Genetic Algorithms: Evolutionary Computation for Optimisation

Introduction to Genetic Algorithms

Genetic Algorithms (GAs) are a class of adaptive heuristic search algorithms inspired by Charles Darwin's theory of natural evolution. They were first conceptualised by John Holland in the 1970s.

- Search heuristic for solving complex optimisation problems by mimicking biological processes.
- Used when traditional methods are too slow, infeasible, or struggle with large solution spaces.

Example: The Travelling Salesman Problem (TSP)

The TSP is a classic optimisation challenge: find the shortest possible route that visits a set of cities exactly once and returns to the origin city.

01

A brute-force search is impractical due to factorial complexity, even for a moderate number of cities.

02

GAs evolve candidate routes (chromosomes) over generations, iteratively improving towards an optimal or near-optimal solution.

Types of Genetic Operators: Selection

Selection is the process of choosing individuals from the current generation to be parents for the next.
The goal is to favour fitter individuals.

Roulette Wheel Selection

Individuals are selected with a probability proportional to their fitness score, like a roulette wheel where larger slices represent higher fitness.

Tournament Selection

A small group of individuals is randomly chosen from the population, and the fittest among them is selected as a parent.

Rank Selection

Individuals are ranked based on their fitness, and selection probability is assigned based on their rank rather than absolute fitness. This reduces the chance of early convergence.

Types of Genetic Operators: Crossover

Crossover (or recombination) combines genetic material from two parent chromosomes to create new offspring, promoting the exploration of the search space.

One-Point Crossover

A single random point is chosen along the chromosome. The genetic material after this point is swapped between the two parents.

Multi-Point Crossover

Similar to one-point, but involves multiple cut points, leading to a more thorough mixing of genes between parents.

Uniform Crossover

Each gene (or bit) in the offspring is chosen randomly from either of the corresponding genes of the two parents, often with a 50% probability.

