R.H.SAPAT COLLEGE OF ENGINEERING,
MANAGEMENT & RESEARCH, NASHIK

DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT CODE: 310258

LAB MANUAL
Laboratory Practice-II
(Artificial Intelligence & Cloud Computing)

Semester – II, Academic Year 2021-2022

Subject Teachers:-

Artificial Intelligence: - Dr. Neeta Deshpande and Mr. Rahul Chakre

Cloud Computing: - Mrs. Vrushali Nikam

Department: Computer Engineering		**Class: TE-A & TE-B**
Subject Name: Laboratory Practice-II **Academic Year: 2021 -22**		**Subject Code:310258**
Teaching Scheme:-	**Practical::4 Hours /Week**	
Examination Scheme:-	**Term Work: 50 Marks**	**Practical : 25 Marks**
Name of Faculty:	**Artificial Intelligence:-Dr. Neeta Deshpande and Mr. Rahul Chakre** **Cloud Computing :- Mrs. Vrushali Nikam**	

Course Objectives:

- To learn and apply various search strategies for AI
- To Formalize and implement constraints in search problems
- To understand the concepts of Information Security / Augmented and VirtualReality/Cloud Computing/Software Modeling and Architectures

Course Outcomes:

CO	State ments	Cognitive levelof learning
C318.1	Design a system using different informed search / uninformed searchor heuristic approaches	(Design)
C318.2	Apply basic principles of AI in solutions that require problem solving,inference, perception, knowledge representation, and learning	(Apply)
C318.3	Design and develop an interactive AI application	(Apply)
C318.4	Use tools and techniques in the area of Cloud Computing	(Apply)
C318.5	Use cloud computing services for problem solving	(Apply)
C318.6	Design and develop applications on cloud	(Apply)

List of Laboratory Assignments

Sr. No.	Group A	Page No.	CO
1	Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a Graph or tree data structure.	3	C318.1
2	Implement A star Algorithm for any game search problem	7	C318.1
3	Implement Greedy search algorithm for any of the following application: • Selection Sort • Minimum Spanning Tree • Single-Source Shortest Path Problem • Job Scheduling Problem • Prim's Minimal Spanning Tree Algorithm • Kruskal's Minimal Spanning Tree Algorithm • Dijkstra's Minimal Spanning Tree Algorithm	12	C318.1
4	Implement a solution for a Constraint Satisfaction Problem using Branch andBound and Backtracking for n-queens problem or a graph coloring problem.	16	C318.2
5	Develop an elementary catboat for any suitable customer interaction application.	19	C318.3
6	Implement any one of the following Expert System I. Information management II. Hospitals and medical facilities III. Help desks management IV. Employee performance evaluation V. Stock market OR Design a Mini-project on any one of the following topics not limited to:- • Airline scheduling and cargo schedules • Hospital Management system • Ticket booking system • Food Ordering System • Task management system	21	C318.3

Assignment No.:01

Problem Statement:

Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

Objectives:

1. To study the various Search Algorithms.
2. To study the depth first search algorithm.
3. To study the breadth first search algorithm.

Theory:

Uninformed Search Algorithms:

The search algorithms in this section have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and/or length of actions. Uninformed search is also called **Blind search**.The following uninformed search algorithms are discussed in this section.

1. Depth First Search
2. Breadth First Search
3. Uniform Cost Search

Each of these algorithms will have:

- A problem **graph,** containing the start node S and the goal node G.
- A **strategy,** describing the manner in which the graph will be traversed to get to G.
- A **fringe,** which is a data structure used to store all the possible states (nodes) that you cango from the current states.
- A **tree that** results while traversing to the goal node.
- A solution **plan,** which the sequence of nodes from S to G.

Depth First Search:

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case ofa graph) and explores as far as possible along each branch before backtracking.

Performance Measure:

d = the depth of the search tree = the number of levels of the search tree. n^i ***= number of nodes in level*** .

Time complexity: *Equivalent to the number of nodes traversed in DFS.*
$T(n) = 1 + n^2 + n^3 + n^4 + \ldots.. + n^d = O(n^d)$
Space complexity: *Equivalent to how large can the fringe get.*

*S(n) = O(n*d)*

Completeness: *DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.*

Optimality: *DFS is not optimal, meaning the number of steps in reaching the solution, or the costspent in reaching it is high.*

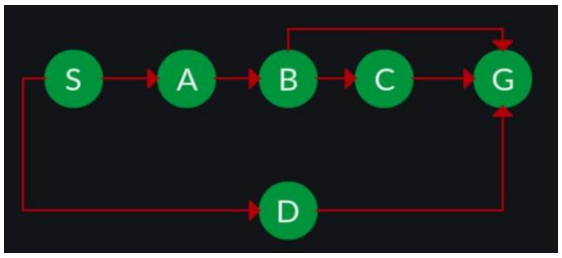
<u>Breadth First Search</u>:
Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a'search key'), and explores all of the neighbor nodes at the present depth prior to moving on tothe nodes at the next depth level.

d = the depth of the shallowest solution.n^i= number of nodes in level

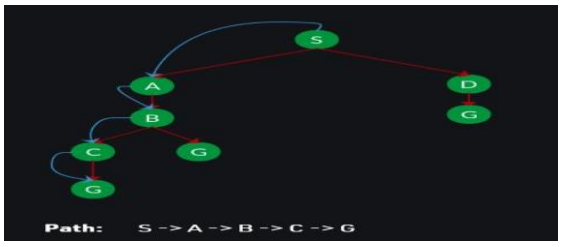
.

