Table of Contents

[**1.** **Contribution Table** 2](#_Toc159106237)

[**2.** **Problem Statement** 2](#_Toc159106238)

[**3.** **Identification & Design of the environment** 3](#_Toc159106239)

[3.1 PEAS Environment 3](#_Toc159106240)

[3.2 Properties of Environment: 4](#_Toc159106241)

[3.3 Problem Solving Agents (PSA): 4](#_Toc159106242)

[**4.** **Algorithms Used for the Solution** 5](#_Toc159106243)

[4.1 Greedy Best-First Search (GBFS) 5](#_Toc159106244)

[4.1.1 Algorithms pseudo code: 5](#_Toc159106245)

[4.1.2 Data structures used: 7](#_Toc159106246)

[4.1.3 Heuristic/Fitness Design: 7](#_Toc159106247)

[4.2 Genetic Algorithm 8](#_Toc159106248)

[4.2.1 Genetic Algorithm pseudo code 9](#_Toc159106249)

[4.2.2 Data structures used: 11](#_Toc159106250)

[4.2.3 Heuristic/Fitness Design: 12](#_Toc159106251)

[5. **Steps to Execute Python Code:** 12](#_Toc159106252)

[**6.** **Test Results** 13](#_Toc159106253)

[6.1 Greedy Best First Search: 13](#_Toc159106254)

[6.2 Genetic Algorithm 14](#_Toc159106255)

[**7.** **Comparative Analysis** 15](#_Toc159106256)

[7.1 Time Complexity: 15](#_Toc159106257)

[7.2 Space Complexity: 15](#_Toc159106258)

[**8.** **Conclusion:** 16](#_Toc159106259)

[8.1 Path Comparison 16](#_Toc159106260)

[8.2 Actual Cost Comparison 16](#_Toc159106261)

[8.3 Time & Space Complexity: 16](#_Toc159106262)

**(S1-23\_AIMLCZG557)**

**(Artificial and Computational Intelligence)**

**Academic Year 2023-2024**

**Assignment 1 – Problem Statement-9  
Autonomous battery-operated micro aquatic boat**

# **Contribution Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl. No.** | **Name (as appears in Canvas)** | **ID NO** | **Email ID** | **Contribution** |
| **1** | **Hemant Kumar Parakh** | **2023AA05741** | 2023aa05741@wilp.bits-pilani.ac.in | **100%** |
| **2** | **Sushil Kumar** | **2023aa05849** | 2023aa05849@wilp.bits-pilani.ac.in | **100%** |
| **3** | **NAGINENI SATISH BABU** | **2023aa05585** | 2023aa05585@wilp.bits-pilani.ac.in | **100%** |
| **4** |  |  |  |  |
| **5** |  |  |  |  |

# **Problem Statement**

Assume that you are given the responsibility to supply food, water, medicines and other amenities to the flooded areas in the city of Chennai. It is your responsibility to supply the needs to all the areas affected by flood where people are suffering. You are provided with a map of the city with flooded land-marks marked (vertices). You are provided with an autonomous battery-operated micro aquatic boat that detects the flooded areas and responds to people who wave to it for help. The autonomous battery-operated micro aquatic boat works on a battery and hence it has to take a path such that all the roads (edges in the graph) are covered, but no road is repeated more than once. After getting the details, the boat agent reports to you, the taken path and the locations where people are requesting for help.

The problem here is to find the shortest route that travels through all the lanes in the area. The places can be visited more than once. The shortest path includes all the roads travelled only once. Help your autonomous battery-operated micro aquatic boat in finding such a path given a starting point and the map.

Here the area map is represented as a graph. The algorithm takes the starting point and the graph as the input and produces the shortest path covering all the edges only once.



Use the following algorithms to solve the problem:

1. Greedy Best first search

2. Genetic algorithm

**Problem Solution**

# **Identification & Design of the environment**

## PEAS Environment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agent | Performance Measure | Environment | Sensors | Actuators |
| Autonomous battery-operated micro aquatic boat | Shortest Path which covers all flooded areas in the city | flooded areas in the city, Boat to navigate the flooded area | Sensors to detect and identify flooded areas, GPS for navigation, cameras or other sensors to detect people waving for help, and battery level sensors to monitor the energy consumption. | Controls of the autonomous battery-operated micro aquatic boat. These actuators control the movement of the boat, allowing it to navigate through the flooded areas based on the calculated path |

## Properties of Environment

**Observable (Fully/Partially):** It’s a **Partially** Observable. The environment is partially observable because the autonomous battery-operated micro aquatic boat can only perceive the flooded areas and neighboring locations within its vicinity using sensors. It may not have complete information about the entire city or all flooded areas at once.

