Chapter 5: Evolution Strategies and Genetic Algorithm

CS-616: Optimization Algorithms

Dr. Mahdi A. Khemakhem & Dr. Esraa M. Eldesouky

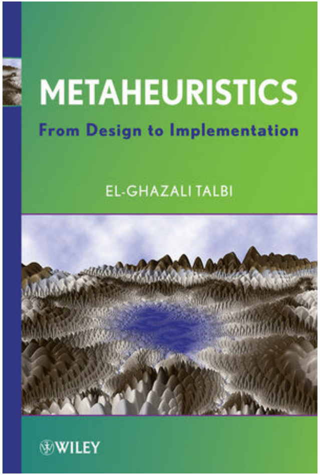
Department of Computer Science
College of Computer Engineering and Science
Prince Sattam bin Abdulaziz University

AY: 2025/2026

جامعة الأمير
سطام بن عبدالعزيز
PRINCE SATTAM BIN ABDULAZIZ UNIVERSITY

Materials

- **Textbook:** ”Metaheuristics: From Design to Implementation” by *El-Ghazali Talbi*
- ➥Read Chapter 3 (Sections 3.2 and 3.3) for this chapter’s material



Outline

Outline

Population-based Methods (1/2)

- Population-based Metaheuristics differ from the previous Single-Solution based Metaheuristics in that they keep around a sample of candidate solutions rather than a single candidate solution.
- Each of the solutions is involved in Tweaking (changing and switching) and quality assessment.
- Most Population-based Metaheuristics inspire concepts from biology.
- One particularly popular set of techniques, collectively known as Evolutionary Computation (EC), borrows liberally from population biology, genetics, and evolution.
  - Evolutionary Algorithm (EA) is one of the popular algorithms in the EC collection.
  - The majority of EAs can be categorized into two types: generational algorithms, which update the entire population once per iteration, and steady-state algorithms, which update the population incrementally by replacing a few candidate solutions at a time.
  - Among the familiar EAs are the Evolution Strategies (ES) and Genetic Algorithm (GA).

Population-based Methods (2/2)

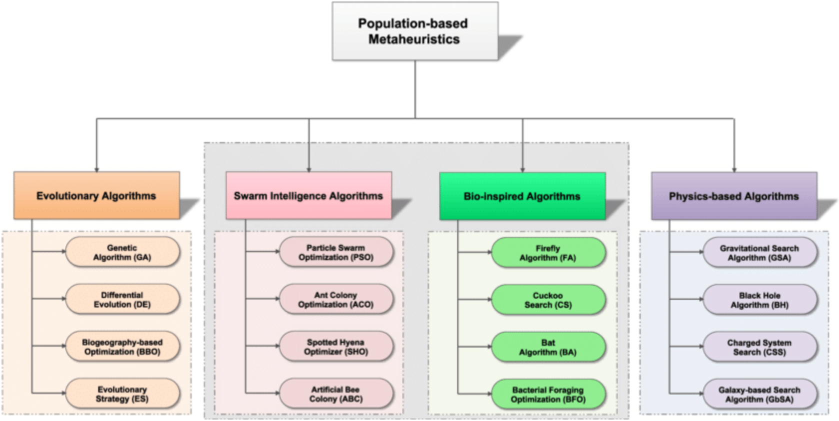


Figure sourced from: **Dhiman, Gaurav. "ESA: a hybrid bio-inspired metaheuristic optimization approach for engineering problems." Published in Engineering with Computers, volume 37 (2021), pages 323-353.**

Evolutionary Computation (EC) (1/3)

Because they are inspired by biology, EC methods tend to use (and abuse) terms from genetics and evolution. The following table presents the meaning of the Common Terms Used in Evolutionary Computation.

Terms	Meaning
Individual	A candidate (potential) solution.
Child and Parent	A *child* is the tweaked (modified) copy of a candidate solution (its *parent*).
Population	Set of candidate solutions.
Fitness	Quality or effectiveness.
Fitness Landscape	Function that measures fitness.
Fitness Assessment or Evaluation	Determining how effective an individual is.
Selection	Choosing potential solutions based on their effectiveness.
Mutation	Simple modification (tweaking) of a candidate, similar to a genetic mutation.