Time complexity: *Equivalent to the number of nodes traversed in BFS until the shallowest solution.*
$T(n) = 1 + n^2 + n^3 + n^4 + \ldots.+n^d = O(n^d)$
Space complexity: *Equivalent to how large can the fringe get.*
$S(n) = O(n^d)$
Completeness: *BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.*

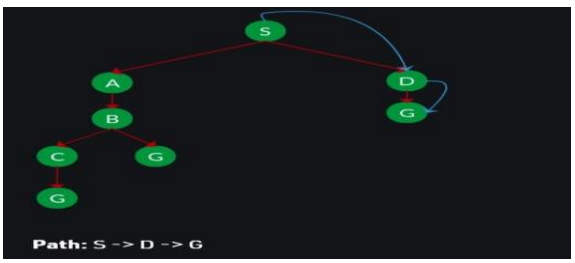
<u>Example</u> **Find path to move from node S to node G *using DFS and BFS.***



DFS



BFS



Algorithm/Flowchart:

DFS:

- **Step 1** – Push a starting node on stack, mark it visited.
- **Step 2** - Visit the adjacent unvisited vertex of start node. Mark it as visited. Display it. Pushit in a stack.
- **Step 3** − If no adjacent vertex is found, pop up a vertex from the stack. Repeat Step 2
- **Step 4** − Repeat Step 2 and Step 3 until the stack is empty.

BFS

- **Step 1** – Insert start node in Queue, mark it visited.
- **Step 2** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Step 3** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Step 4** − Repeat Step 3 and Step 4 until the queue is empty.

Input:

Graph: no of nodes, no of edges
n = 4, e = 6
Enter adjacent node information
0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3

Output:
DFS from vertex 2 – 2, 0, 1, 3
BFS from vertex 2 – 2, 0, 3, 1

Software Requirement:

1. Python3/Java … **GOOGLE COLAB**

Code Editor or IDE – Eclipse/ IntelliJ IDEA/ VSCode

Frequently Asked Questions:

1. What are the commonly used uninformed search algorithms?
2. Which data structure is used in BFS?
3. Which data structure is used in DFS?

What is the performance measure for DFS and BFS?

Conclusion:

We have successfully implemented depth first search and breadth first search algorithm for a graph.

Assignment No.:02

Problem Statement:

Implement A star Algorithm for any game search problem.

Objectives:

1. To study various Search Algorithms
2. To study the A* Algorithm

Theory:

A* Algorithm is the advanced form of the BFS algorithm (Breadth-first search), which searches forthe shorter path first than, the longer paths. It is a **complete** as well as an **optimal** solution for solving path and grid problems. The key feature of the A* algorithm is that it keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time. It also has a list that holdsall the nodes that are left to be explored and it chooses the most optimal node from this list, thus saving time not exploring unnecessary or less optimal nodes. So we use two lists namely open list and closed list the open list contains all the nodes that are being generated and are not existing in the closed list and each node explored after its neighboring nodes are discovered is put in the closed list and the neighbors are put in the open list this is how thenodes expand. Each node has a pointer to its parent so that at any given point it can retrace the path to the parent. Initially, the open list holds the start (Initial) node. The next node chosen from the open list is based on its **f score (f(n))**, the node with the least f-score is picked up and explored.

$$f(n) = g(n) + h(n)$$

Heuristic used in A* Where
g (n) : The actual cost path from the start node to the current node.h (n) : The actual cost path from the current node to goal node.
f (n) : The actual cost path from the start node to the goal node.

Performance Measure:
Optimal – find the least cost from the starting point to the ending point.
Complete – It means that it will find all the available paths from start to end.

A* uses a combination of heuristic value (h-score: how far the goal node is) as well as the g-score (i.e. the number of nodes traversed from the start node to current node).

Example: Puzzle Problem

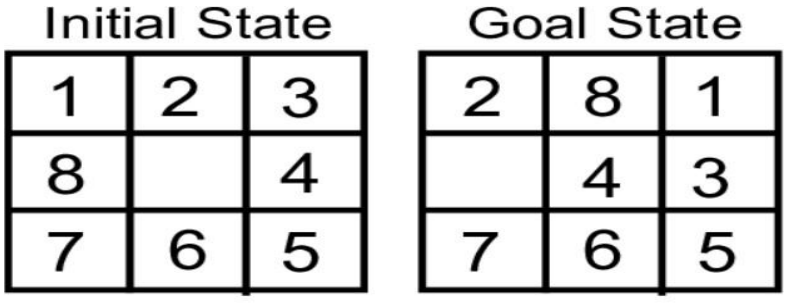


Fig 1. Start and Goal configurations of an 8-Puzzle.

In our 8-Puzzle problem, we can define the **h-score(h(n))** as the number of misplaced tiles by comparing the current state and the goal state or summation of the Manhattan distance between misplaced nodes.**g-score(g(n))** will be number of nodes traversed from a start node to get to the current node.

From Fig 1, we can calculate the **h-score** by comparing the initial (current) state and goal state andcounting the number of misplaced tiles.

Thus, **h-score** = 5 and **g-score** = 0 as the number of nodes traversed from the start node to the currentnode is 0. Similarly, calculate for all nodes.