**Agents (Single/Multi):** It is a **single agent** activity, which is an autonomous battery-operated micro aquatic boat.

**Deterministic (Deterministic/Stochastic):** It is **Stochastic** activity. Because the outcome of actions taken by the agent (such as selecting a random starting node, selecting neighbors based on heuristic values, and randomly choosing paths) may not always be predictable due to randomness or uncertainty in the environment.

**Episodic (Episodic/Sequential):** It is a **sequential** task environment. Because the agent's actions are dependent on the current state of the environment, and the sequence of actions taken by the agent affects future states. The agent must navigate through multiple flooded areas sequentially to provide assistance effectively.

**Static (Static/Semi/Dynamic):** It is a **semi-dynamic** activity. Because while the flooded areas (vertices) may change over time due to factors like water levels changing or new areas becoming flooded, the overall structure of the city's graph (vertices and edges representing roads) remains relatively static during the agent's operation.

**Discrete (Discrete/Continuous):** It is a **Discrete** activity. Because the flooded areas, roads (edges), and nodes (vertices) in the city's graph are discrete and finite. The agent moves from one discrete location to another, and actions such as selecting neighboring nodes or removing edges are discrete operations.

## Problem Solving Agents (PSA)

|  |  |
| --- | --- |
| Navigating all the flooded areas via shortest Path | |
| Initial State | Agent(S) entry point (any) – selected by user |
| Possible Actions | Moving from one flooded area (node) to another connected flooded area.  Providing assistance to people waving for help in the flooded areas.  Navigating along the roads (edges) of the graph to cover all flooded areas efficiently. |
| Transition Model | [S, Move from 'current\_node' to 'next\_node'] = h(n)  [S, Assist people in 'current\_node'] = 'updated\_state' |
| Goal Test | All roads (edges) in the graph have been traversed exactly once and reach to start node if possible. Else goal node can be any node as per shortest path covering all the roads. |
| Path Cost | Sum of all path cost of all the visited edges i.e. 92 |

# **Algorithms Used for the Solution**

## Greedy Best-First Search (GBFS)

Greedy Best-First Search (GBFS) is a search algorithm that uses a heuristic function to guide the search towards the goal. It is called a "greedy" algorithm because it always chooses the next step that appears to be the closest to the goal, regardless of whether that step leads to the goal. Like Breadth-First Search (BFS), GBFS explores all the nodes at a given depth level before moving on to the next level. However, it uses a heuristic function only and doesn’t consider the true cost of the move to determine the order in which to explore the nodes at each level, rather than exploring all the nodes at a given level before moving on to the next level. GBFS is typically used for problems where the solution is not guaranteed to be optimal, but a satisfactory solution is still desired.

Greedy BFS can also only provide an optimal solution if the heuristic chosen is admissible and consistent.

Admissible i.e. for a node n, heuristic h(n) to goal will always be underestimated compared to the true cost h\*(n) of reaching the goal. So h(n) <= h\*(n)

Consistent i.e. when going from neighboring nodes n to n’, the heuristic difference or path cost is never overestimated over the actual path cost.

### Algorithms pseudo code:

function: BEST-FIRST-SEARCH (problem, f) returns a solution node or failure

node←- NODE (STATE= problem.INITIAL)

frontier←- a priority queue ordered by f, with node as an element

reached←- a lookup table, with one entry with key problem.INITIAL and value node

while not IS-EMPTY (frontier) do

node←- POP (frontier)

if problem.IS-GOAL (node.STATE) then return node

for each child in EXPAND (problem, node) do

s←- child.STATE

if s is not in reached or child.PATH-COST&lt; reached[s].PATH.COST then

reached[s]←- child

add child to frontier

return failure

function EXPAND (problem, node) yields nodes

s←- node.STATE

for each action in problem.ACTIONS(s) do

s’←- problem.RESULT(s, action)

cost←- node.PATH-COST+problem.ACTION-COST(s, action, s’)

yield NODE (STATE = s’, PARENT = node, ACTION = action, PATH-COST = cost)

GBFS focuses on visiting all the nodes once and does not repeat visited notes. But as per the problem statement we must visit all the edges once and can visit nodes again if required. As per the given connected graph we must visit a few nodes (flooded area) again to cover all the edges. To meet the requirement of covering all the edges, we have modified the GBFS algorithm as Chinese postman problem.

