**Algorithm: BFS (Breadth-First Search)**

**Input:**

* **graph: An adjacency list representing the graph**
* **startnode: The starting node for the search**
* **goalnode: The goal node to be reached**

1. **Initialize visited as an empty list.**
2. **Initialize queue as an empty list.**
3. **Set flag = 0.**
4. **Append startnode to visited.**
5. **Append startnode to queue.**
6. **While queue is not empty:**
   * **a. Dequeue an element m from queue.**
   * **b. For each neighbour in graph[m]:**
     + **i. If flag is 0:**
       - **If neighbour is not in visited:**
         * **Append neighbour to visited.**
         * **Append neighbour to queue.**
         * **Print "Visited:" and the current state of visited.**
       - **If neighbour is equal to goalnode:**
         * **Print "Goal reached".**
         * **Print "The path taken is:" and visited.**
         * **Set flag = 1.**
7. **If flag is still 0:**
   * **a. Print "Goal is not reached".**

** Initialize visited as an empty list.**

** Define a function DFS(visited, graph, currentnode, goalnode):**

* **a. Initialize flag = 0.**
* **b. If currentnode is not in visited:**
  + **Append currentnode to visited.**
  + **If currentnode is equal to goalnode:**
    - **Set flag = 1.**
    - **Print "Goal is reached".**
    - **Print "The path taken to reach the goal is:" and visited.**
  + **If flag is still 0:**
    - **For each neighbour in graph[currentnode]:**
      * **Recursively call DFS(visited, graph, neighbour, goalnode).**

** Call DFS(visited, graph, startnode, goalnode).**

** After the call, if flag is still 0:**

* **Print "Goal has not been reached".**

**Algorithm: Best First Search**

* **Initialize SuccList with the given adjacency information (node connections and their respective costs).**
* **Set Start to the starting node.**
* **Set Goal to the goal node.**
* **Initialize Explored as an empty list.**
* **Define SUCCESS and FAILURE constants.**
* **Initialize State as FAILURE.**

**2. Define GENCHILD(N) Function**

* **Input: N (a node).**
* **If N exists in SuccList, return its corresponding list of child nodes.**
* **If N does not exist in SuccList, return an empty list.**

**3. Define GOALTEST(N) Function**

* **Input: N (a node).**
* **If N is equal to Goal, return True (goal reached).**
* **Otherwise, return False.**

**4. Define APPEND(L1, L2) Function**

* **Input: L1 and L2 (two lists).**
* **Return a new list containing all elements of L1 followed by all elements of L2.**

**5. Define SORT(L) Function**

* **Input: L (a list of nodes with costs).**
* **Sort L based on the second element (cost) of each inner list using a lambda function.**
* **Return the sorted list.**

**6. Define BestFirstSearch() Function**

* **Initialize Frontier list with the starting node and its cost (e.g., [(Start, cost)]).**
* **Initialize EXPLORED list as empty.**
* **Loop until Frontier is empty or State becomes SUCCESS:**
  + **a. Pop the first node N from Frontier.**
  + **b. If GOALTEST(N[0]) is True, then:**
    - **Set State to SUCCESS.**
    - **Exit the loop.**
  + **c. Add the current node N to EXPLORED.**
  + **d. If the goal is not reached:**
    - **Generate child nodes of N using GENCHILD(N), storing them in CHILD.**
    - **Remove nodes from CHILD that are already in EXPLORED.**
    - **Remove nodes from CHILD that are already in Frontier.**
    - **Append CHILD nodes to Frontier.**
    - **Sort Frontier using the SORT function.**

**7. Calling the Algorithm**

* **Call BestFirstSearch() to execute the search.**
* **Print the Explored list and State (either SUCCESS or FAILURE).**

**Algorithm: Map Coloring with Degree Heuristic**

**1. Initialize Data**

* **Define domain\_colors: a list of available colors.**
* **Define variable\_states: a list of states that need to be colored.**
* **Create a neighbors dictionary specifying the neighboring relationships between states.**
* **Create an empty dictionary finalstateswithcolor to store the assigned colors for each state.**