Algorithm/Flowchart:

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the
    open list (you can leave its f
    at zero)

3.  while the open list is not empty
    a) find the node with the least f
       onthe open list, call it "q"
    b) pop q off the open list
    c) generate q's 8 successors and set
       theirparents to q

    d) for each successor
        i) if successor is the goal, stop search

        ii) else, compute both g and h for
successorsuccessor.g = q.g + distance between
                              successor and q
successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

successor.f = successor.g + successor.h

        iii) if a node with the same position as
           successor is in the OPEN list which has a
```

lower f than successor, skip this
successor

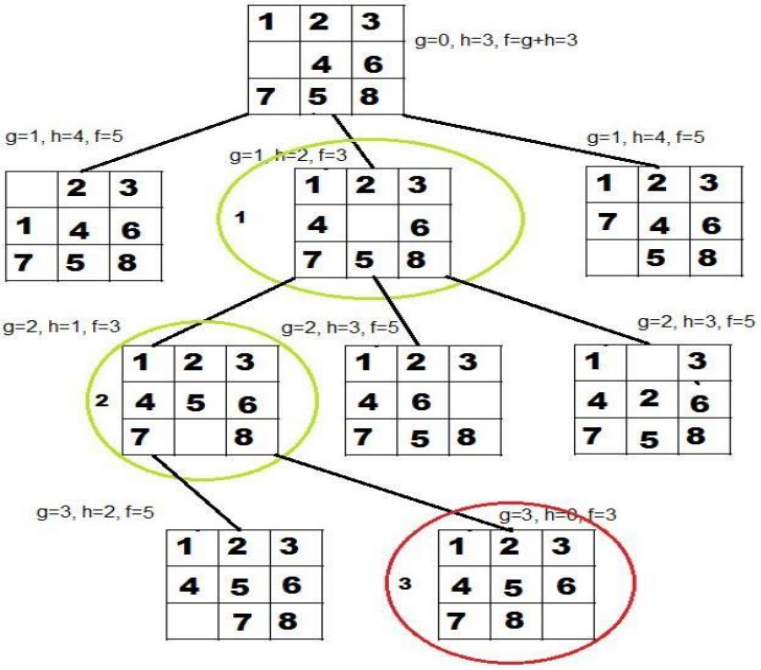
iV) if a node with the same position as
Successor is in the CLOSED list which
has a lower f than successor, skip this
successorotherwise, add the node to the
open list
end (for loop)

e) push q on the closed listend (while loop)

Input:

Start State:
1 2 3
_ 4 6
7 5 8
Goal State:
1 2 3
4 5 6
7 8 _

Sample Solution Stepwise:



Frequently Asked Questions:

1. What is the heuristic used in A* Algorithm?
2. Explain the performance measures of A* Algorithm.
3. List applications of A* Algorithms.
4. What are Informed Search Algorithms? Name the commonly used Informed Search Algorithms.

Conclusion:

Successfully implemented A* Algorithm for 8-Puzzles Problem

Assignment No.:03

Problem Statement:

Implement Greedy search algorithm for any of the following application:

- Selection Sort
- Minimum Spanning Tree
- Single-Source Shortest Path Problem
- Job Scheduling Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm

Objectives:

To study various Greedy Search Algorithms

Theory:

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy. The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem thatdemands either maximum or minimum results. Let's understand through some terms.

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is atechnique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future. This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solutionwhich is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

The following are the characteristics of a greedy method:

- Construct the solution in an optimal way, this algorithm creates two sets where one setcontains all the chosen items, and another set contains the rejected items.
- A Greedy algorithm makes good local choices in the hope that the solution should be eitherfeasible or optimal.

The components that can be used in the greedy algorithm are:

- Candidate set: A solution that is created from the set is known as a candidate set.

Selection function: This function is used to choose the candidate or subset which can be addedin the solution.

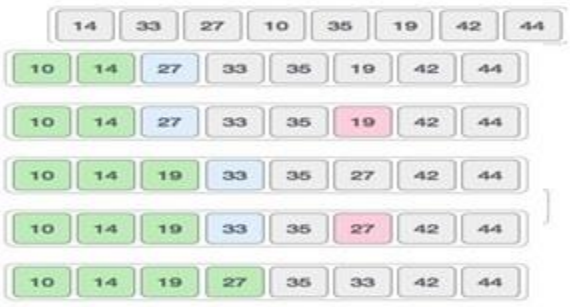
- Feasibility function: A function that is used to determine whether the candidate or subset canbe used to contribute to the solution or not.
- Objective function: A function is used to assign the value to the solution or the partial solution.
- Solution function: This function is used to intimate whether the complete function has beenreached or not.

Applications of Greedy Algorithm

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal'salgorithm.
- It is used in job sequencing with a deadline.

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted arrayboundary by one element to the right. This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where n is the number of items. Consider the following depicted array as an example. For the first position in the sorted list, the whole list is scanned sequentially. The first positionwhere 14 is stored presently, we search the whole list and find that 10 is the lowest value. So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in thelist, appears in the first position of the sorted list. For the second position, where 33 is residing, we start scanning the rest of the list in a We find that 14 is the second lowest value in the list and it should appear at the second place. Weswap these values. After two iterations, two least values are positioned at the beginning in a sorted manner.The same process is applied to the rest of the items in the array linear manner.



Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Algorithm/Flowchart:

```
procedure
selection sortlist  :
array of items n:
size of list


for i = 1 to n - 1
/* set current element as
  minimum*/min = i

  /* check the element to be

  minimum */for j = i+1 to n
    if list[j] < list[min] then
      mi=j
    ;end if
  end for

  /* swap the minimum element with the current
  element*/if indexMin != i then
swap list[min] and list[i]end if
```

Frequently Asked Questions:

1. What is the Greedy Algorithm?
2. Explain the selection sort.

Conclusion:

Successfully implemented greedy approach for selection sort algorithm

Assignment No.:04

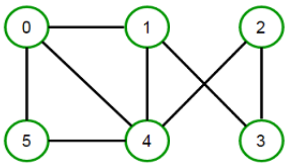
Problem Statement:
Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.

Objectives:
1.To study Constraint Satisfaction Problem using Branch and Bound Algorithms

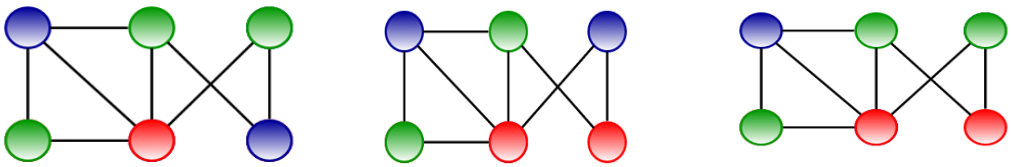
Theory:
We have seen so many techniques like Local search, Adversarial search to solve different problems. The objective of every problem-solving technique is one, i.e., to find a solution to reach the goal. Although, in adversarial search and local search, there were no constraints on the agents while solving the problems and reaching to its solutions. In this section, we will discuss another type of problem-solving technique known as Constraint satisfaction technique. By the name, it is understood that constraint satisfaction means solving a problem under certain constraints or rules. Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. Such type of technique leads to a deeper understanding of the problem structure as well as its complexity.
Constraint satisfaction depends on three components, namely:

- X: It is a set of variables.
- D: It is a set of domains where the variables reside. There is a specific domain for each variable.
- C: It is a set of constraints which are followed by the set of variables.

In constraint satisfaction, domains are the spaces where the variables reside, following the problem specific constraints. These are the three main elements of a constraint satisfaction technique. The constraint value consists of a pair of {scope, rel}. The scope is a tuple of variables which participate in the constraint and rel is a relation which includes a list of values which the variables can take to satisfy the constraints of the problem.
Graph coloring problem involves assigning colors to certain elements of a graph subject to certain restrictions and constraints. In other words, the process of assigning colors to the vertices such that no two adjacent vertexes have the same color is caller Graph Coloring. This is also known as **vertex coloring**. Graph coloring (also called vertex coloring) is a way of coloring a graph's vertices such that no two adjacent vertices share the same color. This post will discuss a greedy algorithm for graph coloring and minimize the total number of colors used.
For example, consider the following graph:



We can color it in many ways by using the minimum of 3 colors.



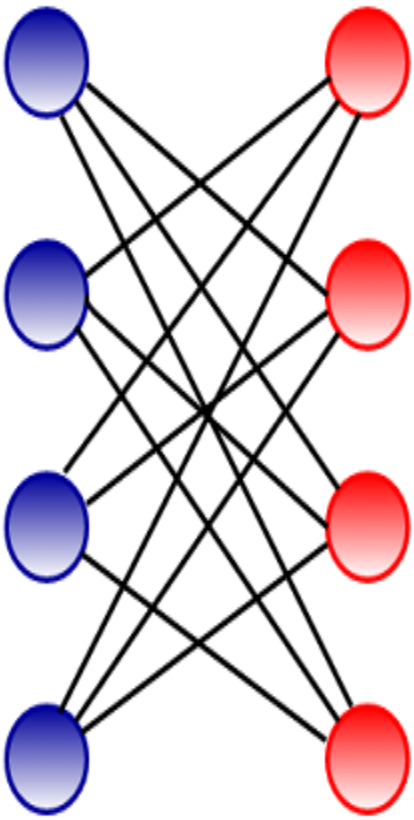
Please note that we can't color the above graph using two colors. Before discussing the greedy algorithm to color graphs, let's talk about basic graph coloring terminology.

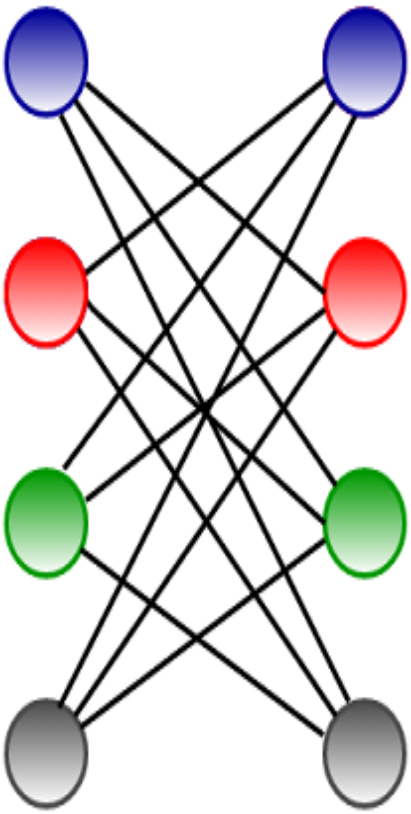
K–colorable graph:
A coloring using at most k colors is called a (proper) *k*–coloring, and a graph that can be assigned a (proper) *k*–coloring is *k*–colorable.

K–chromatic graph:
The smallest number of colors needed to color a graph G is called its chromatic number, and a graph that is *k*–chromatic if its chromatic number is exactly k.

Greedy coloring *considers the vertices of the graph in sequence and assigns each vertex its first available color,* i.e., vertices are considered in a specific order v_1, v_2, … v_n, and v_i and assigned the smallest available color which is not used by any of v_i's neighbors.

Greedy coloring doesn't always use the minimum number of colors possible to color a graph. For a graph of maximum degree x, greedy coloring will use at most $x+1$ color. Greedy coloring can be arbitrarily bad; for example, the following crown graph (a complete bipartite graph), having n vertices, can be 2–colored (refer left image), but greedy coloring resulted in $n/2$ colors (refer right image).





N-Queens Problem

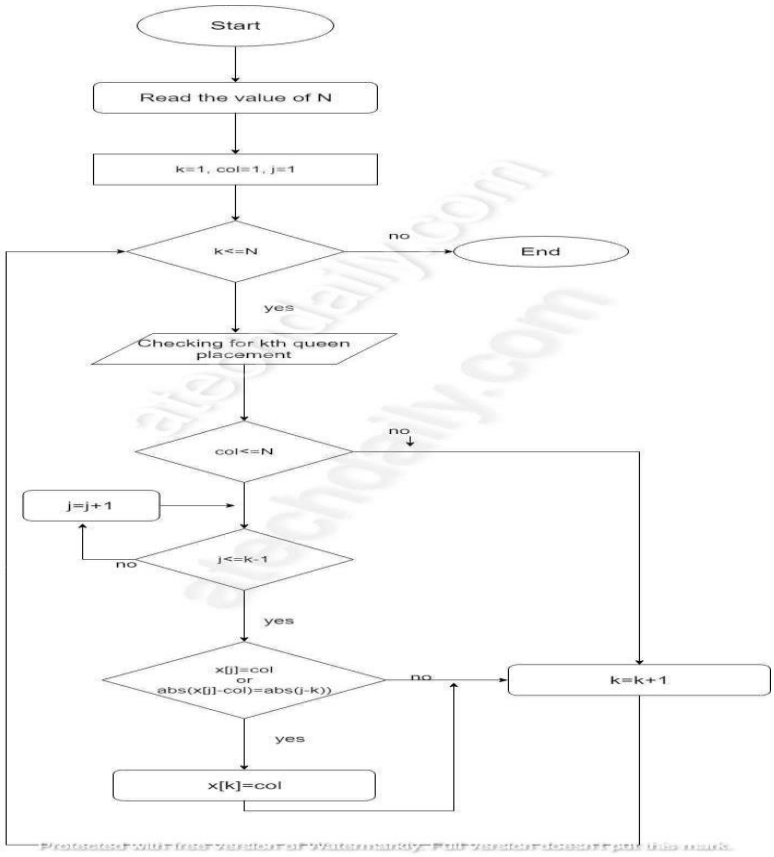
Implement N Queen's problem using Back Tracking

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken. For example, in a maze problem, the solution depends on all the steps you take one-by-one. If any of those steps is wrong, then it will not lead us to the solution. In a maze problem, we first choose a path and continue moving along it. But once we understand that the particular path is incorrect, then we just come back and change it. This is what backtracking basically is.In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.Thus, the general steps of backtracking are:

- Start with a sub-solution
- Check if this sub-solution will lead to the solution or not
- If not, then come back and change the sub-solution and continue again

N Queen Problem: N Queens Problem is a famous puzzle in which n-queens are to be placed on a nxn chess board such that no two queens are in the same row, column or diagonal. The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.

Flowchart for N Queen problem



Algorithm for N Queens Problem

Step 1: Start

Step 2: Given n queens, read n from user and let us denote the queen number by k. k=1,2,..,n.

Step 3: We start a loop for checking if the kth queen can be placed in the respective column of the kth row.