function: FIND-EULERIAN-CIRCUIT(graph) returns a circuit list

heuristic ← CALCULATE-HEURISTIC(graph)

circuit ← empty list

random\_node ← SELECT-RANDOM-NODE(graph)

stack ← [random\_node]

while stack is not empty do

current\_node ← stack[-1]

if graph[current\_node] is not empty then

neighbors ← SORT-NEIGHBORS-BY-HEURISTIC(graph[current\_node], heuristic)

next\_node ← neighbors[0][1]

stack.push(next\_node)

REMOVE-EDGE(graph, current\_node, next\_node)

else

circuit.append(stack.pop())

return REVERSE(circuit)

function: CHINESE-POSTMAN(graph) returns a eulerian\_circuit list

eulerian\_circuit ← FIND-EULERIAN-CIRCUIT(graph)

return eulerian\_circuit

function: CALCULATE-PATH-COST(graph, circuit) returns total\_cost

total\_cost ← 0

visited\_edges ← empty set

for i from 0 to length(circuit) - 2 do

current\_node ← circuit[i]

next\_node ← circuit[i + 1]

if EDGE-EXISTS(graph, current\_node, next\_node) and EDGE-NOT-VISITED(current\_node, next\_node, visited\_edges) then

edge ← SORT-NODES-TO-TUPLE(current\_node, next\_node)

total\_cost += graph[current\_node][next\_node]

visited\_edges.add(edge)

else

PRINT-WARNING(current\_node, next\_node)

return total\_cost

graph ← INITIALIZE-GRAPH()

graph\_for\_cost\_cal ← DEEP-COPY(graph)

START-MEMORY-TRACKING()

START-TIME-TRACKING()

chinese\_postman\_circuit ← CHINESE-POSTMAN(graph)

path\_cost ← CALCULATE-PATH-COST(graph\_for\_cost\_cal, chinese\_postman\_circuit)

END-TIME-TRACKING()

STOP-MEMORY-TRACKING()

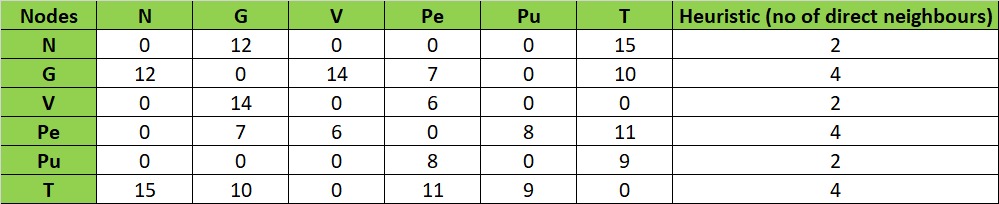
PRINT-RESULTS(chinese\_postman\_circuit, path\_cost, time\_taken, memory\_used)

### Data structures used

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable name** | **Data structure** | **Purpose** | **Details** |
| **graph** | Dictionary of dictionaries | Represents the graph structure of the city, where keys are the names of locations (vertices) and values are dictionaries representing adjacent vertices and their corresponding edge weights (roads) | Each key-value pair in the outer dictionary represents a vertex and its neighbouring vertices with associated edge weights |
| **heuristic** | Dictionary | Stores the heuristic values for each node in the graph. | It's used to determine the next node to traverse based on the heuristic value. The heuristic values are calculated as the degree of each node, representing the number of neighbouring vertices |
| **circuit** | List | Stores the sequence of nodes representing the Eulerian circuit | The algorithm iteratively constructs the circuit, appending nodes as they are visited |
| **stack** | List | Keeps track of the nodes being traversed during the Eulerian circuit construction | Nodes are pushed onto the stack as they are visited and popped off when backtracking is necessary |
| **eulerian\_circuit** | List | Stores the Eulerian circuit computed by the **find\_eulerian\_circuit** function | This list represents the order in which vertices are visited in the Eulerian circuit |
| **visited\_edges** | Set | Tracks visited edges to avoid counting edge weights more than once | As edges are traversed in the circuit, they are added to this set to ensure each edge weight is considered only once |

### Heuristic Design

Heuristic h(n) = highest number of neighbor heuristic



Here's an explanation of how this heuristic is designed and utilized:

**Definition**: The heuristic function calculates an estimate of the cost from the current node to the goal node based on the number of neighbors each node has. In this case, the heuristic value assigned to a node is the number of neighbors it has. Nodes with a higher number of neighbors are considered to be closer to the goal, and nodes with fewer neighbors are considered to be farther away.

