GRPA

PRACTICAL NO-01

* Title:- Implementation of DFS & BFS Algorithms in Python

* Objective:- To implement depth first search algorithm and breadth first search algorithm using an undirected graph & develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

* Introduction:-

- DFS and BFS are two fundamental algorithms used in graph theory for traversing & searching the nodes or vertices of a graph data structure.
- DFS is a graph traversal algorithm that visits all the vertices of a graph or tree data structure recursively by exploring as far as possible along each branch before backtracking.
- DFS uses a stack data structure to keep track of the visited vertices & the next to be visited.

- BFS on the other hand, is another graph travers al algorithm that visits all the vertices of a graph or tree data structure in a breadth-first order.
- It starts at an arbitrary vertex & visits all vertices that are at the same level before preceeding to the next level.
- BFS uses a queue data structure to keep track of the visited vertices & the next vertex to be visited.
- Both BFS & DFS have their own advantages & disadvantages & they are used in different applications based on their characteristics.
- DFS is more suitable for finding a path b/w two nodes, & is more efficient compared to DFS.

- BFS is suitable for finding the shortest path b/w 2 nodes & is best suited for finding all nodes within a certain distance from the starting node.

* Software:-
- Python environment
- Code editor such as Visual Studio Code or Pytharm.

* Procedure:-
Implementing Undirected Graph.
- Create an undirected graph using the adjacency matrix or adjacency list representation
- Implement a function to print the graph.

* Depth First Search Algorithm:-
- Implement a function for DFS traversal using a stack and recursion.
- Implement a function to print the DFS traversal result.

* Breadth First Search Algorithm:
- Implement a function for BFS traversal using a queue.
- Implement a function to print the BFS traversal result.

* Recursive algorithm for traversing all the vertices of a graph or tree:
- Implement a function to traverse all the vertices of graph or tree.
- Implement a function to print the recursive traversal result.

conclusion:-

From the code and output, we can conclude the following:

1) DFS & BFS are 2 different algorithms used for traversing an undirected graph.

2) DFS is a recursive algorithm that starts from a given vertex and explores as far as possible along each branch before backtracking.

) BFS is an iterative algorithm that starts from a given vertex and explores all the vertices at a given distance from the starting vertex before moving on to the vertices at the next distance.

) Both algorithms visit all the vertices of the graph exactly once.

) The order in which the vertices are visited can be different for DFS and BFS, depending on the structure of the graph and starting vertex.

PRACTICAL NO-02

* Title: Implement A star Algorithm for any game search problem.

* Objective:
Implementing the A^* algorithm for any game search problem is to find an optimal solution path from an initial state to goal state while minimizing the cost path. The A^* algorithm uses heuristic function to estimate the remaining cost from the current state to goal state, which allows it to search the most promising paths first. The ultimate goal is to find the optimal solution path with the least possible cost, in terms of time and space complexity.

* Introduction:-
The A^* algorithm is a popular and widely used search algo. in AI & game development. It is a heuristic search algo. designed to find the shortest path b/w 2 points while taking into account the estimated cost of each node in the search space. The A^* algorithm has proven to be very effective in solving game search problems, including the 8-queens problem, the Sokoban game & the sliding puzzle game.
The A^* algo. uses a heuristic f^n to estimate the remaining cost from the current state to the goal state, which allows it to search the most promising paths first. This makes it more efficient than other search algorithms, such as BFS and DFS. The A^* algorithm is also guaranteed to find the optimal solution path with the least possible cost, making

it a preferred choice for game search algorithm.

By using a heuristic function to estimate the remaining cost from the current state to the goal state, the A* algorithm can efficiently search for the optimal solution path, making it a powerful tool for game development and artificial intelligence applications.

* Software:
- Python environment
- Code Editor such as Visual Studio Code or Pycharm.

* Procedure:-

Here's a general procedure for implementing the A* algorithm for any game search problem:

1) Define the state representation: start by defining the state representation for the game search problem. This will include all the information needed to represent the current state of the game.
2) Define the heuristic function: The heuristic function estimates the cost from the current state to the goal state. It should take state as input and return a heuristic value.
3) Define the generate states function: The generate states function generates all possible successor states of the current state. It should take a node as input & return a list of all possible successor states of the current state.
4) Define the is-goal funct^n: The is-goal function checks if the current state is the goal state. It should take a state as input & return true if the current state is goal state, & False otherwise.