**2. Color Assignment Functions**

* **Define two functions for color assignment and validation:**
  + **getthecolor(state):**
    - **Purpose: Finds and returns a suitable color for the specified state.**
    - **Process: Iterates through domain\_colors and calls assigncolor to check if each color is valid for state.**
  + **assigncolor(state, color):**
    - **Purpose: Checks if it’s valid to assign the given color to state.**
    - **Process: Ensures that none of the neighboring states (from neighbors[state]) has the same color.**

**3. Main Function**

* **In the main function:**
  + **Use the degree heuristic to sort variable\_states by the number of neighbors in descending order, prioritizing states with more neighbors.**
  + **Iterate through the sorted list of states.**
  + **For each state, call getthecolor to find a suitable color and store the assignment in finalstateswithcolor.**
  + **Print finalstateswithcolor to display each state with its assigned color, ensuring that neighboring states have different colors.**

**Algorithm: Symptom-Based Diagnosis**

**1. Define Symptom-Checking Functions**

* **Define four functions to check for specific illnesses based on combinations of symptoms:**
  + **measles(a, b, c, d, e):**
    - **Checks if symptoms match the known symptoms of measles.**
    - **Returns "measles" if the symptoms match; otherwise, returns None.**
  + **flu(a, b, c, d, e, f, g, h):**
    - **Checks if symptoms match the known symptoms of flu.**
    - **Returns "flu" if the symptoms match; otherwise, returns None.**
  + **cold(c, j, k, d, h):**
    - **Checks if symptoms match the known symptoms of the common cold.**
    - **Returns "cold" if the symptoms match; otherwise, returns None.**
  + **chickenpox(a, h, g, b):**
    - **Checks if symptoms match the known symptoms of chickenpox.**
    - **Returns "chickenpox" if the symptoms match; otherwise, returns None.**

**2. User Input and Symptom Gathering**

* **Create a function called run\_diagnosis() to interact with the user:**
  + **Prompt the user to enter their name.**
  + **Ask the user a series of yes/no questions about their symptoms.**
  + **Store the user's responses in variables (e.g., a, b, c, etc.), ensuring all responses are stored in lowercase ('y' or 'n').**

**3. Diagnosis Initialization**

* **Initialize an empty list called diagnosis to store possible diagnoses based on the symptoms.**

**4. Diagnosis for Each Illness**

* **Use each symptom-checking function (measles(), flu(), cold(), chickenpox()) to check if the user's symptoms match any known illnesses.**
* **If a function returns a diagnosis (i.e., not None), append the result to the diagnosis list.**

**5. Final Diagnosis**

* **If the diagnosis list contains any possible illnesses, print a message to the user, including their name and the list of possible diagnoses.**
* **If the diagnosis list is empty, inform the user that no diagnosis could be made based on the given symptoms.**

**6. Execution**

* **Call the run\_diagnosis() function to execute the symptom-based diagnosis process.**

**Traveling sales\_man**

**1. Import Necessary Libraries**

* **Import maxsize from the sys module to represent the initial maximum path value.**
* **Import permutations from itertools to generate all possible permutations of the vertices.**

**2. Initialize Graph and Variables**

* **Define the total number of vertices, V.**
* **Create a graph matrix to represent distances between vertices, where graph[i][j] is the distance from vertex i to vertex j.**
* **Set the starting vertex (s) as 0.**

**3. Define the Traveling Salesman Function**

* **Function Name: travellingSalesmanProblem(graph, s)**
* **Create a list of vertices excluding the starting vertex s.**
* **Initialize min\_path to maxsize to track the shortest path found.**
* **Initialize best\_tour to store the tour with the minimum path.**

**4. Iterate Through Permutations**

* **For each permutation of vertices:**
  + **Calculate the total weight of the path starting from s, going through the permutation, and returning to s.**
  + **If the calculated path is shorter than min\_path, update min\_path and store the permutation in best\_tour.**

**5. Return the Result**

* **The function should return min\_path (the shortest path) and best\_tour (the corresponding sequence of vertices).**

**6. Invoke the Function and Print the Result**

* **Call the travellingSalesmanProblem(graph, s) function.**
* **Print the minimum path and the best tour.**