x,y) = d(y,x)$ The distance from x to y is the same as from y to x.
- **Non-negativity:** $d(x,y) \geq 0$ Distance is always non-negative.
- **Triangular Inequality:** $d(x,z) \leq d(x,y) + d(y,z)$ The shortest path is the direct one.

Within a metric space, we can define geometric concepts like segments and balls:

- A ***segment*** $S(x,y)$ between two points x and y is the set of all points z for which holds:

$$S(x,y) = \{z \in S \mid d(x,z) + d(z,y) = d(x,y)\}$$

- A ***ball*** $B(x,r)$ of radius r centered in x is the set of all points y whose distance from x is at most r:

$$B(x,r) = \{y \in S \mid d(x,y) \leq r\}$$

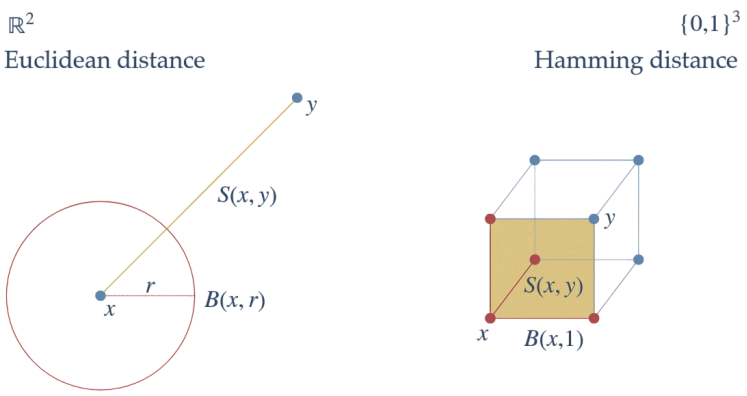


Figure 6.1: Segments and balls in different metric spaces (Euclidean, ($\mathbb{R}^2$) and Hamming, $\{0,1\}^3$)

6.1.1 Geometric Crossover and Mutation

Using these definitions, we can formally characterize genetic operators:

- A ***geometric crossover*** operator, when applied to parents x and y, must produce an offspring z that lies on the segment between them: $z \in S(x,y)$.
- A ***geometric mutation*** operator, when applied to a parent x, must produce an offspring y that lies within a ball of a defined radius r around it: $y \in B(x,r)$.

Observation: ***Exploration vs. Exploitation***

This geometric framework provides a clear distinction between the roles of operators.

- **Crossover is for Exploitation:** A geometric crossover operator generates offspring only within the bounds defined by the parents. It cannot create anything truly new, but rather explores the space "between" existing solutions. Over generations, it is confined to the convex hull of the initial population.
- **Mutation is for Exploration:** A geometric mutation operator is capable of producing offspring outside the segment of the parents, effectively enlarging the search space and allowing the algorithm to escape the initial population's convex hull.

6.1.2 Are Standard GP Operators Geometric?

Many operators in GAs are naturally geometric. For instance, one-point crossover on binary strings with Hamming distance is a geometric crossover. However, this is **not the case for standard GP**.

The standard subtree crossover in GP is ***not geometric***. A simple counterexample is crossing a tree T with itself. In a metric space, the segment $S(T,T)$ contains only the point T itself. Yet, applying subtree crossover to two identical copies of T can produce a different tree T', meaning $T' \notin S(T,T)$. This violates the definition of a geometric crossover.

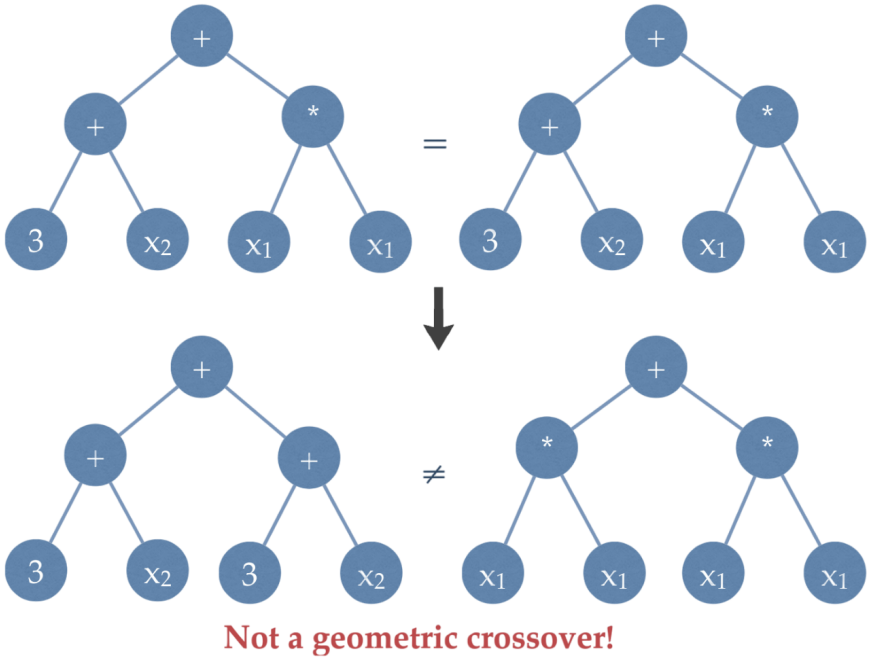
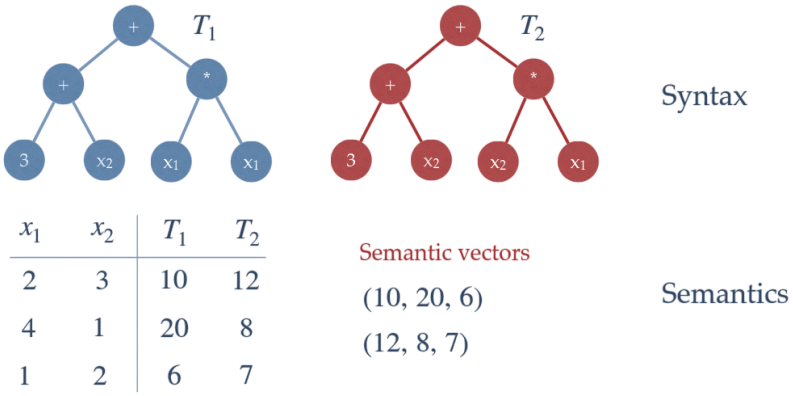


Figure 6.2: An example demonstrating that standard GP subtree crossover is not geometric. Crossing a tree with itself produces a new tree, which is impossible for a geometric operator.

6.2 Geometric Semantic Operators

Standard GP operators act on the ***syntactic space*** (all possible program trees), but what matters is the ***semantic space*** (all possible program behaviors). A program's ***semantics*** are its output vector on a given set of fitness cases (inputs).



x_1	x_2	T_1	T_2
2	3	10	12
4	1	20	8
1	2	6	7

Figure 6.3: The distinction between syntax and semantics. Two different trees, T_1 and T_2 (different syntax), produce two different vectors of outputs for the same set of inputs (different semantics).

The core idea of GSGP is to design operators that are geometric in the *semantic space*, not the syntactic one. This requires:

1. Defining a distance metric on the semantics of programs (e.g., Euclidean distance between their semantic vectors).
2. Designing new crossover and mutation operators that construct an offspring tree whose *semantics* are guaranteed to lie on the segment between the parent semantics (for crossover) or in a ball around the parent's semantics (for mutation).

6.2.1 Geometric Semantic Crossover (GSC)

GSC generates an offspring T_{off} whose semantics are a linear interpolation of the semantics of its parents, T_1 and T_2. This is achieved by constructing the offspring tree as:

$$T_{off} = (R \times T_1) + ((1 - R) \times T_2)$$

Where R is a randomly generated program tree whose outputs are guaranteed to be in the range $[0, 1]$. For any input, the output of T_{off} will be on the segment between the outputs of T_1 and T_2.

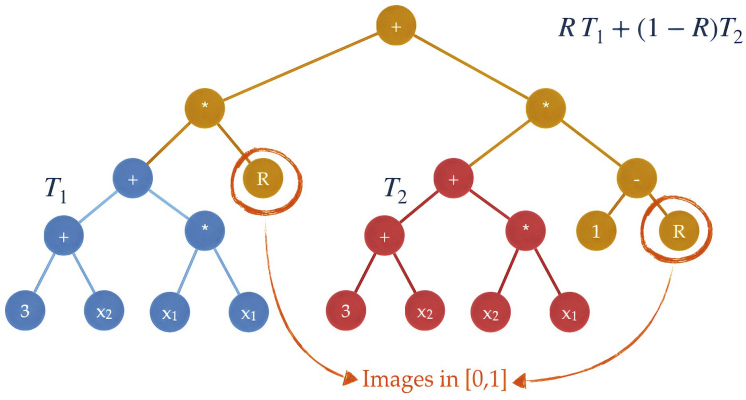


Figure 6.4: The structure of Geometric Semantic Crossover.

The effect on the semantic space is that the offspring's behavior is bounded by the behavior of its parents, providing a highly controlled and exploitative search.

6.2.2 Geometric Semantic Mutation (GSM)

GSM generates a mutated offspring T_{off} by adding a bounded random perturbation to the semantics of a single parent T_1. The construction is:

$$T_{off} = T_1 + (m \times R')$$

Where m is the ***mutation step*** (a constant that defines the mutation radius) and R' is a randomly generated tree whose outputs are guaranteed to be in the range $[-1, 1]$. This ensures the offspring's semantics lie within a "ball" of a specified radius around the parent's semantics.

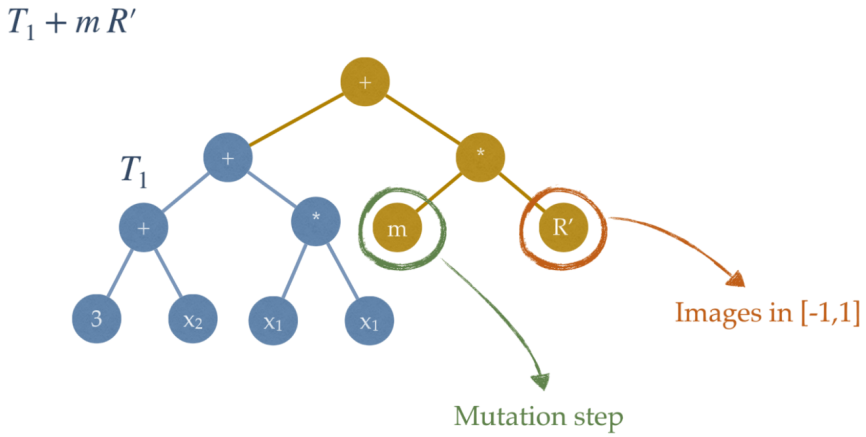


Figure 6.5: The structure of Geometric Semantic Mutation. The offspring tree is created by adding a scaled random component (controlled by mutation step m) to the parent tree T_1.

6.3 Challenges and Advancements in GSGP

While semantically powerful, GSGP faces a significant practical challenge: the trees it produces grow exponentially with each generation. The formula for GSC, for example, combines three trees to make one, leading to rapid and unsustainable bloat.

This has led to the development of more advanced implementations like ***Fast GSGP***, which use techniques such as **subtree sharing** and optimized data structures to manage this complexity, making the approach viable in practice. These methods have shown that GSGP can significantly outperform standard GP on many benchmark problems, especially in terms of generalization to unseen data.

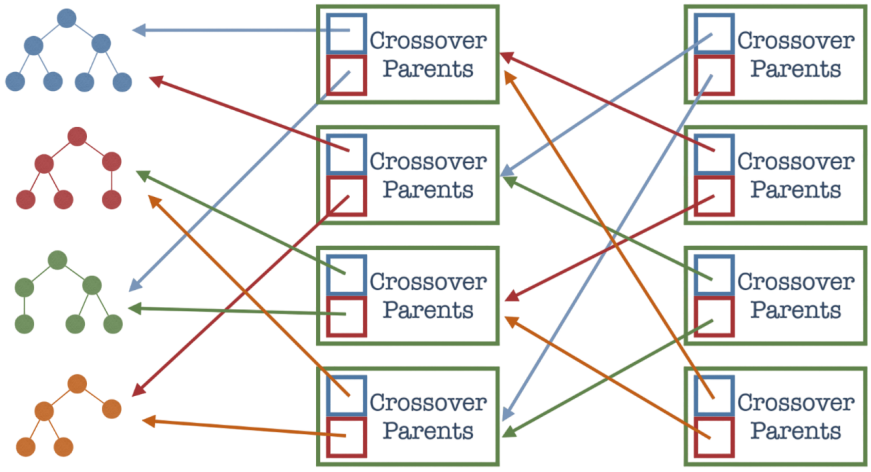


Figure 6.6: Fast GSGP illustration.

7 Estimation of Distribution Algorithms

7.1 From Implicit to Explicit Models

Traditional evolutionary algorithms, such as Genetic Algorithms (GAs), employ operators like selection, crossover, and mutation to navigate the search space. Collectively, these operators induce an *implicit* model of solution quality: selection favors promising individuals, while crossover and mutation explore their neighborhoods to discover new candidates. However, this implicit modeling limits the algorithm's ability to directly capture and exploit the structure of high-quality solutions.