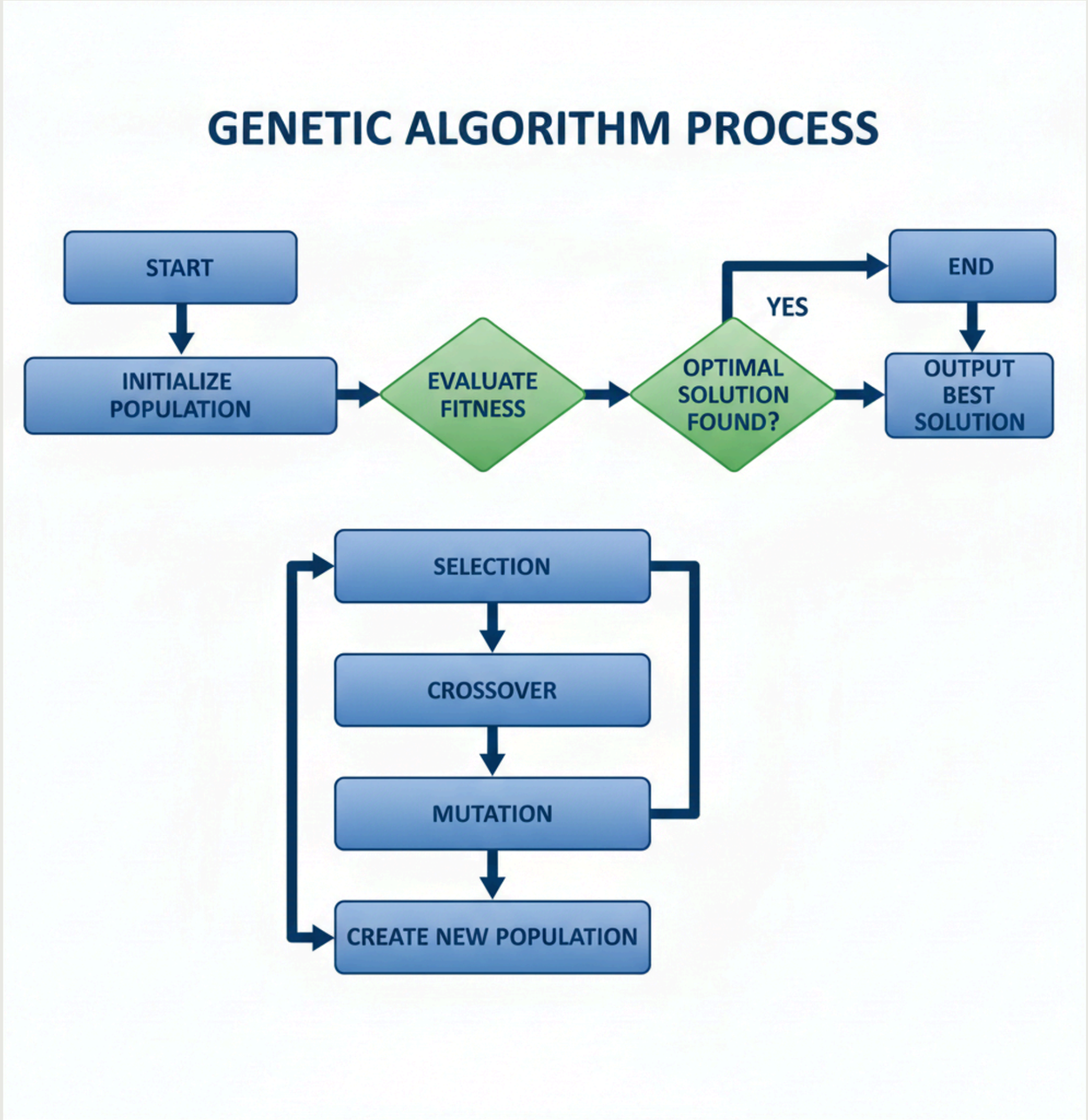
Types of Genetic Operators: Mutation

Mutation introduces random alterations to the genetic material of offspring, preventing premature convergence and maintaining diversity within the population.

- **Bit-flip mutation:** Flips a randomly selected bit (0 to 1, or 1 to 0) in binary encoded chromosomes.
- **Swap mutation:** Swaps the positions of two randomly chosen genes in a chromosome, ideal for permutation problems like TSP.
- **Random resetting:** Replaces a randomly selected gene with a new randomly generated value within the allowed range.

Flowchart of Genetic Algorithm Process

This flowchart visually represents the iterative process of a Genetic Algorithm, from initial population generation to the termination condition.



Genetic Algorithm: Core Steps

1. Initialisation

Create a random population of candidate solutions (chromosomes).

2. Fitness Evaluation

Assess the quality of each individual solution.

3. Selection

Choose parents based on their fitness to breed the next generation.

4. Crossover

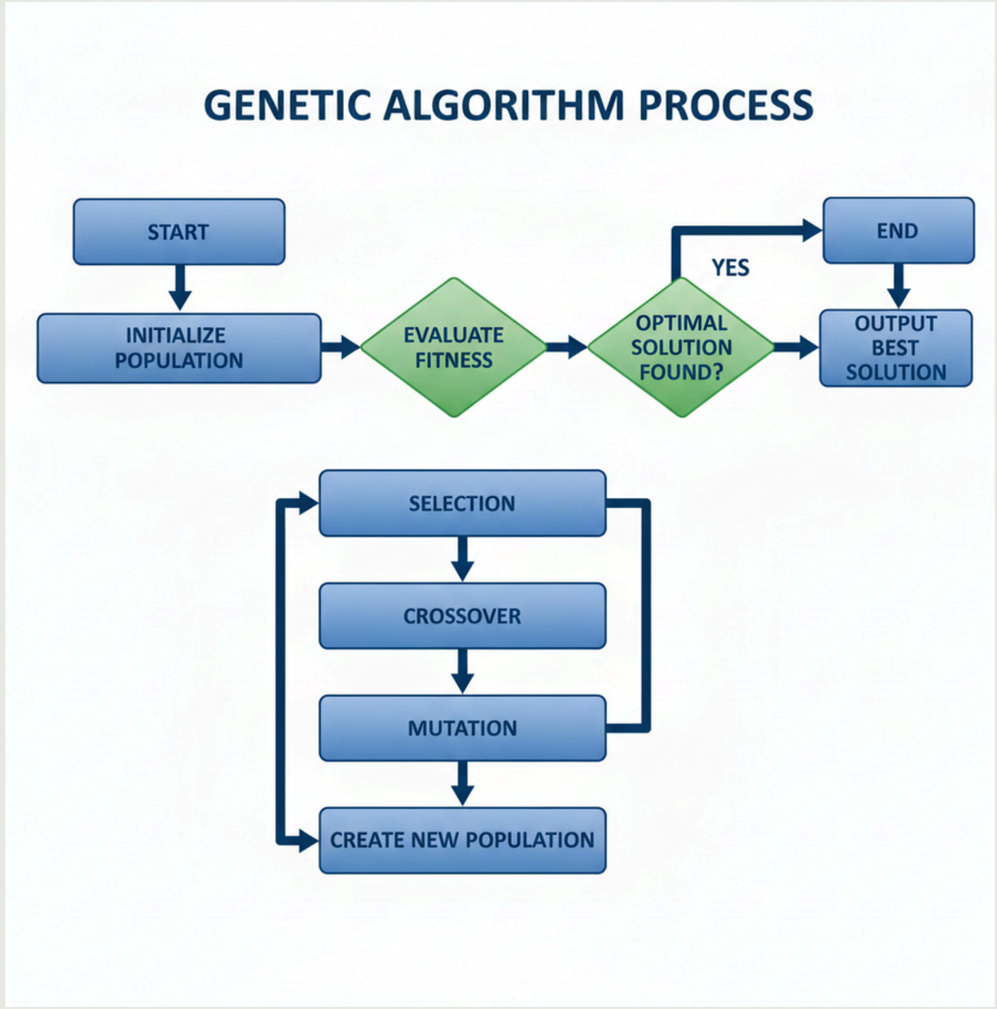
Combine genetic material from parents to produce offspring.

5. Mutation

Introduce random changes to maintain diversity.

6. New Generation & Termination

Replace the old population; repeat until stopping criteria are met (e.g., max generations or fitness threshold).



