

Unit III Evolutionary Computing

3.1 Introduction, Evolutionary Computing, Terminologies of Evolutionary Computing, Genetic Operators

1. Introduction to Evolutionary Computing

Evolutionary Computing (EC) is a subfield of artificial intelligence that uses evolutionary principles—such as reproduction, mutation, recombination, and selection—to solve optimization and search problems.

- Inspired by natural selection and genetic evolution in biological systems.
 - Utilizes populations of candidate solutions (individuals).
 - Applied in problems where traditional approaches are hard (e.g., NP-hard problems, optimization without clear gradients).
-

2. Evolution of Evolutionary Computing

The concept began with biological inspirations in the 1950s and evolved as follows:

Era	Contribution
1950s	Early concepts of genetic algorithms (GA) introduced.
1960s	Genetic Algorithms by John Holland; Evolutionary Strategies in Germany.
1970s	Emergence of Evolutionary Programming.
1980s–90s	Combination into the umbrella of Evolutionary Computing.
2000s onwards	Integration with neural networks, swarm intelligence, and hybrid systems.

3. Terminologies of Evolutionary Computing

Term	Description
Individual/Chromosome	A single solution in the population.
Gene	A part of the chromosome, representing a specific parameter.
Population	A set of individuals.
Fitness Function	A function to evaluate how good a solution is.
Selection	Choosing individuals for reproduction.
Crossover (Recombination)	Combining parts of two parents to create offspring.
Mutation	Random change in a gene to maintain genetic diversity.
Generation	One iteration of the algorithm, where offspring are created.
Convergence	When the population reaches similar or optimal solutions.

4. Genetic Operators

Genetic operators are key components that drive the evolution process in Genetic Algorithms:

a) Selection

- Chooses the fittest individuals for reproduction.
- Methods:
 - Roulette Wheel Selection
 - Tournament Selection
 - Rank Selection

b) Crossover (Recombination)

- Produces new offspring by combining genes of parents.
- Types:
 - Single-point crossover
 - Two-point crossover
 - Uniform crossover

c) Mutation

- Randomly alters genes to explore new solutions.
- Maintains diversity in the population.
- Types:
 - Bit Flip Mutation (for binary encoding)
 - Swap Mutation (for permutations)

d) Elitism

- Ensures the best individuals are carried over to the next generation.

3.2 Evolutionary Algorithms: - Genetic Algorithm, Evolution Strategies, Evolutionary Programming, Genetic Programming, Performance Measures of EA, Evolutionary Computation versus Classical Optimization

Evolutionary Algorithms (EAs)

Evolutionary Algorithms are a family of **population-based metaheuristic optimization algorithms** inspired by biological evolution, such as selection, crossover, and mutation.

1. Types of Evolutionary Algorithms

a) Genetic Algorithm (GA)

- Developed by **John Holland (1975)**.
- Works on a **population of candidate solutions** encoded as chromosomes (typically binary strings).

- Key processes: **Selection → Crossover → Mutation → Evaluation**.
- Commonly used for combinatorial and function optimization.

b) Evolution Strategies (ES)

- Originated in **Germany** by Rechenberg and Schwefel (1960s).
- Focus on **real-valued optimization problems**.
- Typically uses **self-adaptive mutation strategies**, with fewer operators than GA.
- Represented as:
 - **(μ , λ)-ES**: Select best μ from λ offspring.
 - **(μ + λ)-ES**: Best μ from combined parents + offspring.

c) Evolutionary Programming (EP)

- Introduced by **Lawrence Fogel (1960s)**.
- Focuses more on **evolving behavior** (e.g., finite state machines).
- Operates mainly with **mutation, no crossover**.
- Widely used for continuous function optimization and machine learning.

d) Genetic Programming (GP)



- Extension of GA, developed by **John Koza**.
- Individuals are **computer programs** (represented as trees).
- Uses **tree-based crossover and mutation**.
- Solves problems like symbolic regression, classification, and automated program generation.

2. Performance Measures of Evolutionary Algorithms

Measure	Description
Fitness	Value of the objective function; measures solution quality.

Measure	Description
Convergence Rate	Speed at which the algorithm approaches the optimal solution.
Diversity	Variability among solutions; prevents premature convergence.
Robustness	Consistency of performance over multiple runs or environments.
Scalability	Ability to handle larger or more complex problem spaces.
Execution Time	Total time taken for convergence or to reach a solution threshold.

