**Artificial and Computational Intelligence Assignment 1**

**Problem solving by Uninformed & Informed Search**

1. **1.1 PEAS Description**

**Performance Measure (P):**

* Primary Objective: Find a valid path (a sequence of moves) that starts at "Admission Office" and ends at "Exit" while visiting every node exactly once.
* Optimization Goal: Minimize the total distance (or travel cost) of the path.
* Additional Considerations:
  + The solution must be valid (all nodes are included exactly once).
  + The time (or computational efficiency) taken to compute the solution might also be considered in a practical setting.

**Environment (E):**

* + Nature: The environment is a static, fully observable, and deterministic graph.
  + Representation:
    - Nodes: Represent locations such as "Admission Office," "Hostel Office," "Hostel visit," "Library," "Dept visit," "Canteen," and "Exit."
    - Edges: Represent paths between locations, each with an associated travel cost (or distance).
  + Domain: The campus map where each connection is pre-defined.

**Actuators (A):**

* Actions Available:
  + Rohan (the agent) can move from its current node to any adjacent node that is connected by an edge.
  + Each action corresponds to choosing one of the available outgoing edges from the current location.
* Example:
  + If at "Admission Office," Rohan can walk to "Hostel Office" or "Library"

**Sensors (S):**

* Perception:
  + Rohan (the agent) senses its current location (i.e., which node it is currently at).
  + Rohan perceives the complete graph structure (the entire list of nodes and the edges between them along with their travel costs) through knowledge of the map or Campus signboards like distances between the locations.
  + Awareness of the destination location
* Observability:
  + The environment is fully observable, so Rohan always has complete knowledge of all locations and paths.
  + This includes knowing the travel cost associated with moving along any given edge.

**1.2 Task Environment**

* **Fully** **Observable vs. Partially Observable (Sensor based):**
* **Fully Observable:**  
  Rohan has full knowledge of the campus layout and the distances between various locations in the campus and his current position. There’s no hidden information from Rohan in the given scenario.
* **Deterministic vs. Stochastic (Action & State based):**
* **Deterministic:**  
  The outcome of each action (i.e., moving from one location to another in the campus) is predictable. Walking from point A to point B always results in the same distance and time under normal scenario.
* **Episodic vs. Sequential (Action based):**
* **Sequential:**  
  Each action (visiting a particular location) affects the future action. The sequence of visits matters because Rohan must visit all locations exactly once before exiting.
* **Static vs. Dynamic (Action & State based):**
* **Static:**  
  The environment doesn’t change while Rohan is navigating. The locations and distances remain constant throughout his task.
* **Discrete vs. Continuous (State based):**
* **Discrete:**  
  The set of locations Rohan must visit is finite and clearly defined (Admission Office, Hostel Office, Hostel, Canteen, Department, Library, Campus Exit).
* **Single Agent vs. Multi-Agent (Agent based):**
* **Single Agent:**  
  Rohan is the only decision-maker in this problem. No other agents influence his navigation path.

1. **2.1 Heuristics**

In this case, we can consider using the minimum distance from each location to the exit as a heuristic. The path graph data is as below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **From/To** | **Admission office** | **Hostel office** | **Library** | **Hostel visit** | **Canteen** | **Dept visit** | **Exit** |
| **Admission office** | 0 | 2 | 4 | No path | No path | No path | No path |
| **Hostel office** | 2 | 0 | 4 | 2 | 6 | No path | No path |
| **Library** | 4 | 4 | 0 | No path | 7 | 3 | No path |
| **Hostel visit** | No path | 2 | No path | 0 | 6 | No path | 4 |
| **Canteen** | No path | 6 | 7 | 6 | 0 | 2 | 8 |
| **Dept visit** | No path | No path | 3 | No path | 2 | 0 | 5 |
| **Exit** | No path | No path | No path | 4 | 8 | 5 | 0 |

Considering ‘Exit’ as the destination point heuristics calculation comes out to be as follows. To calculate the Heuristics we will consider the direct distance between current position and destination wherever applicable, else we will calculate the shortest available path from current position to the destination.

In Greedy Best First search algorithm (GBFS) selection of the next node is based on the lowest heuristic value of the available nodes for traversal without considering the actual path cost to the next node.

Considering the above, the heuristics comes out as below.

a). Admission office (Admissions Office–>Hostel Office –>Hostel Visit –> Exit) = 2 + 2 + 4 = 8

b). Hostel office (Hostel Office->Hostel Visit->Exit) = 2 + 4 = 6

c). Library (Library->Exit) = 3+ 5 = 8

d). Hostel visit (Hostel Visit->Exit) = 4

e). Canteen (Canteen->Dept. visit->Exit) = 2 + 5 = 7

f). Dept. visit (Dept. visit->Exit) = 5

g). Exit (Exit->Exit) = 0

Now we need to check the consistency and admissibility of the derived Heuristics.

