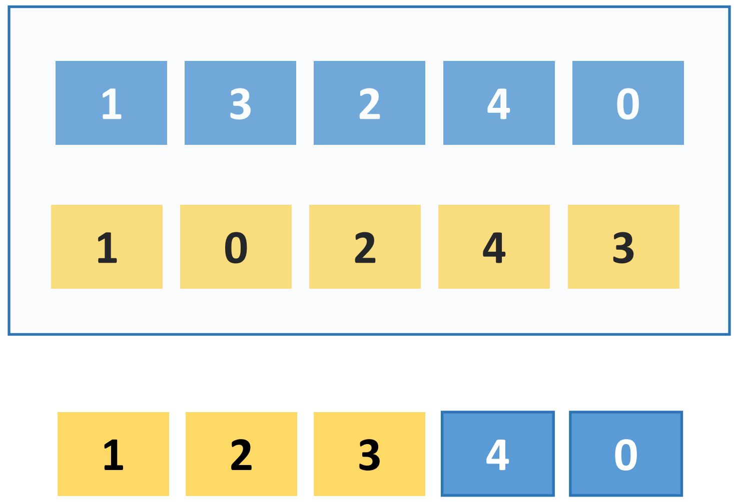
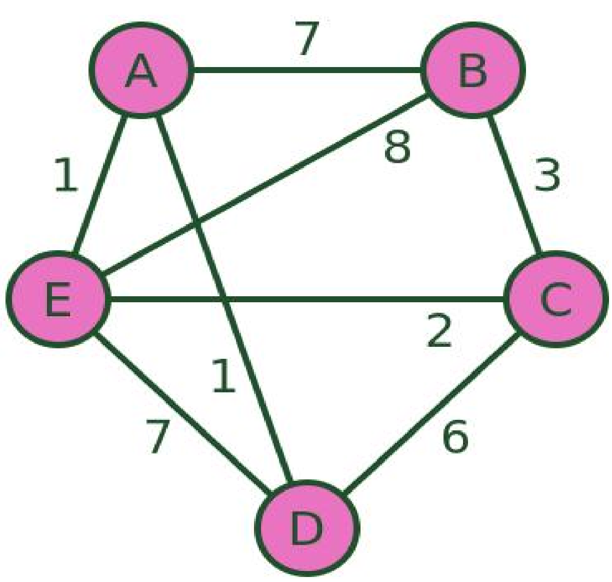


Genetic Algorithm With Implementation

Travelling Salesman Problem (TSP)



