Memetic Algorithm (MA)

A memetic algorithm (MA) is an evolutionary optimization method that extends traditional genetic algorithms (GAs) by incorporating a local search (or refinement) process into the evolutionary cycle. It is inspired by the concept of memes in cultural evolution—ideas that evolve and improve as they spread within a population. In a memetic algorithm, individuals not only undergo global exploration (via crossover and mutation) but also benefit from local exploitation that refines their solutions. This combination often leads to faster convergence and higher-quality solutions, particularly on complex, multimodal problems.

1. Rationale Behind the Memetic Algorithm

Global vs. Local Search

- **Global Search (Exploration):**
  Like in standard GAs, a population of candidate solutions is evolved using operators such as selection, crossover, and mutation. This global process helps explore the vast search space and prevents premature convergence by maintaining diversity.

- **Local Search (Exploitation):**
  Each individual (or a subset of individuals) is further refined using a local search procedure. This step "exploits" the neighborhood of a solution to find a nearby, possibly improved, solution. This hybridization is especially powerful for problems where a good solution lies in a "basin of attraction" around a candidate, such as combinatorial optimization problems.

The Motivation

- **Quality and Convergence:**
  By combining a global search with local improvement, MAs can converge to high-quality solutions more quickly than pure evolutionary algorithms. They balance the exploration–exploitation trade-off more effectively.

- **Avoiding Local Optima:**
  The evolutionary process helps jump out of local optima, while local search fine-tunes promising candidates.

- **Cultural Evolution Analogy:**
  The term "memetic" comes from the idea that cultural information (memes) spreads and adapts in a population much like genes. In MAs, each candidate solution is "educated" (improved) locally, and then these improved traits are recombined in subsequent generations.

2. Detailed Pseudocode

Below is a step-by-step pseudocode that outlines a typical memetic algorithm. The pseudocode highlights the integration of local search within a genetic framework.

```
Algorithm MemeticAlgorithm
    Input: Population size N, maximum generations G_max,
           crossover rate p_c, mutation rate p_m,
           local search probability p_ls
    Output: Best solution found

    // Step 1: Initialize Population
    P ← GenerateInitialPopulation(N)
    EvaluateFitness(P)

    // Step 2: Main Loop
    for generation = 1 to G_max do
        // Optionally, perform local search on individuals
        for each individual x in P do
            if Random(0, 1) < p_ls then
                x' ← LocalSearch(x)
                // Accept the improved solution if it has better fitness
                if Fitness(x') > Fitness(x) then
                    x ← x'
                end if
            end if
        end for

        // Step 3: Create Offspring via Selection, Crossover, and Mutation
        Q ← Ø
        while Size(Q) < N do
            // Selection: e.g., roulette wheel or tournament selection
            parent1 ← SelectIndividual(P)
            parent2 ← SelectIndividual(P)

            // Crossover
            if Random(0, 1) < p_c then
                [child1, child2] ← Crossover(parent1, parent2)
            else
                child1 ← parent1
                child2 ← parent2
            end if

            // Mutation
            child1 ← Mutate(child1, p_m)
            child2 ← Mutate(child2, p_m)

            // Evaluate fitness of offspring
            EvaluateFitness(child1)
            EvaluateFitness(child2)

            // Optionally, apply local search on offspring
            if Random(0, 1) < p_ls then
                child1 ← LocalSearch(child1)
                child2 ← LocalSearch(child2)
            end if

            Q ← Q ∪ {child1, child2}
        end while

        // Step 4: Population Replacement
        P ← ReplacePopulation(P, Q)
    end for

    return BestIndividual(P)
End Algorithm
```

Explanation of Key Steps

- **Population Initialization:**
  A set of N candidate solutions is generated randomly or using a heuristic. Their fitness is then evaluated by the objective function $f(x)$.

- **Local Search:**
  The local search operator, denoted as $LS(x)$, explores the neighborhood of an individual x to find a local improvement. For example, in continuous optimization, this could be a gradient descent step; in combinatorial problems like the Traveling Salesman Problem (TSP), it might be a 2-opt swap.

- **Selection:**
  Candidates are selected based on their fitness. A common method is roulette wheel selection, where the probability of selecting an individual x_i is:

  $$p_i = \frac{f(x_i)}{\sum_{j=1}^{N} f\left(x_j\right)}$$

  This ensures that individuals with higher fitness are more likely to be chosen.

