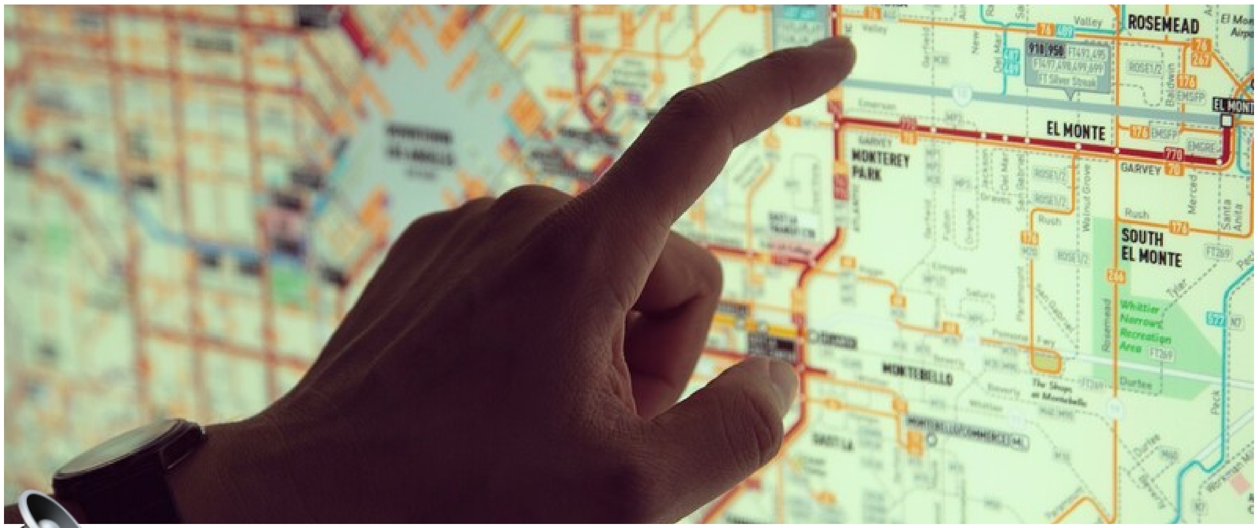


EVOLUTIONARY ALGORITHMS

Type of problem : Constrained Optimisation Problem [COP]

Large-Scale Route Optimization Problem using an Evolutionary Algorithm

1)Project Idea

Problem Description

The problem revolves around efficiently delivering orders from a manufacturing company's warehouses to clients.

and this problem is: `ConstrainedOptimization`

Here are the key components:

1. **Warehouses and Orders:**
    - The company has multiple warehouses located in different areas.
    - When orders come in from clients, a planner needs to assign items to delivery trucks for order fulfillment.
2. **Delivery Truck Assignment:**
    - The planner must decide which delivery truck to use for each order.

- Each truck type has an associated cost (e.g., fuel efficiency, maintenance) and a capacity limit (both in terms of area and weight).
- There is no limit on the number of trucks available for each type.

3. **Route Planning:**
   - For each delivery truck, the planner determines the optimal route (sequence of stops) to deliver orders to their respective destinations.
   - The goal is to minimize the overall delivery cost, considering both truck type costs and travel distances.
4. **Constraints:**
   - Items are only available at specific times. A truck can start its route only when all assigned items are available.
   - The time difference between the earliest and latest available items in the same truck should be less than a user-defined limit (e.g., 4 hours).
   - All items must be delivered before their respective deadlines.
   - Some products can be loaded together in the same truck, while others cannot.
   - Each truck can have at most N stops (where N is user-defined).
   - Each stop incurs a fixed cost (in addition to the delivery cost) and requires the truck to stay for M hours to unload the items (where M is user-defined).

The Vehicle Routing Problem or VRP

The first classic VRP, known as the Traveling Salesman Problem (TSP), dates back to the early 1800s and gained popularity at that time. As VRP evolved, it encompassed more complex tasks involving large volumes of data.

The challenge in VRP is to design the most efficient routes from a depot to a set of destinations, each with specific business constraints. These constraints may include limitations on vehicles, cost controls, time windows, and resource limitations related to the loading process at the depot.

In the Vehicle Routing Problem (VRP), the objective is to determine the optimal routes for multiple vehicles visiting a set of locations. When there is only one vehicle, it simplifies to the Traveling Salesperson Problem.

One way to define optimal routes is to minimize the length of the longest single route among all vehicles. This definition is appropriate when the goal is to complete all deliveries as quickly as possible. The VRP example below illustrates how to find optimal routes based on this definition.

Route optimization is a solution for so-called vehicle routing problems (VRPs).

> **Route optimization** is the process of determining the most cost-efficient route. You may think that it means finding the shortest path between two points, but it’s rarely that simple: You must

account for all relevant factors involved such as the number and location of all stops on the route, arrival/departure time gap, effective loading, etc.

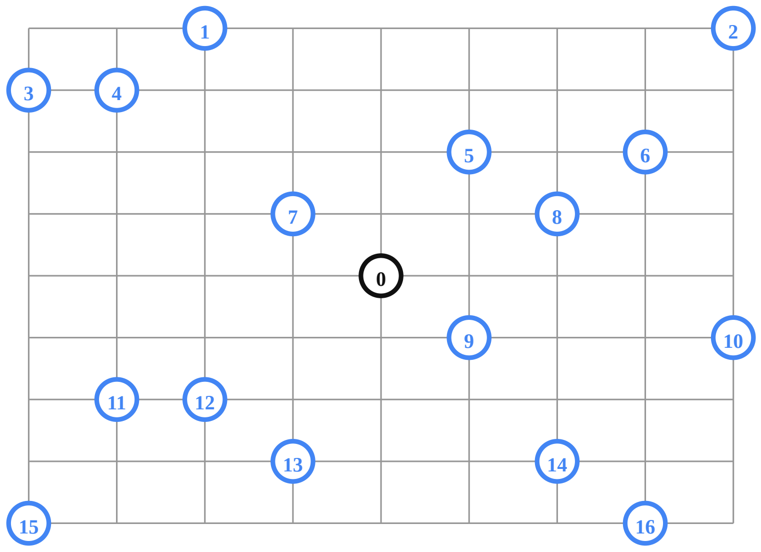
ways of generalizing the TSP by adding constraints on the vehicles, including:

- Capacity constraints: the vehicles need to pick up items at each location they visit, but have a maximum carrying capacity.
- Time windows: each location must be visited within a specific time window.

VRP example

This section presents an example of a VRP in which the goal is to minimize the longest single route.

Imagine a company that needs to visit its customers in a city made up of identical rectangular blocks. A diagram of the city is shown below, with the company location marked in black and the locations to visit in blue.



2) Main functionalities

1)Representation :

represent Truck_details such as (`different attributes of the trucks such as inner size, weight capacity, cost per kilometer, and speed`)

then represent order:

- The order data is preprocessed by dividing the "Weight" and "Area" columns by 10000 to normalize the values.
- The unique order IDs are extracted from the order data and stored in a DataFrame named `order_geenpool`

Individuals Representation:

- Individuals are represented as a list of tuples named `individuals`. Each tuple in the list represents a truck and its assigned orders.
- The first element of the tuple represents the truck type, which is randomly selected from [0, 1, 2].
- The second element of the tuple is a list of order IDs assigned to that truck. These order IDs are randomly selected from the `order_geenpool` DataFrame, ensuring that the selected orders satisfy the truck's weight and area capacities.
- The `init_individuals` function initializes individuals by iterating until there are orders left in the `order_geenpool`. It selects orders for each truck until the truck's weight and area capacities are exceeded or there are no more orders available.

```
def init_individuals():
    individuals = []
    order_geenpool = order_ids
    while len(order_geenpool) > 0:  # while there are orders
        truck_type = random.choice([0, 1, 2])
        truck_details = truck_detail(truck_type)
        inner_size_components = truck_details["Inner Size"].split('x')
        truck_area = float(inner_size_components[0]) * float(inner_size_com
        order_list = []
        sum_weight = 0
        sum_area = 0
        while len(order_geenpool) > 0:
            random_order_idx = random.randint(0, len(order_geenpool) - 1)
            order_id = order_geenpool.iloc[random_order_idx, 0]
            order_details = order_data[order_data["Order_ID"] == order_id].
            if (order_details["Weight"] + sum_weight) <= truck_details["Wei
               (order_details["Area"] + sum_area) <= truck_area:
                order_list.append(order_id)
                sum_weight += order_details["Weight"]
                sum_area += order_details["Area"]
                order_geenpool = order_geenpool[order_geenpool['Order_ID']
            else:
                break
        individuals.append((truck_type, order_list))
    return individuals
    individuals = init_individuals()
individuals
```

