**DFS**

**Initialization: Begin from the root node, marking it as visited.**

**Exploration: Recursively visit each adjacent unvisited node, marking it as visited.**

**Backtracking: If a node has no unvisited adjacent nodes, backtrack to the last visited node with unvisited neighbors and continue the exploration.**

**Termination: The process continues until all nodes have been visited**

**BEST FIRST SEARCH**

**Initialization:**

**Define SuccList (graph), Start, Goal, Explored, SUCCESS, FAILURE, and State.**

**GENCHILD(N):**

**Return the child nodes of N from SuccList. If N is not found, return an empty list.**

**GOALTEST(N):**

**Return True if N is the Goal, otherwise False.**

**APPEND(L1, L2):**

**Combine and return the lists L1 and L2.**

**SORT(L):**

**Sort list L by the second element (cost).**

**BestFirstSearch():**

**Frontier: The list of nodes to be explored. It starts with the Start node, which is Arad.**

**Sort Frontier: The frontier is sorted based on the heuristic values (distances) so that the node with the smallest distance is explored first.**

**Check Goal: If the node at the front of the frontier is the goal (Bucharest), the search is marked successful, and the loop exits.**

**Expand Node: If the node is not the goal, it is added to the Explored list. The algorithm then generates its child nodes (neighbors) using GENCHILD().**

**Filter Child Nodes: Any child node that has already been explored or is already in the frontier is removed to avoid redundant exploration.**

**Add Child Nodes to Frontier: The remaining child nodes are added to the frontier.**

**Loop: The process repeats until either the goal is found or the frontier is empty (meaning no path to the goal exists).**

**Run BestFirstSearch() and print the result and explored nodes**

**BACK TRACKING (COLOR )**

**The available colors are Red, Blue, and Green, which will be used to color the states.**

**defines the neighbors for each state. If two states share a border, they are considered neighbors. For example:**

**Western Australia (wa) borders Northern Territory (nt) and South Australia (sa).**

**getthecolor(state):**

**This function assigns a valid color to a given state.**

**It iterates through the available domain\_colors (Red, Blue, Green) and calls the assigncolor function to check if the color is valid for that state.**

**If a valid color is found, it is returned. If none is valid (which shouldn’t happen here), it returns None.**

**assigncolor**

**This function checks if a color can be assigned to the state without violating the constraints.**

**It goes through each neighbor of the given state. If any neighbor already has the same color, the function returns False (i.e., the color is not valid).**

**If no neighbor has that color, it returns True (i.e., the color is valid).**

**sorts the states based on the degree heuristic. States with more neighbors are processed first because they are more constrained, and it’s more important to color them early.**

**Finally, it prints the final state-to-color mapping, showing which state received which color.**

**LOCAL SERCH**

**Create a recursive function that takes the current index, number of vertices, and output color array.**

**If the current index is equal to the number of vertices, check if the output color configuration is safe (adjacent vertices don’t share a color).**

**If conditions are met, print the configuration and break.**

**Assign a color to a vertex (1 to m).**

**For each assigned color, recursively call the function with the next index and number of vertices.**

**If any recursive function returns true, break the loop and return true.**

**Knowledge Representation**

**Initialize Symptom-Checking Functions:**

**Define functions for each illness (e.g., measles, flu, cold, chickenpox).**

**Each function takes specific symptom inputs (yes/no) and returns the illness name if the symptoms match, otherwise returns None.**

**Define Main Function (run\_diagnosis):**

**Prompt the user to enter their name.**

**Gather responses for each symptom by asking the user a series of yes/no questions.**

**Store the user's responses in a dictionary for easy access.**

**Diagnosis Initialization:**

**Create an empty list diagnosis to hold the possible illnesses based on the user's symptoms.**

**Check for Illnesses:**

**For each illness (measles, flu, cold, chickenpox), call the corresponding symptom-checking function with the appropriate symptoms.**

**If the function returns a valid illness (not None), append the illness name to the diagnosis list.**

**Filter Diagnoses:**

**Remove any None values from the diagnosis list to ensure only valid diagnoses are retained.**

**Generate Final Diagnosis:**

**If the diagnosis list contains illnesses, print the user's name along with the possible diagnoses.**

**If the diagnosis list is empty, inform the user that no diagnosis could be made based on the provided symptoms.**

**Execute the Diagnosis:**

**Call the run\_diagnosis function to initiate the symptom collection and diagnosis process.**