In contrast, ***Estimation of Distribution Algorithms (EDAs)***, also referred to as ***Probabilistic Model-Building Genetic Algorithms (PMBGAs)***, adopt a fundamentally different strategy. Rather than relying on traditional genetic operators, EDAs construct an *explicit* probabilistic model that characterizes the distribution of promising solutions identified thus far. This model is then used to sample a new population of candidate solutions, which subsequently informs the next iteration of model refinement. The evolutionary process is thereby reframed as a cycle of model construction, sampling, and iterative improvement.

7.1.1 Model Fitting via Classification

A principled approach to explicit modeling is to recast the problem as a classification task. Here, the objective is to train a machine learning classifier to discriminate between "good" and "bad" regions of the search space, using the fitness values of individuals in the current population as labels.

Learnable Evolution Models

The ***Learnable Evolution Model (LEM)*** framework formalizes this classification-based approach. Its iterative procedure can be summarized as follows:

1. Execute several standard evolutionary steps to generate a diverse population.
2. Partition the population into two classes according to fitness: a "fit" set and an "unfit" set.
3. Train a classifier (e.g., decision tree, support vector machine, or neural network) to distinguish between fit and unfit individuals. This classifier serves as an explicit model of high-fitness regions in the search space.
4. Eliminate unfit individuals and generate new candidates by sampling from regions identified by the classifier as "fit."
5. Repeat this process until a predefined termination criterion is satisfied.

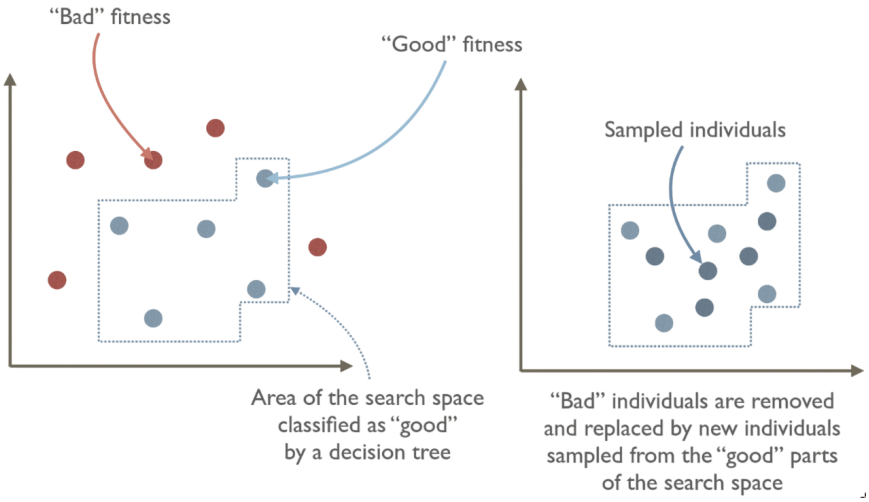


Figure 7.1: The classification process in a Learnable Evolution Model.

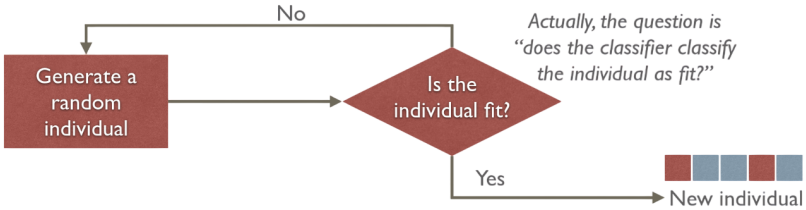
Generating New Individuals

A key challenge in this approach is how to generate new individuals using the trained model. This depends on whether the model is ***generative*** or ***discriminative***.

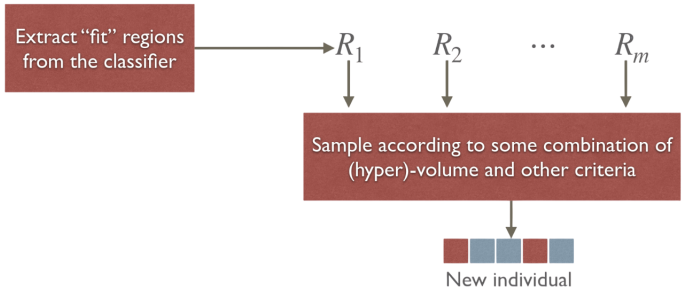
- ***Generative models*** can directly produce new samples that conform to the learned data distribution.
- ***Discriminative models***, such as most classifiers, can only determine whether a given, existing individual belongs to the ”fit” class or not.

Since classifiers are typically discriminative, two main sampling techniques are used:

- **Rejection Sampling**: new individuals are generated randomly from the entire search space and are then passed to the classifier. If the classifier labels an individual as ”fit,” it is accepted into the new population; otherwise, it is rejected, and the process repeats. This method is simple but can be highly *inefficient*, especially late in the search when the ”fit” region may be very small compared to the entire space, leading to a high rejection rate.



- **Region-Based Sampling**: New individuals are generated by sampling directly from regions of the search space identified as ”fit” by the classifier. For classifiers such as decision trees, these regions correspond to explicit rules or hyper-rectangles. Sampling is restricted to these regions, improving efficiency by avoiding the generation of individuals from unpromising areas. The effectiveness of this method depends on the classifier's ability to define such regions.



7.2 Probabilistic Model-Based EDAs

Instead of using a classifier, the most common family of EDAs builds and samples from an explicit ***probability distribution*** that represents the distribution of promising individuals in the population.

7.2.1 Representing the Distribution of Promising Solutions

The core task of an EDA is to build a probabilistic model that captures the features of high-performing individuals. The primary challenge lies in creating a representation that is both expressive enough to guide the search effectively and compact enough to be computationally feasible. A full, explicit joint probability distribution is almost always intractable due to the curse of dimensionality. Several strategies have been developed to address this.

Histogram-Based Models

A conceptually simple way to model the distribution is to create a multi-dimensional histogram. The search space is partitioned into a grid of discrete hypercubes (or "bins"), and the model stores the average fitness or count of promising solutions that fall into each bin. New solutions can then be sampled from the bins with higher scores.

However, this method's utility is severely limited by its exponential complexity. If each of the n dimensions of the search space is divided into just d intervals, the total number of hypercubes is d^n. This number grows so rapidly that the approach becomes unfeasible even for a small number of dimensions, necessitating more compact representations.

Gaussian Mixture Models for Continuous Spaces

For problems in continuous domains (e.g., $\mathbb{R}^n$), a more scalable approach is to model the probability distribution of good solutions as a ***Gaussian Mixture Model (GMM)***. Instead of a discrete grid, the distribution is represented as the sum of a fixed number, b, of multivariate Gaussian distributions. This allows the model to represent multiple promising regions in the search space simultaneously.

A full n-dimensional Gaussian distribution is defined by:

- A mean vector $\vec{\mu}$ of length n.
- A covariance matrix Σ of size $n \times n$, which captures the dependencies between variables.

While more compact than a histogram, the space required to store the full covariance matrix for each component is n^2. The total space complexity to represent the distribution as a sum of b Gaussians is $b(n+n^2)$, which can still be prohibitive if all variable interactions are modeled.

Marginal Distribution Models

To drastically reduce complexity, many EDAs adopt a powerful simplifying assumption: that the variables (genes) are ***mutually independent***. This assumption allows the high-dimensional joint probability distribution to be factorized into the product of n one-dimensional ***marginal distributions***.

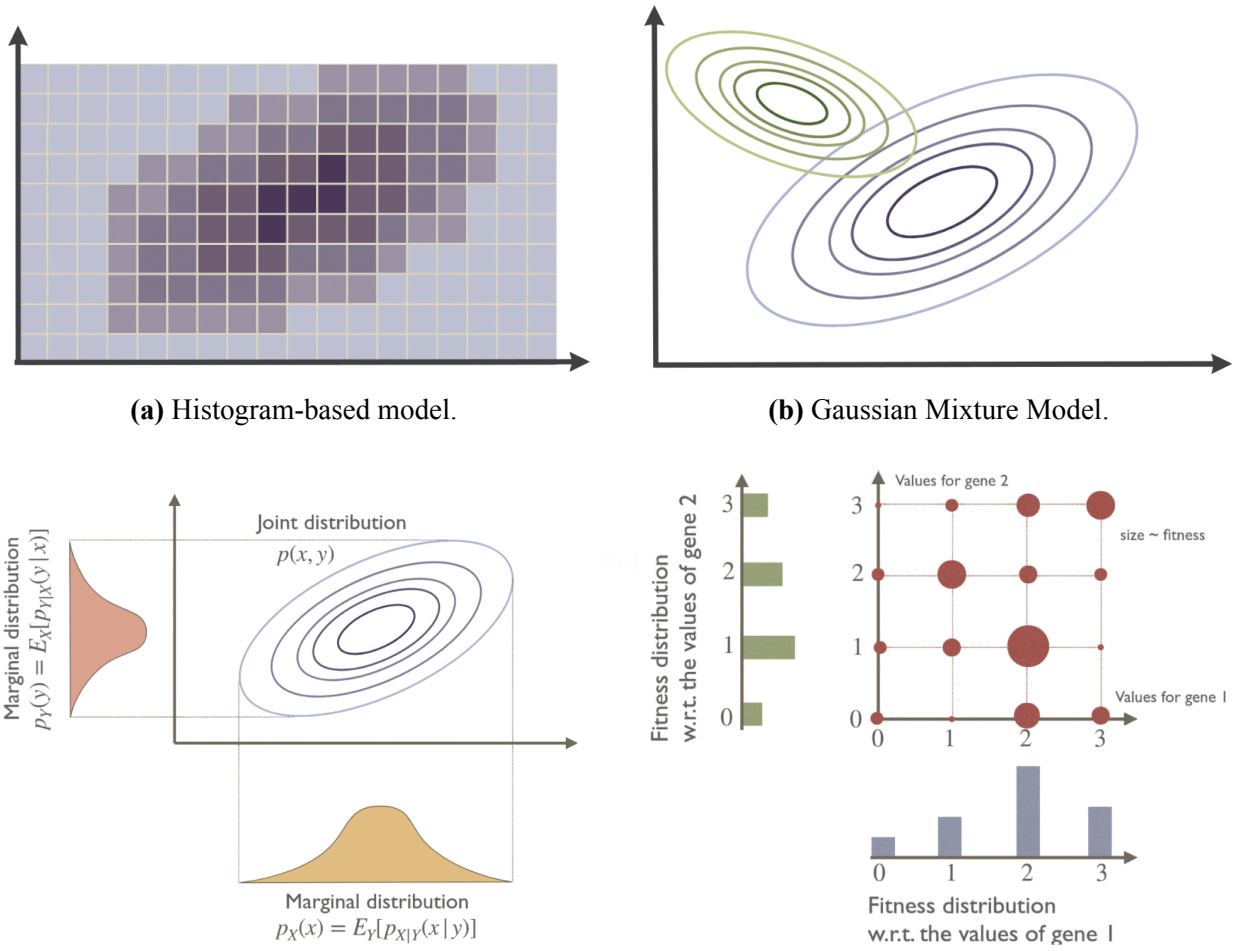
$$P(x_1,\ldots,x_n) = P_1(x_1)\cdot P_2(x_2)\cdots P_n(x_n)$$

This is the foundation of ***univariate EDAs***. Under this model, the complexity of a Gaussian mixture is greatly reduced. Each of the n marginal distributions is a 1D Gaussian mixture, which only requires storing a mean and a variance for each of its b components. The total space complexity falls from $O(bn^2)$ to a much more manageable $O(bn)$.

Models for Finite Discrete Spaces

The concept of using marginal distributions extends naturally to discrete domains.

- If a gene can take one of k possible discrete values, its marginal distribution can be represented by a probability vector of length $k-1$.
- For the very common case of ***Boolean spaces*** ($\{0,1\}^n$), the model is particularly simple. The marginal distribution for each gene can be described by a single value: the probability of that gene being 1. The entire probabilistic model is thus captured by a single n-dimensional vector of probabilities, which is extremely compact.



(a) Histogram-based model.

(b) Gaussian Mixture Model.

(c) Marginal distributions from a joint distribution.

(d) Marginal distributions for a finite discrete space.

Figure 7.2: Four different approaches to modeling the distribution of promising solutions in EDAs.

7.2.2 Univariate EDAs

Univariate EDAs adopt a standard approach in which the high-dimensional distribution over all genes is simplified by marginalizing everything to one dimension by modeling only the distribution of each gene individually. This means we are making a strong assumption: the value of each gene can be determined independently from the values of the other genes. While this independence assumption is often not strictly true in real-world problems, it allows for very simple and computationally efficient algorithms. We will examine two classic univariate EDA algorithms:

- ***Population-Based Incremental Learning (PBIL)***
- ***Compact Genetic Algorithm (cGA)***

Population-Based Incremental Learning (PBIL)

Population-Based Incremental Learning (PBIL) is a foundational univariate EDA that iteratively refines a probabilistic model to guide the search for optimal solutions.

For a problem of dimension n, PBIL maintains a vector of marginal distributions $D_1, D_2, \ldots, D_n$. The initial distribution is typically uniform. At each iteration:

- A population of new individuals is generated by independently sampling each gene j from D_j.
- The b fittest individuals are selected from the population (truncated selection).
- For each gene j, let N_j denote the empirical distribution of gene j among the b selected individuals.
- The probability vector is updated towards the observed distribution in the selected set:

$$D_j \leftarrow (1-\alpha)D_j + \alpha N_j$$

  where $\alpha \in [0,1]$ is a learning rate that controls the speed of adaptation.