**Admissibility**: A heuristic is admissible if it never overestimates the cost to reach the goal. The heuristic should always be less than or equal to the true cost from any node to the goal.

To check that the heuristic is admissible we need to verify that the actual shortest distance does not exceed the heuristic value.

As our criteria for selection is same as that of the admissibility condition the Admissibility of the given Heuristic holds true. Here actual distance from every node to exit would be equal to the heuristic value.

**Consistency:** A heuristic is **consistent** if, for every node **n** and its neighbour **n'**, the following holds:

**h(n) ≤ c(n,n′) + h(n′)**

Where:

* **h(n)** is the heuristic value of the current node.
* **c(n, n')** is the cost of moving from **n** to **n'**.
* **h(n')** is the heuristic value of the neighbour.

Consistency Check for some Example Path:

1. Admission Office → Hostel Office

h(Admission Office) = 8 ≤ 2 + 6 = 8 (True)

1. Hostel Office → Hostel Visit

h(Hostel Office) = 6 ≤ 2 + 4 = 6 (True)

1. Library → Dept Visit

h (Library) = 8 ≤ 3 + 5 = 8(True)

1. Canteen → Dept Visit

h(Canteen) = 7 ≤ 2 + 5 = 7 (True)

1. Hostel Visit → Exit

h(Hostel Visit) = 4 ≤ 4 + 0 = 4 (True)

The heuristic is **consistent** because it satisfies the condition for all connected nodes.

Hence the derived Heuristic is Consistent and Admissible and could be used in the GBFS and Genetic algorithm.

**2.2 Fitness function**

In the given problem our objective is to minimise the distance travelled from the Admission office to exit while traversing through all the given location/nodes. Now the best fitness score would be the path with least distance. As both are inversely related, we may define the fitness function as the inverse of distance.

Fitness function = (1 / Total travelled distance)

Where the Total distance is the sum of actual distances travelled while traversing through the proposed path.

To optimise we may add some kind of penalty for the calculation if the proposed route do not exist by adding some extra weightage to the distance travelled.

We can define the Fitness function as **F (n) = [ 1 / ( D(n)+P )]**

Where P=0, if the proposed path is valid.

P is some large value to penalise, if the proposed path does not exist.

**Code Execution Flow:**

* 1. **Setting Initial State by handing dynamic inputs**

|  |
| --- |
| **# Code Block : Set Initial State (Must handle dynamic inputs)**  **graph = {}**  **def set\_initial\_state():**  **"""**  **Dynamically sets the initial state for the PSA agent.**  **This includes:**  **- Creating a graph where each node and its outgoing edges (with weights) are provided by the user.**  **- Defining the start and goal nodes.**    **Returns:**  **graph: Dictionary representing the graph.**  **start: Start node (string).**  **goal: Goal node (string).**  **"""**  **#graph = {}**  **# Get the number of nodes and their names**  **num\_nodes = int(input("Enter the number of nodes in the graph: "))**  **nodes = []**  **for i in range(num\_nodes):**  **node = input(f"Enter the name for node {i+1}: ").strip()**  **nodes.append(node)**  **graph[node] = {}  # Initialize empty edge dictionary for this node**  **print("\nNow, enter the edges in the format: <source> <target> <weight>")**  **print("Type 'done' when you have finished adding edges.")**  **while True:**  **edge\_input = input("Edge: ").strip()**  **if edge\_input.lower() == "done":**  **break**  **try:**  **source, target, weight = edge\_input.split()**  **weight = float(weight)**  **if source not in graph:**  **print(f"Source node '{source}' not recognized. Please enter a valid source node.")**  **continue**  **if target not in graph:**  **print(f"Target node '{target}' not recognized. Please enter a valid target node.")**  **continue**  **graph[source][target] = weight**  **except ValueError:**  **print("Invalid input format. Please use: <source> <target> <weight>")**    **# Set the start and goal nodes**  **start = input("\nEnter the start node: ").strip()**  **goal = input("Enter the goal node: ").strip()**  **# Validate start and goal**  **if start not in graph:**  **raise ValueError(f"Start node '{start}' is not in the graph.")**  **if goal not in graph:**  **raise ValueError(f"Goal node '{goal}' is not in the graph.")**  **return graph, start, goal**  **# Example usage:**  **if \_\_name\_\_ == "\_\_main\_\_":**  **graph, start, goal = set\_initial\_state()**  **print("\nInitial Graph:")**  **for node, edges in graph.items():**  **print(f"{node} -> {edges}")**  **print(f"\nStart Node: {start}")**  **print(f"Goal Node: {goal}")** |