**Implementation**: The heuristic function is implemented as a part of the search algorithm, such as Greedy Best First Search (GBFS). When exploring the graph, GBFS uses this heuristic information to prioritize nodes with a higher number of neighbors, assuming that they are more likely to lead towards the goal state.

**Heuristic Guidance**: During the search process, GBFS selects nodes to expand based on their heuristic values. Nodes with a higher heuristic value (indicating more neighbors) are considered more promising, and GBFS prioritizes exploring these nodes first. This heuristic guidance helps GBFS efficiently navigate towards the goal state by focusing on paths that are likely to lead to quicker solutions.

**Admissibility**: It's important to note that the "highest number of neighbor heuristics" is admissible if it never overestimates the true cost to reach the goal node. Since the number of neighbors provides an indication of proximity to the goal, this heuristic tends to be admissible in many cases. However, it may not always accurately reflect the true distance to the goal, especially in complex graphs with irregular structures.

## Genetic Algorithm

Genetic algorithms are problem-solving techniques inspired by natural selection and evolution. They work by simulating the process of natural selection, where candidate solutions to a problem are represented as individuals in a population. Through processes like crossover (recombination) and mutation, new generations of solutions are produced and evaluated based on their "fitness" or how well they solve the problem. Over successive generations, the population evolves towards better solutions.

Below are the steps of Genetic algorithm solution.

**Initialization**: Create an initial population of valid edges between nodes (flooded areas).

**Evaluation**: Evaluate the fitness of each individual path in the population based on how well it solves the problem.

**Selection**: Select random individuals from the current population to serve as parents for the next generation. Selection is typically based on the individual's fitness score, with better-fit individuals having a higher chance of being selected.

**Crossover**: Create new individuals (offspring) by combining genetic material from pairs of selected parents. This is typically done by exchanging genetic information (genes or traits) between parents to produce offspring with characteristics from both parents.

**Mutation**: Introduce random changes to the genetic material of the offspring to promote diversity in the population. Mutation helps explore new regions of the solution space that may lead to better solutions.

**Replacement**: Create the next generation population by combining the offspring with some of the existing population.

**Termination**: Repeat the process for a fixed number of generations or until a termination condition is met (e.g., reaching a satisfactory solution, no significant improvement over multiple generations).

**Admissibility**: Genetic algorithms don't typically adhere to the concept of admissibility as seen in GBFS algorithm. Instead, it focusses on finding good solutions based on fitness score through evolutionary processes such as mutation, crossover, and selection. Fitness score = min cost of connected edges.

**Consistency**: Similarly, the notion of consistency in genetic algorithms differs from GBFS. Genetic algorithms aim to efficiently explore the solution space to find satisfactory solutions.

### Genetic Algorithm pseudo code

function: GENETIC-SEARCH(graph, starting\_point) returns a list of edges representing the best route

initialize\_population(graph)

for \_ in range(NUM\_GENERATIONS):

evolve\_population(population, graph, starting\_point)

if population is empty:

return None

best\_individual ← find\_best\_individual(population)

fixed\_best\_individual ← fix\_sequence(best\_individual)

cleaned\_best\_individual ← remove\_invalid\_edges(fixed\_best\_individual)

final\_best\_individual ← avoid\_reverse\_order(cleaned\_best\_individual)

return final\_best\_individual

function: evolve\_population(population, graph, starting\_point)

selected\_parents ← selection(population, graph)

offspring\_population ← []

for i from 0 to length(selected\_parents) - 2 (step by 2):

parent1 ← selected\_parents[i]

parent2 ← selected\_parents[i + 1]

offspring1, offspring2 ← crossover(parent1, parent2, graph, starting\_point)

if offspring1 is not None:

mutate(offspring1, graph)

add offspring1 to offspring\_population

if offspring2 is not None:

mutate(offspring2, graph)

add offspring2 to offspring\_population

if offspring\_population is empty:

return population

else:

return offspring\_population

function: initialize\_population(graph)

population ← []

all\_edges ← generate\_all\_possible\_edges(graph)

repeat POPULATION\_SIZE times:

randomly\_select\_9\_edges(all\_edges)

add selected\_edges to population

function: selection(population, graph)

selected\_parents ← []

repeat for each individual in population:

tournament ← randomly\_select\_individuals(population, tournament\_size)