This process is repeated until a stopping criterion is met.

Population-Based Incremental Learning (PBIL) can be ***extended to continuous domains***, such as $\mathbb{R}^n$, by representing each marginal distribution as a Gaussian distribution parameterized by a mean μ_{D_j} and a variance $\sigma^2_{D_j}$. After the selection step, for each gene j, the sample mean μ_{N_j} and sample variance $\sigma^2_{N_j}$ are computed from the selected individuals:

$$\mu_{N_j} = \frac{1}{|P|}\sum_{P_i \in P} P_{i,j} \qquad \sigma^2_{N_j} = \frac{1}{|P|-1}\sum_{P_i \in P}(P_{i,j} - \mu_{N_j})^2$$

The parameters of the Gaussian are then updated according to the following rules:

$$\mu_{D_j} \leftarrow (1-\alpha)\mu_{D_j} + \alpha\mu_{N_j}, \qquad \sigma^2_{D_j} \leftarrow (1-\alpha)\sigma^2_{D_j} + \alpha\sigma^2_{N_j}$$

This approach enables PBIL to adapt both the central tendency and the dispersion of each marginal distribution, thereby facilitating efficient exploration and exploitation in continuous search spaces.

PBIL can also be extended to discrete spaces with more than two values per gene by representing each marginal as a categorical distribution and updating each probability component accordingly. In all cases, the learning rate α allows for gradual adaptation, balancing exploration and exploitation.

Compact Genetic Algorithm (cGA)

The ***Compact Genetic Algorithm (cGA)*** is a streamlined EDA designed for Boolean search spaces. Unlike traditional genetic algorithms, cGA does not maintain an explicit population. Instead, it iteratively updates a probability vector through pairwise competitions.

1. Probability Vector: Maintain a vector D, where D_j is the probability of gene j being `1`.

2. Sampling: At each iteration, sample two individuals, P_i and P_k, independently from D.

3. Competition: Evaluate P_i and P_k and identify the winner (U) and loser (V) based on fitness.

4. Update: For each gene j where $U_j \neq V_j$, adjust D_j by a small step $1/d$ such that:

$$\begin{aligned} &\text{If } U_j = 1 \text{ and } V_j = 0: \quad D_j \leftarrow D_j + 1/d \\ &\text{If } U_j = 0 \text{ and } V_j = 1: \quad D_j \leftarrow D_j - 1/d \end{aligned}$$

Repeat this process for a fixed number of iterations or until convergence. The cGA thus efficiently evolves the probability vector to represent promising solutions without explicitly storing a population.

7.2.3 Issues of Univariate EDAs and Beyond

Univariate EDAs assume that the distribution of each gene can be found independently from all the other genes: they do not model any linkage or interaction between genes. While this assumption simplifies the model, it is often unrealistic: in many real-world problems, genes interact in complex ways, and optimizing them independently is not effective.

- **Gene independence is rarely true:** If genes were truly independent, we could optimize each one separately. In practice, dependencies between genes are common and important.
- **Local optima:** Because univariate EDAs cannot capture these dependencies, they may get stuck in local optima—situations where changing a single gene does not improve the solution, but changing several together would.
- **Need for multivariate models:** To address this, ***multivariate EDAs*** have been developed. These use more advanced probabilistic models that can represent and exploit dependencies between variables, allowing for more effective search in complex landscapes.

One of the most well-known multivariate EDAs is the ***Bayesian Optimization Algorithm (BOA)***. Unlike PBIL, which uses marginal distributions, BOA employs a Bayesian network to model dependencies between genes. This network is used to generate new samples and is updated at each generation.

Tip: ***BOA: Further Reading***

The following paper [2] describes the BOA algorithm:

Pelikan, Martin, David E. Goldberg, and Erick Cantú-Paz.
"BOA: The Bayesian optimization algorithm."
Proceedings of the genetic and evolutionary computation conference GECCO-99.
Vol. 1. 1999.

8 Covariance Matrix Adaptation ES

The primary limitation of univariate Estimation of Distribution Algorithms (EDAs) is their inability to model dependencies between variables. To address this, multivariate EDAs were developed. The ***Covariance Matrix Adaptation Evolution Strategy (CMA-ES)*** stands as arguably the most successful and widely used multivariate EDA, representing the state of the art for optimization in continuous domains.

CMA-ES blends concepts from both Evolution Strategies and EDAs. At its core, it is an EDA that uses a full multivariate Gaussian distribution to model the population of promising solutions. At each generation, it adapts all the parameters of this distribution:

- The ***mean vector*** $\vec{m}$, which represents the center of the search distribution (the current best guess).
- The ***covariance matrix*** C, which models the pairwise dependencies between variables and determines the shape and orientation of the distribution's contour lines (isodensity ellipses).
- A global ***step-size*** σ, which controls the overall scale of the distribution.

By adapting the full covariance matrix, CMA-ES can learn complex, non-axis-parallel relationships between variables, making it exceptionally powerful on difficult, non-separable, and ill-conditioned optimization problems.

Tip: ***Further Reading***

The definitive guide to CMA-ES is the tutorial by Nikolaus Hansen. It provides a comprehensive, in-depth explanation of the algorithm's mechanics and rationale.

Hansen, Nikolaus. "The CMA evolution strategy: A tutorial." arXiv preprint arXiv:1604.00772 (2016).

https://arxiv.org/abs/1604.00772

8.1 The $(\mu/\mu_w, \lambda)$-CMA-ES Algorithm

The standard CMA-ES algorithm follows a procedure analogous to a (μ, λ)-ES, but with weighted recombination of the selected parents. The main cycle is as follows:

1. **Sample:** Generate a population of λ new candidate solutions by sampling from the current multivariate normal distribution, $\mathcal{N}(\vec{m}, \sigma^2 C)$.
2. **Evaluate:** Assess the fitness of each of the λ new solutions.
3. **Select and Recombine:** Select the μ best individuals from the population and compute a new mean vector $\vec{m}$ as their weighted average.
4. **Update Strategy Parameters:** Use the selected μ individuals to update the covariance matrix C and the step-size σ.

Repeat this process until a termination criterion is met.

Sampling New Individuals

New candidate solutions (individuals) x_k are not sampled directly. Instead, they are generated in a two-step process that decouples the mean, step-size, and covariance:

1. A point y_k is sampled from a standard multivariate normal distribution with zero mean and covariance matrix C: $y_k \sim \mathcal{N}(\vec{0}, C)$.
2. This point is then scaled by the step-size σ and shifted by the mean vector $\vec{m}$:

$$x_k = \vec{m} + \sigma y_k$$

The fitness is evaluated for the point x_k. The step-size σ controls the scale of the distribution; larger values lead to a wider spread of sampled individuals, encouraging exploration.

Observation: ***Efficient Sampling via Eigendecomposition***

To sample efficiently from $\mathcal{N}(\vec{0}, C)$, CMA-ES uses the eigendecomposition of the covariance matrix. Since C is symmetric and positive-definite, it can be written as $C = BD^2B^T$, where B is an orthogonal matrix whose columns are the eigenvectors of C, and D is a diagonal matrix with the square roots of the corresponding eigenvalues.
A sample y_k can then be generated as $y_k = BDz_k$, where $z_k \sim \mathcal{N}(\vec{0}, I)$ is a sample from a standard normal distribution with an identity covariance matrix, which is trivial to generate. This decomposition is also key to efficiently updating C.

8.2 Updating the Distribution Parameters

Updating the Mean Vector

After selecting the μ best individuals, the new mean vector $\vec{m}$ is computed as their weighted average. Let the selected individuals, ordered by fitness, be $x_{1:\lambda}, \ldots, x_{\mu:\lambda}$. The new mean is:

$$\vec{m} \leftarrow \sum_{i=1}^{\mu} w_i x_{i:\lambda}$$

The weights w_i are predefined, positive, and sum to one ($\sum w_i = 1$). They are chosen to give more influence to the better individuals (e.g., $w_1 > w_2 > \ldots > w_\mu$). A common choice is:

$$w_i = \frac{\ln(\frac{\lambda+1}{2i})}{\sum_{j=1}^{\mu} \ln(\frac{\lambda+1}{2j})}$$

Updating the Covariance Matrix C

The adaptation of the covariance matrix is the most sophisticated part of CMA-ES. A naive approach would be to simply re-estimate the covariance from the selected points at each generation. However, this can lead to premature convergence. Instead, CMA-ES uses two separate, gradual update mechanisms: the **rank-μ update** and the **rank-one update**.

- **The Rank-μ Update**
  The rank-μ update incorporates the variance information from the current generation's selected steps. To avoid premature shrinkage of the distribution, it crucially uses the **old mean vector** $\vec{m}_{\text{old}}$ as its reference. The update rule is a gradual one:

$$C \leftarrow (1 - c_\mu)C + c_\mu \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda} (\mathbf{y}_{i:\lambda})^T$$

where c_μ is the learning rate for this update. The term $\sum w_i y_{i:\lambda} (y_{i:\lambda})^T$ is the weighted covariance of the selected steps $y_i = (x_i - \vec{m}_{\text{old}})/\sigma$, and is a matrix of rank (at most) μ.

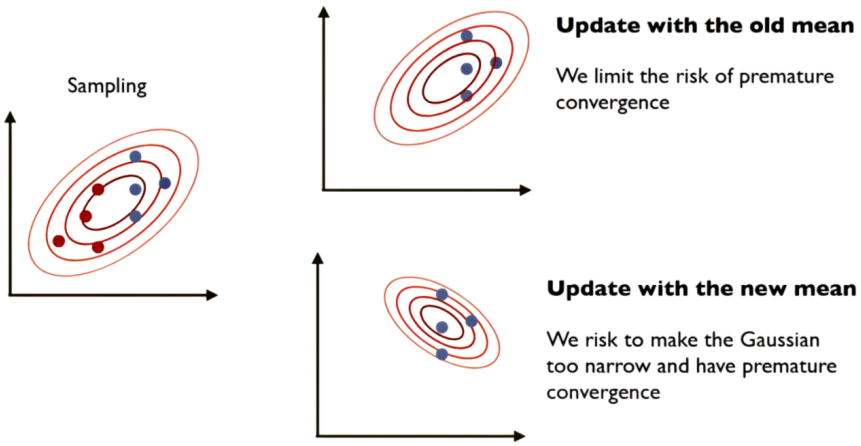


Figure 8.1: Covariance matrix update

- **The Rank-One Update and the Evolution Path**
  The rank-μ update only uses information from the current generation. To incorporate information about the correlation of successful steps *over time*, CMA-ES uses an ***evolution path***, $\vec{p}$. This vector is an exponentially fading record of the average direction the mean vector has been moving:
  $$\vec{p} \leftarrow (1 - c_c)\vec{p} + c_c \frac{\vec{m} - \vec{m}_{\text{old}}}{\sigma}$$
  where c_c is a learning rate.
  The evolution path $\vec{p}$ indicates if successive steps are correlated. The outer product $\vec{p}(\vec{p})^T$ is a rank-one matrix that captures this directional information. It is used to update the covariance matrix in what is called the ***rank-one update***.
- **The Combined Covariance Matrix Update**
  The final update rule for C combines the gradual decay of the old matrix, the rank-one update (from the evolution path), and the rank-μ update (from the current generation):
  $$C \leftarrow (1 - c_1 - c_\mu)C + c_1 \vec{p}(\vec{p})^T + c_\mu \sum_{i=1}^{\mu} w_i y_{i:\lambda} (y_{i:\lambda})^T$$
  where c_1 and c_μ are learning rates for the rank-one and rank-μ updates, respectively.

Step-Size Adaptation

The step-size σ is also adapted automatically. This adaptation uses a separate evolution path, $\vec{p}_\sigma$, to determine whether the search is making directional progress or just moving randomly.

- If the evolution path is consistently long, it means successive steps are not canceling each other out, indicating persistent directional movement. In this case, the step-size σ should be **increased** to move faster in that direction.
- If the evolution path is short, it means successive steps are uncorrelated and tend to cancel each other out, similar to random noise. This suggests the algorithm may be close to an optimum, so σ should be **decreased** to allow for finer-grained local search.

This compares the length of the evolution path to its expected length under random selection and adjusts σ accordingly, providing a parameter-free way to control the scale of the search.

9 Policy Optimisation

In many real-world problems, an optimal solution is not a single static object but a ***policy***: a strategy that dictates the best action to take in any given situation. Policy optimisation is a core challenge in reinforcement learning. The foundational concepts include agents, states, and actions, with Q-learning serving as a classic method. Evolutionary approaches, particularly ***Learning Classifier Systems (LCS)***, use rule-based systems to evolve robust and generalisable policies.

9.1 The Reinforcement Learning Framework

This problem is typically framed within the context of an agent interacting with an environment. The goal is to learn a behavior that maximizes a cumulative reward signal over time.

The interaction is formalized by the following components:

- A finite set of ***states*** $\mathcal{S}$, representing all possible situations the agent can be in.
- A finite set of ***actions*** $\mathcal{A}$, representing all possible moves the agent can make.
- A ***transition function*** $P(s'|s,a)$, the probability of moving to state s' after an action a in state s.
- A ***reward function*** $R(s,a)$, which is the immediate reward received for taking action a in state s.