Evolutionary Computation (EC) (2/3)

Terms	Meaning
Crossover or **Recombination**	A special Tweak which takes two parents, swaps sections of them, and (usually) produces two children.
Breeding	Producing one or more children from a population of parents through an iterated process of selection and Tweaking (typically mutation or recombination)
Genotype or **Genome**	The genetic makeup (data structure) of an individual.
Chromosome	A representation of the genetic makeup.
Gene	A specific part of the genetic makeup.
Allele	A particular setting of a gene.
Phenotype	How an individual behaves or performs
Generation	One cycle of evaluating, modifying, and regenerating potential solutions.

Evolutionary Computation (EC) (3/3)

- EC techniques are generally resampling techniques: new samples (populations) are generated or revised based on the results from older ones.
- The basic generational EC algorithm first constructs an initial population, then iterates through three procedures.
  - First, it assesses the fitness of all the individuals in the population.
  - Second, it uses this fitness information to breed a new population of children.
  - Third, it joins the parents and children in some fashion to form a new next-generation population, and the cycle continues.

Evolutionary Algorithm (EA) (1/3)

Algorithm 1 shows the general schema of a Generational Evolutionary Algorithm (EA)

Algorithm 1: Template of a Generational Evolutionary Algorithm (EA)

```
Input: Initial population P; /* Built by creating some n individuals at random.   */
Output: Best solution found Best
Best = ∅; /* ∅ means "nobody yet"                                                  */
repeat
    AssessFitness(P);
    for each individual P_i ∈ P do
        if Best == ∅ OR Fitness(P_i)<Fitness(Best) then ; /* Fitness is just Quality */

            Best = P_i; /* Update the best known solution Best if improved by P_i   */
        end
    end
    P = Join(P, Breed(P));
until Best is the ideal solution or we have run out of time;
```

Evolutionary Algorithm (EA) (2/3)

- Notice that, unlike the Single-Solution Based Metaheuristics, we now have a separate `AssessFitness` function. This is because typically we need all the fitness values of all individuals before we can `Breed` them.
- Evolutionary Algorithms differ from one another largely in how they perform the `Breed` and `Join` operations.
- The `Breed` operation usually has two parts:
  - Selecting parents from the old population. then
  - Tweaking them (usually Mutating or Recombining them in some way) to make children.
- The `Join` operation usually either
  - Completely replaces the parents with the children. or
  - Includes able-bodied parents along with their children to form the next generation.

Evolutionary Algorithm (EA) (3/3)

Population Initialization

The Population Initialization typically involves creating a set of individuals to explore the solution space effectively. There are several common strategies for population initialization:

- **Random Initialization:** This approach involves randomly generating individuals within the solution space.
- **Uniform Initialization:** In this method, each parameter of an individual is initialized uniformly within its specified bounds (ensure even distribution).
- **Latin Hypercube Sampling:** Latin hypercube sampling divides the solution space into equally sized regions and selects one point randomly within each region (ensure diversification).
- **Heuristic Initialization:** Heuristic methods use problem-specific knowledge or heuristics to generate initial solutions.

⇒The choice of population initialization strategy can significantly impact the performance and convergence properties of the evolutionary algorithm.

Outline

Evolution Strategies (ES)

- The family of algorithms known as **Evolution Strategies (ES)** were developed by **Ingo Rechenberg** and **Hans-Paul Schwefel** at the Technical University of Berlin in the mid 1960s.
- ES algorithms employ a simple procedure for **selecting individuals** called `TruncationSelection`, and (usually) only uses `Mutation` as the **Tweak operator**.

