ARTIFICIAL AND COMPUTATIONAL INTELLIGENCE

ASSIGNMENT 1

GPS AGENT

PARTICIPANTS (BITS ID):

1. **Karamveer Singh (2023aa05649)**
2. **Ravi Sharma (2023aa05622)**
3. **Anshuman Gaonsindhe - 2023ab05150**
4. **Amit Kumar - 2023aa05262**
5. **Vishal Pankaj - 2023aa05260**

ALGORITHMS USED TO SOLVE THE PROBLEM:

- **Greedy Best first search**
- **Genetic algorithm**

QUESTION 1:

PEAS FOR THE GPS AGENT:

PERFORMANCE

- The output which we get from the agent. All the necessary results that an agent gives after processing comes under its performance.
- The performance measure in this context could be the quality of the path found by the algorithm. It could be quantified by factors such as:
    - ✓ Length of the path (number of nodes traversed).
    - ✓ Time taken to find the path.
    - ✓ Optimality of the path (e.g., whether it is the shortest path).
    - ✓ Number of nodes expanded during the search.

ENVIRONMENT:

- All the surrounding things and conditions of an agent fall in this section. It basically consists of all the things under which the agents work.
- In this case, the environment is a graph representing a geographical area with cities as nodes and roads as connecting lines. The agent can move from one city to another along the lines of the graph.

ACTUATORS:

- The devices, hardware or software through which the agent performs any actions or processes any information to produce a result are the actuators of the agent.
- In this case, the agent can perform the following actions:
    - ✓ Expand nodes (traverse edges) to explore the search space.
    - ✓ Select the next node to expand based on the heuristic values.

SENSORS:

- The devices through which the agent observes and perceives its environment are the sensors of the agents.
- In this case, the agent uses sensors to:
    - ✓ Obtain information about the connectivity of nodes (neighbors) in the graph.
    - ✓ Obtain heuristic values for nodes to guide the search.

PROBLEM-SOLVING AGENT (PSA):

GOAL:

The goal of the agent is to find the shortest path from a given start city to a goal city in a graph representing a geographical area.

PERCEPTION:

The agent perceives the current state of the search algorithm, which includes:

- The current node is being expanded.
- The list of nodes has expanded so far.
- The data structure represents the frontier, containing nodes to be explored.

KNOWLEDGE BASE:

The agent has knowledge about the graph structure, representing the connectivity between cities. It knows the start city and the goal city. It may have access to additional information, such as heuristic values for nodes.

SEARCH STRATEGY:

The agent employs a specific search algorithm, such as Greedy best-First Search (GBFS) and Genetic algorithm to explore the search space and find the shortest path.

ACTION:

The main action of the agent is to expand nodes in the search space according to the chosen search algorithm.

EXECUTION:

The agent executes the chosen search algorithm until either the goal city is reached or the frontier becomes empty. It iteratively expands nodes, updating its state and the frontier until the goal city is found or no path exists.

FEEDBACK:

If the goal city is reached, the agent reports the shortest path found from the start city to the goal city. If no path exists, the agent reports that no path is found.

QUESTION 2:

You decide to use the 'haversine' formula to calculate the great-circle distance between two points – that is, the shortest distance over the earth's surface between two points. Using the below latitude and longitude data for the cities, create a function which calculates the heuristic distance from each city to the destination city (refer link https://www.movable-type.co.uk/scripts/latlong.html for more information on Haversine formula).

OUTPUT HEURISTICS:

```
{'Panji': 744.0351557019155,
 'Raichur': 468.7090788152219,
 'Mangalore': 586.9476757845566,
 'Bellari': 427.4793247479098,
 'Tirupati': 110.33441493944744,
 'Kurnool': 388.599828847867,
 'Kozhikode': 528.508520052874,
 'Bangalore': 290.1720249530604,
 'Nellore': 154.2976211368278,
 'Chennai': 0.0}
```

CODE EXECUTION FLOW:

GREEDY BEST FIRST SEARCH:

INITIALIZATION:

- The GBFS algorithm dynamically accepts start and goal cities as input parameters, providing flexibility in defining the search's initial and final points. This process is facilitated through a function call **(get_start_end_states())** that sets these parameters, alongside initializing a priority queue for guiding the search based on heuristic distances.
- In the given execution example, Panaji is selected as the start city and Chennai as the goal city, illustrating the algorithm's adaptability to user-defined conditions.