A key assumption is that the environment is ***Markovian***, meaning the transition probabilities and rewards depend only on the current state and action, not on the sequence of events that preceded them. This ”memoryless” property simplifies the problem significantly: the agent does not need to remember its entire history. Instead, its behavior can be defined by a ***policy***, π. A policy is a function that maps states to actions:

$$\pi : \mathcal{S} \to \mathcal{A}$$

Given a state s, $\pi(s)$ is the action the agent will perform. The ultimate goal is to find the ***optimal policy***, π^*, which maximizes the expected cumulative reward over the agent's lifetime.

9.1.1 Dense Policy Optimisation: Q-Learning

Q-learning is a foundational, non-evolutionary technique for finding the optimal policy. Instead of learning the policy π directly, it learns a state-action value function, called the ***Q-function***.

The Q-function, $Q(s,a)$, represents the expected total future reward of taking action a in state s and then following the optimal policy thereafter. In its simplest form, for discrete states and actions, this function can be stored in a lookup table called a ***Q-table***, with a row for each state and a column for each action.

Once the optimal Q-function, Q^*, is known, the optimal policy π^* can be derived directly by always choosing the action with the highest Q-value in the current state:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a)$$

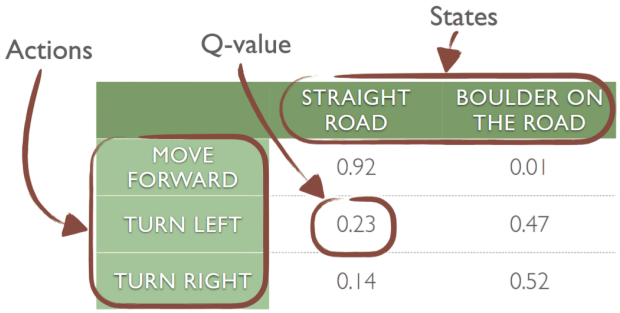


Figure 9.1: A Q-table stores the Q-value for every state-action pair.

The approach to finding the optimal Q-function, Q^*, depends on whether the agent has access to a model of the environment's dynamics.

- **Model-Based Learning**
  When the agent knows the transition probabilities $P(s'|s,a)$ and the reward function $R(s,a)$, it can compute Q^* using dynamic programming techniques as the Bellman equation:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q^*(s',a')$$

  Here, $\gamma \in [0,1)$ is the ***discount factor***, which determines how much future rewards are valued relative to immediate rewards. This equation expresses a self-consistency: the value of taking action a in state s equals the immediate reward plus the expected value of the best possible action in the next state, weighted by the probability of reaching each possible s'. By repeatedly applying this update to all state-action pairs (***value iteration***), the Q-table converges to Q^*.
- **Model-Free Learning**
  In most real-world scenarios, the agent does not know the environment's transition probabilities or reward function in advance. Instead, it must learn Q^* directly from experience by interacting with the environment. This ***model-free*** approach uses the following update rule:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left[r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') \right]$$

  where $\alpha \in (0,1]$ is the ***learning rate***. Over time, as the agent explores the environment and updates its Q-table, the values will eventually converge to Q^*.

Tip: ***Intuition***

Model-free Q-learning enables agents to learn optimal behavior even in complex, unknown environments, simply by trial and error. The update rule ensures that the agent continually refines its estimates of long-term reward, balancing past experience with new observations.

Tabular Q-learning, while foundational, has two major **drawbacks**:

1. **Lack of Generalization**: It requires a distinct entry in the Q-table for every possible state-action pair. It cannot generalize to unseen states or handle continuous state spaces. The size of the Q-table explodes as the state space grows.
2. **Exploration vs. Exploitation**: The standard update is purely exploitative. To ensure the agent explores the environment sufficiently, techniques like ε-greedy action selection (where the agent takes a random action with probability ε) must be added.

These limitations motivate the use of ***sparse policy optimisation*** techniques, which can generalize across states. **Learning Classifier Systems** are a powerful evolutionary approach to this problem.

9.1.2 Learning Classifier Systems (LCS)

Learning Classifier Systems (LCS) are a family of rule-based systems that use evolutionary algorithms to evolve a set of rules that collectively form a policy. The core idea is to aggregate states using rules, allowing for generalization.

Rule-Based Representations

An LCS works with a set of rules, each typically in the form: `IF <condition> THEN <action>`.

- The ***condition*** part (the rule body) specifies a subset of the state space where the rule applies.
- The ***action*** part (the rule head) specifies the action to be taken if the condition is met.

A single rule can cover many states, and a single state can be covered by multiple rules. This requires an ***arbitration scheme*** to decide the rule when multiple ones match the current state.

For rule conditions in different spaces:

- **Boolean Spaces**: Conditions are often represented using a ternary alphabet $\{0, 1, \#\}$, where `#` is a ”don’t care” symbol that matches both 0 and 1. For example, the condition `10#` matches states where the first variable is 1, the second is 0, and the third can be anything.
- **Real-Valued Spaces**: Conditions can be defined by geometric shapes, such as hyper-rectangles (”boxes”), hyper-ellipsoids, or hyperplanes.

The Pitt vs. Michigan Approach

There are two main architectural paradigms for LCS:

1. **The Pittsburgh (Pitt) Approach**: Each individual in the evolutionary population is an *entire set of rules* (a complete policy). The fitness is evaluated for the rule set as a whole.
2. **The Michigan Approach**: Each individual is a *single rule*. The entire population of rules forms the policy. This is a form of cooperative co-evolution, where rules must collaborate to form a good solution.

9.1.3 The Pitt Approach: SAMUEL

SAMUEL (Strategy Acquisition Method Using Empirical Learning) is a classic system that exemplifies the Pittsburgh-style LCS. In SAMUEL, each individual in the population is a complete rule set, representing an entire policy. The system’s lifecycle combines phases of local, intra-generational rule improvement with a traditional evolutionary process acting on the entire population.

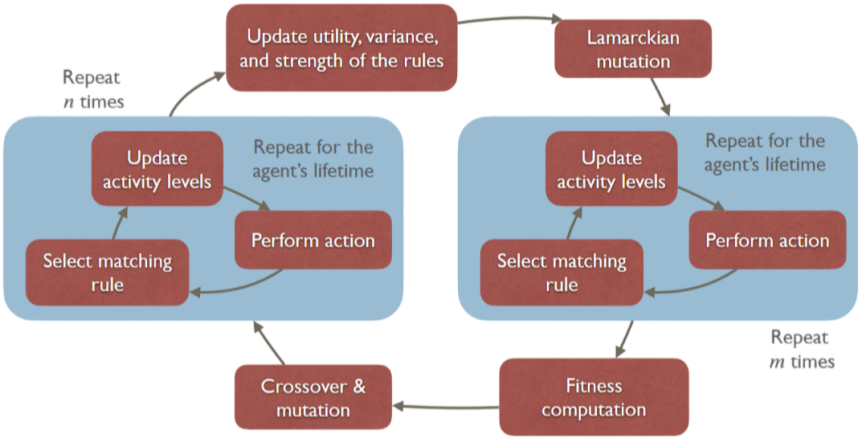


Figure 9.2: The SAMUEL cycle.

Rule Selection

When the agent finds itself in a particular state, the process of selecting which **action to perform** in SAMUEL involves two key steps:

1. **Matching:** Each rule in the individual's rule set is evaluated to compute a ***match score***, which quantifies how well the rule's condition aligns with the current state. This allows for partial or fuzzy matches, so that rules can generalize across similar states rather than requiring an exact match.
2. **Selection:** From all available rules, a smaller ***match set*** is constructed, typically by selecting the rule(s) with the highest match scores (a form of truncated selection). The final rule to execute is then chosen from this match set, often stochastically, with the probability of selection proportional to each rule's match score.

This two-step process enables SAMUEL to flexibly arbitrate between overlapping or competing rules, and to adapt to uncertainty or noise in the environment.

Rule Quality Metrics

SAMUEL maintains and updates several distinct **metrics** for each rule, allowing it to assess and refine the rule set over time:

- ***Fitness***: The fitness of the entire rule set (the individual) is simply the sum of all rewards obtained by the agent during its evaluation period.
- ***Utility***: Similar to a Q-value, this measures the expected reward for applying a rule R_i. It is updated without discounting future rewards:

  $$\text{Utility}(R_i) \leftarrow (1-\alpha)\text{Utility}(R_i) + \alpha r$$

  where r is the immediate reward received and α is a learning rate controlling the update speed.
- ***Utility Variance***: This metric tracks the variability or consistency of a rule's payoff, which is crucial for distinguishing between reliable and unreliable rules:

  $$\text{UtilityVariance}(R_i) \leftarrow (1-\alpha)\text{UtilityVariance}(R_i) + \alpha(\text{Utility}(R_i) - r)^2$$

  A low variance indicates that the rule's performance is stable, and vice versa.
- ***Strength***: This composite metric provide an overall assessment of a rule's quality:

  $$\text{Strength}(R_i) \leftarrow \text{Utility}(R_i) + \gamma\text{UtilityVariance}(R_i), \quad \text{where } 0 \leq \gamma < 1$$

  The parameter γ controls the influence of variance on the strength score, allowing the system to penalize rules that are inconsistent even if their average reward is high.
- ***Activity***: This metric records how frequently a rule is active (i.e., selected and applied). It is used to identify and prune rules that are rarely or never used, helping to keep the rule set efficient and focused. When a rule proposing action a is applied, the activity of all rules with action a in their head is reinforced, while the activity of all other rules decays:

  $$\begin{aligned} &\text{Activity}(R_i) \leftarrow (1-\beta)\text{Activity}(R_i) + \beta && \text{(if rule } R_i \text{ proposes action } a\text{)} \\ &\text{Activity}(R_j) \leftarrow \delta \cdot \text{Activity}(R_j) && \text{(if rule } R_j \text{ does not propose action } a\text{)} \end{aligned}$$

  Here, β is a reinforcement rate and $0 < \delta < 1$ is a decay factor. At the start of the agent's lifetime, all rules are initialized with the same activity: $Activity(R_i) = \frac{1}{2}$.

Together, these metrics enable SAMUEL to not only evaluate the effectiveness of individual rules, but also to adaptively refine the rule set by promoting strong, reliable, and frequently used rules while eliminating weak or redundant ones.

Genetic Operators in SAMUEL

SAMUEL features a rich set of genetic operators, divided into two distinct phases: a local, exploitative **Lamarckian** phase that modifies rules within a single individual, and a standard **Classical** phase that operates on the population of individuals.

- **Lamarckian Mutation Operators**
  These operators are applied during the agent's evaluation lifetime to perform local improvements on its rule set. The parent rule is typically kept in the population alongside the newly created one.
  - ***Rule Deletion***: An old rule with a low activity value may be selected for deletion.
  - ***Rule Specialisation***: If a rule is too general (covers a very large number of states), more restrictive conditions can be added to create a more specialized version.
  - ***Rule Generalisation***: If a rule is too specific (covers very few states), some of its conditions can be removed to make it more general.
  - ***Rule Covering***: If a rule frequently fires with a partial match, the conditions that are not being matched can be removed to improve its applicability.
  - ***Rule Merging***: If two strong rules have the same action and cover overlapping sets of states, they can be merged into a single, more general rule.
- **Classical Mutation and Recombination**
  These operators are applied during the evolutionary phase to create new individuals (rule sets) from existing ones.
  - ***Standard Mutation***: Changes parts of a rule without keeping the parent.
  - ***Creep Mutation***: Makes small, random changes to a rule's conditions, mimicking a local hill-climbing search.
  - ***Uniform Crossover***: Two parent rule sets exchange a fixed number, k, of rules.
  - ***Clustered Crossover***: This operator first identifies pairs or sequences of rules that frequently lead to rewards. It then performs uniform crossover but ensures these beneficial clusters of rules are kept together and not broken apart.

Population-Level Operations

Selection Selection in SAMUEL is a two-stage process. It first establishes a dynamic fitness baseline:

$$\text{baseline} \leftarrow (1-\nu)\text{baseline} + \nu(\mu_{\text{fitness}} + \psi\sigma^2_{\text{fitness}})$$

This baseline is updated using the current population's mean and variance of fitness. Only individuals whose fitness is *above* this baseline are considered for the subsequent standard selection procedure (e.g., roulette wheel or tournament selection), which creates the parent pool for the next generation.

Initialisation SAMUEL supports several strategies for creating the initial population of rule sets:

1. **Random Rules**: The initial rule sets are populated with randomly generated rules.
2. **Domain-Specific Rules**: The system can be seeded with a set of human-designed, heuristic rules that are known to be useful, potentially speeding up the initial phase of evolution.
3. **Generalist Rules**: Each individual can be initialized with a set of highly general rules of the form `IF <all states> THEN a` for each possible action a. These rules cover the entire state space and are subsequently refined and specialized by the Lamarckian and classical operators during the evolutionary process.

9.1.4 The Michigan Approach

In the Michigan approach, the paradigm shifts: each individual in the evolving population is a *single rule*, and the policy is represented by the *entire population of rules*. This setup is a form of cooperative coevolution, where rules must collaborate and compete to form an effective global solution. We will examine two Michigan-style systems: ZCS and its powerful successor, XCS.