**Output:**



* 1. **Setting the matrix for transition & cost**

|  |
| --- |
| #Code Block : Set the matrix for transition & cost (as relevant for the given problem)  def set\_transition\_cost\_matrices(graph):      """      Given a graph as a dictionary of dictionaries, this function creates:        - A transition matrix, where each cell [i][j] is 1 if there is an edge from node i to node j, else 0.        - A cost matrix, where each cell [i][j] is the cost of the edge from node i to node j if it exists;          otherwise, float('inf') if i != j and 0 if i == j.        Matrices are represented as nested dictionaries with keys as node names.      """      nodes = list(graph.keys())      # Initialize matrices: transition matrix with 0's and cost matrix with infinity.      transition\_matrix = {node: {other: 0 for other in nodes} for node in nodes}      cost\_matrix = {node: {other: float('inf') for other in nodes} for node in nodes}        # Set diagonal values: no cost to remain at the same node.      for node in nodes:          cost\_matrix[node][node] = 0        # Populate matrices based on the graph's edges.      for source in graph:          for target, cost in graph[source].items():              transition\_matrix[source][target] = 1  # There is a valid transition from source to target.              cost\_matrix[source][target] = cost       # Set the cost for that transition.        return transition\_matrix, cost\_matrix  # Example usage:  if \_\_name\_\_ == "\_\_main\_\_":      # Example graph (this can also be obtained dynamically as in the initial state block)      transition\_matrix, cost\_matrix = set\_transition\_cost\_matrices(graph)        print("Transition Matrix:")      for source in transition\_matrix:          print(f"{source}: {transition\_matrix[source]}")        print("\nCost Matrix:")      for source in cost\_matrix:          print(f"{source}: {cost\_matrix[source]}") |

**Output:**

**A black and white text

AI-generated content may be incorrect.**

* 1. **Defining the function to design the Transition Model/Successor function. These functions would be called when search algorithms are implemented. Here, we get dynamic user input for state and goal, then generate successors using the Transition Model**

|  |
| --- |
| import shlex  def get\_state\_and\_goal():      """      Prompts the user to enter the current state and the goal state.        Returns:        tuple: (current\_state, goal\_state)      """      current\_state = input("Enter the current state: ").strip()      goal\_state = input("Enter the goal state: ").strip()      return current\_state, goal\_state  def transition\_model(state, graph, goal):      """      Given the current state, returns a list of valid successor states based on the graph.        Each state is represented as a tuple:        (current\_node, path, visited)        - current\_node: The current location (string)        - path: A list of nodes representing the sequence visited so far        - visited: A set of nodes that have been visited        Parameters:        state : tuple            The current state as (current\_node, path, visited).        graph : dict            The graph represented as a dictionary of dictionaries, where each key is a node and            its value is another dictionary mapping neighboring nodes to edge costs.        goal : str            The goal node.        Returns:        successors : list            A list of successor states, each in the form (next\_node, new\_path, new\_visited).      """      current\_node, path, visited = state      successors = []          for neighbor, cost in graph[current\_node].items():          # Only add the neighbor if it hasn't been visited yet.          if neighbor not in visited:              new\_state = (neighbor, path + [neighbor], visited | {neighbor})              successors.append(new\_state)        return successors  # Example usage when state and goal are provided by user input.  if \_\_name\_\_ == "\_\_main\_\_":      # Define the graph as specified.        # Get dynamic input for the current state and goal.      current\_state, goal\_state = get\_state\_and\_goal()      print("\nSelected Current State:", current\_state)      print("Selected Goal State:", goal\_state)        # Construct the initial state representation for the search:      # (current\_node, path, visited)      initial\_state = (current\_state, [current\_state], {current\_state})        # Generate and print the successor states using the transition model.      successors = transition\_model(initial\_state, graph, goal\_state)      print("\nSuccessor States for state '{}' with goal '{}':".format(current\_state, goal\_state))      for succ in successors:          print(succ) |

**Output:**

**A black screen with white text

AI-generated content may be incorrect.**

* 1. **Function to handle goal test using dynamic inputs.**

|  |
| --- |
| def goal\_test(state, goal, graph):      """      Checks whether the current state meets the goal criteria.        A state is represented as a tuple:          (current\_node, path, visited)      where:          - current\_node: the node currently being explored (string)          - path: a list of nodes representing the sequence visited so far          - visited: a set of nodes that have been visited      The goal is achieved if:        1. The current node is the goal node.        2. All nodes in the graph have been visited (dynamically determined from the graph).      Parameters:        state : tuple            The current state in the form (current\_node, path, visited).        goal : str            The goal node.        graph : dict            The graph represented as a dictionary of dictionaries, where keys are node names.        Returns:        bool: True if the state meets the goal criteria, False otherwise.      """      current\_node, path, visited = state      return current\_node == goal and len(visited) == len(graph)  # Example usage with dynamic user input:  if \_\_name\_\_ == "\_\_main\_\_":      # Define an example graph for demonstration (this can be replaced with dynamic graph input)        # Get dynamic input for the current state and the goal state.      current\_node = input("Enter the current state: ").strip()      goal = input("Enter the goal state: ").strip()        # Construct the state as (current\_node, path, visited)      state = (current\_node, [current\_node], {current\_node})        # Execute the goal test function and print the result.      result = goal\_test(state, goal, graph)      print(f"Current Node: {current\_node}, Goal: {goal}")      print("Goal Test Result:", result) |

**Output:**

A black background with white text

AI-generated content may be incorrect.