Step 4: For checking that whether the queen can be placed or not, we check if the previous queens are not in diagonal or in same row with it.

Step 5: If the queen cannot be placed backtracking is done to the previous queens until a feasible solution is not found.

Step 6: Repeat the steps 3-5 until all the queens are placed.

Step 7: The column numbers of the queens are stored in an array and printed as a n-tuple solution

Step 8: Stop

Explanation:
In the above algorithm,

1. For the n queen problem we take input of n, lets say n=4 so, k=1,2,3,4.
2. For placing the first queen i.e k=1,we start a loop for n columns i.e n=4 so till the fourth column.
3. The first queen can be placed at first column only.
4. Then we move for the second queen and place it seeing that the first queen is not in the same column or in diagonal with the second queen.
5. Similarly, the third queen and the fourth queen are placed. But if the fourth queen cannot be placed as it lies in same column or is in diagonal with other queens then back-tracking is done to the previous queens in order of 3,2,1 to achieve the unique feasible solution.
6. For an n problem queen the same way all the n queens are placed and if the n^{th} cannot be placed back-tracking is done and the queens are re-ordered and solution is obtained.

Frequently Asked Questions:

1. What is the Constraint satisfaction Algorithm?
2. Explain Graph coloring Problem in detail

Conclusion:
Successfully implemented constraint satisfaction algorithm

Assignment No.:05

Problem Statement:
Develop an elementary chatbot for any suitable customer interaction application.

Objectives:
1.To study elementary chatbot for any suitable customer interaction application

Theory:
Artificial intelligence chatbots are text- or voice-based interfaces that provide support and connect human users with the services or information they need by simulating a traditional person-to-person conversation. Text-based chatbots are often deployed online on websites and social media platforms to provide customer support and outreach. Voice-based chatbots, on the other hand, are most typically used for call deflection and sorting or over-the-phone customer service. Most smartphones come equipped with a built-in chatbot, and smart speakers with chatbot functionality have been trendy gift-giving items for several years. The most typical chatbot interaction occurs on a business site. These customer service bots usually pop up after a human user navigates around a site for a few minutes or exhibits behaviors that show that they have become "lost" or are having trouble connecting with the information they need. Once the chatbot window presents itself, the user can enter their question in plain, syntactical English. The bot's language recognition functions break down the question and, at the speed of light, compares the query to its data bank of previously asked questions to look for ways customers have achieved satisfying results in similar situations. When that search is complete, the chatbot shares the best and most relevant information with the user.

Applications

Chatbots are valuable for both businesses and consumers, as they reduce barriers to data access created by everything from physical disability to tech savviness, streamline navigation to connect users with results as quickly as possible, and provide a cost-effective alternative to staffing massive numbers of support professionals or contracting with expensive call centers. In fact, in many scenarios, artificial intelligence chatbot services can actually provide a faster, more straightforward experience than dealing with a human professional. If that sounds too good to be true, here are some examples of ways AI-based chatbots are deployed every day:

- **Software Requirements: Google Colab**

Assignment No.:06

Problem Statement:

Implement any one of the following Expert System

I. Information management
II. Hospitals and medical facilities
III. Help desks management
IV. Employee performance evaluation
V. Stock market trading

OR

Design a Miniproject on any one of the following topics not limited to:-

- Airline scheduling and cargo schedules
- Hospital Management system
- Ticket booking system
- Food Ordering System
- Task management system

Library Management System

Objectives:

To study Expert System/ Designed System

Theory:

A system that uses human expertise to make complicated decisions.Simulates reasoning by applying knowledge and interfaces.Uses expert's knowledge as rules and data within the system.Models the problem solving ability of a human expert.

Components of an ES:

1. Knowledge Base
    i. Represents all the data and information inputed by experts in the field.
    ii. Stores the data as a set of rules that the system must follow to make decisions.
2. Reasoning or Inference Engine
    i. Asks the user questions about what they are looking for.
    ii. Applies the knowledge and the rules held in the knowledge base.
    iii. Appropriately uses this information to arrive at a decision.
3. User Interface
    i. Allows the expert system and the user to communicate.
    ii. Finds out what it is that the system needs to answer.
    iii. Sends the user questions or answers and receives their response.
4. Explanation Facility
    i. Explains the systems reasoning and justifies its conclusions.

Frequently Asked Questions:

What is the Expert System?

Conclusion:

Successfully implemented Expert system application.

1. Code for BFS_for_Undirected_Graph

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
```

```
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                  " (starting from vertex 2)")
g.BFS(2)
```

