MANGALAM COLLEGE OF ENGINEERING

DEPARTMENT OF ARTIFICIAL INTELLIGENCE & MACHINE LEARNING (AM)

LAB MANUAL

AIL 333 AI ALGORITHM LAB

SEMESTER V

Submitted By

SNEHA SEBASTIAN

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

MANGALAM COLLEGE OF ENGINEERING

Mangalam Hills Vettimukal P O Kottayam

Affiliated to APJ Abdul Kalam Technological University Approved By the All India Council For Technical Education(AICTE)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time),those who come after 5 minutes will not be allowed into the lab.

2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis / program / experiment details.

3. Student should enter into the laboratory with:

a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output,etc.,) filled in for the lab session.

b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.

c. Proper Dress code and Identity Card.

4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.

5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.

6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.

7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.

8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.

9. Students must take the permission of the faculty in case of any urgency to go out ;if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.

10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

LIST OF EXPERIMENTS

Sl No	Experiments	Page No
1	Installation and working on various AI tools viz. Python, R, GATE, NLTK, MATLAB etc*	
2	Implement basic search strategies for selected AI applications*	
3	Implement state space search algorithms*	
4	Implement informed search algorithms*	
5	Implement backtracking algorithms for CSP*	
6	Implement local search algorithms for CSP*	
7	Implement propositional logic inferences for AI tasks*	
8	Implementation of Knowledge representation schemes*	
9	Implement travelling salesman problem*	
10	Develop a program to construct a pruned game tree using Alpha-Beta pruning. Take the sequence, [10,5,7,11,12,8,9,8,5,12,11,12,9,8,7,10] of MINIMAX values for the nodes at the cutoff depth of 4 plies. Assume that branching factor is 2, MIN makes the first move, and nodes are generated from left to right.	
	CONTENT BEYOND	
1	Implement knowledge base with supervised learning	

Experiment 1
Implementation and working on various AI Tools

AIM:

To understand the implementation and working on various AI Tools like Python,R and Gate

I. Python

1. **Python Installation:** First, you need to install Python. Download the latest version from the official Python website (https://www.python.org/downloads/) and follow the installation instructions for your operating system.
2. **Package Management:** Python has a rich ecosystem of AI libraries. The most popular package manager for Python is **pip**, which allows you to install packages with ease. It comes installed with Python by default.
3. **AI Libraries:** Here are some popular Python AI libraries:

- NumPy: For numerical computing.
- Pandas: For data manipulation and analysis.
- Matplotlib and Seaborn: For data visualization.
- Scikit-learn: For machine learning.
- TensorFlow or PyTorch: For deep learning.

4. **Installation:** You can install these libraries using **pip**. Open a terminal or command prompt and use the following commands:

```bash
pip install numpy pandas matplotlib seaborn scikit-learn tensorflow torch
```

5. **Working with AI Tools:** After installation, you can use these libraries in your Python scripts. For example, to use NumPy:

```python
import numpy as np

# Create a NumPy array
data = np.array([1, 2, 3, 4, 5])

# Perform operations with NumPy arrays
mean = np.mean(data)
print(mean)
```

II. R

1. **R Installation:** To get started with R, download the latest version of R from the official website (https://www.r-project.org/) and install it on your system.
2. **Package Management:** R uses **install.packages()** to install packages. By default, R comes with a package manager called **install.packages()**.

3. **AI Libraries:** R also has several AI libraries available. Some popular ones are:
- dplyr: For data manipulation.
- ggplot2: For data visualization.
- caret: For machine learning.
- tensorflow or keras: For deep learning.

4. **Installation:** You can install these libraries using the following commands in the R console:

```R
install.packages("dplyr")
install.packages("ggplot2")
install.packages("caret")
install.packages("tensorflow") # or install.packages("keras")
```

5. **Working with AI Tools:** After installation, you can load the packages and use them in your R scripts. For example:

```R
library(dplyr)
data <- c(1, 2, 3, 4, 5)
mean_val <- mean(data)
print(mean_val)
```

III. GATE (General Architecture for Text Engineering)

GATE is a Java-based platform used for natural language processing and text analysis.