5) Implement the A* algorithm: Using the above defined f^n's, implement the A* algo. The algo. should keep track of the open and closed lists, generate all possible successor states of the current state, evaluate each successor state using the heuristic f^n & add the successor states to the open list. The algo should also keep track of path from start to current state.

6) Return sol^n path: Once the A* algo. reaches the goal state, return the sol^n path from the start to the goal state.

7) Test the implementation: Finally test the implementation using different test cases to ensure that it works correctly.

* conclusion:-

- In conclusion, the A* algo is a powerful search algo that can efficiently find an optimal sol^n path from an initial state to goal state while minimizing the cost of the path. The above code provides a well structured and efficient implementation of the A* algo for solving the 8-queens problem. By defining a Node class, a heuristic funct^n and an A star fu^n, the code demonstrates how to use the A* algo to search through the space of possible sol^n's & select the most promising path to the goal state.
- The code also shows how to generate all possible successor states of current state & evaluate them using the cost & heuristic value. Finally the code tests the A* algo by calling it with an initial state & printing the sol^n path if found.
- Overall the above code demonstrates the power & effectiveness of the A* algo in solving game search problems, and provides a good starting point for implementing the A* algo. for any other game search problem.

PRACTICAL NO-03

* Title:- Implement: Greedy search algorithm for any of the foll applications: Minimum Spanning Tree

* objective:- The objective of this Python code is to implement the Greedy Algorithm to find the minimum Spanning Tree of an undirected graph.
The program creates a graph object with a specified number of vertices and edges, and uses the prim_mst method to compute the total cost of the MST. The implementation relies on a priority queue to select the edges with the smallest weight and it marks the visited vertices to avoid cycles in the tree.

* Introduction:-
A Minimum Spanning Tree (MST) is a tree that connects all the vertices of a graph with the minimum possible total weight of its edges. The MST has many practical appl's, such as in N/W design, circuit layout, and transportation planning. One way to find the MST of a graph is by using the Greedy algorithm, which starts from an arbitrary vertex & iteratively adds the edge with the smallest weight that connects a visited vertex to an unvisited one. In this Python code, we implement the Greedy algorithm to find the MST of an undirected graph using a priority queue to select the edges with the smallest weight. The program defines a Graph class that allows us to add edges to the graph, and implements the prim_mst method that computes the total cost of the MST. The O/P of the program is the total

weight of the minimum spanning tree.

* Software:-
- Python environment
- code editor such as Visual Studio code or Pycharm.

* Procedure:-
1. Define the Graph class that represents an undirected graph with a specified number of vertices(V), and a list of adjacency lists(graph) that store the edges and weights for each vertex.
2. Implement the add_edge method of the Graph class that adds edges to the graph by appending a tuple with the destination vertex (v) & the weight (w) to the adjacency list of the source vertex (u).
3. Implement the prim_mst method of the Graph class that computes the minimum spanning tree of the graph using the Greedy algorithm.
4. Initialize the visited list to keep track of the visited vertices, and the min_heap priority, queue to store the edges with the smallest weight.
5. Add a tuple with the starting vertex (0) & its weight to the min_heap priority queue.
6. Initialize the cost variable to 0 to accumulate the total weight of the MST.
7. While the min_heap priority queue is not empty, do the following:
   a) Pop the vertex with the smallest weight from the min heap priority queue.
   b) If the vertex has already been visited, skip it. Otherwise, mark it as visited & add its weight to the total cost.
   c) Iterate over the neighbors of the vertex & add them

to the min-heap priority queue if they haven't been visited yet.

8. Return the cost variable, which corresponds to the total weight of the MST.

9. The procedure outlined above implements the Greedy algorithm to find the MST of an undirected graph using a priority queue to select the edges to the graph, & the prim_mst method computes the total cost of the MST. The program output is the total weight of the minimum spanning tree.

* Conclusion:-

- In conclusion, the implementation of the Greedy algorithm to find the Minimum spanning Tree of an undirected graph in Python provides an efficient and straightforward approach to solving this problem. The use of priority queue to select the edges with the smallest weight allows us to reduce the time complexity of the algorithm, making it suitable for large-scale graphs.
- The Graph class provides a flexible and modular way to represent & manipulate the graph data, which enhances the scalability and readability of the implementation. Overall, this Python code provides a practical and useful tool for solving real-world problems that require the optimization of a network or a transportation system.