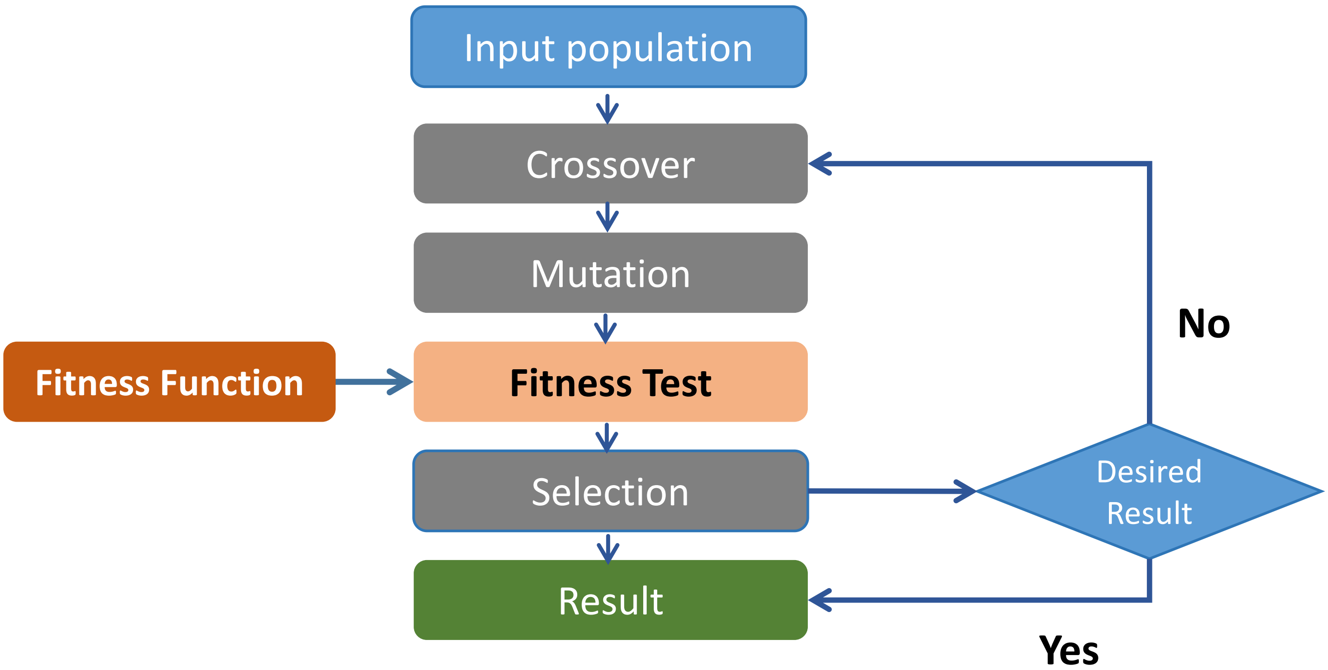
Algorithm Series

Practical Python

Introduction

A genetic algorithm (GA) is a search heuristic that mimics the process of natural selection. It is used to generate high-quality solutions for optimization and search problems by relying on bio-inspired operations such as selection, crossover (recombination), and mutation.

Key Components



Applications:

Optimization Problems: Traveling Salesman Problem, Knapsack Problem, etc.

Machine Learning: Tuning neural network architectures.

Game Development: Creating AI players.

Engineering Design: Optimizing parameters in engineering problems.

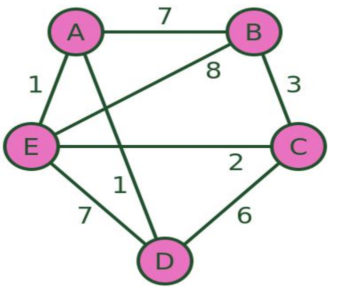
Traveling Salesman Problem (TSP)

In the TSP, a salesman must visit a set of cities exactly once and return to the starting city. The goal is to minimize the total distance traveled. The problem becomes more difficult as the number of cities increases, making brute force solutions impractical due to the large number of possible routes.

A GA is well-suited for the TSP because it can search through a large number of potential solutions (routes) and use evolutionary principles to find a good (if not optimal) solution efficiently.

CityA (0), CityB (1), CityC (2),

CityD (3), CityE (4), CityF (5)



Distance matrix (in kilometers):

```
[
    [ 0,  29, 20, 21, 50, 31 ],
    [ 29, 0,  50, 29, 55, 40 ],
    [ 20, 50, 0,  15, 14, 25 ],
    [ 21, 29, 15, 0,  19, 36 ],
    [ 50, 55, 14, 19, 0,  27 ],
    [ 31, 40, 25, 36, 27, 0  ]
]
```

Key Steps in a Genetic Algorithm

Generate Trial Structures

Population Initialization: The algorithm starts by generating an initial population of candidate solutions. These solutions are often represented as strings (chromosomes), which can be binary strings, real numbers, or other encodings depending on the problem domain.

Diversity: A diverse population increases the chances of finding a good solution. Initializing with random values or using heuristics to create a varied population can help.

Each individual (route) is represented as a permutation of the city indices.

[0, 3, 2, 1, 4, 5] represents visiting the cities in the order A → D → C → B → E→ F.

Initial Pool

[0, 3, 2, 5, 1, 4]	[2, 0, 3, 1, 4, 5]
[4, 5, 1, 0, 2, 4]	[0, 5, 1, 3, 2, 3]
[3, 1, 2, 0, 4, 5]	[2, 1, 3, 5, 0, 4]
[2, 5, 4, 3, 0, 1]	[2, 0, 4, 3, 1, 5]

Fitness Function

Each candidate solution is evaluated using a fitness function that quantifies how good that solution is with respect to the problem being solved. The fitness function is problem-specific and can involve various metrics.

Selection Process: After evaluation, the selection process determines which individuals will be parents for the next generation.

The fitness of a route is the inverse of the total distance traveled. The total distance is calculated by summing the distances between consecutive cities in the route, including the return to the starting city.