Sample of output:

[(1, ['A240329', 'A260568', 'A260902', 'A230525', 'A250346', 'A250581']),
(0, ['A250122', 'A010530', 'A240108', 'A010531', 'A150401', 'A140109']),
(1, ['A230658', 'A190225', 'A260604', 'A270200', 'A240104', 'A230645']),
(1, ['A250555', 'A180370', 'A240112']),
(2, []),
(0,
['A190226','A260582','A220342','A220309','A260111','A260899','A240519','A250212','A260114','2]),

Function `init_population(size)`, initializes a population of individuals for a genetic algorithm. Here's a breakdown of what it does:

- **Input:** `size` is the desired size of the population, i.e., the number of individuals in the population.
- **Output:** The function returns a list `population` containing `size` number of individuals.
- **Return:** Finally, it returns the populated list `population`, which contains `size` number of individuals, each initialized using the `init_individuals()` function.

```
def init_population(size):
    population = []
    for i in range(size) :
        idi = init_individuals()
        population.append(idi)
    return population
```

Function `calculate_distance_route(route)`, calculates the total distance for a given route of deliveries in a vehicle routing problem (VRP) scenario. Here's a breakdown of what it does:

1. **Input:** `route` is a list of order IDs representing the sequence of deliveries to be made.
2. **Output:** The function returns the total distance (in meters) for the given route.
3. **Calculation:** It iterates over each order in the `route` and calculates the distance between consecutive stops. It uses two datasets, `order_data` and `distance_data`, to get the destination for each order and the distance between locations.

```
def calculate_distance_route(route):
    total_distance = 0
    source = "City_61"  # Assuming the starting location is the warehouse

    # Iterate over each order in the route
    for order_id in route:
        # Get the destination for the current order from the order_data datase
        destination = order_data.loc[order_data["Order_ID"] == order_id, "Dest

        # Get the distance between the source and destination from the distanc
        distance = distance_data[(distance_data["Source"] == source) & (distar

        # Check if the distance is found
        if not distance.empty:
            # Add the distance to the total distance
            total_distance += distance.iloc[0]["Distance(M)"]
            # Update the source for the next order
            source = destination
        elif(source == destination):
            total_distance +=0
        else:
            print(f"Error: Distance not found for route {source} -> {destinati
            return None  # Return None if distance is not found

    # Add the distance from the last destination back to the warehouse
    distance_back = distance_data[(distance_data["Source"] == source) & (dista
    if not distance_back.empty:
        total_distance += distance_back.iloc[0]["Distance(M)"]
    elif(source == "City_61"):
        total_distance +=0
    else:
        print(f"Error: Distance not found for route {source} -> city_61")
        return None  # Return None if distance is not found

    return total_distance

# Example usage
route = ['A210204', 'A150402', 'A250599', 'A090422', 'A190598']
distance = calculate_distance_route(population[0][0][1])
print("Total distance:", distance)
```

Total distance: 4043.6059999999998

Function `costForTruck(truck_type, distance)`, calculates the cost for using a specific type of truck to travel a given distance. Here's a breakdown of what it does:

1. **Inputs:** `truck_type`: An integer representing the type of truck. Assumed values are 0, 1, or any other value. `distance`: The distance traveled by the truck, assumed to be in a specific unit (e.g., meters).

2. **Output:** The function returns the cost (in some unit of currency) for using the specified type of truck to travel the given distance.
3. **Return:** The calculated cost is returned as the output of the function.

```
def costForTruck(truck_type,distance):
    if truck_type == 0 :
        cost = distance * 3
    elif truck_type == 1:
        cost = distance * 2
    else:
        cost = distance
    return cost
```

```
costForTruck(0,distance)
```

12130.818

Function `calculate_arrival_time(route)`, calculates the estimated arrival time (in days) for each order in a given route based on the distance traveled and a predefined speed. Here's a breakdown of what it does:

1. **Input:** `route` is a list of order IDs representing the sequence of deliveries to be made.
2. **Output:** The function returns a list `travel_time_days_per_orders` containing the estimated arrival time (in days) for each order in the route.
3. **Calculation:** For each order in the route, the function calculates the total distance traveled up to that order using the `calculate_distance_route(route[0:i+1])` function (assuming it is defined elsewhere).
4. **Return:** The function returns the list `travel_time_days_per_orders` containing the estimated arrival time for each order in the route.

```
def calculate_arrival_time(route):
    travel_time_days_per_orders = []

    for i in range(len(route)) :
        total_distance = calculate_distance_route(route[0:i+1])
        if i == 0:
            days = (total_distance / speed) / 24
            travel_time_days_per_orders.append(round(days))
        else:
            days = ((total_distance / speed) + 2*i) / 24
            travel_time_days_per_orders.append(round(days))
    return travel_time_days_per_orders

calculate_arrival_time(['A220356', 'A250272'])
```

Function `spiltDate(date_string)`, takes a date string as input and extracts the day component from it. Here's a breakdown of what it does:

1. **Input:** `date_string` is a string representing a date in a specific format.
2. **Output:** The function returns an integer representing the day component of the date.
3. **Return:** The function returns the `day` component extracted from the date string.

```
def spiltDate(date_string):
    date_components = date_string.split()[0]
    # Split the date components by '/' to extract month, day, and year
    month, day, year = map(int, date_components.split('/'))
    return day
```

Function `fitness_function(individual)`, calculates the fitness of an individual solution in a genetic algorithm for the vehicle routing problem (VRP). Here's a breakdown of what it does:

1. **Input:** `individual` is a list representing a solution to the VRP. Each element in the list corresponds to a truck route, containing the type of truck and the list of order IDs in the route.
2. **Output:** The function returns the fitness value of the individual solution, which is a measure of how good the solution is. Lower values indicate better solutions.
3. **Calculation:**

- For each truck in the individual's solution:
  - It calculates the distance traveled by the truck using `calculate_distance_route(truck[1])` and the cost for using the truck on that route using `costForTruck(truck[0], distance_for_truck)`.
  - If the distance calculation fails (`distance_for_truck` is `None`), it penalizes the solution by setting the distance to 0.
  - It adds the cost of using the truck on the route to the total cost of the individual (`cost_individual`).
- It calculates the arrival time for each order in the routes and checks if any order arrives after its deadline. If so, it penalizes the solution by adding a penalty cost (`10` in this case) for each such violation.

4. **Return:** The function returns the rounded total cost (`cost_individual`) as the fitness value.

```
def fitness_function(individual):
    cost_individual = 0.0
    for truck in individual:
        distance_for_truck = calculate_distance_route(truck[1])
        if distance_for_truck is None:
            distance_for_truck = 0  # Penalize the solution if distance calcul
        cost_for_truck = costForTruck(truck[0], distance_for_truck) # Only get
        cost_individual += cost_for_truck
        if cost_for_truck is not None:
            cost_individual += cost_for_truck
        else:
            cost_individual += 0  # Penalize the solution if cost calculation

    arrival_time_for_ord =  calculate_arrival_time(truck[1])
    i=0
    for ord_id in truck[1]:
        Available_Time = order_data[order_data["Order_ID"] == ord_id].iloc[0,5
        Deadline = order_data[order_data["Order_ID"] == ord_id].iloc[0,6]
        Available_day = spiltDate(Available_Time)
        Deadline_day = spiltDate(Deadline)
        arrival_time = arrival_time_for_ord[i] + Available_day
        # this is penlity
        if arrival_time > Deadline_day:
            cost_individual += 10
    return round(cost_individual)  # Higher fitness for lower cost
```

```
# Example usage
fitness_score = fitness_function(population[1])
print("Fitness score:", fitness_score)
```

Fitness score: 2817626

Function `tournament_selection(population)`, implements a tournament selection strategy for selecting parents in a genetic algorithm. Here's a breakdown of what it does:

1. **Input:** `population` is a list of individuals (solutions) in the genetic algorithm.
2. **Output:** The function returns a list `selected_parents` containing the selected parent individuals.
3. **Selection Process:**
    - For each parent to be selected (twice in this case, as indicated by `for _ in range(2)`):
        - It randomly selects 5 candidates from the population using `random.sample(population, 5)`.