PRACTICAL NO-04

* **Title:-** Implement a solution for a constant satisfaction Problem using Branch and Bound Backtracking for n-queens problem.

* **Objective:-**
To implement a Python program to solve n-queens problem using Branch & Bound Backtracking algorithms. The program ~~to~~ should define a class NQueens that includes methods to check if a given position is a valid loc^n for a queen on the board & to solve the n-queens problem using a recursive backtracking algorithm. The program should take an input n, representing the size of the board, and o/p all valid sol^n's to the problem. The program should utilize the branch and bound technique to optimize the backtracking process & reduce the number of unnecessary calculations. The o/p should include the number of sol^n's found and the details of each sol^n. The program should be written in Python & should be efficient and bug-free.

Introduction:-
The n-queens problem is a classic problem in comp sci. & mathematics, which requires placing n queens on an nxn chessboard in such a way that no 2 queens can attack each other. The sol^n to the problem involves finding all possible arrangements of n queens on the board that satisfy the constraints of the problem. The program will use the NQueens class to implement the algorithms & include the solve method to initiate the backtracking process. The program will also include the is_valid method to check if a given position is a valid loc^n for queen on the board. The output will include the number of sol^n's found &

the details of each solution.

* Software:-
- Python Environment
- Code Editor such as Visual studio code or Pycharm

* Procedure:-
1) Define the N Queens class with the following methods:
a) __init__(self,n): initialize the class with the board size n & create an empty list to store the solⁿ.
b) is_valid (self,row,col): check if the given position (row,col) is a valid locⁿ for a queen on the board.
c) solve(self, col): solve the n queens problem using a recursive backtracking algo. with branch & bound algorithm.
2) Implement the is_valid ~~algo~~ method to check if the given position (row,col) is a valid locⁿ for a queen on the board by checking if there is another queen in the same row, column or diagonal.
3) Implement the solve method to solve the n-queens problem using a recursive backtracking algo. ~~to~~ w/ branch and bound optimization The method should take the branch and bound optimization The method should take ~~one~~ col parameter to indicate the column number to start placing queens.
4) In the solve method, use a loop to iterate through all possible rows in the current column col. If the position(row, col) is a valid location for a queen, add the posⁿ to the solutⁿ list and recursively call the solve method with the next column col+1.
5) In the solve method, use the branch & bound technique to optimize the backtracking process. If at any point the number of queens placed on the board len(solution) equals the board size self.n, append the solⁿ to the list of valid solⁿs and return If the no of queens placed on the board plus the number of remaining

empty columns is less than self.n, return without further exploring that branch of the search tree.

6) Once all possible sol^n's have been explored, output the number of valid solutions found and the details of each solution.

7) Test the program with various board sizes to ensure that it works correctly & efficiently.

8) optimize the program by using techniques such as memoization, dynamic programming or parallelization to improve performance if needed.

9) Document the code and write comments to make it easy for other developers to understand the implementation and make modifications if necessary.

Conclusion:-

In this project, we have successfully implemented a Python program to solve the n-queens problem using branch and bound & backtracking algorithms. The program takes an input n, representing the size of the board, and outputs all valid sol^n's to the problem. We utilized the NQueens class to implement the algorithms and included the solve method to initiate the backtracking process. We also included the is_valid method to check if a given position is a valid loc^n for a queen on the board. We utilised the branch & bound technique to ~~utilise~~ optimise the backtracking process and reduce the number of unnecessary calculations.

The output includes the number of sol^n's found and the details of each sol^n.

PRACTICAL NO-05

Title:- Develop an elementary chatbot for any suitable customer interaction application.

Objective:- To develop an elementary chatbot for a suitable customer interaction appl^n that can effectively and efficiently engage with customers, provide them with relevant info, & improve their overall experience.
Understanding and responding to customer inquiries; The chatbot should be able to accurately understand and respond to customer inquiries in a timely manner. It should be able to interpret the intent of the customer's message & provide relevant information.

Introduction:-
In today's digital age, businesses are increasingly turning to chatbots to enhance customer experience and streamline customer interactions. chatbots are computer programs designed to simulate conversation with human users, using AI & NLP.
The objective of this assignment is to develop an elementary chatbot for a suitable customer interaction appl^n. The chatbot will be designed to engage with customers, provide them with relevant information, and improve their overall experience. To achieve this chatbot will be equipped with features such as understanding and responding to customer inquires, providing ~~continuously~~ personalised recommendations and continuously improving based on feedback and usage data.