The Zeroth-Level Classifier System (ZCS)

ZCS is an early, influential Michigan-style LCS. It maintains a population of `IF ... THEN ...` rules and uses a steady-state Genetic Algorithm to evolve them.

- The population is a ***single set of rules*** (each individual is a rule).
- The fitness of a rule is directly tied to its ***utility*** (its predicted payoff).
- ZCS requires an ***exact match*** for a rule's condition to be satisfied.
- It includes a ***covering*** mechanism: if the agent encounters a state s for which no rule matches, a new rule that covers s is created and replaces an existing, low-fitness one in the population.

ZCS operates as a steady-state system. The inner loop consists of agent interaction with the environment and fitness updates. The outer, evolutionary loop is triggered periodically.

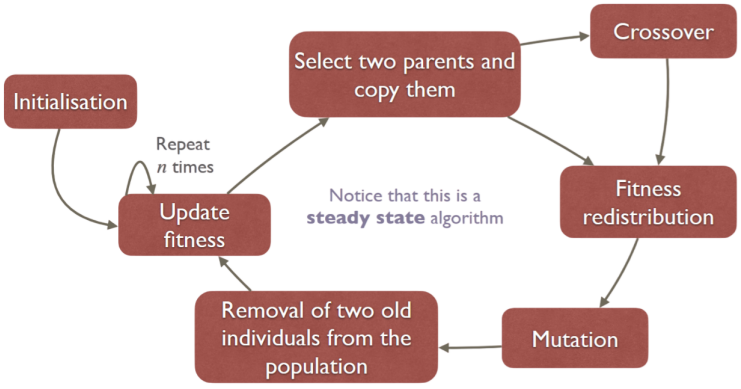


Figure 9.3: The steady-state cycle of the ZCS algorithm.

The **fitness update** in ZCS is inspired by the bucket-brigade algorithm, a precursor to Q-learning.

1. For the current state s, find the ***match set*** M of all rules whose conditions match s.
2. Select one rule from M to apply (e.g. fitness-proportional selection) and extract its action a.
3. Identify the ***action set*** $A \subseteq M$ of all rules in M that also propose action a.
4. Perform action a, receive reward r, and transition to the next state s'.
5. Each rule $A_i \in A$ updates its fitness by sharing some with the previous action set and receiving a portion of the reward r plus discounted future value from A':
$$\text{Fitness}(A_i) \leftarrow (1-\alpha)\text{Fitness}(A_i) + \alpha \frac{1}{|A|}\left(r + \gamma \sum_{A'_j \in A'} \text{Fitness}(A'_j)\right)$$
6. Additionally, rules in M that proposed different actions ($B = M - A$) are penalized:
$$\text{Fitness}(B_i) \leftarrow \beta \cdot \text{Fitness}(B_i), \quad \text{with } \beta \in [0,1]$$

When the GA is triggered, two parents P_1, P_2 are selected for **fitness redistribution**:

- If no crossover occurs, the parents and children (C_1, C_2) share fitness:
$$\text{Fitness}(C_i) \leftarrow \frac{1}{2}\text{Fitness}(P_i) \quad \text{and} \quad \text{Fitness}(P_i) \leftarrow \frac{1}{2}\text{Fitness}(P_i)$$
- If crossover occurs, children receive a portion of the parents' fitness, the parents' fitness is halved:
$$\text{Fitness}(C_i) \leftarrow \frac{1}{4}(\text{Fitness}(P_1) + \text{Fitness}(P_2)) \quad \text{and} \quad \text{Fitness}(P_i) \leftarrow \frac{1}{2}\text{Fitness}(P_i)$$

The XCS Algorithm

XCS is a landmark LCS that builds upon ZCS and introduces several major improvements, most notably the decoupling of a rule's fitness from its predicted utility. This fundamental change transforms the objective of the system.

The central innovation in XCS is the **accuracy-based fitness**: a rule's fitness is based on the ***accuracy*** of its prediction, not the magnitude of the prediction itself. This encourages the system to form a complete and accurate map of the entire state-action-payoff landscape, rewarding rules that are reliable predictors, even if they predict low payoffs. XCS maintains several distinct measures for each rule:

1. ***Rule Utility (Prediction)***: An estimate of the expected payoff if the rule is applied. This is kept separate from fitness.
2. ***Utility Error***: An estimate of the mean absolute error in the utility prediction.
3. ***Accuracy***: A measure derived from the utility error. Lower error means higher accuracy.
4. ***Fitness***: A function of the rule's accuracy. This value is used for selection in the genetic algorithm.

XCS differs from ZCS in four key ways:

- **Action Selection**: XCS makes a more informed decision by considering all rules in the match set M. It first calculates a *system prediction* for each possible action a by taking a fitness-weighted average of the utility predictions of all rules in M that propose action a. The action with the highest system prediction is then chosen (often with an ε-greedy strategy).

$$\text{Score}(a) = \frac{\sum_{r \in R_a} \text{Utility}(r) \times \text{Fitness}(r)}{\sum_{r \in R_a} \text{Fitness}(r)}, \quad \text{where } R_a \subseteq M$$

- **Utility and Fitness Updates**: The utility, error, and accuracy of rules in the current action set are updated using Q-learning-style rules. Fitness is then updated as a function of accuracy.
- **Fitness Redistribution**: When offspring are created, the fitness of the parents is not changed. The offspring inherit a fraction of the parents' fitness, preventing experienced rules from being immediately disadvantaged.
- **Niche-Based GA**: The genetic algorithm for creating new rules operates only within the ***action set*** of the chosen action. This "niche" approach encourages specialization, as rules only compete with other rules that advocate for the same action in the same context.

By prioritizing accuracy, XCS evolves a population of classifiers that are not just effective but also maximally general, forming a compact and comprehensive model of the environment.

10 Distributed Methods, Coevolution

10.1 Distributed Methods

Evolutionary algorithms are often computationally expensive, especially when fitness evaluations are costly or when large populations are required. Distributed and parallel methods allow us to leverage multiple cores, computers, or even specialized hardware (such as GPUs) to accelerate these computations. Beyond speed, distributed models can also improve the quality of evolution by maintaining greater diversity within the population.

There are several reasons to use distributed methods:

- **Efficiency:** Running evolutionary algorithms can be resource-intensive. By distributing the workload across multiple processors or machines, we can significantly reduce computation time.
- **Quality:** Some distributed models help preserve diversity, which can lead to better solutions and avoid premature convergence.
- **Suitability:** Population-based evolutionary algorithms are naturally amenable to parallelization, often more so than other optimization methods.

The simplest way to parallelize evolutionary algorithms is to run multiple independent instances (or runs) of the algorithm in parallel, each on a different core or machine. This is often called **embarrassingly parallel** computation.

10.1.1 Master-Slave Model

A more sophisticated approach, known as the **master-slave** or **client-server** model, is to distribute the fitness evaluation step, which is often the most computationally expensive part of the algorithm.

- The evolutionary process (selection, variation, etc.) is managed by a central node (master).
- Fitness evaluations are delegated to worker nodes (slaves), which may be on different machines.
- Only individuals and their fitness are communicated between nodes, minimizing data transfer.

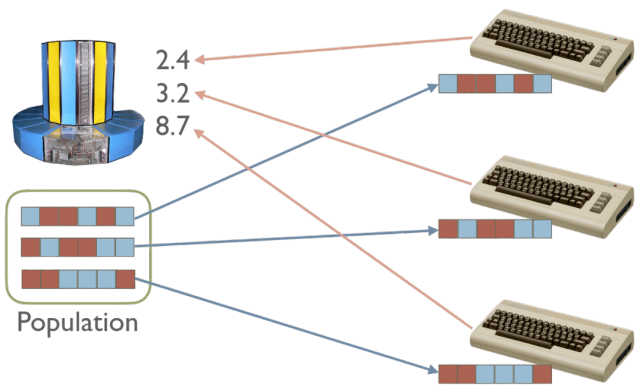


Figure 10.1: Fitness evaluations are distributed, the evolutionary process remains centralized.

The master-slave model offers several key advantages that make it particularly well-suited for evolutionary algorithms with expensive fitness evaluations:

- **Scalability:** If fitness evaluation is expensive, then this method scales well.
- **Fault Tolerance:** The system can be robust to failures of individual worker nodes.
- **Flexibility:** Nodes can be added or removed as needed.
- **Limitation:** If communication time is comparable to evaluation time, the benefits diminish.

Tip: ***Practical Note***

Distributed fitness assessment is most effective when the cost of evaluating individuals dominates the cost of communication. For problems with lightweight fitness functions, the overhead of distribution may outweigh the benefits.

10.1.2 Island Model

Distributed evolution can be inspired by the concept of real-world islands: populations evolve independently on separate islands, with occasional exchange of individuals. This approach, known as the ***Island Model***, introduces additional parameters and topological considerations, enabling both improved scalability and increased diversity.

Model Description and Parameters

In the Island Model, the global population is divided into several subpopulations (islands), each evolving independently. Periodically, individuals migrate between islands, allowing beneficial traits to spread while maintaining diversity.

Key parameters include:

- **Number of islands:** How many subpopulations are maintained.
- **Population size per island:** The number of individuals on each island.
- **Migration topology:** How islands are connected (who can exchange individuals with whom).
- **Migration policy:** Which individuals are exchanged, how many, and how often.

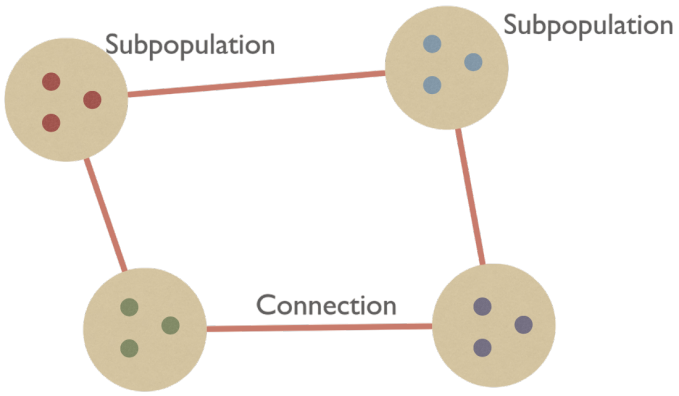


Figure 10.2: Island Model: subpopulations (islands) are connected by migration paths.

Observation: ***Independency of Islands***

There is no need for the islands to share the same parameters or even the same fitness function.

Topologies for Island Models

The migration topology determines the pattern of connections between islands. Common topologies are:

- **Ring:** Each island exchanges individuals with its immediate neighbors.
- **Grid:** Islands are arranged in a 2D grid, exchanging with adjacent islands.
- **Fully connected:** Every island can exchange with every other island.
- **Toroid:** A grid with wrap-around connections.

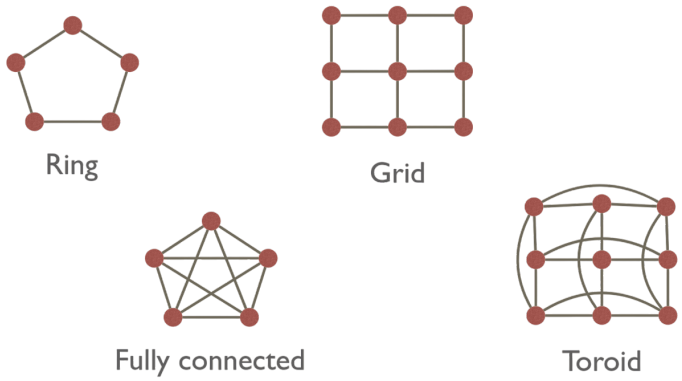


Figure 10.3: Examples of common island model topologies: ring, grid, fully connected, toroid.

A Typical Island Model Algorithm

A typical island model algorithm is:

1. Each island evolves its population independently using a standard evolutionary algorithm.
2. Every K generations, a subset (e.g., the top 5%) of individuals is selected for migration.
3. Migrants are sent to neighboring islands according to the chosen topology.
4. Migration can be synchronous (at the same time) or asynchronous (whenever ready).

Tip: ***Synchronous vs. Asynchronous Migration***

- ***Synchronous migration*** requires all islands to pause and exchange individuals at the same time, which can be less efficient if islands progress at different speeds.
- ***Asynchronous migration*** allows islands to exchange whenever they are ready, improving resource utilization and scalability.

Variants: Central and Collector Topologies

There are two main variants of the island model:

- **One Central Population (Star Topology)**
  In the star topology, each island evolves independently, but periodically sends its best individuals to a central island. The central island acts as a collector and redistributor of genetic material.

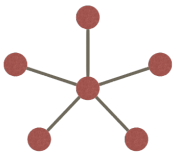


Figure 10.4: Star topology: peripheral islands send individuals to a central island.

- **Collector Topology**
  A more general variant is the collector topology, where islands are arranged in a directed acyclic graph. Individuals move in one direction, allowing for staged or layered optimization (e.g., different islands reward different aspects of a solution).

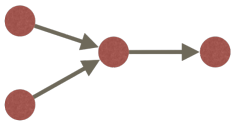


Figure 10.5: Collector topology: individuals move through a directed acyclic graph of islands.

10.1.3 Spatially-Embedded Evolutionary Algorithms

Beyond coarse-grained parallelism (islands), we can also embed individuals in a spatial structure, such as a grid or lattice. In spatially-embedded evolutionary algorithms, each individual interacts only with its immediate neighbors, creating a fine-grained parallel structure that promotes local competition while maintaining global diversity.