****

* 1. **Function to get Start and end state as user inputs:**

def get\_start\_goal\_states(graph):

    """

    Display the possible states to choose from and get user input for the start and goal states.

    Parameters:

      graph : dict

          The graph represented as a dictionary where keys are the available states.

    Returns:

      tuple: (start, goal)

          The start and goal states as entered by the user.

    """

    # Display the available states

    print("Available states:")

    for state in graph.keys():

        print(" -", state)

    # Get dynamic input for the start state

    start = input("Enter the start state from the available states: ").strip()

    while start not in graph:

        print("Invalid start state. Please choose from the available states.")

        start = input("Enter the start state from the available states: ").strip()

    # Get dynamic input for the goal state

    goal = input("Enter the goal state from the available states: ").strip()

    while goal not in graph:

        print("Invalid goal state. Please choose from the available states.")

        goal = input("Enter the goal state from the available states: ").strip()

    return start, goal

if \_\_name\_\_ == "\_\_main\_\_":

    # Get the start and goal states dynamically from the user

    start, goal = get\_start\_goal\_states(graph)

    print("\nSelected Start State:", start)

    print("Selected Goal State:", goal)

**Output:**A black rectangular object with white text

AI-generated content may be incorrect.

* 1. **Search Algorithm 1: Greedy Best First Search (GBFS) Algorithm**

**Overview:**Greedy Best First Search (GBFS) is a heuristic-driven search algorithm that expands the node that appears to be closest to the goal, based solely on a heuristic function. It uses a priority queue to select the next node to explore according to its estimated cost to reach the goal. While GBFS can often find a solution quickly, it is not guaranteed to find the optimal (shortest) path.

**Algorithm Details:**

**Input:**

* A graph (or search space)
* A start node
* A goal node
* A heuristic function h(n) that estimates the cost from node n to the goal

**Output:**

* A path from the start node to the goal node, if one exists

**Pseudocode:**

|  |
| --- |
| **function GBFS (graph, start, goal, h):**  **// Initialize the open list (priority queue) with the start node and its heuristic value.**  **open\_list = PriorityQueue()**  **open\_list.push(start, h(start))**    **// Initialize the closed set to keep track of visited nodes.**  **closed\_set = {}**    **// Optionally, maintain a mapping for reconstructing the path.**  **parent = {}**    **while open\_list is not empty:**  **// Remove the node with the lowest heuristic value.**  **current = open\_list.pop()**    **// If the goal is reached, reconstruct and return the path.**  **if current == goal:**  **return reconstruct\_path(parent, goal)**    **// Mark the current node as visited.**  **closed\_set.add(current)**    **// Process each neighbor of the current node.**  **for each neighbor in graph.neighbors(current):**  **if neighbor is not in closed\_set:**  **parent[neighbor] = current**  **open\_list.push(neighbor, h(neighbor))**    **return "No path found"** |

**Code Implementation of Greedy Best First Search:**

|  |
| --- |
| import heapq  def compute\_cost(path, graph):      """      Compute the total cost of traversing the given path using the graph.      The path is a list of nodes.      """      total\_cost = 0      for i in range(len(path) - 1):          source = path[i]          target = path[i + 1]          total\_cost += graph[source].get(target, float('inf'))      return total\_cost  def greedy\_best\_first\_search(graph, start, goal, heuristic\_values):      """      Implements Greedy Best First Search (GBFS) to find a valid path from start to goal.      The goal is defined as reaching 'Exit' after having visited all nodes.        A modified heuristic is used that rewards states with more visited nodes.      """      def modified\_heuristic(state):          current\_node, path, visited = state          base = heuristic\_values[current\_node]          total\_nodes = len(graph)          bonus = (len(visited) / total\_nodes) \* 100  # Adjust bonus scaling as needed.          return base - bonus      # Initial state: (current\_node, path, visited\_set)      initial\_state = (start, [start], {start})        # Priority queue stores tuples: (modified\_heuristic, state)      open\_list = []      heapq.heappush(open\_list, (modified\_heuristic(initial\_state), initial\_state))        while open\_list:          current\_f, state = heapq.heappop(open\_list)          current\_node, path, visited = state            # Debug (optional): print current state info          # print("Expanding:", current\_node, "Visited:", visited)            if goal\_test(state, goal, graph):              total\_cost = compute\_cost(path, graph)              return path, total\_cost            for succ in transition\_model(state, graph, goal):              heapq.heappush(open\_list, (modified\_heuristic(succ), succ))        return ["Error: Path Not Found"], float('inf')  # Define heuristic values for each node.  # For nodes with a direct edge to "Exit", use that cost; otherwise, assign a high value.  heuristic\_values = {}  for node in graph:      if node == "Exit":          heuristic\_values[node] = 0      else:          heuristic\_values[node] = graph[node].get("Exit", 1000)  # Set start and goal nodes.  start, goal = get\_start\_goal\_states(graph)  if \_\_name\_\_ == "\_\_main\_\_":      path, cost = greedy\_best\_first\_search(graph, start, goal, heuristic\_values)      print("GBFS Path:", path)      print("GBFS Total Cost:", cost)    **NOTE:**  **The following logic has been added to the code implementation of GBFS Algorithm to ensure that the algorithm chooses the Hamiltonian Path in the graph**   * **Transition Model:** The transition\_model function only adds a neighbour if it is not already in the visited set. This prevents the goal ("Exit") from being added twice. * **Goal Test:** The goal\_test function remains the same. It checks that the current node is "Exit" and that the number of visited nodes equals the total number of nodes in the graph. * **Modified Heuristic:** The modified\_heuristic function subtracts a bonus based on the fraction of nodes visited from the base heuristic value. This rewards states that have visited more nodes. |