Installation:

- Download GATE from the official website: https://gate.ac.uk/
- Extract the downloaded archive to a directory on your system.

Running GATE:

- Navigate to the GATE installation directory and run **gate.bat** (Windows) or **gate.sh** (Linux/Mac) from the command line.

Creating and Running Applications:

- GATE uses configurations called "applications" for various text processing tasks.
- Use the GATE Developer GUI to create and manage applications.
- You can also interact with GATE using Java APIs.

RESULT:

Various AI Tools are familiarized.

EXPERIMENT-2

BASIC SEARCH STRATEGY FOR AI APPLICATION

AIM

To implement basic search strategy for the vacuum cleaner world

ALGORITHM:

1. Initialize `final_goal_state` dictionary with both rooms marked as clean.

2. Initialize `cost` to keep track of the performance measure.

3. Take user input for the `location_input` (location of the vacuum) and `status_input` (status of the room at the current location: clean or dirty).

4. Take user input for `status_input_complement` (status of the other room).

5. Determine the goal state `goal_state` based on the location and status inputs.

6. Print the initial status of the agent and the world.

7. If the agent's location is 'A':

   a. If room A is dirty (status_input == '1'):

      i. Clean room A.

      ii. Increment `cost`.

      iii. If room B is dirty (status_input_complement == '1'), clean room B, increment `cost`, and print the goal state and total cost.

      iv. If room B is clean, move to room B and print the goal state and total cost.

   b. If room A is clean:

      i. Move to room B.

      ii. If room B is dirty (status_input_complement == '1'), clean room B, increment `cost`, and print the goal state and total cost.

      iii. If room B is clean, print that both rooms are clean and the goal state is already reached.

8. If the agent's location is 'B':

   a. If room B is dirty (status_input == '1'):

      i. Clean room B.

      ii. Increment `cost`.

iii. If room A is dirty (status_input_complement == '1'), clean room A, increment `cost`, and print the goal state and total cost.

iv. If room A is clean, move to room A and print the goal state and total cost.

b. If room B is clean:

i. Move to room A.

ii. If room A is dirty (status_input_complement == '1'), clean room A, increment `cost`, and print the goal state and total cost.

iii. If room A is clean, print that both rooms are clean and the goal state is already reached.