The key characteristics of spatially-embedded EAs include:

- **Spatial positioning:** Each individual occupies a specific location in the topological structure.
- **Local neighborhoods:** Selection, crossover, and mutation operations are performed using only individuals within a defined local neighborhood (e.g., Moore or von Neumann neighborhoods).
- **Gradual information diffusion:** Good solutions spread gradually across the spatial structure, preventing rapid convergence and maintaining population diversity.
- **Parallel execution:** This fine-grained parallelism maps naturally to parallel hardware architectures, particularly GPUs where thousands of individuals can be processed simultaneously.
- **Asynchronous updates:** Individual locations can be updated independently, allowing for efficient parallel implementation without global synchronization.

The spatial embedding creates natural niches where different regions of the population can explore different areas of the search space, leading to better exploration and reduced premature convergence compared to panmictic populations.

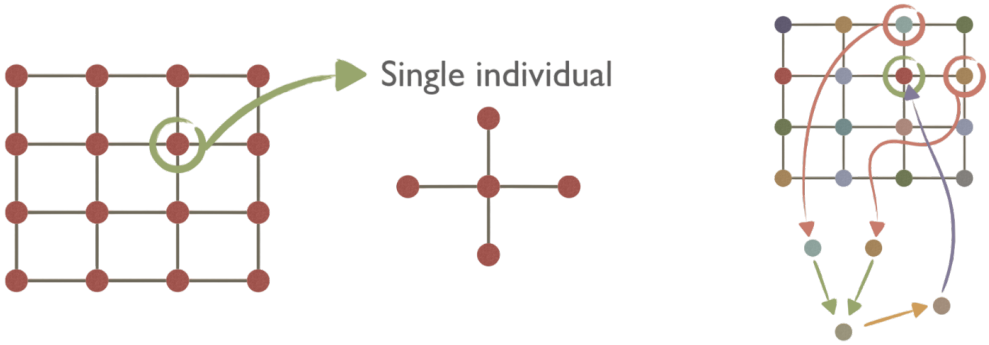


Figure 10.6: In spatially-embedded EAs, each individual interacts only with its neighbors.

A typical update cycle for an individual in spatially-embedded EA involves the following steps:

- **Parent selection:** Select two parent individuals from the local neighborhood
- **Crossover:** Apply crossover operation between the selected parents to generate offspring
- **Mutation:** Apply mutation operators to the offspring to introduce genetic variation
- **Replacement:** Replace the current individual at this spatial location with the generated offspring

10.2 Coevolution

In standard evolutionary algorithms, the fitness of an individual is determined solely by its own performance on the objective function. In ***coevolutionary algorithms***, however, the fitness of an individual is influenced by interactions with other individuals, either within the same population or across multiple populations. This fundamental shift from independent to interdependent fitness evaluation enables the evolution of more complex and adaptive behaviors.

The main idea of coevolution is that the fitness of an individual is affected by external factors:

- **Performance against others:** Fitness may depend on how well an individual performs against other members of the population (e.g., in games or adversarial tasks).
- **Collective performance:** Sometimes, the fitness is based on the performance of the entire population as a group.
- **Similarity penalties:** Too many similar individuals may be penalized to encourage diversity.

This interdependent nature of fitness evaluation means that in coevolution, the distinction between absolute and relative fitness becomes critically important:

- **Absolute fitness:** The fitness we wish to optimize (e.g., winning against all possible opponents).
- **Relative fitness:** The fitness as measured within the current population, which may only reflect performance against current peers.

Tip: ***Fitness in Coevolution***

Progress in coevolution should ideally be measured using both internal (relative) and external (absolute) fitness, to ensure that improvements are meaningful beyond the current population.

10.2.1 One-Population Competitive Coevolution

Competitive coevolution is a type of coevolutionary algorithm where the fitness of an individual is determined by its performance against other individuals in the same population. This approach is often used when evolving agents that play games. Each individual's fitness is determined by competing against other individuals in the same population. For example, agents may play matches against each other, and fitness is based on the number of victories.

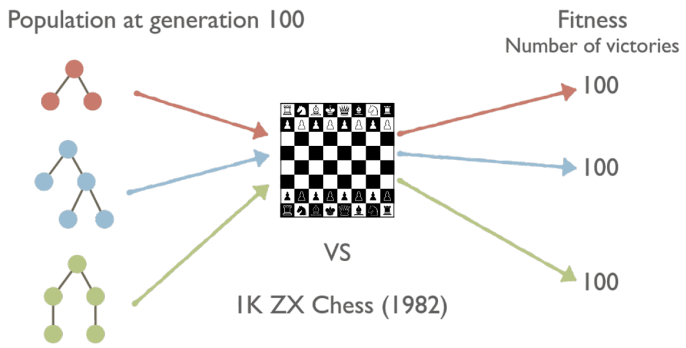


Figure 10.7: Competitive coevolution: example with a too weak or too strong opponent.

A challenge is that evaluating the "real" fitness can be difficult, especially if the population is much weaker or stronger than a fixed opponent. This is known as the problem of using a fixed opponent:

Evaluating the fitness of the individuals can be misleading if the opponent is too weak or too strong.

This can be solved using a ***collection of fixed opponents***, or using other ***individuals as opponents***.

Evaluation Methods

Several methods exist for evaluating individuals in competitive coevolution:

- **Pairwise:** Only make individual n compete with individual $n+1$. This approach is very efficient, requiring only $O(n)$ tests, but the results are highly dependent on the specific adversary each individual faces, which can introduce significant bias.

- **Complete:** Every individual competes with every other individual. This method provides a precise assessment of fitness, as all possible matchups are considered, but it is computationally expensive, requiring $O(n^2)$ tests as the population grows.

- **K-fold:** Every individual competes with k randomly selected individuals. This method offers a tunable trade-off between computational cost and evaluation accuracy. However, some individuals may be tested more than others, potentially introducing uneven selection pressure.

- **Single-elimination tournament:** Every individual proceeds with the competitions until it is beaten. This approach is efficient, as only $O(n)$ tests are needed, and better individuals are likely to be tested more times. However, unlucky pairings can penalize some individuals, making the outcome sensitive to the tournament structure.

Fitnessless Selection

In some cases, the fitness is never explicitly computed, which can be both computationally efficient and conceptually elegant. E.g., in tournament selection, the selected individual is simply the winner of the tournament, without requiring any numerical fitness assignment. This approach bypasses the need to calculate absolute fitness values and instead relies on relative performance comparisons.

10.2.2 Two-Population Competitive Coevolution

In two-population competitive coevolution, two distinct populations are evolved in parallel, each serving a complementary but opposing role.

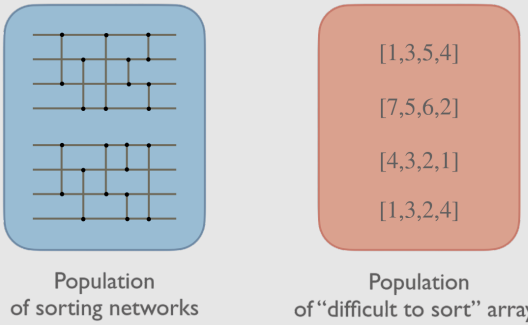
- **Primary population:** The individuals we are actually interested in improving and optimizing (e.g., sorting networks, game-playing strategies, or neural network controllers).
- **Alternative (foil) population:** Individuals that serve as adaptive adversaries, specifically designed to challenge, test, and ”foil” the primary population by exploiting its weaknesses (e.g., arrays that are difficult to sort, counter-strategies, or challenging test cases).

This setup creates a powerful evolutionary arms race where both populations continuously improve together. As the primary population develops better solutions, the foil population evolves more challenging test cases that expose new weaknesses and vice versa.

Example: *Evolving Sorting Networks*

A classic example is the coevolution of sorting networks and difficult-to-sort arrays:

- The primary population consists of candidate sorting networks.
- The foil population consists of arrays that are hard to sort.



For this scenario, fitness is defined as follows:

- **For sorting networks:** Fitness is the number of arrays sorted correctly.
- **For arrays:** Fitness is the number of networks that fail to sort the array.

The two fitnesses are in conflict, making the coevolution competitive.

How the Evolution is Performed

The evaluation process in two-population competitive coevolution is dynamic, as both populations are constantly changing. The process of assessing the fitness can be described in several steps:

1. **Shuffle and Pairing:** Individuals from population P are paired with individuals from population Q, typically by shuffling and pairing them randomly. This pairing is often repeated k times to ensure a robust evaluation, and care is taken to avoid performing the same evaluation more than once.

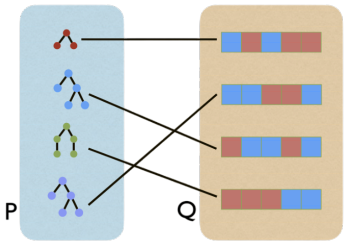


Figure 10.8: Shuffling and pairing individuals from two populations for evaluation.

2. **Comparison with Previous Generation:** To make the evolutionary process more stable and informative, individuals from one population at the current generation can be compared with individuals from the other population at the previous generation. This helps to smooth out abrupt changes and provides a more consistent selection pressure.

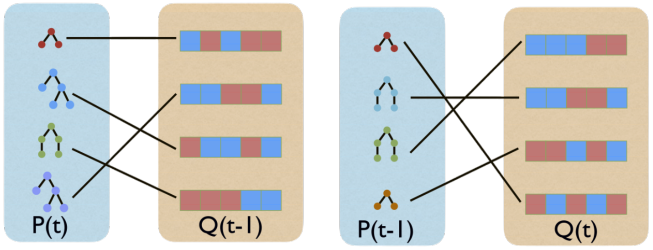


Figure 10.9: Comparing individuals with those from the previous generation.

3. **Combining Best and Random Opponents:** For each individual, it is possible to select k_1 opponents from the best individuals of the other population (from the previous generation) and k_2 opponents randomly. This hybrid approach balances exploitation (testing against the best) and exploration (testing against random opponents).

This multi-faceted evaluation strategy ensures that individuals are tested thoroughly and fairly, promoting robust coevolutionary progress in both populations.

Two-population competitive coevolution can suffer from "**loss of gradient**": If one population becomes much stronger than the other, the weaker population provides no useful feedback, and selection pressure is lost. In such cases, it may be helpful to pause the evolution of the stronger population, allowing the weaker one to "catch up."

10.2.3 N-Population Cooperative Coevolution

In some problems, a solution can be decomposed into multiple interacting sub-solutions, each with its own characteristics. Instead of evolving a single, very large individual, it can be more effective to have multiple interacting populations, each responsible for a different part of the solution. This approach is known as ***N-population cooperative coevolution***.

Some examples include robot soccer (where each player is a sub-solution), or any multiplayer game with more than one role. While it is always possible to create an enormous individual encoding all roles, using multiple populations allows for more flexibility and specialization.

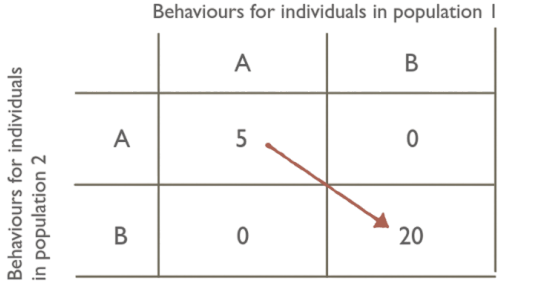
There are two main approaches to evaluating fitness in cooperative coevolution:

- **Parallel methods:** All populations evolve at the same time, and the same methods of evaluation as in competitive coevolution can be used. As usual, it is important to specify how individuals from different populations are paired for evaluation.
- **Sequential methods:** Populations are evolved one at a time (alternating optimization). This is less suitable for competitive coevolution, but can be used in some cooperative settings.

Possible Drawbacks

A key challenge in cooperative coevolution is ***lucky pairs***: an individual may appear to have good fitness simply because it is paired with strong individuals from other populations, even if it does not contribute to the global solution. E.g., in evolving a soccer team, a weak player may still have high fitness if the rest of the team is strong. Another drawback is ***overgeneralization***: individuals may become decent at everything, but not excel at anything. Their average performance is good, but they are not optimal for any specific scenario. A possible solution is to compute the fitness as the max or min (instead of the average) among all tests, but this is not a perfect solution.

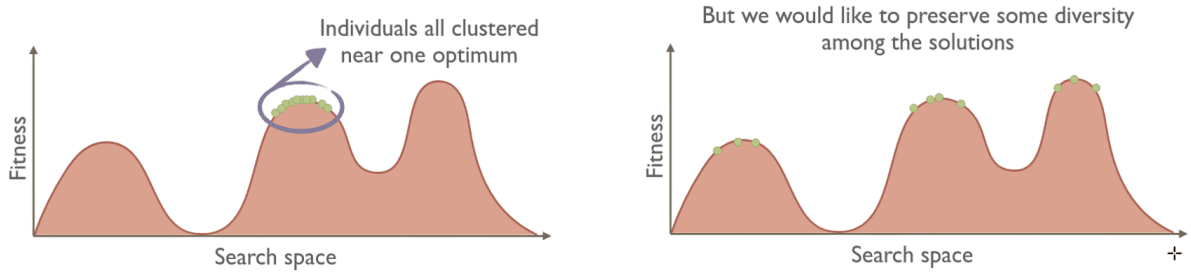
Additionally, moving from a local optimum may require multiple populations to change behavior at the same time, which can be difficult to achieve in practice.