(μ, λ) Evolution Strategy Algorithm (1/5)

- Among the simplest ES algorithms is the (μ, λ).
- μ: Number of parents which survive.
- λ: Number of children that the μ parents make in total (should be a multiple of μ).
- ES practitioners refer to their algorithm by the choice of μ and λ. For example, if $\mu = 5$ and $\lambda = 20$, then we have a (5, 20) Evolution Strategy.
- It begins with a population of λ individuals, generated randomly.
- It iterates as follows:
  1. Assess the fitness of all individuals.
  2. Retain only the μ fittest individuals (`TruncationSelection`).
  3. Each of the μ fittest individuals reproduces λ/μ children through standard Mutation (`Mutate`).
  4. Replace parents with offspring (children).
  5. Repeat the process for the next iteration.

(μ, λ) Evolution Strategy Algorithm (2/5)

Algorithm 2: Template of the (μ, λ) Evolution Strategy Algorithm

```
Input: μ and λ; /* Numbers of parents selected and children generated by the parents. */
Output: Best solution found Best
P = ∅; /* ∅ means "nobody yet" */
for λ times do ; /* Build Initial Population */

    P = P∪{new random individual}
end
Best = ∅; /* ∅ means "nobody yet" */
repeat
    for each individual P_i ∈ P do
        AssessFitness(P_i);
        if Best == ∅ OR Fitness(P_i)<Fitness(Best) then
            Best = P_i; /* Update the best known solution Best if improved by P_i */
        end
    end
    Q = the μ individuals in P whose Fitness() are smallest; /* Truncation Selection */
    P = {} ; /* Join is done by just replacing P with the children */
    for each individual Q_j ∈ Q do
        for λ/μ times do
            P = P∪ {Mutate(Copy(Q_j))}
        end
    end
until Best is the ideal solution or we have run out of time;
```

(μ, λ) Evolution Strategy Algorithm (3/5)

Crossover and Mutation

- Note the use of the function `Mutate` instead of `Tweak` in Algorithm 2.
- Recall that Population-based Metaheuristics offer various ways to perform the `Tweak` operation:
  - Mutation: Similar to Tweaks seen before, it involves converting a single individual into a new individual through a (usually small) random change.
  - Crossover or Recombination: Involves mixing and matching multiple (typically two) individuals to form children.

⇒We will use the terms Mutation and Crossover henceforth to indicate the Tweak operation performed.

(μ, λ) Evolution Strategy Algorithm (4/5)

Exploration vs. Exploitation (1/2)

The (μ, λ) algorithm offers three adjustable parameters that allow us to balance exploration and exploitation.

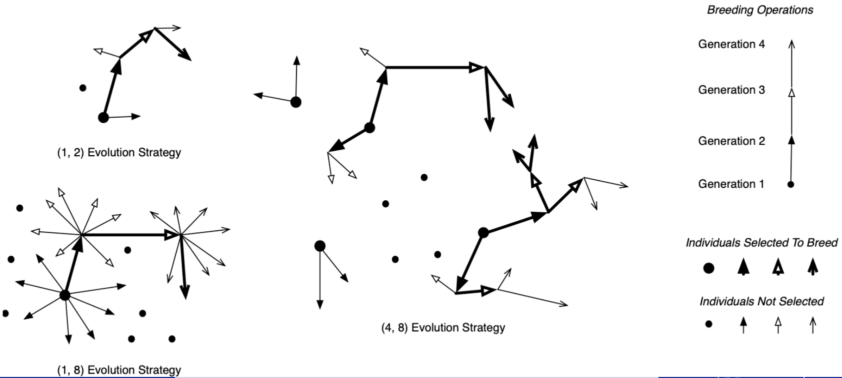
- **The size of λ:** This essentially controls the sample size for each population. At the extreme, as λ approaches ∞, the algorithm approaches exploration.
- **The size of μ:** This controls how selective the algorithm is; low values of μ with respect to λ push the algorithm more towards exploitative search as only the best individuals survive.
- **The extent of Mutation:** When `Mutate` introduces significant noise, offspring deviate considerably from their parents, exhibiting a high degree of randomness irrespective of the selectivity of μ.

(μ, λ) Evolution Strategy Algorithm (5/5)

Exploration vs. Exploitation (2/2)

The following shows the effect of variations with these operations.