9. The algorithm completes when the agent finishes its tasks and reaches the final goal state.

PROGRAM:

```
#INSTRUCTIONS

#Enter LOCATION A/B in captial letters

#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY


def vacuum_world():

    # initializing goal_state

    # 0 indicates Clean and 1 indicates Dirty

  final_goal_state = {'A': '0', 'B': '0'}

  cost = 0

  print("****WELCOME TO THE VACUUM WORLD****")

  location_input = input("Enter Location of Vacuum") #user_input of location vacuum is placed

  status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean

  status_input_complement = input("Enter status of other room")

  print("Here is MY status")

  if location_input=='A':

  # Location is A

    goal_state={'A':status_input,'B':status_input_complement}
```

```
        print("My Initial Location Condition is " + str(goal_state))
        if(status_input=='1'):
        #location is A and is dirty
            print("I find location A is dirty so the dirt need to be sucked")
            print("SUCKING THE DIRT.....PLEASE WAIT...")
            cost+=1
            print("The cost after sucking the dirt in A  is",str(cost))
            if(status_input_complement=='1'):
            #Location B is dirty
                print("I find B is dirty so the dirt need to be sucked")
                print("SUCKING THE DIRT.....PLEASE WAIT...")
                cost+=1
                print("B is clean now ....")
                print("I have completed the goal you have given me...the goal state is..",
final_goal_state )
                print("So the total cost /the total performance measure is",str(cost))
                print("THANK YOU")
            else :
            #Location B is clean
                print("Location B is clean so hence I am moving to A")
                print("I have completed the goal you have given me...the goal state is..",
final_goal_state )
                print("So the total cost /the total performance measure is",str(cost))
                print("THANK YOU")
        else :
       #Location A is clean
            print("Location A is not dirty so moving to location B")
            if(status_input_complement=='1'):
             #Location B is dirty
```

```
            print("I find B is dirty so the dirt need to be sucked")
            print("SUCKING THE DIRT.....PLEASE WAIT...")
            cost+=1
            print("B is clean now ....")
            print("I have completed the goal you have given me...the goal state is..",
final_goal_state )
            print("So the total cost /the total performance measure is",str(cost))
            print("THANK YOU")
        else :
        #Location B is clean
            print("Location B is clean so hence I am moving to A")
            print("As both the rooms were clean i had no work..the goal state is already reached..",
final_goal_state )
            print("So the total cost /the total performance measure is",str(cost))
            print("THANK YOU")
    else :
        goal_state={'A':status_input_complement,'B':status_input}
        print("Initial Location Condition" + str(goal_state))
        if(status_input=='1'):
        #location is B and is dirty
            print("I find location B is dirty so the dirt need to be sucked")
            print("SUCKING THE DIRT.....PLEASE WAIT...")
            cost+=1
            print("The cost after sucking the dirt in B is",str(cost))
            if(status_input_complement=='1'):
            #Location A is dirty
                print("I find A is dirty so the dirt need to be sucked")
                print("SUCKING THE DIRT.....PLEASE WAIT...")
                cost+=1
```

```
            print("A is clean now .... ")
            print("I have completed the goal you have given me...the goal state is..",
final_goal_state )
            print("So the total cost /the total performance measure is",str(cost))
            print("THANK YOU")
        else :
        #Location A is clean
            print("Location A is clean so hence I am moving to B")
            print("I have completed the goal you have given me...the goal state is..",
final_goal_state )
            print("So the total cost /the total performance measure is",str(cost))
            print("THANK YOU")
     else :
    #Location B is clean
        print("Location B is not dirty so moving to location A")
        if(status_input_complement=='1'):
         #Location A is dirty
            print("I find A is dirty so the dirt need to be sucked")
            print("SUCKING THE DIRT.....PLEASE WAIT .. ")
            cost+=1
            print("A is clean now .... ")
            print("I have completed the goal you have given me...the goal state is..",
final_goal_state )
            print("So the total cost /the total performance measure is",str(cost))
            print("THANK YOU")
        else :
        #Location A is clean
            print("Location A is clean so hence I am moving to B")
            print("As both the rooms were clean i had no work..the goal state is already reached..",
final_goal_state )
```

```
        print("So the total cost /the total performance measure is",str(cost))
        print("THANK YOU")

vacuum_world()
```

OUTPUT:

```
****WELCOME TO THE VACUUM WORLD****
Enter Location of Vacuum A
Enter status of A 0
Enter status of other room 1
Here is MY status
My Initial Location Condition is {'A': '0', 'B': '1'}
Location A is not dirty so moving to location B
I find B is dirty so the dirt need to be sucked
SUCKING THE DIRT.....PLEASE WAIT...
B is clean now.....
I have completed the goal you have given me...the goal state is.. {'A': '0', 'B': '0'}
So the total cost /the total performance measure is 1
THANK YOU
```

```
****WELCOME TO THE VACUUM WORLD****
Enter Location of Vacuum B
Enter status of B 1
Enter status of other room 1
Here is MY status
Initial Location Condition{'A': '1', 'B': '1'}
I find location B is dirty so the dirt need to be sucked
SUCKING THE DIRT.....PLEASE WAIT...
The cost after sucking the dirt in B is 1
I find A is dirty so the dirt need to be sucked
SUCKING THE DIRT.....PLEASE WAIT...
A is clean now.....
I have completed the goal you have given me...the goal state is.. {'A': '0', 'B': '0'}
So the total cost /the total performance measure is 2
THANK YOU
```

RESULT:

The basic search strategy for vacuum cleaner world is successfully implemented and the output is verified.