- Code Snippet –

```
# Call the function to get start and end states
start_city, end_city = get_start_end_states(cities_list)

# Prepare initial state
initial_state = set_initial_state(start_city, end_city)
```

- Output –

```
1. Panaji
2. Raichur
3. Mangalore
4. Bellari
5. Tirupathi
6. Kurnool
7. Kozhikode
8. Bangalore
9. Nellore
10. Chennai
Initlializing start city : Panaji and goal city : Chennai
```

CORE LOOP:

- A while loop runs as long as there are nodes to explore in the priority queue. It selects the node closest to the goal based on the heuristic function.
- Code Snippet –

```
# time & space of gbfs
import time
start_time = time.perf_counter()
# Implement GBFS with cost
path_gbfs, total_cost_gbfs, node_explored,edge_explored = greedy_best_first_search(graph, get_heuristics(end_city),initial_s
end_time = time.perf_counter()
```

HEURISTIC CALCULATION:

- For each neighboring city of the current node, the code calculates the heuristic cost (distance to the goal) using the haversine formula. This cost guides the selection of the next city to explore.
- Code Snippet

```python
# Funnction calculating distance based on latitude and logitude in kms.
def haversine(lat1, lon1, lat2, lon2):
    """
    Calculate the great circle distance in kilometers between two points
    on the earth (specified in decimal degrees)
    """
    # Convert decimal degrees to radians
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

    # Haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = math.sin(dlat/2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon/2)**2
    c = 2 * math.asin(math.sqrt(a))
    r = 6371 # Radius of earth in kilometers.
    return c * r

# Function to calculate cost matrix of each city.
def set_transition_and_cost_matrix(cities_info):
    """
    Initializes a matrix/dictionary representing the graph of cities with the cost of travel between them.

    :param cities_info: A dictionary with city names as keys and their lat, long as values.
    :return: A dictionary of dictionaries representing the distances between cities.
    """
    matrix = {}
    for city, (lat1, lon1) in cities_info.items():
        matrix[city] = {}
        for destination, (lat2, lon2) in cities_info.items():
            if city != destination: # Avoid distance to itself
                distance = haversine(lat1, lon1, lat2, lon2)
                matrix[city][destination] = distance
    return matrix

# Function calculating heuristic value from all the nodes to the goal city city node.
def get_heuristics(goal_city):
    heuristics = {}
    for k in cities_info.keys():
        distance = haversine(*cities_info[k],*cities_info[goal_city])
        heuristics[k]= distance
    return heuristics
```

- Output – transition cost matrix

Out[64]:

	Panaji	Raichur	Mangalore	Bellari	Tirupathi	Kurnool	Kozhikode	Bangalore	Nellore	Chennai
Panaji	0.000000	384.703347	307.213146	334.067476	636.349903	452.250193	515.797860	493.225615	671.742877	744.035156
Raichur	384.703347	0.000000	453.789435	127.191089	363.073048	85.058151	575.671548	360.817146	344.493606	468.709079
Mangalore	307.213146	453.789435	0.000000	332.965424	500.196520	471.576519	209.713852	296.849250	579.748933	586.947676
Bellari	334.067476	127.191089	332.965424	0.000000	317.157600	141.997819	448.826986	251.746981	338.511264	427.479325
Tirupathi	636.349903	363.073048	500.196520	317.157600	0.000000	286.159311	474.903874	210.532503	109.240956	110.334415
Kurnool	452.250193	85.058151	471.576519	141.997819	286.159311	0.000000	563.591889	321.186102	259.819051	388.599829
Kozhikode	515.797860	575.671548	209.713852	448.826986	474.903874	563.591889	0.000000	274.174956	577.222190	528.508520
Bangalore	493.225615	360.817146	296.849250	251.746981	210.532503	321.186102	274.174956	0.000000	305.804085	290.172025
Nellore	671.742877	344.493606	579.748933	338.511264	109.240956	259.819051	577.222190	305.804085	0.000000	154.297621
Chennai	744.035156	468.709079	586.947676	427.479325	110.334415	388.599829	528.508520	290.172025	154.297621	0.000000

PATH SELECTION:

- Once the goal is reached, the code reconstructs the path from the start city to Chennai by backtracking through the explored nodes.
- Code Snippet

```
# GBFS Loop
while frontier:

    print("Open List:", [(heuristic, cost, city) for heuristic, cost, city in frontier])  # Print the open list
    # Choose the city with the lowest heuristic value
    _, current_cost, current_city = heapq.heappop(frontier)

    nodes_explored += 1  # Increment nodes explored when a node is popped from the frontier

    if current_city == goal:
        print("Closed List:", explored)  # Print the closed list upon reaching the goal
        return reconstruct_path(came_from, start, goal), current_cost, nodes_explored, edges_explored

    # Mark the city as explored
    explored.add(current_city)
    print("Closed List:", explored)  # Print the closed list every time a city is explored

    successors = transition_model(current_city, graph)

    # Add neighboring cities to the frontier if they haven't been explored or
    # if a cheaper path to them has been found
    # Use the transition_model function to get successors and their costs
    for neighbor, distance in successors:
        edges_explored += 1  # Increment edges explored for each neighbor examined
        new_cost = current_cost + distance
        if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
            cost_so_far[neighbor] = new_cost
            heapq.heappush(frontier, (heuristics[neighbor], new_cost, neighbor))
            came_from[neighbor] = current_city


# If the goal was never reached, return None
return None, None,nodes_explored, edges_explored
```

- **Output –**

```
Open List: [(744.0351557019155, 0, 'Panaji')]
Closed List: {'Panaji'}
Open List: [(468.7090788152219, 457, 'Raichur'), (586.9476757845566, 365, 'Mangalore')]
Closed List: {'Raichur', 'Panaji'}
Open List: [(110.33441493944744, 910, 'Tirupathi'), (586.9476757845566, 365, 'Mangalore'), (388.599828847867, 557, 'Kurnoo
l')]
Closed List: {'Tirupathi', 'Raichur', 'Panaji'}
Open List: [(0.0, 1063, 'Chennai'), (388.599828847867, 557, 'Kurnool'), (427.4793247479098, 1289, 'Bellari'), (586.9476757845
566, 365, 'Mangalore')]
Closed List: {'Tirupathi', 'Raichur', 'Panaji'}
Path found from GBFS:
Panaji -> Raichur -> Tirupathi -> Chennai
Total cost: 1063 km
```

GENETIC ALGORITHM:

POPULATION INITIALIZATION:

- The Genetic Algorithm (GA) begins by dynamically generating an initial population, based on input parameters, consisting of random valid paths between cities. Each path, constructed from these dynamic inputs, represents a potential solution to efficiently reach a user-defined goal city, showcasing the algorithm's flexibility and adaptability to varying scenarios.
- Code Snippet

```
import random
# Function to calculate valid path for initial population.
def generate_valid_path(start_city, goal_city, graph):
    """
    Generate a valid path from start to city to goal city based on map connections.

    :param start_city: The starting city for the path.
    :param goal_city: The goal city for the path.
    :param graph: A dictionary or similar structure representing valid connections between cities.
    :return: A list representing a valid path.
    """
    path = [start_city]
    current_city = start_city
    while current_city != goal_city:
        # Get valid next steps from current city
        next_cities = [city for city in graph[current_city] if city not in path]
        if not next_cities:  # No valid next city.
            break
        next_city = random.choice(next_cities)
        path.append(next_city)
        current_city = next_city
    return path

# Function to create the initial state for ga: a population of paths
def set_initial_state_for_ga(start, end, cities, population_size):
    # Initialize the population with random valid paths
    population = []
    for _ in range(population_size):
        path = generate_valid_path(start, end, cities)
        population.append(path)
    return population
```

In [76]:

```
#Invoke algorithm 2 (Should Print the solution, path, cost etc., (As mentioned in the problem))

start_city, goal_city = get_start_end_states(cities_list)

import time
start_time_ga = time.perf_counter()
population_size = 100
population = set_initial_state_for_ga(start_city, goal_city, graph, population_size)
print("Population \n",population)
```