**Output:**

**A black screen with white text

AI-generated content may be incorrect.**

* 1. **Algorithm 2: Genetic Algorithm**

**Overview:**A Genetic Algorithm is a search heuristic inspired by the process of natural selection. It is used for solving optimization and search problems by evolving a population of candidate solutions over several generations using bio-inspired operators such as selection, crossover, and mutation.

**Algorithm Details:**

**Input:**

* A population of candidate solutions (individuals), typically represented as strings, arrays, or permutations.
* A fitness function to evaluate each candidate's quality.
* Parameters such as population size, number of generations, crossover probability, and mutation probability.

**Output:**

* A candidate solution that approximates the optimal solution to the problem.

**Pseudocode:**

|  |
| --- |
| **function GeneticAlgorithm(population\_size, generations, mutation\_rate):**  **// Initialize a random population of candidate solutions.**  **population = generate\_random\_population(population\_size)**    **for generation from 1 to generations:**  **// Evaluate the fitness of each individual in the population.**  **fitness\_scores = evaluate\_fitness(population)**    **// Select individuals for reproduction based on their fitness.**  **selected = selection(population, fitness\_scores)**    **// Create a new population using crossover and mutation.**  **new\_population = []**  **while size(new\_population) < population\_size:**  **parent1, parent2 = select\_two(selected)**  **child = crossover(parent1, parent2)**  **child = mutate(child, mutation\_rate)**  **new\_population.append(child)**    **population = new\_population**    **// Return the best individual from the final population.**  **best\_individual = select\_best(population)**  **return best\_individual** |