EXPERIMENT 3

Implement State Space Search Algorithms

AIM

To implement state space search algorithms like BFS and DFS

BFS(Breadth First Search)

ALGORITHM

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)

Output: Prints whether the goal node is reached and the path taken (if reached)

1. visited = empty list
2. queue = empty list
3. flag = 0
4. Append startnode to visited
5. Append startnode to queue
6. While queue is not empty:
   a. m = dequeue 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 to 1
7. If flag is 0:

a. Print "Goal is not reached"

8. End

Call BFS(visited, graph, startnode, goalnode)

PROGRAM

```
#Breadth first Search
graph = {
  'a' : ['b','c'],
  'b' : ['a', 'c','d'],
  'c' : ['a', 'b','d'],
  'd' : ['b', 'c']

}
#graph = {
#'5' : ['3','7'],
#'3' : ['2', '4'],
#'7' : ['8'],
#'2' : [],
#'4' : ['8'],
#'8' : []
#}

visited = []
queue = []
def bfs(visited, graph, startnode,goalnode):
    flag=0
    visited.append(startnode)
    queue.append(startnode)
```

```
    while queue:
        m = queue.pop(0)
        for neighbour in graph[m]:
            if flag==0:
                if neighbour not in visited:
                    visited.append(neighbour)
                    queue.append(neighbour)
                    print("visited", visited)
                    if neighbour==goalnode:
                        print("Goal  reached")
                        print("The path taken is",visited)
                        flag=1
    if flag==0:
            print("Goal is not reached")


print("Breadth-First Search")
bfs(visited, graph, 'a','e')
```

OUTPUT

Startnode=a,goalnode=e

```
Breadth-First Search
visited ['a', 'b']
visited ['a', 'b', 'c']
visited ['a', 'b', 'c', 'd']
Goal is not reached
```

Startnode=a,goalnode=d

```
Breadth-First Search
visited ['a', 'b']
visited ['a', 'b', 'c']
visited ['a', 'b', 'c', 'd']
Goal reached
The path taken is ['a', 'b', 'c', 'd']
```

DFS(Depth First Search)

ALGORITHM

Algorithm DFS (Depth-First Search):

Input: graph (an adjacency list representing the graph),

startnode (the starting node for the search),

goalnode (the goal node to be reached)

Output: Prints whether the goal node is reached and the path taken (if reached)

1. visited = empty list
2. Define a function DFS(visited, graph, currentnode, goalnode):
   a. flag = 0
   b. If currentnode is not in visited:
      - Append currentnode to visited
      - If currentnode is equal to goalnode:
        * Set flag to 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)
3. Call DFS(visited, graph, startnode, goalnode)
4. If flag is 0:
   a. Print "Goal has not reached"
5. End

Call DFS(visited, graph, startnode, goalnode)

PROGRAM:

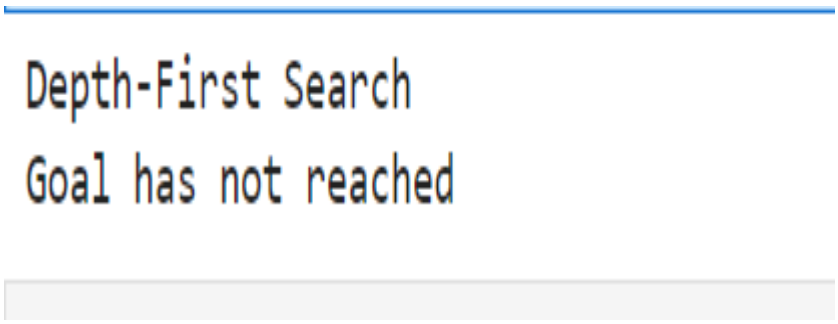
```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
```

```
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
 visited = []
 def dfs(visited, graph, startnode,goalnode):
    flag=0
    if startnode not in visited:
        visited.append(startnode)
        if startnode==goalnode:
            flag=1
            print("Goal is reached")
            print("The path taken to reach the goal is",visited)
        elif flag==0 :
             for neighbour in graph[startnode]:
                dfs(visited, graph, neighbour,goalnode)
    elif flag==0:
        print("Goal has not reached")
print("Depth-First Search")
dfs(visited, graph, '5','7')
```

OUTPUT

```
Depth-First Search
Goal is reached
The path taken to reach the goal is ['5', '3', '2', '4', '8', '7']
```

```
dfs(visited, graph, '5','10')
```



RESULT:

The state space search algorithms Breadth First Search(BFS) and Depth First Search(DFS) is implemented and the output is verified

EXPERIMENT 4

Implement Informed Search Algorithm

AIM

To implement informed search algorithms like BFS and A* algorithms

BFS(Best First Search)

ALGORITHM

1. Initialization:
   - Initialize `SuccList` with the given adjacency information.
   - 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:
   - Takes a node `N` as input.
   - If `N` exists in `SuccList`, return its corresponding list of child nodes.
   - Return an empty list if `N` does not exist.