Three (μ, λ) Evolution Strategy variations. Each generation, μ individuals are selected to breed, and each gets to create λ/μ children, resulting in λ children in total.



$(\mu + \lambda)$ Evolution Strategy Algorithm (1/3)

- The second Evolution Strategy algorithm is called $(\mu + \lambda)$ algorithm.
- It differs from (μ, λ) algorithm in only one respect: the `Join` operation.
  - Recall that in (μ, λ) algorithm the parents are simply replaced with the children in the next generation.
  - In $(\mu + \lambda)$ algorithm, the next generation consists of the μ parents plus the λ new children.
  - That is, the parents compete with the kids next time around. Thus the next and all successive generations are $\mu + \lambda$ in size.

$(\mu + \lambda)$ Evolution Strategy Algorithm (2/3)

Algorithm 3: Template of the $(\mu + \lambda)$ Evolution Strategy Algorithm

```
Input: μ and λ; /* Numbers of parents selected and children generated by the parents. */
Output: Best solution found Best
P = ∅; /* ∅ means "nobody yet" */
for λ times do ; /* Build Initial Population */

    P = P ∪ {new random individual}
end
Best = ∅; /* ∅ means "nobody yet" */
repeat
    for each individual P_i ∈ P do
        AssessFitness(P_i);
        if Best == ∅ OR Fitness(P_i) < Fitness(Best) then
            Best = P_i; /* Update the best known solution Best if improved by P_i */
        end
    end
    Q = the μ individuals in P whose Fitness() are smallest; /* Truncation Selection */
    P = Q ; /* The Join operation is the only difference with (μ, λ) */
    for each individual Q_j ∈ Q do
        for λ/μ times do
            P = P ∪ {Mutate(Copy(Q_j))}
        end
    end
until Best is the ideal solution or we have run out of time;
```

$(\mu + \lambda)$ Evolution Strategy Algorithm (3/3)

- Generally speaking, $(\mu + \lambda)$ algorithm may be more exploitative than (μ, λ) algorithm because able-bodied parents persist to compete with the children.
- This has risks:
  - Scenario: A highly fit (able-bodied) parent may dominate the population repeatedly.
  - Consequence: This dominance can lead the entire population to prematurely converge towards descendants of that parent.
  - Result: The population becomes trapped in a local optimum surrounding the dominating parent.

Outline

Genetic Algorithm (GA) (1/3)

- The Genetic Algorithm (GA) was invented by John Holland at the University of Michigan in the 1970s.
- It is similar to a (μ, λ) Evolution Strategy in many respects: It iterates through fitness assessment, selection and breeding, and population reassembly.
- The primary difference is in how selection and breeding take place: whereas Evolution Strategies select the parents and then creates the children, the Genetic Algorithm little-by-little selects a few parents and generates children until enough children have been created.
- To breed,
  - It begins with an empty population of children.
  - It then selects two parents from the original population, copy them, cross them over with one another, and mutate the results.
  - This forms two children, which they then added to the child population.
  - This process is repeated until the child population is entirely filled.

Genetic Algorithm (GA) (2/3)

Algorithm 4 shows the general schema of the Genetic Algorithm

Algorithm 4: Template of the Genetic Algorithm

```
Input: popsize; /* This is basically λ. Make it even.                */
Output: Best solution found Best
P = ∅;
for popsize times do
    P = P∪{new random individual}
end
Best = ∅;
repeat
    for each individual P_i ∈ P do
        AssessFitness(P_i);
        if Best == ∅ OR Fitness(P_i)<Fitness(Best) then
            Best = P_i;
        end
    end
    Q = {};
    for popsize/2 times do
        Parent P_a = Select(P);
        Parent P_b = Select(P);
        Children C_a, C_b = Crossover(Copy(P_a), Copy(P_b));
        Q = Q∪ {Mutate(C_a), Mutate(C_b)}; /* End of deviation       */
    end
    P = Q;
until Best is the ideal solution or we have run out of time;
```

Genetic Algorithm (GA) (3/3)

Solution Representation

- To represent individuals, the classical Genetic Algorithm (GA) works with fixed-length boolean vectors (arrays).
- Algorithm 5 shows how GA can generate random boolean vector.