* Software:
- Python Environment
- Code editor such as Visual studio code or Pycharm.

* Procedure :-
- The procedure for developing an elementary chatbot for a suitable customer interaction appl^n can be divided into the foll steps:
- Identify the customer interaction appl^n: The first step is to identify the customer interaction appl^n for which the chatbot will be developed. This could be customer service, e-commerce, marketing, or any other appl^n that involves customer interactions.
- Determine the chatbot's purpose and features: Once the customer interact^n appl^n has been identified, determine the purpose and features of the chatbot.
- Choose a chatbot development platform: There are several chatbot development platforms available, such as Dialogflow, IBM Watson and Microsoft Bot Framework. Chose a platform that meets the requirements of the chatbot's purpose and features.
- Create the chatbot's conversational flow: The conversational flow is the backbone of the chatbot. It determines how the chatbot will interact with users and how it will respond to different user inputs.
- Train the chatbot: ~~Once the chatbot has been~~ To ensure that the chatbot can accurately interpret user inputs and provide relevant responses, it must be trained on a dataset of sample conversations.
- Test the chatbot: Once the chatbot has been trained, it must be tested to ensure that it performs as expected.

- Deploy the chatbot: After testing, deploy the chatbot to the chosen customer interaction application.
- continuously improve the chatbot: A chatbot is never truly finished. Continuously gather feedback and usage data to make improvements to the chatbot's conversational flow.

* conclusion:-

This procedure provides a solid solution & foundation for developing an elementary chatbot for a customer interaction application. However, the specific details of each step may vary depending on the chosen chatbot development platform and customer interaction application.

PRACTICAL NO-06

* **Title:** Implement Expect system for Information

* **Objective:** To implement an expert system for information management using Python that allows users to manipulate a knowledge base of books in a library by adding, viewing searching and deleting books. The expect system will provide an interface that enables users to interact with the knowledge base using a set of predefined rules. The system will help users manage the library's book inventory efficiently and effectively.

* **Introduction:**
An expert system is a computer program that emulates the decision-making ability of a human expert. In this project, we will implement an Expert System for Information Management using Python. The system will be designed to manage a library's book inventory by allowing users to add, view, search and delete books. The system will provide a simple and intuitive interface that enables users to interact with the knowledge base of books. The objective of this project is to create an efficient & effective system that will help users manage the library's book inventory. The system will use a set of predefined rules to manipulate the knowledge base and provide users with accurate & relevant info. about books in the library.

* Software:
Python environment
Code editor such as Visual Studio code or Pycharm.

* Procedure:
1) The implementation of the expert system for Information Management using Python will involve the following steps:
2) Define the knowledge base:
Create a knowledge base that will store information about books in the library. The knowledge base will be implemented as a Python dictionary that contains information about each book such as title, author, ISBN and publisher.
3) Define the rules:
Create a set of rules that the expert system will use to manipulate the knowledge base. The rules will include functions that allow users to add, view, search and delete books from the knowledge base.
4) Create the expert system:
Create an expert system ~~to~~ that uses the rules to manipulate the knowledge base. The system will provide an interface that enables users to interact with the knowledge base using a set of predefined commands. The expert system will be implemented as a Python function that takes user i/p and executes the appropriate rule based on the user's input.
5) Test the system:
Test the system by adding, viewing, searching and deleting books from the knowledge base. Verify that the system is providing necessary, accurate & relevant information about books in the library. Make any necessary changes or improvements to the system to ensure that it is functionally correct.

6) Document the system:

Document the system by providing a detailed description of its design and implementation. Include examples of how to use the system and any limitations or known issues w/ the system. Provide instructions for users on how to use the system effectively.

★ Observation / conclusion:

- The Expert system for Information Management using Python presented in the previous sections provides a practical and effective solution for managing a library's book inventory.
- The system uses a set of predefined rules to manipulate a knowledge base in the form of a Python dictionary, allowing users to add, view, search & delete books from the inventory.
- The code is well-structured and easy to read, with clear documentation of each step in the process. The system also provides an intuitive interface for users to interact with, making it easy for users to manage the library's book inventory efficiently and effectively.
- Overall, the Expert System for Information Management using Python provides a flexible and practical solution for managing a library's book inventory and demonstrates the power and effectiveness of Expert Systems in solving complex problems in a variety of fields.