3. Define GOALTEST(N) Function:
   - Takes a node `N` as input.
   - If `N` equals the `Goal`, return `True`, indicating the goal is reached.
   - Otherwise, return `False`.

4. Define APPEND(L1, L2) Function:
   - Takes two lists `L1` and `L2` as input.
   - Returns a new list containing all elements of `L1` followed by all elements of `L2`.

5. Define SORT(L) Function:

- Takes a list `L` as input.

- Sorts the list `L` based on the second element (cost) of each inner list using the `lambda` function.

- Returns the sorted list.

6. Define BestFirstSearch() Function:

- Initializes the `Frontier` list with the starting node and its cost.

- Initializes the `EXPLORED` list.

- Enters a loop that continues until either the `Frontier` list is empty or the `State` becomes `SUCCESS`.

  - Pops the first node `N` from the `Frontier`.

  - If `GOALTEST(N[0])` is `True`, the goal is reached, and the `State` becomes `SUCCESS`.

    - The current node `N` is added to `EXPLORED`.

  - If the goal is not reached:

    - The current node `N` is added to `EXPLORED`.

    - Generates child nodes of `N` using the `GENCHILD` function.

    - Removes nodes from `CHILD` that are already in `EXPLORED`.

    - Removes nodes from `CHILD` that are already in `Frontier`.

    - Appends the child nodes to the `Frontier`.

    - Sorts the `Frontier` using the `SORT` function.

7. Calling the Algorithm:

- Calls the `BestFirstSearch()` function to execute the search algorithm.

- The `Explored` list and the `result` (SUCCESS or FAILURE) are printed.