For the route [0, 1, 2, 3, 4, 5], the total distance would be:

Distance =

distance(A → B) +
distance(B → C) +
distance(C → D) +
distance(D → E) +
distance(D → F) +
distance(F → A)

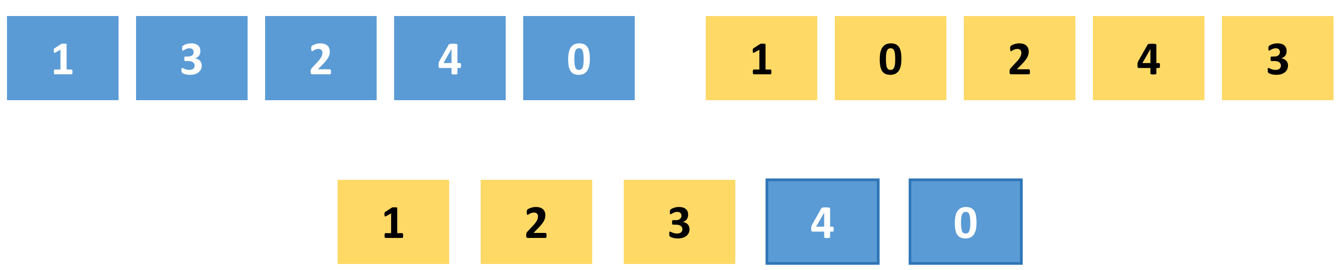
10 + 35 + 30 + 25 + 10 = 171 km

The fitness of this route would be 1 / 171 ≈ 0.0058.

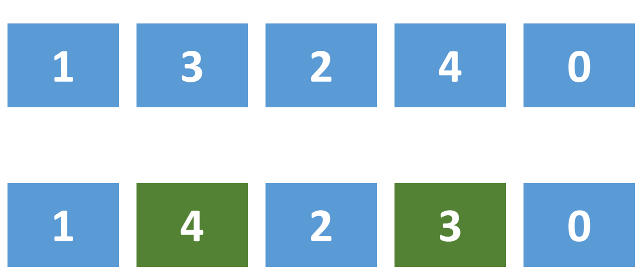
Create Offspring

Genetic Operators: New individuals (offspring) are created from the selected parents using genetic operators:

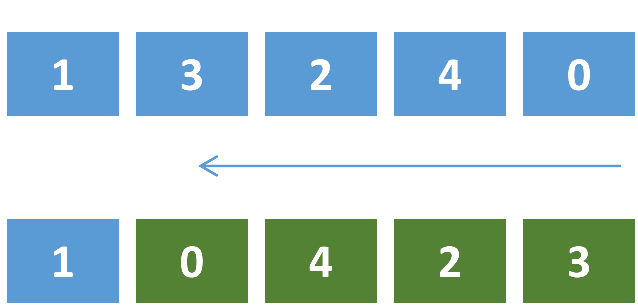
Crossover (Recombination): Combines the genetic information of two parents to produce offspring. For example, a one-point crossover takes a segment from each parent to create a new solution.



Mutation: Introduces random changes to an individual's genes to maintain genetic diversity and explore new areas of the search space. For instance, flipping a bit in a binary representation or adding a small random value to a number.



Inversion: Changes the order of genes in the chromosome. This operator can help escape local optima by altering the structure of a solution.



Initialize the Distance matrix and write a function to create random initial Samples

```python
import random

# Define the city names and distance matrix
cities = ['A', 'B', 'C', 'D', 'E', 'F']
distance_matrix = [
    [0,  29, 20, 21, 50, 31],
    [29, 0,  50, 29, 55, 40],
    [20, 50, 0,  15, 14, 25],
    [21, 29, 15, 0,  19, 36],
    [50, 55, 14, 19, 0,  27],
    [31, 40, 25, 36, 27, 0]
]

def create_initial_population(pop_size):
    """Create initial population with shuffled routes."""
    population = []
    for _ in range(pop_size):
        # Route contains indices of cities
        route = list(range(len(cities)))
        random.shuffle(route)
        population.append(route)
    return population

print(create_initial_population(3))
```

```
[[4, 1, 0, 2, 5, 3], [3, 4, 2, 5, 0, 1], [5, 0, 4, 3, 2, 1]]
```