**Code Implementation of Genetic Algorithm:**

|  |
| --- |
| import random  def genetic\_algorithm(graph, start, goal, population\_size=100, generations=500, mutation\_rate=0.1):      """      Implements a Genetic Algorithm to find a valid path that visits every intermediate node exactly once,      starting at 'start' and ending at 'goal'. The intermediate nodes are permuted.        Parameters:        graph (dict): The graph represented as an adjacency list (dictionary of dictionaries).        start (str): The starting node.        goal (str): The goal node.        population\_size (int): Number of individuals in the population.        generations (int): Number of generations to evolve.        mutation\_rate (float): Probability of mutation for an individual.        Returns:        tuple: (best\_path, best\_distance, best\_fitness\_history)               best\_path is a list representing the sequence of nodes from start to goal.               best\_distance is the total cost of that path.               best\_fitness\_history is a list recording the best fitness value (negative cost) per generation.               If no valid path is found, returns (["Error: Path Not Found"], float('inf'), best\_fitness\_history).      """      # Extract intermediate nodes: all nodes except start and goal.      nodes = list(graph.keys())      nodes.remove(start)      nodes.remove(goal)        def fitness\_function(path):          """          Compute the fitness of an individual (a permutation of intermediate nodes).          The total cost is computed as:            cost from start to first node +            cost along consecutive nodes in the path +            cost from last node to goal.          If the path does not cover all intermediate nodes exactly once, return infinity as a penalty.            Fitness is defined as the negative total cost, so a lower cost gives a higher fitness.          """          try:              total\_cost = graph[start][path[0]]  # cost from start to first intermediate              for i in range(len(path) - 1):                  total\_cost += graph[path[i]][path[i+1]]              total\_cost += graph[path[-1]][goal]   # cost from last intermediate to goal              # Ensure all intermediate nodes are used exactly once.              if len(set(path)) != len(nodes):                  return float('inf')              return -total\_cost          except KeyError:              return float('inf')        def create\_individual():          """Create a random individual by shuffling the intermediate nodes."""          individual = nodes[:]          random.shuffle(individual)          return individual        def mutate(individual):          """Mutate an individual by swapping two randomly chosen nodes with probability mutation\_rate."""          if random.random() < mutation\_rate:              i, j = random.sample(range(len(individual)), 2)              individual[i], individual[j] = individual[j], individual[i]        def crossover(parent1, parent2):          """          Perform crossover between two parent individuals using one-point crossover.          The child is built by taking the first part from parent1 and then filling in the remaining nodes          in the order they appear in parent2, avoiding duplicates.          """          cut = random.randint(1, len(parent1) - 1)          child = parent1[:cut]          child += [node for node in parent2 if node not in child]          return child        # Initialize the population with random individuals.      population = [create\_individual() for \_ in range(population\_size)]      best\_fitness\_history = []        for generation in range(generations):          # Sort population by fitness (lower total cost -> higher fitness, since fitness = -total\_cost).          population.sort(key=lambda ind: fitness\_function(ind))          best\_fitness = fitness\_function(population[0])          best\_fitness\_history.append(best\_fitness)            # Elitism: retain the top 10% of individuals.          elite\_count = max(1, population\_size // 10)          next\_generation = population[:elite\_count]            # Generate new individuals until the population is replenished.          while len(next\_generation) < population\_size:              parent1, parent2 = random.sample(population[:max(2, population\_size // 2)], 2)              child = crossover(parent1, parent2)              mutate(child)              next\_generation.append(child)          population = next\_generation        best\_individual = min(population, key=lambda ind: fitness\_function(ind))      if fitness\_function(best\_individual) == float('inf'):          return ["Error: Path Not Found"], float('inf'), best\_fitness\_history        best\_path = [start] + best\_individual + [goal]      best\_distance = -fitness\_function(best\_individual)      return best\_path, best\_distance, best\_fitness\_history  start, goal = get\_start\_goal\_states(graph)  # --- Main Execution Example ---  if \_\_name\_\_ == "\_\_main\_\_":      best\_path, best\_distance, best\_fitness\_history = genetic\_algorithm(graph, start, goal, population\_size=100, generations=500, mutation\_rate=0.1)      print("Genetic Algorithm Best Path:", best\_path)      print("Genetic Algorithm Best Total Cost:", best\_distance) |

* 1. **Code Implementation of Bidirectional Search Algorithm and why this cannot be implemented in this scenario:**

|  |
| --- |
| import heapq  def compute\_cost(path, graph):      """      Compute the total cost of traversing the given path using the graph.      The path is a list of nodes.      """      total\_cost = 0      for i in range(len(path) - 1):          source = path[i]          target = path[i + 1]          total\_cost += graph[source].get(target, float('inf'))      return total\_cost  def build\_reverse\_graph(graph):      """      Build and return the reverse of the given directed graph.      For each edge u -> v with cost, the reverse graph has an edge v -> u with the same cost.      """      reverse\_graph = {node: {} for node in graph}      for u in graph:          for v, cost in graph[u].items():              reverse\_graph[v][u] = cost      return reverse\_graph  def bidirectional\_search(graph, start, goal):      """      Performs Bidirectional Search to find a Hamiltonian path from 'start' to 'goal'      (i.e., a path that visits every node exactly once, with the final node equal to goal).      Returns:          (path, total\_cost): Tuple containing the complete path and its total cost,                              or (["Error: Path Not Found"], inf) if no valid path is found.      """      # Build reverse graph for backward expansion.      reverse\_graph = build\_reverse\_graph(graph)        # Each state is represented as (node, path, visited)      # Initialize forward frontier from start and backward frontier from goal.      forward\_frontier = [(start, [start], {start})]      backward\_frontier = [(goal, [goal], {goal})]        # Maintain dictionaries for visited states (mapping node to state) in each direction.      forward\_visited = {start: (start, [start], {start})}      backward\_visited = {goal: (goal, [goal], {goal})}        while forward\_frontier and backward\_frontier:          # ----- Expand Forward Frontier -----          new\_forward\_frontier = []          for state in forward\_frontier:              current\_node, path, visited = state              for neighbor, cost in graph[current\_node].items():                  if neighbor not in visited:                      new\_state = (neighbor, path + [neighbor], visited | {neighbor})                      if neighbor not in forward\_visited:                          forward\_visited[neighbor] = new\_state                          new\_forward\_frontier.append(new\_state)          forward\_frontier = new\_forward\_frontier          # Check for intersection between forward and backward frontiers.          intersect = set(forward\_visited.keys()).intersection(set(backward\_visited.keys()))          if intersect:              meet = intersect.pop()              f\_state = forward\_visited[meet]  # (node, f\_path, f\_visited)              b\_state = backward\_visited[meet]  # (node, b\_path, b\_visited)              # Merge paths. b\_state[1] is the path from goal to meet, so reverse it (excluding the meeting node)              combined\_path = f\_state[1] + b\_state[1][-2::-1]              combined\_visited = f\_state[2] | b\_state[2]              if len(combined\_visited) == len(graph):                  return combined\_path, compute\_cost(combined\_path, graph)            # ----- Expand Backward Frontier -----          new\_backward\_frontier = []          for state in backward\_frontier:              current\_node, path, visited = state              for neighbor, cost in reverse\_graph[current\_node].items():                  if neighbor not in visited:                      new\_state = (neighbor, [neighbor] + path, visited | {neighbor})                      if neighbor not in backward\_visited:                          backward\_visited[neighbor] = new\_state                          new\_backward\_frontier.append(new\_state)          backward\_frontier = new\_backward\_frontier          # Check for intersection again.          intersect = set(forward\_visited.keys()).intersection(set(backward\_visited.keys()))          if intersect:              meet = intersect.pop()              f\_state = forward\_visited[meet]              b\_state = backward\_visited[meet]              combined\_path = f\_state[1] + b\_state[1][-2::-1]              combined\_visited = f\_state[2] | b\_state[2]              if len(combined\_visited) == len(graph):                  return combined\_path, compute\_cost(combined\_path, graph)        return ["Error: Path Not Found"], float('inf')  # -------------------------------  # Example Usage  # -------------------------------  if \_\_name\_\_ == "\_\_main\_\_":          start, goal = get\_state\_and\_goal()        path, cost = bidirectional\_search(graph, start, goal)      print("Bidirectional Search Path:", path)      print("Bidirectional Search Total Cost:", cost) |