Output: Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

2. Code for DFS_for_Undirected_Graph

```
# Python3 program to print DFS traversal
# from a given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation


class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS  current node v and entire set of visited nod
es
    def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self, v):

        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function
```

```
        # to print DFS traversal
        self.DFSUtil(v, visited)

# Driver code


# Create a graph given
# in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

Output: Following is DFS from (starting from vertex 2)
2 0 1 3

3. Code for A Star - 8 puzzle

```
class Node:
    def __init__(self, data, level, fval):
        """ Initialize the node with the data, level of the node and the c
alculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank s
pace
            either in the four directions {up,down,left,right} """
        x, y = self.find(self.data, '_')
        """ val_list contains position values for moving the blank space i
n either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
                child_node = Node(child, self.level+1, 0)
                children.append(child_node)
        return children

    def shuffle(self, puz, x1, y1, x2, y2):
        """ Move the blank space in the given direction and if the positio
n value are out
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.d
ata):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None

    def copy(self, root):
        """ Copy function to create a similar matrix of the given node"""
```

```
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self, puz, x):
        """ Specifically used to find the position of the blank space """
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if puz[i][j] == x:
                    return i, j


class Puzzle:
    def __init__(self, size):
        """ Initialize the puzzle size by the specified size,open and clos
ed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self, start, goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) +
g(x) """
        return self.h(start.data, goal)+start.level

    def h(self, start, goal):
        """ Calculates the different between the given puzzles """
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp
```

```
    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()

        start = Node(start, 0, 0)
        start.fval = self.f(start, goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/")
            print("  V  \n")
            for i in cur.data:
                for j in i:
                    print(j, end=" ")
                print("")
            """ If the difference between current and goal node is 0 we ha
ve reached the goal node"""
            if(self.h(cur.data, goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i, goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]

            """ sort the opne list based on f value """
            self.open.sort(key=lambda x: x.fval, reverse=False)


puz = Puzzle(3)
puz.process()
```

Output:

```
Enter the start state matrix

1 2 3
5 6 _
7 8 4
Enter the goal state matrix

1 2 3
5 8 6
_ 7 4


  |
  |
 \'/
  V

1 2 3
5 6 _
7 8 4

  |
  |
 \'/
  V

1 2 3
5 _ 6
7 8 4

  |
  |
 \'/
  V

1 2 3
5 8 6
7 _ 4

  |
  |
 \'/
  V

1 2 3
5 8 6
7 4
```

4. Code for N-Queen Problem

```
# Python program to solve N Queen
# Problem using backtracking

global N
N = 4

def printSolution(board):
  for i in range(N):
    for j in range(N):
      print (board[i][j],end=' ')
    print()
# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):