- **Crossover and Mutation:**
  Genetic operators combine and slightly modify the candidate solutions to explore new areas of the search space. The crossover operator recombines parts of two parent solutions, while mutation introduces random changes.

- **Population Replacement:**
  A new population is formed by selecting from the offspring (and sometimes by combining with current individuals) for the next generation.

3. Equations and Their Components

Fitness Function

The fitness function $f(x)$ evaluates the quality of a candidate solution x. It is problem-dependent:

- **Example (Maximization):**
$$f(x) = \text{ObjectiveValue}(x)$$
- **Example (Minimization):**
Often converted to a maximization problem by using:
$$f(x) = \frac{1}{1 + \text{Cost}(x)}$$

Selection Probability

For roulette wheel selection:

$$p_i = \frac{f(x_i)}{\sum_{j=1}^{N} f\left(x_j\right)}$$

- $f(x_i)$**:** Fitness of individual i.
- $\sum_{j=1}^{N} f\left(x_j\right)$**:** Sum of fitness values of all N individuals.
- p_i**:** The probability that individual i is selected.

Local Search Improvement Operator

Let $LS(x)$ denote a local search operator. The new solution after applying local search is:

$$x' = LS(x)$$

- x**:** Original candidate solution.
- x'**:** Improved candidate after local search.
- **Local Search Criteria:** The algorithm accepts x' if:
$$f(x') > f(x)$$
This condition ensures that only improvements are retained.

Mutation Operator

If a mutation operator alters x to produce x'', this can be represented as:

$$x'' = x + \Delta$$

- Δ**:** A small random change vector (or perturbation) applied to x.

Imperialist Competitive Algorithm (ICA)

The Imperialist Competitive Algorithm (ICA) is a population-based metaheuristic inspired by the socio-political process of imperialistic competition. In ICA, candidate solutions are envisioned as “countries” that form empires, with the best countries acting as imperialists and the others as colonies. Over successive iterations, colonies are assimilated toward their imperialist (mimicking cultural or technological assimilation), while random “revolution” operations introduce diversity. Meanwhile, empires compete for colonies—weak empires lose colonies to stronger ones until only one (or a few) powerful empire remains, representing the best solution(s).

Below is a detailed breakdown of the rationale, equations, pseudocode, and an example.

1. Rationale Behind ICA

ICA is built on two main principles:

- **Assimilation:** Colonies move toward their imperialist, analogous to how colonies adopt the cultural and technological traits of their dominant nation. This step exploits the search space by guiding candidate solutions toward better regions.

- **Competition:** Just as empires compete for resources and influence, weaker empires lose colonies to stronger ones. This process helps the algorithm escape local optima by periodically restructuring the population and allowing promising regions of the search space to expand.

Together, these processes balance exploration (via revolution and competition) and exploitation (through assimilation), making ICA a robust tool for solving complex optimization problems.

2. Detailed Equations and Their Explanations

2.1. Assimilation Equation

During the assimilation step, each colony is moved closer to its imperialist. A common formulation is:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \beta \cdot \left(\mathbf{x}_{\text{imperialist}} - \mathbf{x}_{\text{old}}\right) + d \cdot \tan(\theta)$$

where:

- $\mathbf{x}_{\text{new}}$: New position (vector) of the colony.
- $\mathbf{x}_{\text{old}}$: Current position of the colony.
- $\mathbf{x}_{\text{imperialist}}$: Position of the imperialist country.

- β**:** Assimilation coefficient (a positive constant or random factor controlling the step size toward the imperialist).
- d**:** Distance between the colony and the imperialist, i.e., $d = \| \mathbf{x}_{\text{imperialist}} - \mathbf{x}_{\text{old}} \|$.
- θ**:** A small random angle (in multidimensional problems, this introduces a deviation perpendicular to the direct line toward the imperialist) that helps maintain exploration.

Explanation:
This equation moves a colony toward its imperialist, but the added deviation term prevents the search from becoming too deterministic, thereby improving the chance of escaping local optima.