   - It calculates the fitness of each candidate using the `fitness_function(sol)` function and stores these fitness values in `candidates_fitness`.
   - It identifies the candidate with the best (lowest) fitness (`best_fitness`) and its index (`idx_best`) in the `candidates` list.
   - It selects the chosen one (`chosen_one`) as the candidate with the best fitness and adds it to the `selected_parents` list.
4. **Return** The function returns the list `selected_parents` containing the two selected parent individuals.

```
def tournament_selection(population):
    selected_parents = []

    for _ in range(2):
        candidates = random.sample(population, 5)
        candidates_fitness = []
        for sol in candidates:
            candidates_fitness.append(fitness_function(sol))

        # Assess the prowess of each candidate, and let the brilliance shine f
        best_fitness = min(candidates_fitness)  # Select the one with the high
        idx_best =  candidates_fitness.index(best_fitness)
        chosen_one = candidates[idx_best]
        # Marvel at the chosen one and add it to the list of selected parents
        selected_parents.append(chosen_one)

    return selected_parents
```

```
# Example usage:
parents = tournament_selection(population)
print("Selected parents:", parents)
```

Selected parents: 2

Function `roulette_wheel_selection(population)`, implements a selection strategy known as roulette wheel selection in genetic algorithms. Here's a breakdown of what it does:

1. **Input:** `population` is a list of individuals (solutions) in the genetic algorithm.
2. **Output:** The function returns a single individual selected from the population based on their fitness.
3. **Selection Process:**

- It first calculates the total fitness score of the population by summing the fitness scores of all individuals in the population using a generator expression: `sum(fitness_function(individual) for individual in population)`.
- It then generates a random number (`spin`) between 0 and the total fitness score using `random.uniform(0, total_fitness)`.
- The function then iterates over the population, summing the fitness scores of individuals until the cumulative sum is greater than or equal to the random number `spin`.
- When this condition is met, it returns the individual at that point, effectively selecting an individual with a probability proportional to its fitness score.

4. **Return:** The function returns the selected individual based on the roulette wheel selection mechanism.

```
def roulette_wheel_selection(population):
    # Calculate the total fitness score of the population
    total_fitness = sum(fitness_function(individual) for individual in populat

    # Generate a random number between 0 and the total fitness score
    spin = random.uniform(0, total_fitness)

    # Spin the roulette wheel and select the individual based on their cumulat
    cumulative_fitness = 0
    for individual in population:
        cumulative_fitness += fitness_function(individual)
        if cumulative_fitness >= spin:
            return individual
```

```
# Example usage:
selected_individual = roulette_wheel_selection(population)
print("Selected individual:", selected_individual)
```

Integer value

Function `crossover(parent1, parent2)`, implements a crossover operation for genetic algorithms. Here's a breakdown of what it does:

1. **Inputs:** `parent1` and `parent2` are two parent individuals (solutions) that will undergo crossover.
2. **Output:** The function returns two offspring individuals resulting from the crossover operation.
3. **Crossover Process:**
    - It chooses a random crossover point (`crossover_point`) between 1 and the minimum length of `parent1` and `parent2` minus 1. This point determines where the genes of the parents will be

exchanged.

- It creates two offspring (`offspring1` and `offspring2`) by combining the genes of the parents before and after the crossover point.
    - `offspring1` is created by taking the genes of `parent1` before the crossover point and the genes of `parent2` after the crossover point.
    - `offspring2` is created by taking the genes of `parent2` before the crossover point and the genes of `parent1` after the crossover point.

4. **Return:** The function returns `offspring1` and `offspring2`, the two offspring resulting from the crossover operation.

```
def crossover(parent1, parent2):
    # Choose a random crossover point
    crossover_point = random.randint(1, min(len(parent1), len(parent2)) - 1)

    # Create the offspring by combining the parents' genes
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]

    return offspring1, offspring2

offspring1, offspring2 = crossover(population[0],population[1])

fp1 = fitness_function(population[0])
fp2 = fitness_function(population[1])

fo1 = fitness_function(offspring1)
fo2 = fitness_function(offspring2)

print(fp1,"\n",fp2,"\n",fo1,"\n",fo2,"\n")
```

```
2715017
2817626
2743803
2788840
```

Function `uniform_crossover(parent1, parent2, probability=0.5)`, implements a uniform crossover operation for genetic algorithms. Here's a breakdown of what it does:

1. **Inputs:**

- `parent1` and `parent2` are two parent individuals (solutions) that will undergo crossover.

- `probability` is the probability of selecting a gene from `parent1` or `parent2` for the offspring. The default value is 0.5, meaning an equal probability for each parent.

2. **Output:** The function returns two offspring individuals resulting from the crossover operation.
3. **Crossover Process:**
    - It iterates over the genes of `parent1` and `parent2` simultaneously using the `zip` function.
    - For each pair of genes, it randomly selects a gene from either parent based on the given `probability`:
        - If a random number between 0 and 1 is less than `probability`, it selects the gene from `parent1` for `offspring1` and from `parent2` for `offspring2`.
        - Otherwise, it selects the gene from `parent2` for `offspring1` and from `parent1` for `offspring2`.
4. **Return:** The function returns `offspring1` and `offspring2`, the two offspring resulting from the uniform crossover operation.

```
def uniform_crossover(parent1, parent2, probability=0.5):
    # Create the offspring by selecting genes from parents with a given probab
    offspring1 = []
    offspring2 = []

    for gene1, gene2 in zip(parent1, parent2):
        if random.random() < probability:
            offspring1.append(gene1)
            offspring2.append(gene2)
        else:
            offspring1.append(gene2)
            offspring2.append(gene1)

    return offspring1, offspring2

offspring1, offspring2 = uniform_crossover(population[0],population[1])
fp1 = fitness_function(population[0])
fp2 = fitness_function(population[1])

fo1 = fitness_function(offspring1)
fo2 = fitness_function(offspring2)

print(fp1,"\n",fp2,"\n",fo1,"\n",fo2,"\n")
```

```
2715017
2817626
2936069
2513507
```

Function `shuffle_mutation(individual)`, performs a mutation operation known as shuffle mutation on an individual in a genetic algorithm. Here's a breakdown of what it does:

1. **Input:** `individual` is a solution (or individual) in the genetic algorithm. It likely represents a solution to the vehicle routing problem, where each element in the individual represents a truck route.
2. **Output:** The function returns the mutated individual after applying shuffle mutation.
3. **Mutation Process:**
    - For each truck in the individual (accessed by iterating over the range of the individual's length):
        - It shuffles the order of the orders (or genes) in the truck's route. This is done by shuffling the second element of the truck element (`individual[truck][1]`), which likely represents the list of orders in the route.
4. **Return:** The function returns the mutated individual after shuffling the order of the orders in each truck's route.

```
def shuffle_mutation(individual):
    for truck in range(len(individual)):
        random.shuffle(individual[truck][1])
    return individual
```

Function `mutation(individual)`, performs a mutation operation on an individual in a genetic algorithm. Here's a breakdown of what it does:

1. **Input:** `individual` is a solution (or individual) in the genetic algorithm. It likely represents a solution to the vehicle routing problem, where each element in the individual represents a truck route.
2. **Mutation Process:**
    - It first checks if the length of the individual is greater than or equal to 2. This is to ensure that there are enough genes (or elements) in the individual to perform a mutation.
    - If the length condition is met:
        - It randomly selects a subset size (`subset_size`) between 1 and the minimum of 3 and the length of the individual. This determines how many genes will be shuffled.
        - It randomly selects `subset_size` indices (`subset_indices`) from the range of indices of the individual.
        - It shuffles the `subset_indices` to randomize the order in which genes will be shuffled.