- Output –

```
1. Panaji
2. Raichur
3. Mangalore
4. Bellari
5. Tirupathi
6. Kurnool
7. Kozhikode
8. Bangalore
9. Nellore
10. Chennai
Initlializing start city : Panaji and goal city : Chennai
Population
 [['Panaji', 'Mangalore', 'Bangalore', 'Kozhikode'], ['Panaji', 'Raichur', 'Tirupathi', 'Bellari', 'Bangalore', 'Kozhikode',
'Mangalore'], ['Panaji', 'Raichur', 'Kurnool', 'Nellore', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaj
i', 'Raichur', 'Kurnool', 'Nellore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Kozhikode'], ['Panaji', 'Mangalore',
'Kozhikode', 'Bangalore', 'Bellari', 'Tirupathi', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Ra
ichur', 'Kurnool', 'Nellore', 'Chennai'], ['Panaji', 'Raichur', 'Kurnool', 'Nellore', 'Chennai'], ['Panaji', 'Raichur', 'Kurn
ool', 'Nellore', 'Chennai'], ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Bellari', 'Tirupathi', 'Raichur', 'Kurnool',
'Nellore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Be
llari', 'Tirupathi', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Kozhikode'], ['Panaji', 'Mangalore', 'Bangalore', 'Koz
hikode'], ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Bellari', 'Tirupathi', 'Chennai'], ['Panaji', 'Mangalore', 'Kozh
ikode', 'Bangalore', 'Bellari', 'Tirupathi', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaji', 'Mangalor
e', 'Kozhikode', 'Bangalore', 'Chennai'], ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Bellari', 'Tirupathi', 'Chenna
i'], ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Bellari', 'Tirupathi', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi',
'Bellari', 'Bangalore', 'Kozhikode', 'Mangalore'], ['Panaji', 'Raichur', 'Tirupathi', 'Bellari', 'Bangalore', 'Kozhikode', 'M
angalore'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaji', 'Raichu
r', 'Tirupathi', 'Bellari', 'Bangalore', 'Mangalore', 'Kozhikode'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaji',
'Raichur', 'Tirupathi', 'Bellari', 'Bangalore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Raic
hur', 'Kurnool', 'Nellore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Bellari', 'Tirupathi', 'Raichur', 'Kurnool', 'N
ellore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Kozhikode'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Pana
ji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Raichu
r', 'Tirupathi', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chenna
i'], ['Panaji', 'Raichur', 'Tirupathi', 'Bellari', 'Bangalore', 'Mangalore', 'Kozhikode'], ['Panaji', 'Mangalore', 'Kozhikod
e', 'Bangalore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi', 'Bellari',
'Bangalore', 'Kozhikode', 'Mangalore'], ['Panaji', 'Raichur', 'Tirupathi', 'Bellari', 'Bangalore', 'Mangalore', 'Kozhikode'],
['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Mangalore', 'Ba
ngalore', 'Chennai'], ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Chennai'], ['Panaji', 'Raichur', 'Kurnool', 'Nellor
```

FITNESS EVALUATION:

- The fitness of each path is evaluated based on its total distance. Shorter paths receive a higher fitness score, indicating a better solution.
- Code Snippet –

```python
def calculate_total_distance(path, distances):
    """
    Calculate the total distance of the given path based on the road distances.

    :param path: A list of tuples representing the path (e.g., [('City1', 'City2'), ('City2', 'City3'), ...]).
    :param distances: A dictionary with city tuples as keys and distances as values.
    :return: The total distance of the path.
    """
    total_distance = 0
    for i in range(len(path) - 1):
        city_pair = (path[i], path[i+1])
        total_distance += distances.get(city_pair, 0)
    return total_distance

def fitness_function(path, distances):
    total_distance = calculate_total_distance(path, distances)
    if total_distance <= 0:
        return 0
    return 1 / total_distance
```

EVOLUTIONARY OPERATIONS:

- The code performs selection, crossover, and mutation operations to generate new populations. Selection picks the fittest paths, crossover mixes segments of paths between pairs of selected paths, and mutation introduces random changes to paths to explore new solutions.
- Code Snippet –

```python
# Function to select parents from population based their fitness score.
def selection(population, fitness_scores, selection_size):
    # Normalize fitness scores to use as probabilities for selection
    total_fitness = sum(fitness_scores)

    # Ensure total fitness is greater than zero and is finite
    if total_fitness <= 0 or not math.isfinite(total_fitness):
        raise ValueError("Invalid total fitness score: Must be greater than zero and finite.")

    selection_probabilities = [score / total_fitness for score in fitness_scores]

    # Select individuals based on their normalized fitness scores
    selected_indices = random.choices(
        range(len(population)),
        weights=selection_probabilities,
        k=selection_size
    )

    # Create a new list containing the selected individuals
    selected_population = [population[i] for i in selected_indices]
    return selected_population

#Function to generate the offspring/child from selected parents.
def crossover(parent1, parent2, cities):
    # Ensure paths are long enough for crossover
    min_length = min(len(parent1), len(parent2))
    if min_length <= 3:
        # If paths are too short, return them as is or handle differently
        return parent1, parent2

    # Proceed with crossover
    crossover_point = random.randint(1, min_length - 2)
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]

    # Ensure the offspring are valid paths
    offspring1 = [city for city in cities if city in offspring1]
    offspring2 = [city for city in cities if city in offspring2]

    return offspring1, offspring2
```

```python
#Function to generate strongest child by mutating offsprings/children.
def mutate(path, mutation_rate):
    """
    Mutate a path by swapping two cities with a given probability.

    :param path: The path to mutate.
    :param mutation_rate: The probability of mutating a given gene.
    :return: The mutated path.
    """
    mutated_path = path.copy()
    for i in range(len(mutated_path)):
        if random.random() < mutation_rate:
            swap_index = random.randint(0, len(mutated_path) - 1)
            # Swap the cities at i and swap_index
            mutated_path[i], mutated_path[swap_index] = mutated_path[swap_index], mutated_path[i]
    return mutated_path
```

- Output -

```
Parents: ['Panaji', 'Raichur', 'Kurnool', 'Nellore', 'Chennai'], ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Chennai']
| Offspring after crossover: ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Chennai'], ['Panaji', 'Raichur', 'Kurnool',
'Nellore', 'Chennai']
Gene before mutation: ['Panaji', 'Raichur', 'Kurnool', 'Nellore', 'Chennai'] | Gene after mutation: ['Panaji', 'Chennai', 'Ku
rnool', 'Nellore', 'Raichur']
Parents: ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Bellari', 'Tirupathi', 'Chennai'], ['Panaji', 'Raichur', 'Kurnoo
l', 'Nellore', 'Chennai'] | Offspring after crossover: ['Panaji', 'Mangalore', 'Kozhikode', 'Nellore', 'Chennai'], ['Panaji',
'Raichur', 'Bellari', 'Tirupathi', 'Kurnool', 'Bangalore', 'Chennai']
Gene before mutation: ['Panaji', 'Raichur', 'Bellari', 'Tirupathi', 'Kurnool', 'Bangalore', 'Chennai'] | Gene after mutation:
['Raichur', 'Panaji', 'Chennai', 'Tirupathi', 'Kurnool', 'Bangalore', 'Bellari']
Parents: ['Panaji', 'Mangalore', 'Kozhikode', 'Bangalore', 'Bellari', 'Tirupathi', 'Chennai'], ['Panaji', 'Raichur', 'Kurnoo
l', 'Nellore', 'Chennai'] | Offspring after crossover: ['Panaji', 'Mangalore', 'Kozhikode', 'Nellore', 'Chennai'], ['Panaji',
'Raichur', 'Bellari', 'Tirupathi', 'Kurnool', 'Bangalore', 'Chennai']
Parents: ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'] | Offspring after cro
ssover: ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai']
Gene before mutation: ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'] | Gene after mutation: ['Panaji', 'Bangalore', 'Mangalo
re', 'Chennai']
Parents: ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'] | Offspring after cro
ssover: ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Chennai']
No crossover applied for parents: ['Panaji', 'Raichur', 'Kurnool', 'Nellore', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi',
'Chennai']
Parents: ['Panaji', 'Mangalore', 'Bangalore', 'Bellari', 'Tirupathi', 'Raichur', 'Kurnool', 'Nellore', 'Chennai'], ['Panaji',
'Raichur', 'Kurnool', 'Nellore', 'Chennai'] | Offspring after crossover: ['Panaji', 'Mangalore', 'Bangalore', 'Nellore', 'Che
nnai'], ['Panaji', 'Raichur', 'Bellari', 'Tirupathi', 'Kurnool', 'Nellore', 'Chennai']
Gene before mutation: ['Panaji', 'Mangalore', 'Bangalore', 'Nellore', 'Chennai'] | Gene after mutation: ['Chennai', 'Nellor
e', 'Bangalore', 'Mangalore', 'Panaji']
Parents: ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi', 'Bellari', 'Bangalore', 'Kozhikod
e', 'Mangalore'] | Offspring after crossover: ['Panaji', 'Raichur', 'Mangalore', 'Bellari', 'Tirupathi', 'Kozhikode', 'Bangal
ore'], ['Panaji', 'Raichur', 'Tirupathi', 'Chennai']
Gene before mutation: ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'] | Gene after mutation: ['Chennai', 'Raichur', 'Tirupath
i', 'Panaji']
No crossover applied for parents: ['Panaji', 'Raichur', 'Tirupathi', 'Chennai'], ['Panaji', 'Mangalore', 'Bangalore', 'Kozhik
ode']
No crossover applied for parents: ['Panaji', 'Mangalore', 'Bangalore', 'Chennai'], ['Panaji', 'Raichur', 'Tirupathi', 'Chenna
i']
```