calculate\_fitness\_scores(tournament, graph)

add best\_individual to selected\_parents

return selected\_parents

function: crossover(parent1, parent2, graph, starting\_point)

offspring1, offspring2 ← perform\_crossover(parent1, parent2)

if offspring1 is None or offspring2 is None:

return None, None

if not connected\_to\_starting\_point(offspring1, starting\_point) or not connected\_to\_starting\_point(offspring2, starting\_point):

return None, None

if not valid\_edges(offspring1, offspring2, graph):

return None, None

return offspring1, offspring2

function: mutation(individual, graph)

randomly\_choose\_mutation\_points(individual)

swap\_selected\_points(individual)

if new\_individual\_is\_valid(individual, graph):

return individual

else:

return None

function: fix\_sequence(individual)

fixed\_individual ← []

add first\_edge\_to\_fixed\_individual(individual)

repeat for each edge in individual from second\_edge:

if edge\_is\_connected\_to\_previous\_edge(edge, fixed\_individual):

add edge\_to\_fixed\_individual(edge)

return fixed\_individual

function: remove\_invalid\_edges(individual)

cleaned\_individual ← []

for each edge in individual:

if edge\_is\_valid(edge, graph):

add edge\_to\_cleaned\_individual(edge)

return cleaned\_individual

function: avoid\_reverse\_order(individual)

final\_individual ← []

visited\_edges ← set()

for edge in individual:

if reverse\_edge\_not\_visited(edge, visited\_edges):

add edge\_to\_final\_individual(edge)

add edge\_to\_visited\_edges(edge)

return final\_individual

### Data structures used

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable name** | **Data structure** | **Purpose** | **Details** |
| graph | Dictionary of dictionaries | Represents the graph structure of the city, where keys are the names of locations (vertices) and values are dictionaries representing adjacent vertices and their corresponding edge weights (roads) | Each key-value pair in the outer dictionary represents a vertex and its neighbouring vertices with associated edge weights |
| population | Lists of Tuples | Stores population of potential solutions with valid edges(routes). | Each individual in the population is a list of tuples, where each tuple represents an edge between two locations. The list of individuals forms the population. |
| fitness\_scores | List | Represents the fitness scores of individuals in the genetic algorithm, where each tuple contains the fitness score and the individual's route | The algorithm iteratively constructs the circuit, appending nodes as they are visited |
| selected\_parents | List | Stores the selected parent individuals | Represents the individuals chosen to undergo crossover for producing offspring |
| offspring1, offspring2 | List of tuples | Holds the offspring resulting from the crossover operation between selected parent individuals. | Represent the offspring generated through crossover in the genetic algorithm, where each tuple contains the offspring's route |
| visited\_edges | Set | Tracks visited edges to avoid counting edge weights more than once | As edges are traversed in the circuit, they are added to this set to ensure each edge weight is considered only once |

### Heuristic/Fitness Design

**Fitness score** = minimum of total cost of the path represented by an individual

Here's how the fitness score is calculated and used in the code:

**Calculation**: The fitness score is calculated based on the total cost of the path represented by an individual. The lower the total cost, the better the fitness score. The cost of a path is determined by summing up the weights of the edges traversed in the path.

**Usage in selection**: The fitness score is used in the selection process, specifically in the selection function. In this function, a tournament selection mechanism is employed to select individuals for reproduction. Each individual in the population participates in a tournament, and the one with the lowest fitness score (i.e., the lowest total cost) is selected as a parent. This process helps to bias the selection towards individuals with better fitness scores, thus favoring better solutions.

# **Steps to Execute Python Code**

1. Step-1 – Open the code in Jupyter Notebook/ google colab
2. Step-2 – Run the code step by step.
3. Step-3 – at Cell of DYNAMIC INPUT, code will ask you to select Starting node. Please select the starting node from the given list of flooded areas.  
   if wrong input is given, it would ask you again to input starting node.