   - It creates a list of genes to be shuffled (`shuffled_genes`) by extracting the genes at the `subset_indices`.
   - It shuffles the `shuffled_genes` to randomize their order.
   - It updates the individual with the shuffled subset by assigning each gene in `shuffled_genes` back to the corresponding index in `subset_indices`.
3. **Return:** The function returns the mutated individual after shuffling a subset of genes.

```
def mutation(individual):
    # Randomly shuffle a subset of genes
    if len(individual) >= 2:
        subset_size = random.randint(1, min(3, len(individual)))  # Randomly s
        subset_indices = random.sample(range(len(individual)), subset_size)  #
        random.shuffle(subset_indices)  # Shuffle the subset indices
        # Shuffle genes at selected indices
        shuffled_genes = [individual[i] for i in subset_indices]
        random.shuffle(shuffled_genes)
        # Update individual with shuffled subset
        for i, index in enumerate(subset_indices):
            individual[index] = shuffled_genes[i]
    return individual
mutation(population[0])
```

Function `elitism_selection(population, num_elites)`, implements an elitism selection strategy in a genetic algorithm. Here's a breakdown of what it does:

1. **Inputs:**
   - `population` is a list of individuals (solutions) in the genetic algorithm.
   - `num_elites` is the number of top individuals (elites) that will be preserved from the population.
2. **Output:** The function returns a list `elites` containing the top `num_elites` individuals from the population based on their fitness.
3. **Selection Process:**
   - It first sorts the population based on the fitness of the individuals in descending order using the `sorted` function and a lambda function as the key for sorting (`key=lambda ind: ind[1]`, assuming the fitness value is at index 1 in each individual tuple).
   - It then selects the top `num_elites` individuals (elites) from the sorted population to survive and form the next generation.
4. **Return:** The function returns the list `elites`, which contains the top `num_elites` individuals selected based on their fitness, implementing elitism in the genetic algorithm.

```
def elitism_selection(population, num_elites):
    # Sort the population based on fitness in descending order
    sorted_population = sorted(population, key=lambda ind: ind[1], reverse=Tru
    # Select the top individuals (elites) to survive
    elites = sorted_population[:num_elites]

    return elites
```

```
# Example usage:
num_elites = 50  # Set the number of elites to survive
elites = elitism_selection(population, num_elites)
print("Elite individuals:", len(elites))
```

Function `generational_replacement(population, offspring)`, implements a generational replacement strategy in a genetic algorithm. Here's a breakdown of what it does:

1. **Inputs:**
    - `population` is a list of individuals (solutions) in the current generation of the genetic algorithm.
    - `offspring` is a list of new individuals (offspring) generated from the current population through genetic operations such as crossover and mutation.
2. **Output:** The function returns a new population (`new_population`) containing the top individuals selected from the combined population of the current population and the offspring.
3. **Replacement Process:**
    - It combines the current population and the offspring to create a `combined_population`.
    - It sorts the `combined_population` based on the fitness of the individuals in descending order using the `sorted` function and a lambda function as the key for sorting (`key=lambda ind: ind[1]`, assuming the fitness value is at index 1 in each individual tuple).
    - It selects the top individuals from the sorted `combined_population` to survive and form the new population (`new_population`). The number of individuals selected is equal to the size of the original population (`len(population)`), ensuring that the size of the new population remains the same as the original population.
4. **Return:** The function returns the `new_population`, which contains the top individuals selected based on their fitness from the combined population, implementing a generational replacement strategy in the genetic algorithm.

```
def generational_replacement(population, offspring):
    # Combine the current population and the offspring
    combined_population = population + offspring
    # Sort the combined population based on fitness in descending order
    sorted_population = sorted(combined_population, key=lambda ind: ind[1], re
    # Select the top individuals to survive, equal to the size of the original
    new_population = sorted_population[:len(population)]

    return new_population
```

3) Similar applications in the market

1. **Electric Vehicle Routing Problem (EVRP)**:
    - The EVRP is a specialized variant of the VRP that considers the battery capacity of electric vehicles. It aims to optimize routes for electric vehicles while ensuring they can reach their destinations without running out of charge. EVRPs play a crucial role in sustainable transportation and logistics.
2. **OptimoRoute**:
    - OptimolRoute is a web-based route planning and optimization tool.
    - It allows businesses to create efficient delivery routes, considering factors like delivery windows, vehicle capacities, and traffic.
    - Ideal for small to medium-sized businesses.
3. **Work Wave Route Manager**:
    - Work Wave offers a comprehensive route planning solution for field service and delivery businesses.
    - It considers constraints like time windows, vehicle capacities, and driver availability.
    - Provides real-time tracking and reporting.
4. **Waze**:
    - Waze is a community-based navigation app that provides real-time traffic information.
    - It offers route suggestions, alerts about accidents, road closures, and police presence.
    - Particularly useful for delivery drivers and couriers.
5. **Google Maps**:
    - Google Maps is widely used for navigation, route planning, and real-time traffic updates.
    - It helps users find optimal routes for deliveries, considering traffic conditions, distance, and estimated time of arrival.

- Available on both Android and iOS platforms.

6. **Other VRP Variants**:
    - Beyond the classic VRP, there are several variants that address specific constraints:
        - **Capacitated VRP (CVRP)**: Introduces vehicles with limited load capacity.
        - **VRP with Time Windows (VRPTW)**: Considers time constraints for customer visits.
        - **VRP with Heterogeneous Fleets (HFVRP)**: Involves vehicles with different capacities or speeds.
        - **Time-Dependent VRP (TDVRP)**: Accounts for varying travel times based on traffic conditions.
        - **Multi-Depot VRP (MDVRP)**: Involves multiple depots for vehicle dispatch2.

4) 7 literature reviews of Academic publications (papers)

1. "Large-scale evolutionary optimization"
    - **Authors**: Jing Liu, Ruhul Sarker, Saber Elsayed, Daryl Essam, and Nurhadi Siswanto.
    - **Abstract**: The paper provides a comprehensive review of large-scale evolutionary optimization, discussing algorithms for single-objective, multi-objective, and sparse multi-objective large-scale problems. It includes a comparative study, benchmarks, applications, challenges, and future research directions in the field. The abstract emphasizes the importance of designing efficient algorithms to solve complex real-world problems that involve a large number of decision variables. The paper also highlights the need for scalability, efficacy, and the ability to handle constraints and dynamic changes in large-scale optimization problems.
2. "Efficiently solving very large-scale routing problems"
    - **Authors**: Florian Arnold, Michel Gendreau, and Kenneth Sörensen1.
    - **Abstract**: It discusses the development of a local search heuristic for solving very large-scale capacitated vehicle routing problems (CVRP) efficiently23. The heuristic is designed to handle instances with up to 30,000 customers and outperforms previous methods, with a focus on minimizing time and space complexity. The paper also introduces new realistic VRP instances based on real-world data to encourage further research in this domain.
3. "Online Vehicle Routing: The Edge of Optimization in Large-Scale Applications"
    - Authors: Dimitris Bertsimas, Patrick Jaillet, Sebastien Martin
    - Abstract: This paper focuses on the challenges posed by ride-sharing companies that offer transportation on demand at a large scale. The authors develop an optimization framework

and a novel backbone algorithm that allow real-time dispatching of thousands of taxis serving over 25,000 customers per hour. They demonstrate improved performance compared to existing heuristics using historical simulations based on New York City routing network and yellow cabs data

4. **A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems**
    - Authors: Not specified
    - Abstract: In this paper, the authors propose a metaheuristic called FILO to solve large-scale instances of the Capacitated Vehicle Routing Problem. Their approach combines the Iterated Local Search paradigm with acceleration techniques, resulting in an effective solution strategy
5. **"The Vehicle Routing Problem: State-of-the-Art Classification and Review"**:
    - **Authors**: Shi-Yi Tan and Wei-Chang Yeh
    - **Abstract**: This study analyzes literature published between 2019 and August 2021 using a taxonomic framework. The authors review research based on models and solutions, categorizing models into customer-related, vehicle-related, and depot-related. Solution algorithms are classified as exact, heuristic, or meta-heuristic. The study provides a classification table for easy access to relevant literature and recent trends in VRPs and their variants..
6. **"Improving Inbound Logistic Planning for Large-Scale Real-World Routing Problems: A Novel Ant-Colony Simulation-Based Optimization"**:
    - **Authors**: Giovanni Calabrò, Vincenza Torrisi, Giuseppe Inturri, and Matteo Ignaccolo.
    - **Abstract**: The study presents an agent-based model to solve a Capacitated Vehicle Routing Problem (CVRP) for inbound logistics using an Ant Colony Optimization (ACO) algorithm. It focuses on optimizing routes for transporting palletized fruit and vegetables from farms to a main depot in South Italy, aiming to minimize the total distance traveled by trucks. The results show the method’s effectiveness in reducing costs and improving scheduling for large-scale freight transport operations.
7. **"The Vehicle Routing Problem: A Taxonomic Review":**
    - **Authors**: Burak Eksioglu, Arif Volkan Vural, and Arnold Reisman.
    - **Abstract**: The paper presents a comprehensive taxonomy for classifying the literature on the Vehicle Routing Problem (VRP). It defines the VRP domain broadly, encompassing managerial, physical, geographical, and informational considerations, as well as theoretical disciplines. The taxonomy aims to organize the disjointed VRP literature and identify gaps for future research. The paper also compares previous taxonomies and demonstrates the descriptive power and parsimony of the proposed taxonomy through sample articles.