function to calculate distance basis the route

```python
def calculate_distance(route):
    """total distance of a route based on distance matrix."""
    distance = 0
    for i in range(len(route) - 1):
        distance += distance_matrix[route[i]][route[i + 1]]
    # Return to start distance
    distance += distance_matrix[route[-1]][route[0]]
    return distance

print(calculate_distance([3, 4, 2, 5, 0, 1]))
```

```
147
```

```python
def crossover(parent1, parent2):
    """Perform ordered crossover between two parents."""
    child = [-1] * len(cities)
    start_pos, end_pos = sorted(random.sample(range(len(cities)), 2))

    # Copy part of parent1 into the child
    child[start_pos:end_pos] = parent1[start_pos:end_pos]

    # Fill the remaining cities from parent2 in the same order
    parent2_pos = 0
    for i in range(len(cities)):
        if child[i] == -1:
            while parent2[parent2_pos] in child:
                parent2_pos += 1
            child[i] = parent2[parent2_pos]

    return child
```

```python
print(crossover([3, 4, 2, 5, 0, 1], [4, 2, 3, 1, 0, 5]))
```

```
[4, 3, 2, 5, 0, 1]
```

```python
def mutate(route, mutation_rate):
    """Randomly swap cities in route using mutation probability."""
    for i in range(len(route)):
        if random.random() < mutation_rate:
            swap_with = random.randint(0, len(route) - 1)
            route[i], route[swap_with] = route[swap_with], route[i]
    return route
```

```python
print(mutate([4, 3, 2, 5, 0, 1], .5))
```

```
[4, 1, 2, 5, 0, 3]
```

```python
def rank_population(population):
    """Rank basis fitness (lower distance is better)."""
    fitness_results = [(route, calculate_distance(route))
                       for route in population]
    return sorted(fitness_results, key=lambda x: x[1])
```

```python
print(rank_population([[4, 1, 0, 2, 5, 3],
                       [3, 4, 2, 5, 0, 1]]))
```

```
[([3, 4, 2, 5, 0, 1], 147), ([4, 1, 0, 2, 5, 3], 184)]
```

```python
def selection(ranked_pop, elite_size):
    """Select elite routes for further stage"""
    selection_results = [ranked_pop[i][0]
                         for i in range(elite_size)]
    return selection_results
```

```python
selection([([3, 4, 2, 5, 0, 1], 147),
           ([4, 1, 0, 2, 5, 3], 184)
          ], 1)
```

```
[[3, 4, 2, 5, 0, 1]]
```

```python
def evolve_population(population, elite_size, mutation_rate):
    """Evolve the population through selection, crossover, and mutation."""
    ranked_pop = rank_population(population)
    selection_pool = selection(ranked_pop, elite_size)

    # Create next generation using crossover and mutation
    children = []
    for i in range(len(population) - elite_size):
        parent1 = random.choice(selection_pool)
        parent2 = random.choice(selection_pool)
        child = crossover(parent1, parent2)
        children.append(mutate(child, mutation_rate))

    # Add the elite individuals directly to the next generation
    new_generation = selection_pool + children
    return new_generation
```

```python
evolve_population([[4, 1, 3, 5, 0, 2],[0, 5, 2, 4, 1, 3],
                   [0, 1, 4, 5, 3, 2],[3, 5, 0, 2, 4, 1],
                   [4, 1, 0, 3, 5, 2]], 2, 0.01)
```

```
[[0, 5, 2, 4, 1, 3],
 [4, 1, 0, 3, 5, 2],
 [4, 1, 0, 3, 5, 2],
 [4, 0, 5, 2, 1, 3],
 [0, 5, 2, 4, 1, 3]]
```

```python
population_size = 6
elite_size = 3
mutation_rate = 0.1
generations = 50
distance_cal = [ ]

# Create initial population
population = create_initial_population(population_size)

# Evolve the population through generations
for gen in range(generations):
    population = evolve_population(population, elite_size, mutation_rate)
    best_route = rank_population(population)[0]
    distance_cal.append(best_route[1])

# Display the best route in terms of city names
best_route = rank_population(population)[0][0]
distance = rank_population(population)[0][1]
best_route_cities = [cities[i] for i in best_route]
print(distance_cal)
print("Best Route:", " -> ".join(best_route_cities), distance)
```

```
[177, 175, 148, 148, 148, 148, 148, 148, 146, 146, 146, 146, 146, 146, 146, 146,
146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146,
146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146,
146, 146]
Best Route: F -> B -> A -> D -> C -> E 146
```