**Output:**

****

**Reason why Bidirectional Search cannot be used for the given scenario:  
  
1. Premature Intersection:**

The algorithm checks for an intersection between the forward and backward frontiers and immediately attempts to merge the paths when a common node is found. However, if the intersection happens early, for example, at a node that is reached before all nodes have been visited in either search, the merged visited set will be incomplete. For instance, in the current run, an intersection might occur at "Dept\_visit" or "Hostel\_visit" with only a subset of nodes visited. Since the merged visited set does not equal the full set of nodes in the graph, the goal test fails and the search continues. **2. Complexity of Hamiltonian Path Problems:**

Bidirectional search is most effective for standard shortest-path problems in which you only need to meet in the middle, with a simple intersection condition. For Hamiltonian path problems, however, the search space is exponentially large (the problem is NP-complete), and the two searches must combine in such a way that the union of their visited sets covers all nodes. This is much more challenging because a mere node intersection does not guarantee that the two partial paths can be merged into one complete path that visits every node exactly once.

**Output:**

A black rectangular object with white text

AI-generated content may be incorrect.

* 1. **Calling the search algorithms – GBFS, Genetic Algorithm and Bidirectional Search**

|  |
| --- |
| import time  def main():        # Get dynamic input for the start and goal states.      start, goal = get\_start\_goal\_states(graph)      print("\nSelected Start State:", start)      print("Selected Goal State:", goal)        # Precompute heuristic values for GBFS.      # For each node, if there's a direct edge to the goal, we use its cost; otherwise, a high value is assigned.      heuristic\_values = {}      for node in graph:          if node == goal:              heuristic\_values[node] = 0          else:              heuristic\_values[node] = graph[node].get(goal, 1000)  # 1000 is an arbitrarily high cost      # -------------------------------      # Call Greedy Best First Search (GBFS)      # -------------------------------      print("\nCalling Greedy Best First Search (GBFS)...")      start\_time = time.time()      gbfs\_path, gbfs\_cost = greedy\_best\_first\_search(graph, start, goal, heuristic\_values)      gbfs\_runtime = time.time() - start\_time      print("GBFS Path:", gbfs\_path)      print("GBFS Total Cost:", gbfs\_cost)      print("GBFS Runtime: {:.6f} seconds".format(gbfs\_runtime))        # -------------------------------      # Call Genetic Algorithm (GA)      # -------------------------------      print("\nCalling Genetic Algorithm...")      start\_time = time.time()      ga\_path, ga\_cost, ga\_fitness\_history = genetic\_algorithm(graph, start, goal,                                                               population\_size=100,                                                               generations=500,                                                               mutation\_rate=0.1)      ga\_runtime = time.time() - start\_time      print("Genetic Algorithm Best Path:", ga\_path)      print("Genetic Algorithm Total Cost:", ga\_cost)      print("Genetic Algorithm Runtime: {:.6f} seconds".format(ga\_runtime))        # -------------------------------      # Placeholder for Bidirectional Search (if implemented)      # -------------------------------        print("\nCalling Bidirectional Search...")      start\_time = time.time()      bd\_path, bd\_cost = bidirectional\_search(graph, start, goal)  # Function to be implemented      bd\_runtime = time.time() - start\_time      print("Bidirectional Search Path:", bd\_path)      print("Bidirectional Search Total Cost:", bd\_cost)      print("Bidirectional Search Runtime: {:.6f} seconds".format(bd\_runtime))    # Call main if this module is run directly.  if \_\_name\_\_ == |