5)About Dataset

Route Optimization - A Real World Scenario

The example demonstrated in this solution accelerator is inspired by a real-world optimization scenario. The customer is a manufacturing company. They have warehouses in different locations. When they receive orders from their clients, a planner need to plan the item-to-truck assignment for order delivery. The planner also need to decide the route of each delivery truck, namely, the sequence of the stops to deliver orders to different destinations. A delivery assignment has its associated cost determined by the type of the assigned delivery truck and the corresponding travelling distance. The optimization objective here is to minimize the overall delivery cost.

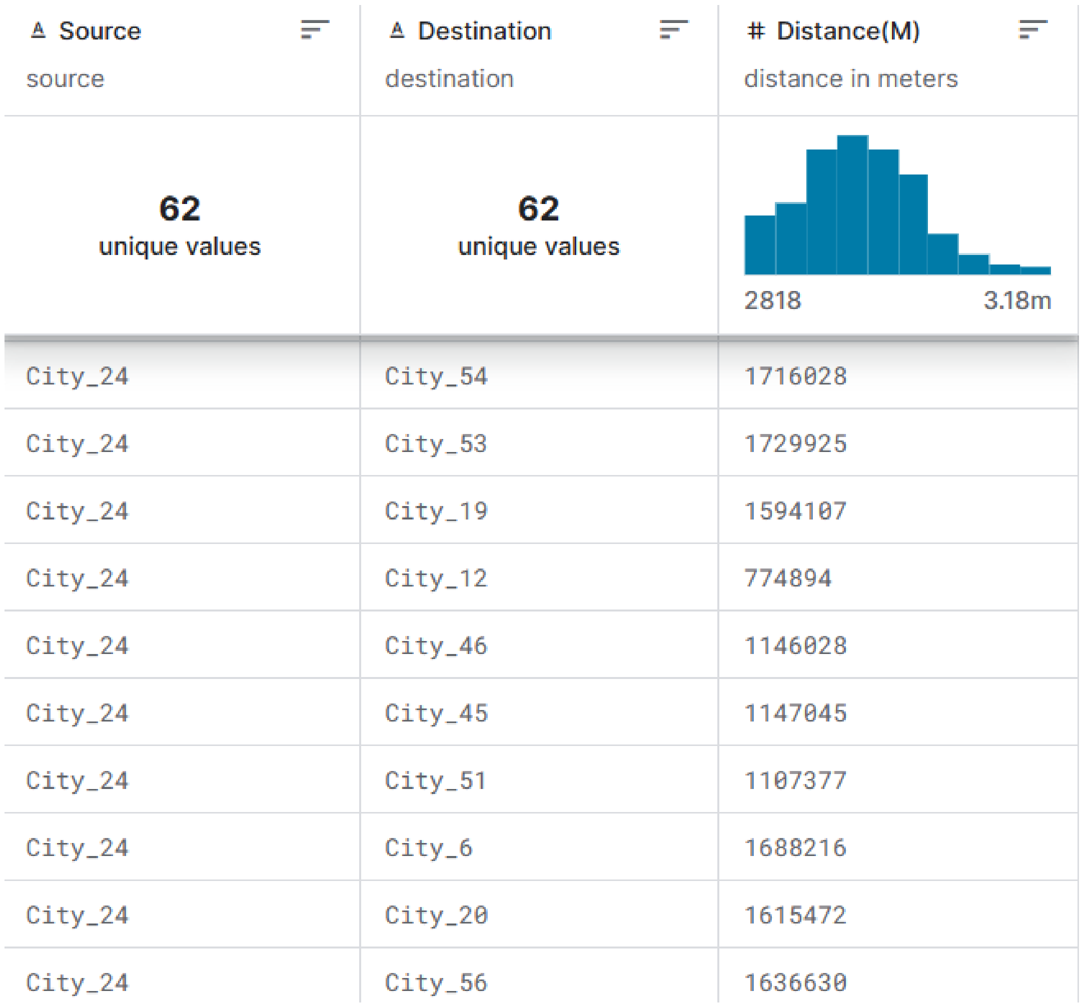
The constraints modeled in our example are:

1. There are different types of trucks we can choose from. A truck has capacity limit on both area and weight. (We assume that there is no limit on the number of trucks for each type)
2. An item is only available by a specific time. A truck can start only when all items assigned to it are available.
3. The available time difference between the earliest and last available items in the same truck should be less than a user defined limit (e.g., 4 hours).
4. All items need to be delivered to their destinations before their deadlines.
5. Depending on the properties of products, some items can put in the same truck, but some cannot.
6. A truck can have at most N stops, where N is a user defined number.
7. A truck need to stay at each stop for M hours to unload the items, where M is a user defined number. Each stop will incur a fixed amount of cost in addition to the delivery cost.

Datasets :

1) distance data

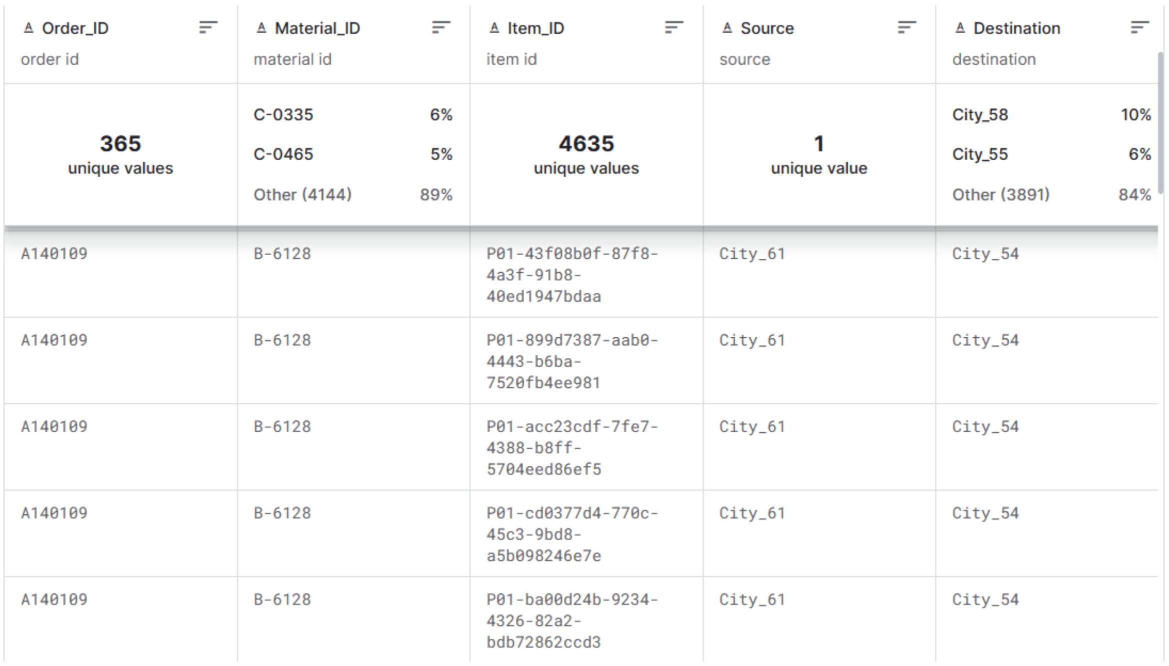
Untitled



Source	Destination	Distance(M)
source	destination	distance in meters
62 unique values	62 unique values	2818 – 3.18m
City_24	City_54	1716028
City_24	City_53	1729925
City_24	City_19	1594107
City_24	City_12	774894
City_24	City_46	1146028
City_24	City_45	1147045
City_24	City_51	1107377
City_24	City_6	1688216
City_24	City_20	1615472
City_24	City_56	1636630