Algorithm 5: Template of Generate a Random Bit-Vector

Input: ℓ ; /* Vector size */
Output: Boolean vector v of size ℓ

$v =$ and new empty vector $< v_1, v_2, \ldots, v_\ell >$;
for $i = 1$ **to** ℓ **do**
    $r =$ a random number chosen uniformly between 0.0 and 1.0 inclusive;
    if $r < 0.5$ **then**
        $v_i =$ true;
    else
        $v_i =$ false;
    end
end

Crossover and Mutation (1/4)

Note how similar the Genetic Algorithm is to (μ, λ), except during the breeding phase. To perform breeding, we need two new functions we've not seen before: `Crossover` and `Mutate`.

Mutation

- A simple way to `Mutate` a boolean vector is bit-flip mutation: march down the vector, and flip a coin of a certain probability (see Algorithm 6)

Algorithm 6: Template of Bit-Flip Mutation

Input: p, $v =< v_1, v_2, \dots, v_\ell >$; /* p:probability of flipping a bit (often is set to $1/\ell$), v: boolean vector to be mutated */

Output: Mutated boolean vector v

for $i = 1$ **to** ℓ **do**
    r = a random number chosen uniformly between 0.0 and 1.0 inclusive;
    if $r < p$ **then**
        $v_i = \overline{v_i}$; /* $\overline{v_i}$: Complementary of v_i */
    end
end

Crossover and Mutation (2/4)

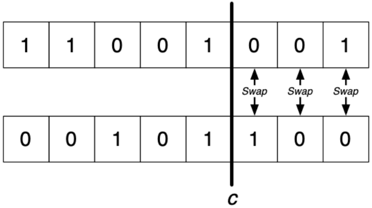
Crossover

- Crossover is the GA's distinguishing feature.
- It involves mixing and matching parts of two parents to form children.
- How to do that mixing and matching depends on the representation of the individuals.
- There are two classic (basic) ways of doing crossover in vectors: One-Point and Two-Point.

Crossover and Mutation (3/4)

One-Point Crossover

Let's say the vector is of length ℓ. One-point crossover picks a number c between 1 and ℓ, inclusive, and swaps all the indexes greater than c, as shown by the Figure on the right-hand side and Algorithm 7.



Algorithm 7: Template of One-Point Crossover

Input: $v =< v_1, v_2, \ldots, v_\ell >$, $w =< w_1, w_2, \ldots, w_\ell >$; /* v, w: vectors to be crossed over */
Output: **Corssedover vectors v and w**

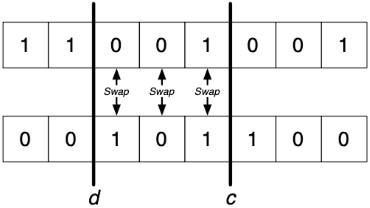
$c =$ **a random integer chosen uniformly between 1 and ℓ inclusive;**
for $i = c$ **to** ℓ **do**
  Swap the values of v_i and w_i;
end

⇒Note that if $c = 1$ or $c = \ell$, no crossover really happens: if $c = 1$ then everything crosses over and if $c = \ell$ nothing crosses over. In either case, the individuals don't change.

Crossover and Mutation (4/4)

Two-Point Crossover

Let's say the vector is of length ℓ. Two-point crossover picks two number c and d between 1 and ℓ, inclusive, and swaps all the indexes between d and c, as shown by the Figure on the right-hand side and Algorithm 8.



Algorithm 8: Template of Two-Point Crossover

```
Input: v =< v_1, v_2, ..., v_ℓ >, w =< w_1, w_2, ..., w_ℓ > ; /* v, w:  vectors to be crossed over */
Output: Corssedover vectors v and w
c = a random integer chosen uniformly between 1 and ℓ inclusive;
d = a random integer chosen uniformly between 1 and ℓ inclusive;
if c < d then
    Swap c and d;
end
for i = d to c do
    Swap the values of v_i and w_i;
end
```

The End