To move from the local optimum 5 the two population must change behavior "at the same time"

Diversity Maintenance

Premature convergence to a sub-optimal solution is a common problem in optimization. One way to avoid this is to increase the population size or adjust algorithm parameters, but it is often more effective to enforce diversity directly within the population.



Diversity can be measured in several ways:

- **Genotype:** Similarity in the encoded structure (e.g., Hamming distance for bit strings).
- **Phenotype:** Similarity in behavior or solution, even if encoded differently.
- **Fitness:** Similarity in fitness values (useful in multi-objective optimization).

To maintain diversity, **fitness sharing** can be used. This technique addresses the tendency of populations to converge prematurely by penalizing individuals that cluster too closely together in the search space. Individuals that are too similar have their fitness reduced by having to "share" it with others. The degree of punishment depends on their similarity:

$$s(x,y) = \begin{cases} 1 - \left(\frac{d(x,y)}{\sigma}\right)^{\alpha} & \text{if } d(x,y) \leq \sigma \\ 0 & \text{otherwise} \end{cases}$$

where $d(x,y)$ is the *distance* between individuals x and y, σ is a *threshold* that defines the niche radius (the range within which individuals are considered similar), and $\alpha > 0$ controls the *degree of punishment* applied to similar individuals.

The shared fitness value is then computed as:

$$f(x) = \frac{r(x)^{\beta}}{\sum_y s(x,y)}$$

where $r(x)$ is the *raw fitness* and the denominator sums the sharing function over all individuals in the population. A *scaling factor* $\beta > 1$ can also be used to amplify fitness differences before sharing occurs.

The parameters α, β, and σ collectively control the ***strength*** and ***range*** of fitness sharing, allowing fine-tuning of the diversity pressure.

This approach encourages the population to spread out across different regions of the search space, creating natural niches where multiple solutions can coexist. By preventing the population from clustering around a single high-fitness region, fitness sharing helps to avoid premature convergence and maintain a diverse set of solutions that can continue to explore different areas of the problem landscape.

11 Multi-Objective Optimisation

12 Neuroevolution

Neuroevolution is a field of machine learning that uses evolutionary algorithms to design artificial neural networks (ANNs). While the dominant paradigm for training ANNs is gradient-based backpropagation, this method primarily optimizes the network's weights for a fixed architecture. Neuroevolution offers a broader approach, capable of evolving not only the weights but also the network's topology (its structure and connections) and other hyperparameters. This makes it a powerful tool for automating neural network design and tackling problems where gradient information is unavailable or unreliable, such as in reinforcement learning.

12.1 A Primer on Artificial Neural Networks

An Artificial Neural Network is a computational model inspired by the structure of biological brains. It is composed of interconnected nodes, called ***neurons***, which are typically organized into layers.
Each neuron receives inputs, performs a computation, and passes the result to other neurons. The computation within a single neuron involves three steps:

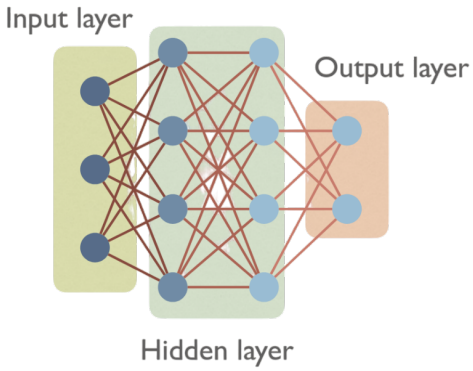


Figure 12.1: A simple feedforward neural network architecture.

1. **Weighted Sum:** The neuron calculates a weighted sum of its inputs.
2. **Bias:** A bias term b_i is added to the weighted sum.
3. **Activation Function:** A non-linear ***activation function*** f is applied to the result.

The output of a neuron i is thus given by $f(b_i + \sum_{j \to i} w_{ji} x_j)$, where w_{ji} are the weights of the connections from input neurons j.

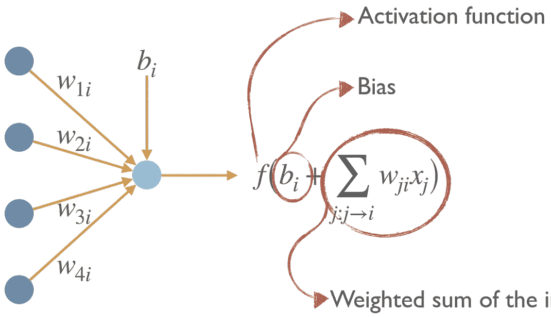


Figure 12.2: The computation performed by a single neuron.

While many networks use **fully connected layers**, where every neuron in one layer connects to every neuron in the next, other architectures are better suited for certain data types. **Convolutional layers**, for example, connect each neuron only to a small, local region of the previous layer—its ***receptive field***. All neurons in a convolutional layer share the same weights, which are applied as the receptive field slides across the input (with a given stride). This local connectivity and weight sharing make convolutional layers highly efficient and effective for spatial data like images, in contrast to the dense and unique connections of fully connected layers.

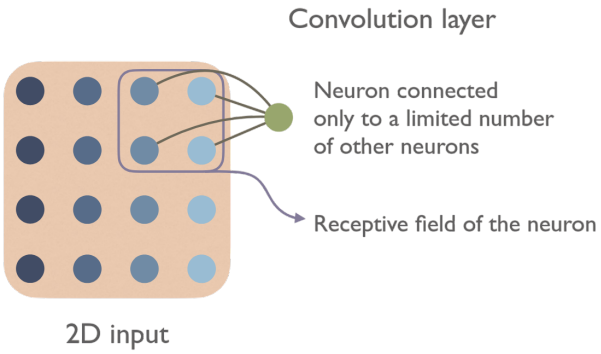


Figure 12.3: A partially connected neural network architecture.

12.1.1 Neural Network Training

The standard method for training a neural network is through an optimization process, typically ***(stochastic) gradient descent***, which iteratively adjusts the network's weights w_{ij} to minimize a loss function (the error). This process, known as ***backpropagation***, requires the loss function L and activation functions to be differentiable.

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$

where η is the learning rate.

However, backpropagation only learns the weights. The ***architecture*** of the network (the number, size, and type of layers) and other critical ***hyperparameters*** (e.g., learning rate, momentum) must be selected manually by the human designer. Neuroevolution provides a way to automate this entire design process.

12.2 Neuroevolution: Evolving Neural Networks

Neuroevolution applies evolutionary algorithms to the design of neural networks. Instead of manual design, the process aims to discover effective networks through evolution. An EA can be used to evolve various components of a neural network:

- **Weights:** As an alternative to backpropagation, especially in domains without gradients.
- **Hyperparameters:** Such as learning rates, momentum, or dropout rates.
- **Activation Functions:** Evolving the mathematical form of the activation functions.
- **Architecture (Topology):** Evolving the connections, layers, and types of layers that form the network structure.

Historically, "classic" neuroevolution (before the rise of deep learning) focused on evolving both the weights and topology of smaller networks. However, directly evolving every single weight and connection in modern, large-scale deep networks is often computationally infeasible, which has led to the development of more sophisticated encoding schemes.

12.2.1 Encoding Schemes

The choice of how to represent a neural network as a genotype is critical, as it determines how genetic operators like crossover and mutation can be applied. Several encoding schemes have been proposed.

- ***GENITOR Encoding:*** For a fixed network topology, each potential edge is encoded as a binary substring. One bit indicates the presence of the edge, while the remaining bits encode its weight. The entire network is represented by concatenating these substrings into a single, long binary string. This allows for standard GA operators but is inflexible as the number of neurons is fixed.
- ***Matrix Encoding:*** The network's connectivity is represented by an adjacency matrix, where each entry indicates the presence or absence of a connection. This allows the connections to be modified, but the number of neurons remains fixed. For large networks, the matrix becomes excessively large (n^2 for n neurons), making this approach not scalable.
- ***Node-Based Encoding:*** The main "unit" of the genome is the node. Each gene represents a neuron and contains information about its outgoing connections and their weights. This representation is flexible, as both new nodes and new connections can be created by adding to the list of genes. A similar encoding is used by the NEAT algorithm.
- ***Path Encoding:*** The network is represented as a collection of paths, where each path traces a route from an input neuron to an output neuron. The full network is reconstructed by overlaying all evolved paths. Recombination is typically done via two-point crossover on the lists of paths.

12.2.2 The Competing Conventions Problem

A fundamental challenge in evolving neural network topologies is the ***competing conventions problem*** (also known as the permutation problem). This occurs when two different genotypes encode networks that are functionally identical but have different structural representations.

For example, two networks might perform the same computation, but the hidden neurons that perform specific sub-functions appear in a different order. If a standard crossover operator is applied to these two parent genotypes, it is likely to combine parts that are not functionally compatible. The resulting offspring can be a meaningless combination of mismatched sub-networks, performing much worse than either parent.

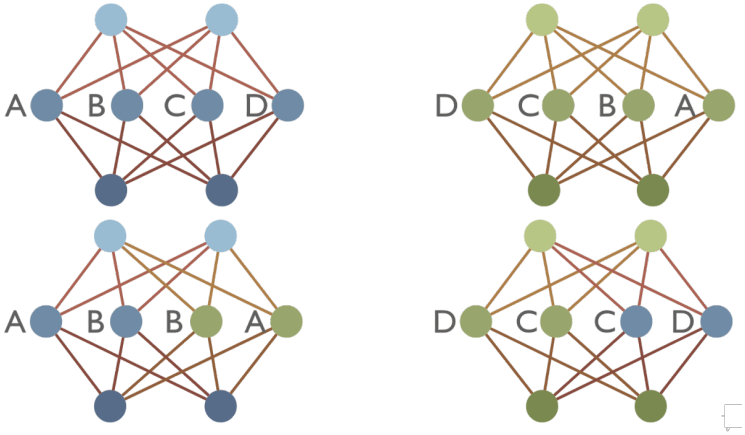


Figure 12.4: The Competing Conventions Problem. The two parent networks are functionally identical (isomorphic), but their hidden neuron functionalities are permuted.

Solving this problem is crucial for effective topological evolution. It requires a crossover mechanism that can intelligently align and combine functionally corresponding genes.

12.3 NEAT: NeuroEvolution of Augmenting Topologies

NEAT (NeuroEvolution of Augmenting Topologies) is a highly influential neuroevolution method specifically designed to solve the competing conventions problem and effectively evolve both network weights and topologies. Its key features are:

1. **Complexifying Evolution**: NEAT starts with a population of simple networks (initially with no hidden nodes) and gradually adds complexity (new nodes and connections) via mutation over generations.
2. **Innovation Numbers**: Each new structural gene (a new node or link) is assigned a globally unique ***innovation number***. These numbers serve as historical markers, allowing NEAT to track the lineage of genes.
3. **Intelligent Crossover**: By aligning genes with matching innovation numbers, NEAT can perform crossover between different topologies in a meaningful way, avoiding the competing conventions problem.
4. **Speciation**: The population is divided into ***species*** based on topological similarity. Mating occurs primarily within species, which protects new innovations from having to immediately compete with different, more established topologies, giving them time to optimize.

12.3.1 The NEAT Genome

The NEAT genotype is a direct encoding consisting of two types of genes:

- ***Neuron Genes***: A list of all neurons in the network. Each gene has an ID and a type (e.g., input, output, hidden).
- ***Link Genes***: A list of all connections. Each link gene specifies the source and destination neuron, the connection's weight, an enabled/disabled status bit, and its unique ***innovation number***.

Crucially, a connection can be present in the genome but be disabled, meaning it does not participate in the network's computation but can be re-enabled by a future mutation.

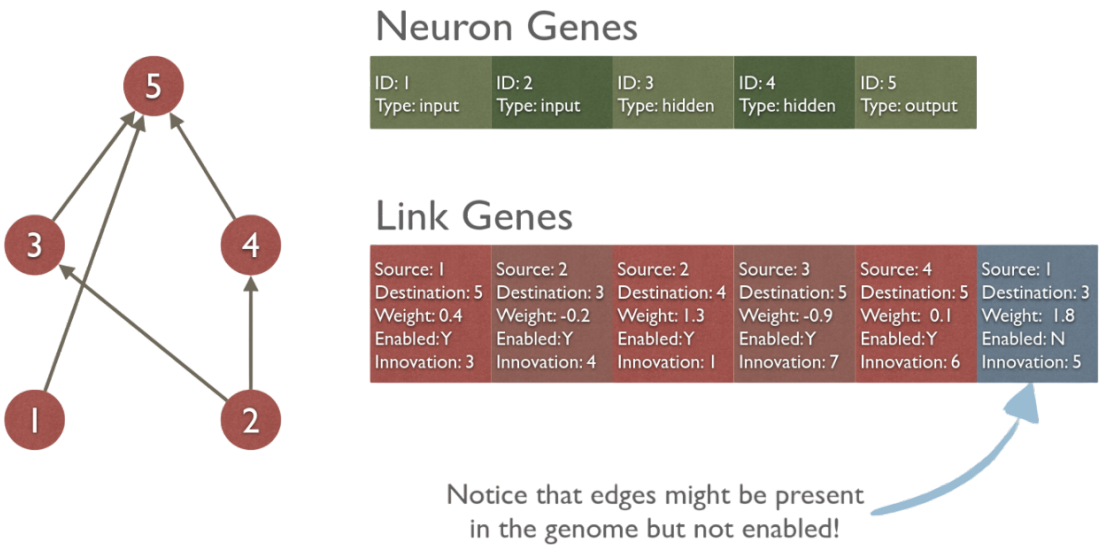


Figure 12.5: A NEAT genome, composed of neuron genes and link genes. Each link gene has an innovation number to track its historical origin.