A screenshot of a computer

Description automatically generated

# **Test Results**

### Greedy Best First Search

#### **Scenario 1**: When starting node is Nungambakkam and Search algorithm is GBFS

##### User Input – Starting Location

A white background with black text

Description automatically generated

##### Heuristic, Best Path, Path Cost

A close up of a text

Description automatically generated

##### Time and Space complexity

A close-up of a number

Description automatically generated

#### **Scenario 2**: When starting node is Tambaram and search algorithm is GBFS

##### User Input – Starting Location

A screenshot of a computer

Description automatically generated

##### Heuristic, Best Path, Path Cost

A close up of a text

Description automatically generated

##### Time and Space Complexity

A screenshot of a computer

Description automatically generated

### Genetic Algorithm

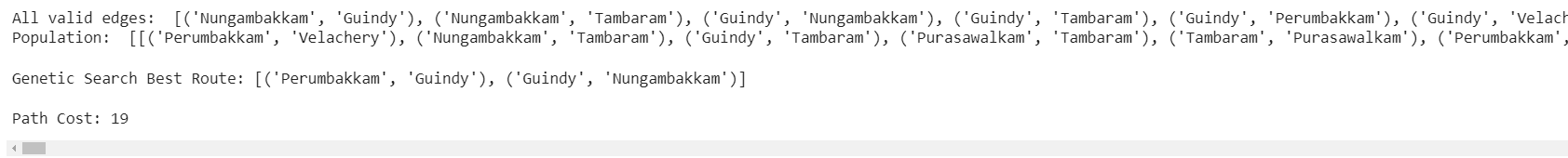
#### **Scenario 1**: When starting node is Nungambakkam and Search algorithm is Genetic search

##### User Input – Starting Location

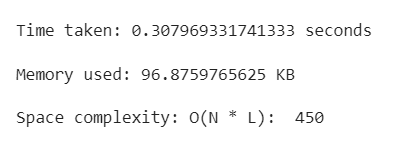
A white background with black text

Description automatically generated

##### Best Path, Path Cost, List of valid Edges



##### Time and Space Complexity



#### **Scenario 2**: When starting node is same Nungambakkam and search algorithm is Genetic search

##### User Input – Starting Location

A close up of a screen

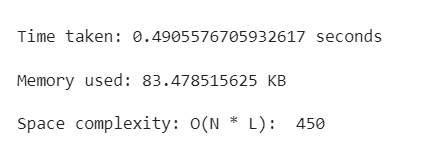
Description automatically generated

##### Best Path, Path Cost, List of valid Edges

A close-up of a message

Description automatically generated

##### Time and Space Complexity



# **Comparative Analysis**

### Time Complexity:

For Search algorithms, time complexity = difference between start and end time and print it.

#### Time Complexity = Time taken by **GBFS** Algorithm

* Scenario 1: 0.0014777183532714844 seconds
* Scenario 2: 0.007776021957397461 seconds

#### Time Complexity = Time taken by **Genetic** search Algorithm

* Scenario 1: 0.307969331741333 seconds
* Scenario 2: 0.4905576705932617 seconds

### Space Complexity:

For Search algorithms, space complexity will be analogous to the maximum number of nodes that will be in the OPEN list or Fringe throughout the search process.

#### Space Complexity = Number of nodes explored by **GBFS** Algorithm

* Scenario 1: 10
* Scenario 2: 10

#### Space Complexity = O(N \* L) by **Genetic** Algorithm, where N=Population size & L=Chromosomes size

* Scenario 1: 450
* Scenario 2: 450

# **Conclusion:**

### Path Comparison

* Greedy BFS utilizes only Heuristic value for navigating an agent. Since, the defined heuristic is same irrespective of random starting location, it always gives different optimal paths but with the path cost (sum of cost of all the edges) for the given problem. It may differ for different problems and situations.
* GBFS tends to follow a focused path towards the goal state based on heuristic guidance, potentially overlooking alternative paths that may lead to better solutions in certain cases.
* Genetic Algo explores a broader range of solutions and traverses multiple paths simultaneously through the population evolution process.
* While GBFS may find a solution quickly if the heuristic provides accurate guidance, Genetic Algo may explore a more diverse set of solutions and potentially discover better solutions through the evolutionary process, albeit at the cost of higher computational resources.

### Actual Cost Comparison

* **GBFS**, Agent performs search and provide different paths based on starting location. But always gives same path cost because we must visit all the edges only once.
* **Genetic Search**, Agent performs search with random set of edges in all the scenarios and provide different path and cost based on best available solution due to local minima.

### Time & Space Complexity:

* Greedy BFS uses only heuristics to navigate the agent. Since the Heuristic function in the problem statement is independent of the nature of the agent, hence the change in agent has no effect on the time and space complexity. Instead, the values are dependent on the scenario/environment defined for the search.
* In case of Genetic algorithms, it uses fitness score to navigate the agent. Since local search does not guarantee for the optimal solution or no solution at all, changing the Mutation rate may affect on time and space complexity. Thes values are dependent on scenario and environment of the situation.