3. Evolutionary Computation vs Classical Optimization

Feature	Evolutionary Computation	Classical Optimization
Nature	Population-based, stochastic	Single-solution, deterministic
Gradient Required?	 No	 Often yes (e.g., gradient descent)
Exploration	Strong global search (explores multiple regions)	Often stuck in local minima
Search Space	Discrete, continuous, mixed	Usually continuous
Robustness	Robust to noisy and non-differentiable functions	May struggle with discontinuities
Parallelism	Naturally parallelizable	Usually sequential
Complexity	Higher computational cost	Generally faster for simple problems

Feature	Evolutionary Computation	Classical Optimization
Use Cases	Optimization, machine learning, robotics, design problems	Calculus-based, convex optimization, linear programming

3.3 Advanced Topics: Constraint Handling, Multi-objective Optimization, Dynamic Environments

1. Constraint Handling

In real-world problems, solutions must often satisfy certain **constraints** (like budget limits, capacity, laws of physics, etc.).

Types of Constraints

- **Equality Constraints** (e.g., $x + y = 10$)
- **Inequality Constraints** (e.g., $x \geq 0$)
- **Boundary Constraints** (e.g., $0 \leq x \leq 1000$)

Constraint Handling Techniques

Method	Description
Penalty Functions	Add a penalty to the fitness if a solution violates constraints.
Repair Methods	Modify infeasible solutions to make them feasible.
Feasibility Rules	Prefer feasible solutions; if all are infeasible, choose least violating one.
Decoder-based Approach	Use a mapping that only generates feasible solutions.

2. Multi-objective Optimization (MOO)

Many problems have **multiple conflicting objectives** (e.g., cost vs quality, speed vs fuel efficiency).

 **Goal:** Find a set of trade-off solutions called the **Pareto Front**, where no objective can be improved without worsening another.

Popular Multi-objective EAs

Algorithm	Key Idea
NSGA-II (Non-dominated Sorting Genetic Algorithm II)	Fast, elitist sorting, crowding distance for diversity.
SPEA2 (Strength Pareto EA 2)	Uses strength and density for fitness assignment.
MOEA/D (Multi-Objective EA based on Decomposition)	Breaks the problem into scalar subproblems.

Performance Metrics

- Hypervolume
- Spacing
- Pareto Spread
- Generational Distance

3. Dynamic Environments

In some scenarios, the **optimization landscape changes over time** (e.g., stock markets, traffic systems, online recommendation engines).

Challenges

- Changing optima
- Need to adapt quickly
- Avoiding re-convergence to outdated solutions

Dynamic EA Strategies

Strategy	Description
Memory-based	Store and reuse good solutions from past environments.
Diversity Maintenance	Keep the population diverse to adapt faster.
Prediction Models	Use trends to anticipate future changes.
Random Immigrants	Periodically add random individuals to the population.

3.4 Swarm Intelligence: Ant Colony Optimization

Swarm Intelligence (SI)

Swarm Intelligence is inspired by how animals like ants, bees, and birds work together to solve problems, even without a leader.

- **Key Idea:** Simple agents (like ants) follow basic rules and interact with each other, leading to complex, smart behavior.
- **Examples:**
 - **Ants** find the shortest path to food.
 - **Bees** find the best flowers.
 - **Birds** fly in a group using simple local rules.

Ant Colony Optimization (ACO)

ACO is an algorithm inspired by how ants find the shortest path between their nest and food. Here's how it works:

1. How ACO Works:

1. **Ants Search for Food:** Ants move randomly and drop a substance called **pheromone** on the path. The more ants follow a path, the stronger the pheromone becomes.

2. **Ants Follow Strong Pheromones:** Ants prefer paths with stronger pheromone trails, but they also explore new paths to avoid getting stuck in one place.
 3. **Pheromone Evaporation:** Over time, the pheromone on paths **decays**, so ants need to keep exploring. This helps the colony adapt if conditions change.
 4. **Updating Pheromones:** Once all ants finish their search, they update the pheromones on the paths they traveled, making better paths even stronger.
-

2. ACO Steps:

1. **Initialization:** Set initial pheromone levels on all paths.
 2. **Ant Movement:** Ants move along the paths and select the next step based on pheromone levels.
 3. **Update Pheromones:** After ants complete their paths, pheromones are updated: evaporated or reinforced based on the quality of the solution.
 4. **Repeat:** This process is repeated over several rounds (iterations) until the algorithm finds a good solution.
-

3. Applications of ACO:

- **Traveling Salesman Problem (TSP):** Find the shortest route that visits all cities.
 - **Routing Problems:** Optimize paths in networks (like internet data routing).
 - **Scheduling:** Assign jobs to machines or workers in the best way.
-

4. Benefits of ACO:

- **Flexibility:** Can solve many complex problems like pathfinding and scheduling.
 - **Adaptability:** It can handle changing environments, like new paths appearing or disappearing.
 - **Parallel:** Multiple ants explore different solutions at once, speeding up the process.
-

5. Limitations of ACO:

- **Computationally Expensive:** Requires many iterations and ants to find a good solution.
- **Sensitive to Parameters:** Works best when the algorithm parameters (like pheromone evaporation rate) are tuned well.