ITERATION AND CONVERGENCE:

- These steps are repeated for a set number of generations or until a satisfactory solution emerges. With each generation, the population is expected to evolve towards more optimal paths.
- Code Snippet –

```python
for generation in range(generations):
    fitness_scores = [fitness_function(path, distances) for path in population]

    elites_indices = sorted(range(len(fitness_scores)), key=lambda i: fitness_scores[i], reverse=True)[:number_of_elites
    elites = [population[i] for i in elites_indices]

    parents = selection(population, fitness_scores, len(population) // 2)
    if len(parents) % 2 != 0:
        parents.pop()

    offspring = []
    for i in range(0, len(parents), 2):
        if random.random() < crossover_rate:
            child1, child2 = crossover(parents[i], parents[i+1], cities)
            print(f"Parents: {parents[i]}, {parents[i+1]} | Offspring after crossover: {child1}, {child2}")
        else:
            child1, child2 = parents[i], parents[i+1]
            print(f"No crossover applied for parents: {parents[i]}, {parents[i+1]}")

        child1_before_mutation = child1[:]
        child1 = mutate(child1, mutation_rate)
        if child1 != child1_before_mutation:
            print(f"Gene before mutation: {child1_before_mutation} | Gene after mutation: {child1}")

        child2_before_mutation = child2[:]
        child2 = mutate(child2, mutation_rate)
        if child2 != child2_before_mutation:
            print(f"Gene before mutation: {child2_before_mutation} | Gene after mutation: {child2}")

        if not is_path_valid(child1, graph):
            child1 = repair_path(child1, start_city, goal_city, graph)
        if not is_path_valid(child2, graph):
            child2 = repair_path(child2, start_city, goal_city, graph)

        offspring.extend([child1, child2])

    new_population = elites + offspring[:population_size - len(elites)]
    population = new_population

    for elite in elites:
        elite_fitness = fitness_function(elite, distances)
        if is_goal_reached(elite[-1], goal_city) and elite_fitness > best_fitness:
            best_path = elite
            best_fitness = elite_fitness
```

SOLUTION EXTRACTION:

- The best path found throughout the generations is presented as the solution. This path is the most efficient route to Chennai discovered by the algorithm.
- Output –

```
Solution Found from Genetic Algorithm:
Path: Panaji -> Raichur -> Kurnool -> Nellore -> Chennai
Cost (Total Distance): 1057.0 km
```

COMPARISON – GREEDY BEST FIRST SEARCH VS GENETIC ALGORITHM:

OPTIMALITY:

GBFS may not always find the optimal solution, especially if the heuristic function is not accurate or the search space is complex. On the other hand, GA has the potential to find better solutions through evolutionary processes but may take longer to converge.

GBFS --> Panaji -> Raichur -> Tirupathi -> Chennai | Total cost: 1063 km

GA --> Panaji -> Raichur -> Kurnool -> Nellore -> Chennai | Total Cost : 1057.0 km

EFFICIENCY:

GBFS is generally more efficient in terms of time complexity for finding a single shortest path, but it may not perform well in complex search spaces or when the heuristic function is not well-suited. GA, although slower, can explore a larger portion of the search space and potentially find better solutions.

GBFS - Time complexity: 0.000112 seconds.

GA - Time complexity: 0.002497 seconds

RESOURCE USAGE:

GBFS typically requires less memory compared to GA, as it only needs to store information about the search frontier and the explored nodes. GA, on the other hand, requires storing a population of potential solutions, which can consume more memory.