  # Check this row on left side
  for i in range(col):
    if board[row][i] == 1:
      return False

  # Check upper diagonal on left side
  for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
    if board[i][j] == 1:
      return False

  # Check lower diagonal on left side
  for i, j in zip(range(row, N, 1), range(col, -1, -1)):
    if board[i][j] == 1:
      return False

  return True

def solveNQUtil(board, col):
  # base case: If all queens are placed
  # then return true
  if col >= N:
    return True

  # Consider this column and try placing
  # this queen in all rows one by one
```

```
    for i in range(N):

      if isSafe(board, i, col):
        # Place this queen in board[i][col]
        board[i][col] = 1

        # recur to place rest of the queens
        if solveNQUtil(board, col + 1) == True:
          return True

        # If placing queen in board[i][col
        # doesn't lead to a solution, then
        # queen from board[i][col]
        board[i][col] = 0

    # if the queen can not be placed in any row in
    # this column col then return false
    return False

  # This function solves the N Queen problem using
  # Backtracking. It mainly uses solveNQUtil() to
  # solve the problem. It returns false if queens
  # cannot be placed, otherwise return true and
  # placement of queens in the form of 1s.
  # note that there may be more than one
  # solutions, this function prints one of the
  # feasible solutions.
  def solveNQ():
    board = [ [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]
        ]
    if solveNQUtil(board, 0) == False:
      print ("Solution does not exist")
      return False

    printSolution(board)
    return True

  solveNQ()
```

Output: 0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0 True

5. Code for Graph Coloring Problem

```
# Adjacent Matrix
G = [[ 0, 1, 1, 0, 1, 0],
     [ 1, 0, 1, 1, 0, 1],
     [ 1, 1, 0, 1, 1, 0],
     [ 0, 1, 1, 0, 0, 1],
     [ 1, 0, 1, 0, 0, 1],
     [ 0, 1, 0, 1, 1, 0]]



# inisiate the name of node.
node = "abcdef"
t_={}
for i in range(len(G)):
  t_[node[i]] = i

# count degree of all node.
degree =[]
for i in range(len(G)):
  degree.append(sum(G[i]))

# inisiate the posible color
colorDict = {}
for i in range(len(G)):
  colorDict[node[i]]=["Blue","Red","Yellow","Green"]


# sort the node depends on the degree
sortedNode=[]
indeks = []

# use selection sort
for i in range(len(degree)):
  _max = 0
  j = 0
  for j in range(len(degree)):
    if j not in indeks:
      if degree[j] > _max:
        _max = degree[j]
        idx = j
  indeks.append(idx)
  sortedNode.append(node[idx])

# The main process
```

```
theSolution={}
for n in sortedNode:
  setTheColor = colorDict[n]
  theSolution[n] = setTheColor[0]
  adjacentNode = G[t_[n]]
  for j in range(len(adjacentNode)):
    if adjacentNode[j]==1 and (setTheColor[0] in colorDict[node[j]]):
      colorDict[node[j]].remove(setTheColor[0])


# Print the solution
for t,w in sorted(theSolution.items()):
  print("Node",t," = ",w)
```

Output:
```
Node a  =  Yellow
Node b  =  Blue
Node c  =  Red
Node d  =  Yellow
Node e  =  Blue
Node f  =  Red
```

6. Code for Kruskal's MST

```
from collections import defaultdict

# Class to represent a graph


class Graph:

  def __init__(self, vertices):
    self.V = vertices # No. of vertices
    self.graph = [] # default dictionary
    # to store graph

  # function to add an edge to graph
  def addEdge(self, u, v, w):
    self.graph.append([u, v, w])

  # A utility function to find set of an element i
  # (uses path compression technique)
  def find(self, parent, i):
    if parent[i] == i:
      return i
    return self.find(parent, parent[i])

  # A function that does union of two sets of x and y
  # (uses union by rank)
  def union(self, parent, rank, x, y):
    xroot = self.find(parent, x)
    yroot = self.find(parent, y)

    # Attach smaller rank tree under root of
    # high rank tree (Union by Rank)
    if rank[xroot] < rank[yroot]:
      parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
      parent[yroot] = xroot

    # If ranks are same, then make one as root
    # and increment its rank by one
    else:
      parent[yroot] = xroot
      rank[xroot] += 1
```

```
# The main function to construct MST using Kruskal's
  # algorithm
def KruskalMST(self):

  result = [] # This will store the resultant MST

  # An index variable, used for sorted edges
  i = 0

  # An index variable, used for result[]
  e = 0

  # Step 1: Sort all the edges in
  # non-decreasing order of their
  # weight. If we are not allowed to change the
  # given graph, we can create a copy of graph
  self.graph = sorted(self.graph,
           key=lambda item: item[2])

  parent = []
  rank = []

  # Create V subsets with single elements
  for node in range(self.V):
    parent.append(node)
    rank.append(0)

  # Number of edges to be taken is equal to V-1
  while e < self.V - 1:

    # Step 2: Pick the smallest edge and increment
    # the index for next iteration
    u, v, w = self.graph[i]
    i = i + 1
    x = self.find(parent, u)
    y = self.find(parent, v)