2) large order dataset

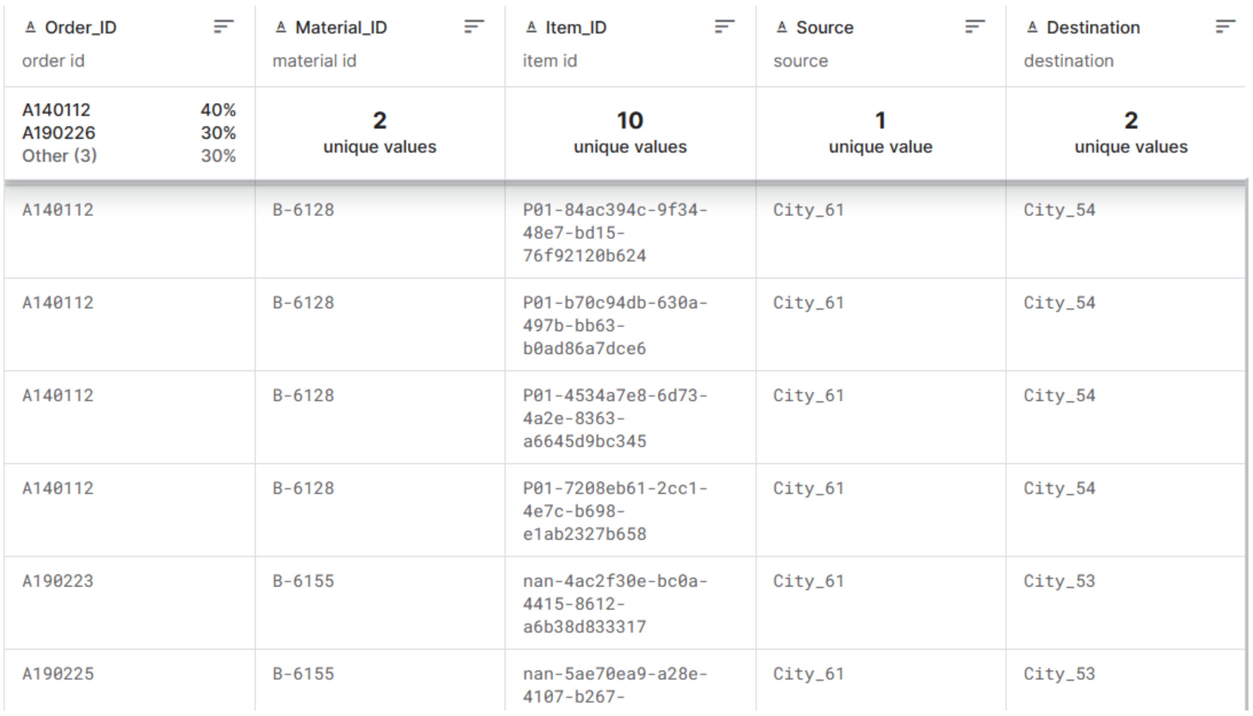
[Untitled]



Order_ID	Material_ID	Item_ID	Source	Destination
order id	material id	item id	source	destination
365 unique values	C-0335 6% C-0465 5% Other (4144) 89%	**4635** unique values	**1** unique value	City_58 10% City_55 6% Other (3891) 84%
A140109	B-6128	P01-43f08b0f-87f8-4a3f-91b8-40ed1947bdaa	City_61	City_54
A140109	B-6128	P01-899d7387-aab0-4443-b6ba-7520fb4ee981	City_61	City_54
A140109	B-6128	P01-acc23cdf-7fe7-4388-b8ff-5704eed86ef5	City_61	City_54
A140109	B-6128	P01-cd0377d4-770c-45c3-9bd8-a5b098246e7e	City_61	City_54
A140109	B-6128	P01-ba00d24b-9234-4326-82a2-bdb72862ccd3	City_61	City_54

3) small order dataset

Untitled



Order_ID	Material_ID	Item_ID	Source	Destination
order id	material id	item id	source	destination
A140112 40% A190226 30% Other (3) 30%	**2** unique values	**10** unique values	**1** unique value	**2** unique values
A140112	B-6128	P01-84ac394c-9f34-48e7-bd15-76f92120b624	City_61	City_54
A140112	B-6128	P01-b70c94db-630a-497b-bb63-b0ad86a7dce6	City_61	City_54
A140112	B-6128	P01-4534a7e8-6d73-4a2e-8363-a6645d9bc345	City_61	City_54
A140112	B-6128	P01-7208eb61-2cc1-4e7c-b698-e1ab2327b658	City_61	City_54
A190223	B-6155	nan-4ac2f30e-bc0a-4415-8612-a6b38d833317	City_61	City_53
A190225	B-6155	nan-5ae70ea9-a28e-4107-b267-	City_61	City_53

4)truck dataset

Untitled

Example Input :

It is a set of items to be delivered, where the Item_ID uniquely defines an item.

Order_ID	Material_ID	Item_ID	Source	Destination	Available_Time	Deadline	Danger_Type	Area (m^2)	Weight (kg)
A140109	B-6128	P01-79c46a02-e12f-41c4-9ec9-25e48597ebfe	City_61	City_54	2022-04-05 23:59:59	2022-04-11 23:59:59	type_1	3.888	3092
A140112	B-6128	P01-84ac394c-9f34-48e7-bd15-76f92120b624	City_61	City_54	2022-04-07 23:59:59	2022-04-13 23:59:59	type_1	3.888	3092
A140112	B-6128	P01-b70c94db-630a-497b-bb63-b0ad86a7dce6	City_61	City_54	2022-04-07 23:59:59	2022-04-13 23:59:59	type_1	3.888	3092

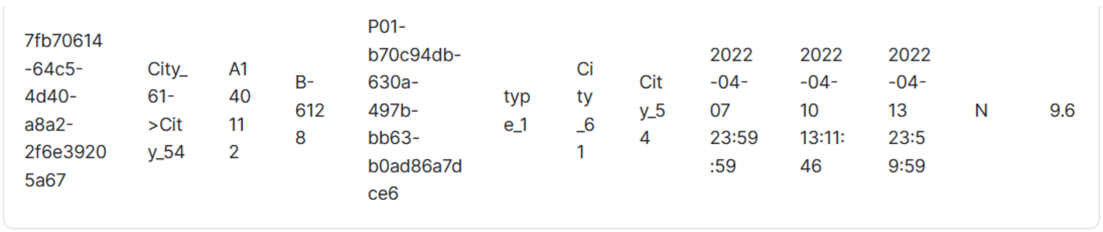
Also assume there are 3 types of trucks available for assignment:

Truck Type (length in m)	Inner Size (m^2)	Weight Capacity (kg)	Cost Per KM	Speed (km/h)
16.5	16.1×2.5	10000	3	40
12.5	12.1×2.5	5000	2	40
9.6	9.1×2.3	2000	1	40

Example Output :

Below is an example output of the route assignment, where Truck_ID uniquely defines a truck. The column Shared_Truck indicates if there are items from different orders sharing the same truck.

Truck_ID	Truck_Route	Order_ID	Material_ID	Item_ID	Danger_Type	Source	Destination	Start_Time	Arrival_Time	Deadline	Shared_Truck	Truck_Type
d27e70e3-e143-4419-8c4a-2faf130e29b3	City_61->City_54	A140109	B-6128	P01-79c46a02-e12f-41c4-9ec9-25e48597ebfe	type_1	City_61	City_54	2022-04-05 23:59:59	2022-04-08 13:11:46	2022-04-11 23:59:59	N	9.6
7fb70614-64c5-4d40-a8a2-2f6e39205a67	City_61->City_54	A140112	B-6128	P01-84ac394c-9f34-48e7-bd15-76f92120b624	type_1	City_61	City_54	2022-04-07 23:59:59	2022-04-10 13:11:46	2022-04-13 23:59:59	N	9.6



Truck_ID	Truck_Route	Order_ID	Material_ID	Item_ID	Danger_Type	Source	Destination	Start_Time	Arrival_Time	Deadline	Shared_Truck	Truck_Type
7fb70614-64c5-4d40-a8a2-2f6e39205a67	City_61->City_54	A140112	B-6128	P01-b70c94db-630a-497b-bb63-b0ad86a7dce6	type_1	City_61	City_54	2022-04-07 23:59:59	2022-04-10 13:11:46	2022-04-13 23:59:59	N	9.6

6)Details of algorithms/approaches used & result

This algorithm is a simple genetic algorithm (GA) used for optimization problems. Here's a detailed explanation of each part:

1. **Constants:**
   - `POPULATION_SIZE`: The number of individuals in each generation.
   - `NUM_GENERATIONS`: The total number of generations.
   - `TOURNAMENT_SIZE`: The size of the tournament selection.
   - `MUTATION_RATE`: The probability of mutation for each gene.
   - `CROSSOVER_RATE`: The probability of crossover for each pair of parents.
2. **Genetic Algorithm Function (`genetic_algorithm`):**
   - **Initialization:** Initializes the population by calling `init_population(POPULATION_SIZE)`, which creates a population of random individuals.