**Output:**

**A black screen with white text

AI-generated content may be incorrect.**

* 1. **Comparative Analysis (Time and Space Complexity) by dynamically computing time and space for both GBFS and Genetic Algorithm**

|  |
| --- |
| import time  import tracemalloc  import heapq  import random  # # ----------------------------------------------  # # Measurement Functions using tracemalloc and time  # # ----------------------------------------------  def measure\_gbfs(graph, start, goal, heuristic\_values):      print("Measuring GBFS...")      tracemalloc.start()      t0 = time.time()      path, cost = greedy\_best\_first\_search(graph, start, goal, heuristic\_values)      runtime = time.time() - t0      current\_mem, peak\_mem = tracemalloc.get\_traced\_memory()      tracemalloc.stop()      print("GBFS Path:", path)      print("GBFS Total Cost:", cost)      print("GBFS Runtime: {:.6f} seconds".format(runtime))      print("GBFS Peak Memory Usage: {:.2f} KB".format(peak\_mem / 1024))      return path, cost, runtime, peak\_mem  def measure\_ga(graph, start, goal, population\_size=100, generations=500, mutation\_rate=0.1):      print("Measuring Genetic Algorithm...")      tracemalloc.start()      t0 = time.time()      path, cost, fitness\_history = genetic\_algorithm(graph, start, goal, population\_size, generations, mutation\_rate)      runtime = time.time() - t0      current\_mem, peak\_mem = tracemalloc.get\_traced\_memory()      tracemalloc.stop()      print("Genetic Algorithm Path:", path)      print("Genetic Algorithm Total Cost:", cost)      print("Genetic Algorithm Runtime: {:.6f} seconds".format(runtime))      print("Genetic Algorithm Peak Memory Usage: {:.2f} KB".format(peak\_mem / 1024))      return path, cost, runtime, peak\_mem  # ----------------------------------------------  # Main Execution: Measure and Print Complexities  # ----------------------------------------------  if \_\_name\_\_ == "\_\_main\_\_":      print("=== Measuring GBFS ===")      measure\_gbfs(graph, start, goal, heuristic\_values)        print("\n=== Measuring Genetic Algorithm ===")      measure\_ga(graph, start, goal, population\_size=100, generations=500, mutation\_rate=0.1) |

**Output:**

**A black screen with white text

AI-generated content may be incorrect.**

* 1. **Greedy Best First Search (GBFS):**

**Time Complexity:**

* + - Worst-case: O(V log V) — assuming each node is inserted into the priority queue once and each insertion/deletion takes O(log V), where V is the number of nodes.
    - In many search problems, GBFS can exhibit exponential worst-case behavior (O(b^d)), where b is the branching factor and d is the solution depth; however, for our problem (a well-defined graph with a unique visit constraint), the search is limited to visiting each node exactly once.
    - **Space Complexity:**
    - O(V), due to the storage required for the priority queue and the visited set.
  1. **Genetic Algorithm :**

**Time Complexity:**

* + - O(G \* P \* E), where:
    - G is the number of generations,
    - P is the population size, and
    - E is the evaluation cost for each individual (in our case, evaluating a candidate path involves computing the cost along the path, which is proportional to the number of nodes).
    - **Space Complexity:**
    - O(P \* L), where L is the length of an individual (typically the number of intermediate nodes to be permuted), plus overhead for storing fitness values and historical data.
  1. **Bidirectional Search (If implemented):**

**Time Complexity:**

* + - O(b^(d/2)), under ideal conditions, since bidirectional search reduces the effective search depth by half, where b is the branching factor and d is the solution depth.
    - **Space Complexity:**
    - O(b^(d/2)), as it maintains two frontiers—one from the start and one from the goal.
  1. **Comparative Discussion:**
* GBFS is advantageous when a good heuristic is available, enabling efficient guidance of the search. However, it may not always find an optimal solution if the heuristic is not admissible.
* The Genetic Algorithm offers a robust approach for exploring large solution spaces and escaping local minima by evolving a population of solutions, although it may require more computational time and careful parameter tuning (e.g., population size, number of generations, mutation rate).
* Bidirectional Search can significantly reduce search time by exploring from both the start and goal simultaneously, but it often requires more memory and can be more complex to implement correctly. The algorithm checks for an intersection between the forward and backward frontiers and immediately attempts to merge the paths when a common node is found. However, if the intersection happens early, for example, at a node that is reached before all nodes have been visited in either search, the merged visited set will have an incomplete visited set and, therefore, will not form a valid complete solution.