    # If including this edge does't
    # cause cycle, include it in result
    # and increment the indexof result
    # for next edge
    if x != y:
      e = e + 1
      result.append([u, v, w])
      self.union(parent, rank, x, y)
```

```
            # Else discard the edge

        minimumCost = 0
        print ("Edges in the constructed MST")
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree" , minimumCost)

# Driver code
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

# Function call
g.KruskalMST()
```

Output: Edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Spanning Tree 19

7. Code for Prim's MST

```
import sys # Library for INT_MAX

class Graph():

  def __init__(self, vertices):
    self.V = vertices
    self.graph = [[0 for column in range(vertices)]
          for row in range(vertices)]

  # A utility function to print the constructed MST stored in parent[]
  def printMST(self, parent):
    print ("Edge \tWeight")
    for i in range(1, self.V):
      print (parent[i], "-", i, "\t", self.graph[i][parent[i]])

  # A utility function to find the vertex with
  # minimum distance value, from the set of vertices
  # not yet included in shortest path tree
  def minKey(self, key, mstSet):

    # Initialize min value
    min = sys.maxsize

    for v in range(self.V):
      if key[v] < min and mstSet[v] == False:
        min = key[v]
        min_index = v

    return min_index

  # Function to construct and print MST for a graph
  # represented using adjacency matrix representation
  def primMST(self):

    # Key values used to pick minimum weight edge in cut
    key = [sys.maxsize] * self.V
    parent = [None] * self.V # Array to store constructed MST
    # Make key 0 so that this vertex is picked as first vertex
    key[0] = 0
    mstSet = [False] * self.V

    parent[0] = -1 # First node is always the root of
```

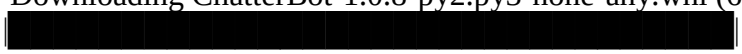
```
    for cout in range(self.V):

        # Pick the minimum distance vertex from
        # the set of vertices not yet processed.
        # u is always equal to src in first iteration
        u = self.minKey(key, mstSet)

        # Put the minimum distance vertex in
        # the shortest path tree
        mstSet[u] = True

        # Update dist value of the adjacent vertices
        # of the picked vertex only if the current
        # distance is greater than new distance and
        # the vertex in not in the shortest path tree
        for v in range(self.V):

            # graph[u][v] is non zero only for adjacent vertices of m
            # mstSet[v] is false for vertices not yet included in MST
            # Update the key only if graph[u][v] is smaller than key[v]
            if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.g
raph[u][v]:
                key[v] = self.graph[u][v]
                parent[v] = u

    self.printMST(parent)

g = Graph(5)
g.graph = [ [0, 2, 0, 6, 0],
        [2, 0, 3, 8, 5],
        [0, 3, 0, 0, 7],
        [6, 8, 0, 0, 9],
        [0, 5, 7, 9, 0]]

g.primMST();
```

Output:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 – 4	5

8. Code for Chatbot

```
!pip install chatterbot
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer

bot = ChatBot('Bot')

trainer = ListTrainer(bot)
trainer.train([
    "Hi, can I help you",
    "Who are you?",
    "I am your virtual assistant. Ask me any questions...",
    "Where do you operate?",
    "We operate from Singapore",
    "What payment methods do you accept?",
    "We accept debit cards and major credit cards",
    "I would like to speak to your customer service agent",
    "please call +65 3333 3333. Our operating hours are from 9am to 5pm, M
onday to Friday"

])
trainer.train([
    "What payment methods do you offer?",
    "We accept debit cards and major credit cards",
    "How to contact customer service agent",
    "please call +65 3333 3333. Our operating hours are from 9am to 5pm, M
onday to Friday"

])

while True:
  request=input('you :')
  if request == 'OK' or request == 'ok':
    print('Bot: bye')
    break
  else:
    response=bot.get_response(request)

    print('Bot:', response)
```

Output: Collecting chatterbot
Downloading ChatterBot-1.0.8-py2.py3-none-any.whl (63 kB)
 63 kB 1.7 MB/s

Requirement already satisfied: pytz in /usr/local/lib/python3.7/dist-packages (from chatterbot) (2022.1)
Collecting mathparse<0.2,>=0.1
  Downloading mathparse-0.1.2-py3-none-any.whl (7.2 kB)
Collecting sqlalchemy<1.4,>=1.3
  Downloading SQLAlchemy-1.3.24-cp37-cp37m-manylinux2010_x86_64.whl (1.3 MB)
     |████████████████████████████████| 1.3 MB 11.1 MB/s
Requirement already satisfied: python-dateutil<2.9,>=2.8 in /usr/local/lib/python3.7/dist-packages (from chatterbot) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil<2.9,>=2.8->chatterbot) (1.15.0)
Installing collected packages: sqlalchemy, mathparse, chatterbot
  Attempting uninstall: sqlalchemy
    Found existing installation: SQLAlchemy 1.4.35
    Uninstalling SQLAlchemy-1.4.35:
      Successfully uninstalled SQLAlchemy-1.4.35
Successfully installed chatterbot-1.0.8 mathparse-0.1.2 sqlalchemy-1.3.24
List Trainer: [####################] 100%
List Trainer: [####################] 100%
you :hi
Bot: I would like to speak to your customer service agent
you :yes
Bot: I am your virtual assistant. Ask me any questions...