12.3.2 Genetic Operators in NEAT

Mutation

NEAT uses a variety of mutation operators to introduce variation:

- **Add Connection**: A new link gene is created between two previously unconnected neurons.
- **Add Node**: An existing connection is ”split”. The old connection is disabled, and a new hidden node is inserted. Two new connections are created: one from the original source to the new node (with weight 1), and one from the new node to the original destination (with the old weight).
- **Weight Mutation**: The weights of existing connections can be either slightly perturbed or assigned a completely new random value.
- **Enable/Disable Mutation**: The status of a link gene is toggled.

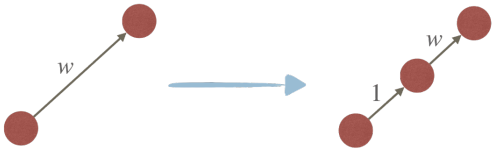


Figure 12.6: The ”add node” mutation in NEAT splits an existing connection.

Crossover and the Innovation Number

The **innovation number** is a unique identifier assigned to each new structural gene (connection or node) created by mutation in NEAT.

This number is incremented globally with every new mutation, ensuring that identical structural changes, such as adding the same connection or splitting the same link in different individuals, receive the same innovation number, even if they occur in different genomes or generations.

This mechanism is crucial for solving the competing conventions problem, as it allows NEAT to track the historical origin of each gene.

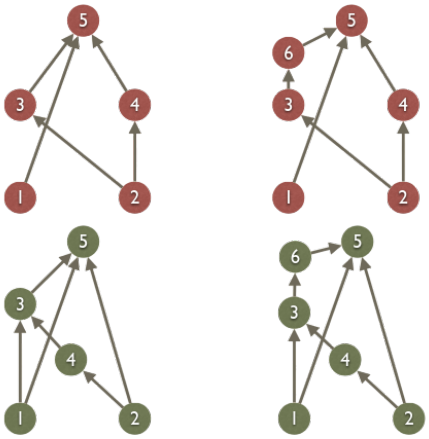


Figure 12.7: The same mutation in two different individuals receives the same innovation number.

During **crossover**, NEAT uses innovation numbers to align the link genes of two parent genomes. The process works as follows:

- ***Matching Genes***: Genes with the same innovation number in both parents are considered matching. For each matching gene, the offspring randomly inherits the gene from either parent.
- ***Disjoint Genes***: These are genes whose innovation numbers fall within the range of the other parent’s innovation numbers but are absent in that parent.
- ***Excess Genes***: These are genes whose innovation numbers are outside the range of the other parent’s innovation numbers.
- ***Inheritance Rule***: Disjoint and excess genes are inherited only from the ***fitter*** parent (the parent with higher fitness).

This alignment and inheritance strategy allows NEAT to recombine genomes with different topologies in a meaningful way, preserving useful structural innovations and ensuring that offspring are viable.

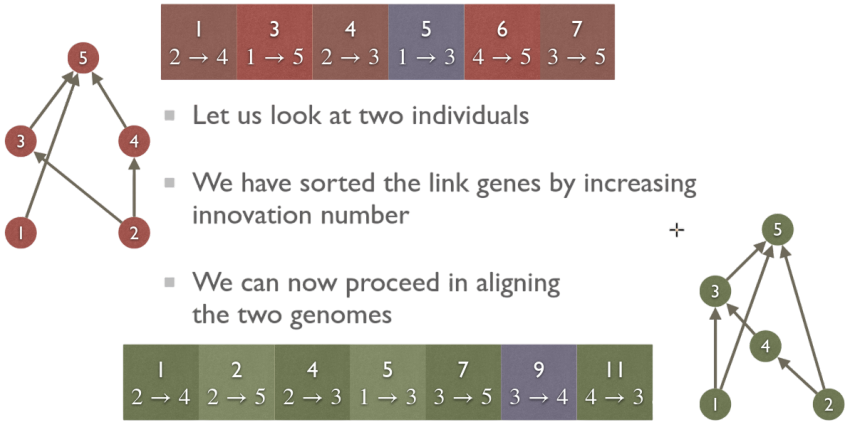


Figure 12.8: Example: Two individuals with their genes sorted by innovation number. Matching genes are aligned, while disjoint and excess genes are identified.

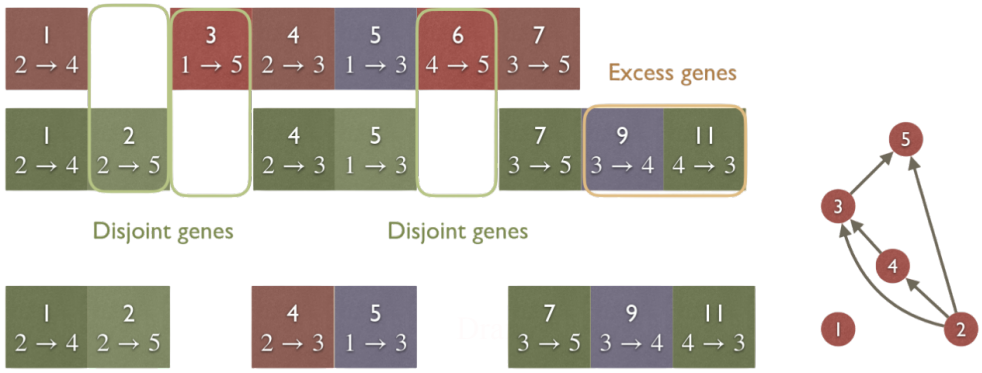


Figure 12.9: Crossover in NEAT: Matching genes (by innovation number) are inherited randomly from either parent. Disjoint and excess genes are inherited from the fitter parent only. This process enables the combination of different topologies while preserving functional structures.

Speciation

A **species** in NEAT is a group of individuals with similar genetic structure, allowing new structural innovations to be protected and optimized before facing competition from the entire population. Speciation in NEAT serves two main purposes:

- It protects new topological innovations by grouping similar individuals together, so that crossover and competition occur primarily within species.
- It prevents premature convergence by allowing diverse solutions to coexist.

Creation of Species

- **Compatibility Distance:** Two individuals will belong to the same species if their *compatibility distance* δ is less than a fixed threshold. The compatibility distance is calculated as:

$$\delta = c_1 \frac{E}{N} + c_2 \frac{D}{N} + c_3 \overline{W}$$

where E is the number of excess genes, D is the number of disjoint genes, $\overline{W}$ is the average weight difference of matching genes, N is the number of genes in the larger genome (set to 1 if both genomes are small), and c_1, c_2, c_3 are coefficients that weight each term.

- **Assignment:** Each existing species is represented by a random genome (the representative) from the previous generation. Each new individual is compared to the representatives, and is assigned to the first compatible species found. If it is not compatible with any existing species, it forms a new species and becomes its representative.

To prevent a single species from dominating the population, NEAT uses **fitness sharing**:

- The raw fitness f_i of each individual i is divided by the number of individuals $|S_i|$ in its species:

$$f_i' = \frac{f_i}{|S_i|}$$

- The number of offspring allocated to each species is proportional to the sum of the adjusted fitnesses of its members.

This mechanism ensures that smaller, innovative species are not overwhelmed by larger, established ones, allowing new structures to mature and compete effectively.

12.4 Indirect Encodings for Large Networks

Direct encodings, such as those used in NEAT, become impractical for modern deep neural networks with millions of parameters. ***Indirect encodings*** address this challenge by evolving a compact set of rules or a generative program that constructs the final, large-scale network. This approach enables the automatic discovery of regularities, symmetries, and repeating patterns.

There are several indirect encoding methods for large networks. The most prominent ones are:

- **Grammar-Based Encoding**
  A grammar (such as a context-free grammar) is evolved to generate the structure of a neural network. The grammar can produce an adjacency matrix or a sequence of construction rules, allowing for the compact specification of complex, regular architectures.

$$S \mapsto \begin{bmatrix} A & B \\ A & C \end{bmatrix} \qquad A \mapsto \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad B \mapsto \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \qquad C \mapsto \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

  This is the same we saw in Section 4.6.3, but here the matrices represent neural networks.
- **Bi-Dimensional Growth Encoding (L-Systems)**
  Bi-dimensional growth encoding uses ***L-systems*** (Lindenmayer systems), recursive rewriting rules originally designed to model plant growth, to generate complex and regular neural connectivity patterns in 2D space.
  Each neuron has an L-system that recursively defines its branching structure (number, angle, and length of branches). Synaptic connections are determined by where branches from different neurons intersect or come close.

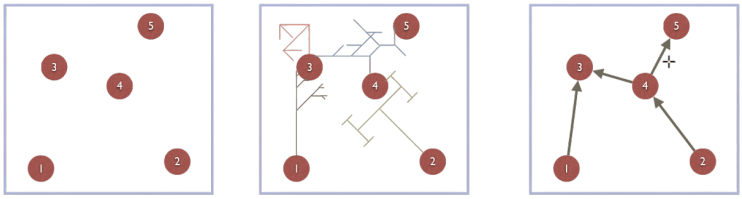


Figure 12.10: Illustration of bi-dimensional growth encoding using L-systems.

  This approach enables the automatic emergence of repeating motifs and spatial regularities that are hard to achieve with direct encodings.

- **HyperNEAT**
  HyperNEAT is a prominent indirect encoding technique that evolves a ***Compositional Pattern-Producing Network (CPPN)***: a small neural network whose inputs are the geometric coordinates of two neurons in the target network (the ”substrate”). The CPPN outputs the weight of the connection between those neurons.

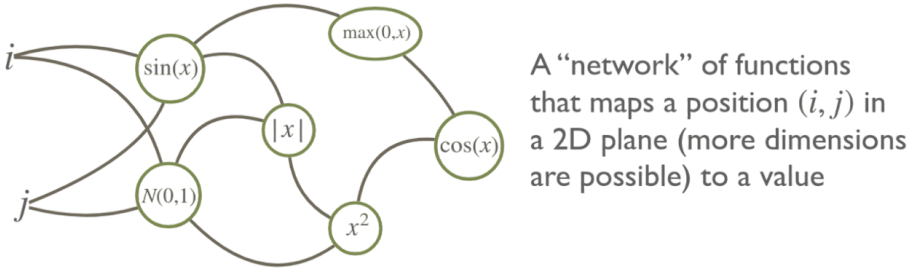


Figure 12.11: CPPN representation of a neural network.

By systematically querying the CPPN for all possible pairs of coordinates, HyperNEAT can generate large, structured neural networks from a compact genome. The CPPN itself is evolved using NEAT, enabling the discovery of geometric and functional regularities in the connectivity pattern.

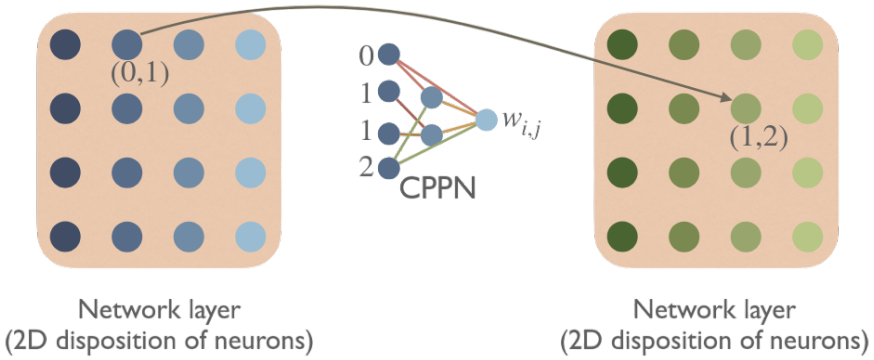


Figure 12.12: HyperNEAT uses a CPPN to generate the network structure.

- **DENSER**
  DENSER (Deep Evolutionary Network Structured Representation) is a method for evolving deep neural network architectures using a hierarchical, layer-based approach. Both the number, type, and parameters of the layers are evolved, while the network weights are learned via backpropagation.
  The process is organized into two evolutionary levels:
  – **Top level:** A genetic algorithm (GA) encodes the overall sequence and general types of layers (such as convolutional, pooling, or fully connected).
  – **Lower level:** For each layer, grammatical evolution (GE), specifically, dynamic structured GE, is used to generate the detailed parameters (e.g., filter size, activation function, etc.).
  Only the best-performing networks are trained for more than a few epochs, focusing computational resources on the most promising candidates. This approach enables the automatic discovery of effective and diverse deep network architectures.

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] Martin Pelikan et al. "BOA: the Bayesian optimization algorithm". In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*. GECCO'99. Orlando, Florida: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1558606114.

[3] 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.

[4] Jingqiao Zhang et al. "JADE: Adaptive Differential Evolution With Optional External Archive". In: *IEEE Transactions on Evolutionary Computation* 13.5 (2009), pp. 945–958. DOI: 10.1109/TEVC.2009.2014613.