2.2. Empire Total Cost Equation

Each empire's strength is evaluated not only by its imperialist's cost but also by the performance of its colonies. A typical formulation is:

$$T = c_{\text{imperialist}} + \xi \cdot \frac{1}{n} \sum_{i=1}^{n} c_i$$

where:

- T**:** Total cost of the empire.
- $c_{\text{imperialist}}$**:** Cost (or fitness) of the imperialist.
- c_i**:** Cost of the ith colony in the empire.
- n**:** Number of colonies in the empire.
- ξ**:** A weighting factor (typically in the range [0, 1]) that scales the influence of the colonies' performance relative to the imperialist's performance.

Explanation:
This total cost helps compare different empires; even if an empire's imperialist is strong, a large average cost among its colonies may indicate that the empire is overall weak and prone to losing colonies in competition.

2.3. Imperialist Power and Colony Allocation

To decide how many colonies each imperialist receives during initialization, we first compute a measure of power:

$$\text{Power}_i = c_{\max} - c_i$$

for each imperialist i, where:

- c_i**:** Cost of the ith imperialist.
- $c_{\max}$**:** The worst (largest) cost among the imperialists.

Next, the normalized power is:

$$P_i = \frac{\text{Power}_i}{\sum_j \text{Power}_j}$$

The number of colonies allocated to the ith imperialist is then approximately:

$$n_i = \text{round}(P_i \times N_{\text{colonies}})$$

where:

- N_{colonies}**:** Total number of colonies available (i.e., the number of countries not selected as imperialists).

Explanation:
This proportional allocation ensures that stronger imperialists (with lower costs) receive more colonies, giving them a larger influence in the search process.

3. Detailed Pseudocode

Below is detailed pseudocode for the ICA:

```
Algorithm ImperialistCompetitiveAlgorithm
Input:
    f(x)             // Cost function to minimize
    N                // Total number of countries (candidate solutions)
    N_imp            // Number of imperialists to select
    β                // Assimilation coefficient
    ξ                // Weighting factor for empire total cost
    p_rev            // Revolution rate (probability of random change)
    max_iter         // Maximum number of iterations
Output:
    x_best           // Best solution found

1. Initialization:
    a. Generate N countries (candidate solutions) randomly.
    b. Evaluate cost f(x) for each country.
    c. Sort the countries based on cost (ascending for minimization).

2. Formation of Empires:
    a. Select the top N_imp countries as imperialists.
    b. Let the remaining (N - N_imp) countries be colonies.
    c. For each imperialist i:
         i. Calculate its power: Power_i = c_max - c_i,
            where c_max is the highest cost among the imperialists.
         ii. Normalize power: P_i = Power_i / Σ(P_j) for all imperialists.
         iii. Allocate colonies: n_i = round(P_i × (N - N_imp)).

3. Main Loop (Iterate until termination condition):
    iter = 0
    While iter < max_iter and number of empires > 1 do:
        For each empire do:
            // Assimilation Step:
            For each colony in the empire:
                a. Compute direction vector: D = (x_imperialist - x_colony)
                b. Compute distance: d = ||D||
                c. Generate a random deviation angle θ (in a pre-defined
range).
                d. Update colony position:
                   x_new = x_colony + β * D + d * tan(θ)
                e. With probability p_rev, perform Revolution:
                   - Replace x_new with a random solution within the search
space.
                f. Evaluate f(x_new).
                g. If f(x_new) < f(x_imperialist), then swap x_new and
x_imperialist.
            End For

            // Update Empire's Total Cost:
            T_empire = f(x_imperialist) + ξ * (Mean cost of colonies)

        End For

        // Imperialistic Competition:
        a. Identify the weakest empire (with highest T_empire).
        b. In the weakest empire, identify the colony with the worst cost.
        c. Transfer this weakest colony to the empire with the best T_empire.
        d. If an empire loses all its colonies, eliminate that empire.

        iter = iter + 1
    End While

4. Termination:
    Return the best imperialist (the solution with the lowest cost).

End Algorithm
```

Explanation of the Pseudocode:

- **Initialization:** Randomly generates the candidate solutions and evaluates their quality.
- **Empire Formation:** The best solutions become imperialists and receive a number of colonies proportional to their strength.
- **Assimilation:** Each colony moves toward its imperialist with an added random deviation (to avoid premature convergence).
- **Revolution:** Occasionally, colonies are randomly altered to introduce diversity.
- **Competition:** Weak empires lose colonies to stronger ones, and empires without colonies are removed.

- **Termination:** The algorithm stops after a fixed number of iterations or when a convergence criterion is met, returning the best solution found.