- **Main Loop:** Iterates through each generation (from 1 to `NUM_GENERATIONS`):
    - **Fitness Evaluation:** Evaluates the fitness of each individual in the population using `fitness_function` and stores the fitness scores in `fitness_scores`.
    - **Parent Selection:** Selects parents for crossover using tournament selection (`tournament_selection`) and stores them in the `parents` list. This is done by iterating `TOURNAMENT_SIZE` times and selecting the best individual each time to be a parent.
    - **Crossover:** Performs crossover on selected parents to create offspring (`offspring`). It uses one-point crossover (`one_point_crossover`) with a probability of `CROSSOVER_RATE`. If crossover occurs, two children are added to the offspring list; otherwise, the parents are added unchanged.
    - **Mutation:** Performs mutation on the offspring with a probability of `MUTATION_RATE`, using the `mutation` function.
    - **Replacement:** Replaces the current population with the offspring.
    - **Printing Best Individual:** Prints the best individual in the current generation based on fitness. The best individual is found using the `min` function with a lambda function as the key to compare individuals based on fitness.
- **Return:** Returns the best individual found across all generations, which is the individual with the minimum fitness score.

3. **Example Usage:** Calls `genetic_algorithm` to find the best solution and prints the best solution found.

```
import random

# Define constants
POPULATION_SIZE = 25
NUM_GENERATIONS = 30
TOURNAMENT_SIZE = 5
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.8

# Genetic algorithm function
def genetic_algorithm():
    # Initialize population
    population = init_population(POPULATION_SIZE)

    # Main loop for the number of generations
    for generation in range(NUM_GENERATIONS):
        # Evaluate fitness of each individual in the population
        fitness_scores = [fitness_function(individual) for individual in popul
        # Select parents for crossover using tournament selection
        parents = [tournament_selection(population) for _ in range(POPULATION_
        # [[p1,p2],[p1,p2]]
        # Perform crossover to create offspring
        offspring = []
        for i in range(0, POPULATION_SIZE, 2):
            if random.random() < CROSSOVER_RATE:
                child1, child2 = one_point_crossover(parents[i][0], parents[i]
                offspring.append(child1)
                offspring.append(child2)
            else:
                     offspring.extend([parents[i][0], parents[i][1]])

        # Perform mutation on the offspring
        for i in range(len(offspring)):
            if random.random() < MUTATION_RATE:
                offspring[i] = mutation(offspring[i])

        # Replace the current population with the offspring
        population = offspring

        # Print the best individual in the current generation
        best_individual = min(zip(population, fitness_scores), key=lambda x: x
        print(f"Generation {generation+1}, Best Individual: {best_individual}
        # print(f"Generation {generation+1}, Best Individual: {best_individual

    # Return the best individual from the final generation
    return min(zip(population, fitness_scores), key=lambda x: x[1])[0]

# Example usage
best_solution = genetic_algorithm()
print("Best solution found:", best_solution)
```

Generation 1, Best Individual: [(0, ['A250556', 'A220301', 'A240104', 'A230676']), (0, ['A230153', 'A240110', 'A240425', 'A240112', 'A250403']), (1, ['A220300', 'A270114', 'A220312']), (2, ['A220493']), (1, ['A250555', 'A260606']), (0, ['A250346', 'A250554', 'A220492', 'A220313', 'A240107']), (0, ['A190223', 'A140112', 'A190225', 'A240042', 'A220319']), (1, ['A240045', 'A260061']), (2, ['A220318']), (1, ['A260078']), (0, ['A140109', 'A260605', 'A220494', 'A140110']), (0, ['A250403', 'A190223', 'A250346', 'A230669', 'A220309']), (0, ['A240281', 'A220492', 'A250554', 'A250556']), (0, ['A240109', 'A230153', 'A240049', 'A230667', 'A190225']), (1, ['A260604', 'A230670', 'A270114']), (0, ['A230668', 'A190226', 'A260603', 'A240104', 'A220319']), (1, ['A240042']), (1, ['A240425', 'A230154']), (1, ['A240107', 'A230675', 'A240112']), (1, ['A230676', 'A220321']), (0, ['A220312', 'A260052', 'A250555']), (0, ['A220303', 'A240110', 'A230151', 'A250402']), (2, ['A080377']), (0,

```
['A250404', 'A240061', 'A140112'])]
fitness= 640322
Generation 2, Best Individual: [(0, ['A220309', 'A220319', 'A220494', 'A080377', 'A240107',
'A260376']), (1, ['A230669']), (1, ['A220303', 'A140112']), (2, ['A240045']), (1, ['A220493', 'A220306',
'A260061']), (1, ['A250401']), (1, ['A220313', 'A220308']), (1, ['A250557', 'A080376']), (2,
['A240046']), (1, ['A260078']), (0, ['A140109', 'A260605', 'A220494', 'A140110']), (0, ['A250403',
'A190223', 'A250346', 'A230669', 'A220309']), (0, ['A240281', 'A220492', 'A250554', 'A250556']),
(0, ['A240109', 'A230153', 'A240049', 'A230667', 'A190225']), (1, ['A240107', 'A230155', 'A240061']),
(0, ['A250404', 'A260052', 'A220306', 'A240111']), (2, ['A240045']), (1, ['A230669', 'A230525']), (1,
['A260376', 'A220319']), (1, ['A220310', 'A220312']), (0, ['A230151', 'A240042', 'A230675']), (0,
['A230668', 'A250558', 'A140110'])]
fitness= 466823
Generation 3, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (0, ['A240111', 'A260606',
'A230155', 'A220300', 'A260376']), (0, ['A250558', 'A240426', 'A230525', 'A220301', 'A220318']), (2,
['A240045']), (1, ['A220493', 'A220306', 'A260061']), (1, ['A250401']), (1, ['A220313', 'A220308']), (1,
['A250557', 'A080376']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (0, ['A240281',
'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110', 'A220309']), (1,
['A250401', 'A240109']), (0, ['A260605', 'A250404', 'A230667', 'A240046']), (2, ['A190225']), (0,
['A220312', 'A260376', 'A240107', 'A260604', 'A230151']), (0, ['A230153', 'A240050', 'A140110',
'A250555'])]
fitness= 417969
Generation 4, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (1, ['A220493', 'A220306', 'A260061']), (1, ['A250401']), (1,
['A220313', 'A220308']), (1, ['A250557', 'A080376']), (2, ['A240046']), (2, ['A190225']), (2,
['A260603']), (0, ['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1,
['A140110', 'A220309']), (1, ['A250401', 'A240109']), (0, ['A260605', 'A250404', 'A230667',
'A240046']), (1, ['A230670']), (0, ['A220312', 'A260376', 'A240107', 'A260604', 'A230151']), (0,
['A230153', 'A240050', 'A140110', 'A250555'])]
fitness= 362085
Generation 5, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (0, ['A240111', 'A260606',
'A230155', 'A220300', 'A260376']), (1, ['A250401']), (2, ['A240045']), (1, ['A220493', 'A220306',
'A260061']), (0, ['A250558', 'A240426', 'A230525', 'A220301', 'A220318']), (1, ['A220313',
'A220308']), (1, ['A250557', 'A080376']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (1, ['A250557']), (2, ['A190225']), (0, ['A220312', 'A260376', 'A240107',
'A260604', 'A230151']), (0, ['A230153', 'A240050', 'A140110', 'A250555'])]
fitness= 373154
Generation 6, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (0, ['A240111', 'A260606',
'A230155', 'A220300', 'A260376']), (0, ['A250558', 'A240426', 'A230525', 'A220301', 'A220318']), (2,
['A240045']), (1, ['A220493', 'A220306', 'A260061']), (1, ['A250401']), (1, ['A220313', 'A220308']), (1,
['A250557', 'A080376']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (0, ['A240281',
'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110', 'A220309']), (2,
['A080377']), (1, ['A250557']), (2, ['A190225']), (0, ['A220312', 'A260376', 'A240107', 'A260604',
'A230151']), (0, ['A230153', 'A240050', 'A140110', 'A250555'])]
fitness= 373154
Generation 7, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1, ['A220303',
'A140112']), (2, ['A240045']), (1, ['A220493', 'A220306', 'A260061']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A250557', 'A080376']), (2, ['A240046']), (2, ['A190225']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (1, ['A250557']), (2, ['A190225']), (0, ['A220312', 'A260376', 'A240107',
'A260604', 'A230151']), (0, ['A230153', 'A240050', 'A140110', 'A250555'])]
fitness= 315725
Generation 8, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (1, ['A220493', 'A220306', 'A260061']), (1, ['A250401']), (1,
['A220313', 'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (1, ['A250557']), (2, ['A190225']), (0, ['A220312', 'A260376', 'A240107',
'A260604', 'A230151']), (0, ['A230153', 'A240050', 'A140110', 'A250555'])]
fitness= 310699
Generation 9, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (1, ['A250557']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A230670']), (2, ['A240046']), (1, ['A250557', 'A080376']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (0, ['A220312', 'A260376', 'A240107', 'A260604', 'A230151']), (2,
['A190225']), (2, ['A080377']), (0, ['A230153', 'A240050', 'A140110', 'A250555'])]
fitness= 307966
Generation 10, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (1, ['A250557']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A230670']), (2, ['A240046']), (1, ['A250557', 'A080376']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (0, ['A220312', 'A260376', 'A240107', 'A260604', 'A230151']), (2,
['A190225']), (2, ['A080377']), (0, ['A230153', 'A240050', 'A140110', 'A250555'])]
fitness= 307966
Generation 11, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A230670']), (2, ['A240046']), (1, ['A250557', 'A080376']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (0, ['A220312', 'A260376', 'A240107', 'A260604', 'A230151']), (2,
['A190225']), (2, ['A080377']), (0, ['A230153', 'A240050', 'A140110', 'A250555'])]
fitness= 303458
Generation 12, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A250557', 'A080376']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (1, ['A250557']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 223049
```

```
Generation 13, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A250557', 'A080376']), (0, ['A240281', 'A250557', 'A240061', 'A260078']), (1,
['A250557', 'A080376']), (2, ['A260603']), (2, ['A240046']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (1, ['A250557']), (2, ['A190225']), (2, ['A080377']), (0, ['A230153',
'A240050', 'A140110', 'A250555'])]
fitness= 257996
Generation 14, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A250557', 'A080376']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (1, ['A250557']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 223049
Generation 15, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A250557', 'A080376']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (1, ['A250557']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 223049
Generation 16, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (1, ['A250557', 'A080376']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (0,
['A240281', 'A250557', 'A240061', 'A260078']), (1, ['A250402', 'A190226']), (1, ['A140110',
'A220309']), (2, ['A080377']), (1, ['A250557']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 223049
Generation 17, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A250402', 'A190226']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 192436
Generation 18, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A250402', 'A190226']), (1, ['A140110', 'A220309']), (2, ['A080377']), (1,
['A250557']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 198003
Generation 19, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A250402',
'A190226']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 192436
```

```
Generation 20, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A250402', 'A190226']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 192436
Generation 21, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 181160
Generation 22, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 181160
Generation 23, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 181160
Generation 24, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 181160
Generation 25, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 181160
Generation 26, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 181160
Generation 27, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 181160
Generation 28, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (1, ['A230670']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 181160
Generation 29, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (2, ['A240046']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 172405
Generation 30, Best Individual: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1,
['A220303', 'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313',
'A220308']), (2, ['A080377']), (2, ['A240046']), (2, ['A240046']), (2, ['A260603']), (1, ['A250557',
'A080376']), (1, ['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2,
['A240046']), (2, ['A190225']), (2, ['A080377']), (1, ['A230669'])]
fitness= 172405
Best solution found: [(1, ['A240108', 'A240050', 'A220310']), (1, ['A230669']), (1, ['A220303',
'A140112']), (2, ['A240045']), (2, ['A080377']), (1, ['A250401']), (1, ['A220313', 'A220308']), (2,
['A080377']), (2, ['A240046']), (2, ['A240046']), (2, ['A260603']), (1, ['A250557', 'A080376']), (1,
['A220313', 'A220308']), (1, ['A140110', 'A220309']), (2, ['A080377']), (2, ['A240046']), (2,
['A190225']), (2, ['A080377']), (1, ['A230669'])]
```