Understanding the Code with an Example

This code solves a simple optimization problem:

maximize the function f(x) = x², where x is an integer between 0 and 31

```python
import random

# Genetic Algorithm to maximize f(x) = x^2

# Parameters
POP_SIZE = 6      # population size
GENES = 5         # chromosome length (5 bits -> numbers 0–31)
GENERATIONS = 10  # number of iterations
MUTATION_RATE = 0.1  # probability of mutation

# Fitness function (objective: maximize x^2)
def fitness(chromosome):
    x = int("".join(map(str, chromosome)), 2)  # convert binary to decimal
    return x * x

# Generate initial population
def create_population():
    return [[random.randint(0, 1) for _ in range(GENES)] for _ in range(POP_SIZE)]
```

Understanding the Code with an Example

- Selection: We can use methods like Roulette Wheel or Tournament to pick the best individuals.
- Crossover: Like mixing parent chromosomes—single-point, two-point, or uniform crossover.
- Mutation: Randomly flipping bits to avoid getting stuck in local solutions.

```
# Selection (Roulette Wheel Selection)
def selection(population):
    total_fitness = sum(fitness(ind) for ind in population)
    pick = random.uniform(0, total_fitness)
    current = 0
    for ind in population:
        current += fitness(ind)
        if current > pick:
            return ind

# Crossover (Single-point crossover)
def crossover(parent1, parent2):
    point = random.randint(1, GENES - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

# Mutation (Flip a random bit)
def mutate(chromosome):
    for i in range(GENES):
        if random.random() < MUTATION_RATE:
            chromosome[i] = 1 - chromosome[i]
    return chromosome
```

Understanding the Code with an Example

- Generational Evolution: In each generation, parents are selected, crossover and mutation are applied, and a new population is formed to improve fitness.

```python
# Main GA loop
def genetic_algorithm():
    population = create_population()
    print("Initial Population:")
    for ind in population:
        print(ind, "Value:", int("".join(map(str, ind)), 2), "Fitness:", fitness(ind))

    for gen in range(GENERATIONS):
        new_population = []
        while len(new_population) < POP_SIZE:
            # Selection
            parent1 = selection(population)
            parent2 = selection(population)

            # Crossover
            child1, child2 = crossover(parent1, parent2)

            # Mutation
            child1 = mutate(child1)
            child2 = mutate(child2)

            new_population.extend([child1, child2])

        population = new_population[:POP_SIZE]
        best = max(population, key=fitness)

        print(f"\nGeneration {gen + 1}:")
        for ind in population:
            print(ind, "Value:", int("".join(map(str, ind)), 2), "Fitness:", fitness(ind))
        print("Best solution so far:", best, "Value:", int("".join(map(str, best)), 2), "Fitness:", fitness(best))

    return max(population, key=fitness)
```

Understanding the Code with an Example

- Best Solution: After all generations, the fittest chromosome is returned as the final optimized solution.
- GA is inspired by evolution.
- Uses Selection, Crossover, Mutation.
- Improves solutions over generations.
- Versatile and powerful for optimization.

```
# Run GA
best_solution = genetic_algorithm()
print("\nFinal Best Solution:", best_solution, "Value:", int("".join(map(str, best_solution)), 2), "Fitness:", fitness(best_solution))
```

Implementation Techniques

Successful GA implementation requires careful consideration of several key parameters and techniques to ensure effective optimisation.

Encoding Schemes

Binary strings, real-valued vectors, or permutations must be chosen based on the problem domain.

Fitness Function Design

Crucial for accurately evaluating solution quality (e.g., shortest route length for TSP, maximal profit).

Parameter Tuning

Population size, crossover rate, and mutation rate must be empirically tuned for optimal performance.

Termination Criteria

Define when the algorithm stops: fixed generations, fitness threshold, or stagnation detection.

Applications of Genetic Algorithms

Genetic Algorithms are versatile tools applied across numerous fields for solving complex optimisation and search problems.

Engineering Design

Optimising aerodynamic shapes, antenna design, and structural components for efficiency.

Machine Learning

Feature selection, hyperparameter optimisation, and neural network training for enhanced model performance.

Scheduling & Resources

Efficiently allocating resources and creating optimal schedules for job shops, transportation, and logistics.

Bioinformatics

Gene sequencing, protein folding prediction, and drug discovery processes to uncover complex biological patterns.

Research Papers

Paper 1: Goldberg (1989) "Genetic Algorithms in Search, Optimization and Machine Learning"

Established GA framework and demonstrated effectiveness on combinatorial problems

Paper 2: Srinivas & Patnaik (1994) "Adaptive probabilities of crossover and mutation in genetic algorithms"

Proposed adaptive operator rates improving convergence speed and solution quality

Conclusion: Genetic Algorithms

- Genetic Algorithms offer powerful, nature-inspired optimisation tools
- Effective for complex, multi-dimensional, and NP-hard problems
- Success depends on careful design of encoding, fitness, and operators
- Continues to evolve with hybrid methods and real-world applications
- Encourages creative problem-solving by mimicking evolution’s success