7)Development platform

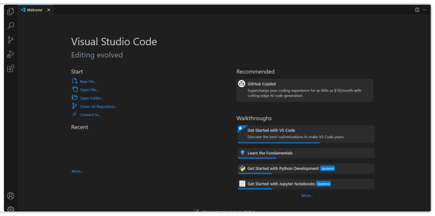
Visual Studio Code, also commonly referred to as **VS Code**,[9] is a source-code editor developed by Microsoft for Windows, Linux, macOS and web browsers.[10][11] Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded version control with Git. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add functionality.

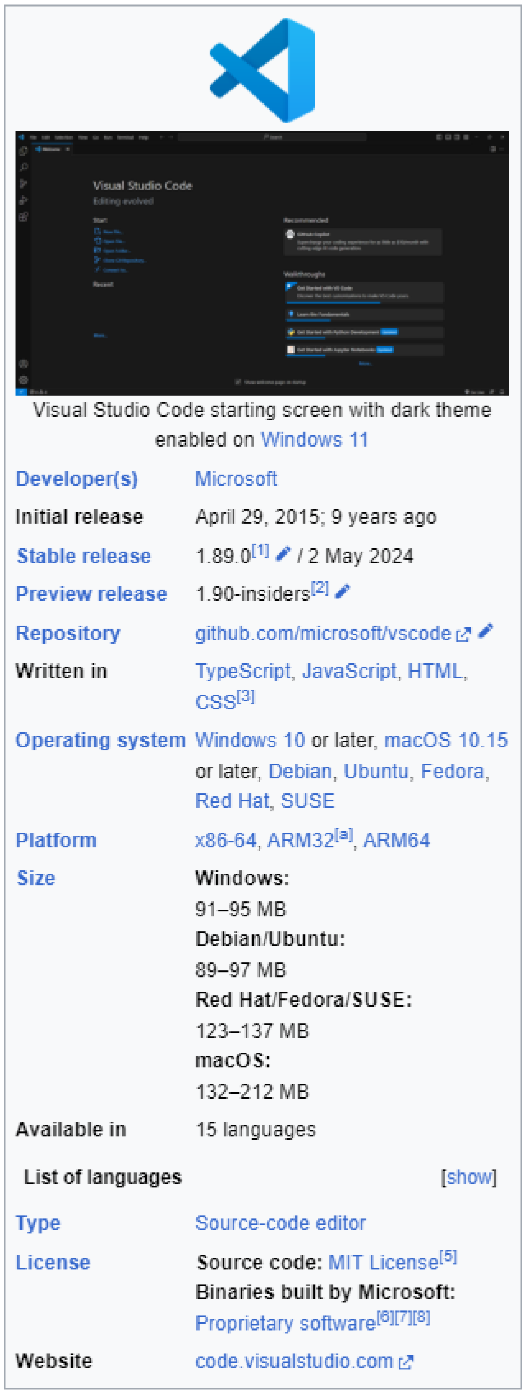
In the Stack Overflow 2023 Developer Survey, Visual Studio Code was ranked the most popular developer environment tool among 86,544 respondents, with 73.71% reporting that they use it. [12]

Visual Studio Code

Visual Studio Code, also commonly referred to as VS Code, is a source-code editor developed by Microsoft for Windows, Linux, macOS and web browsers. Features include support for debugging, syntax highlighting, intelligent code

W https://en.wikipedia.org/wiki/Visual_Studio_Code





Visual Studio Code

Visual Studio Code starting screen with dark theme enabled on Windows 11

Developer(s)	Microsoft
Initial release	April 29, 2015; 9 years ago
Stable release	1.89.0[1] / 2 May 2024
Preview release	1.90-insiders[2]
Repository	github.com/microsoft/vscode
Written in	TypeScript, JavaScript, HTML, CSS[3]
Operating system	Windows 10 or later, macOS 10.15 or later, Debian, Ubuntu, Fedora, Red Hat, SUSE
Platform	x86-64, ARM32[a], ARM64
Size	**Windows:** 91–95 MB **Debian/Ubuntu:** 89–97 MB **Red Hat/Fedora/SUSE:** 123–137 MB **macOS:** 132–212 MB
Available in	15 languages
List of languages	[show]
Type	Source-code editor
License	**Source code:** MIT License[5] **Binaries built by Microsoft:** Proprietary software[6][7][8]
Website	code.visualstudio.com

Reception[edit]

In the 2016 Developers Survey of Stack Overflow, Visual Studio Code ranked No. 13 among the top popular development tools, with only 7% of the 47,000 respondents using it.[33] Two years later, however, Visual Studio Code achieved the No. 1 spot, with 35% of the 75,000 respondents using it.[34] In the 2019 Developers Survey, Visual Studio Code was also ranked No. 1, with 50% of the 87,000 respondents using it.[35] The 2020 Developers Survey did not cover integrated development environments.[36] In the 2021 Developers Survey, Visual Studio Code continued to be ranked No. 1, with 74.5% of the 71,000 respondents using it,[37] 74.48% of the 71,010 responses in the 2022 survey,[38] and 73.71% of the 86,544 responses in the 2023 survey.[39]