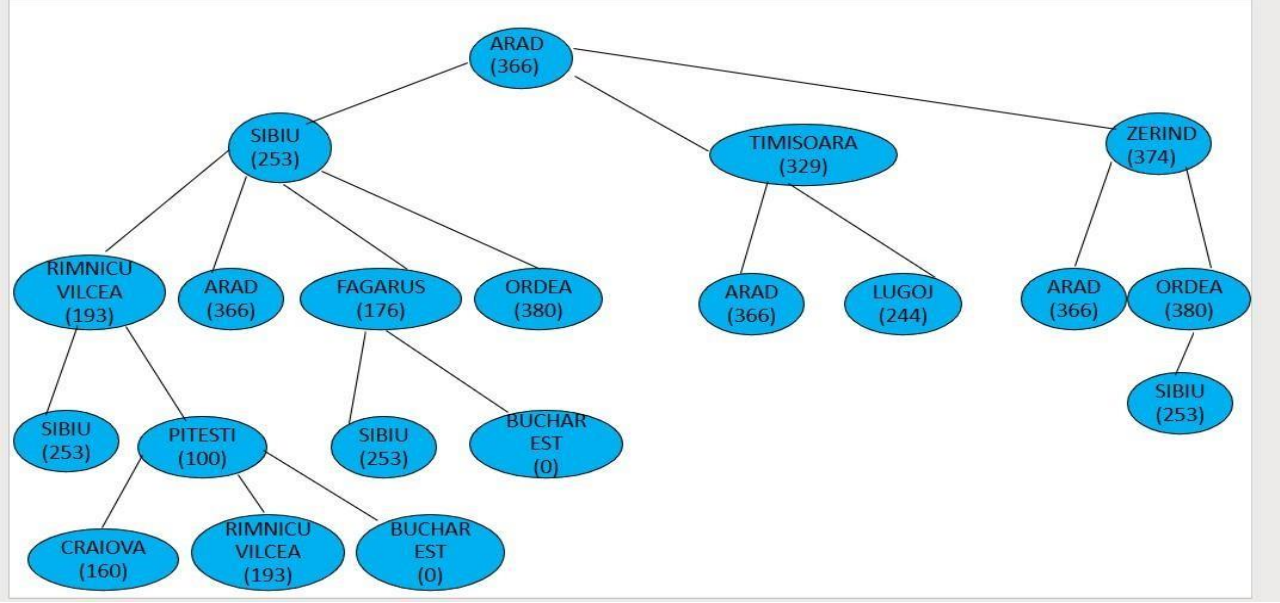


Fig. Heuristic value mentioned from each city to Bucharest

PROGRAM

SuccList={ 'Arad': [['Sibiu',253],['Timisoara',329],['Zerind',374]], 'Sibiu': [['Rimnicu Vilcea',193],['Arad',366],['Fagarus',176],['Oradea',380]], 'Rimnicu Vilcea': [['Sibiu',253],['Pitesti',100]], 'Pitesti': [['Craiova',160],['Rimicu Vilcea',193],['Bucharest',0]], 'Fagarus': [['Sibiu',253],['Bucharest',0]], 'Timisoara': [['Arad',366],['Lugoj',274]], 'Zerind': [['Arad',366],['Ordea',380]], 'Oradea': [['Sibiu',253]]}

Start='Arad'

#Start='Oradea'

#Start='Rimnicu Vilcea'

Goal='Bucharest'

Explored = list()

SUCCESS=True

FAILURE=False

State=FAILURE

def GENCHILD(N):

New_list=list()

if N in SuccList.keys():

```
New_list=SuccList[N]

return New_list

def GOALTEST(N):
if N == Goal:
return True
else:
return False

def APPEND(L1,L2):
New_list=list(L1)+list(L2)
return New_list

def SORT(L):
L.sort(key = lambda x: x[1])
return L

def BestFirstSearch():
Frontier=[[Start,366]]
EXPLORED=list()
global State
global Explored
while (len(Frontier) != 0) and (State != SUCCESS):
print("----------")
N= Frontier[0]
print("N=",N)
del Frontier[0] #delete the node we picked
```

```
if GOALTEST(N[0])==True:
State = SUCCESS
EXPLORED = APPEND(EXPLORED,[N])
print("EXPLORED=",EXPLORED)
else:
EXPLORED = APPEND(EXPLORED,[N])
print("EXPLORED=",EXPLORED)
CHILD = GENCHILD(N[0])
print("CHILD=",CHILD)
for val in EXPLORED:
if val in CHILD:
CHILD.remove(val)
for val in Frontier:
if val in CHILD:
CHILD.remove(val)
Frontier = APPEND(CHILD,Frontier) #append CHILD elements to
                                  FRONTIER
print("Unsorted Frontier=",Frontier)
SORT(Frontier)
print("Sorted Frontier=",Frontier)

Explored=EXPLORED
return State

result=BestFirstSearch() #call search algorithm
print(Explored,result)
```

OUTPUT

Start=Arad, goal=Bucharest

```
------------
N= ['Arad', 366]
EXPLORED= [['Arad', 366]]
CHILD= [['Sibiu', 253], ['Timisoara', 329], ['Zerind', 374]]
Unsorted Frontier= [['Sibiu', 253], ['Timisoara', 329], ['Zerind', 374]]
Sorted Frontier= [['Sibiu', 253], ['Timisoara', 329], ['Zerind', 374]]
------------
N= ['Sibiu', 253]
EXPLORED= [['Arad', 366], ['Sibiu', 253]]
CHILD= [['Rimnicu Vilcea', 193], ['Arad', 366], ['Fagarus', 176], ['Oradea', 380]]
Unsorted Frontier= [['Rimnicu Vilcea', 193], ['Fagarus', 176], ['Oradea', 380], ['Timisoara', 329], ['Zerind', 374]]
Sorted Frontier= [['Fagarus', 176], ['Rimnicu Vilcea', 193], ['Timisoara', 329], ['Zerind', 374], ['Oradea', 380]]
------------
N= ['Fagarus', 176]
EXPLORED= [['Arad', 366], ['Sibiu', 253], ['Fagarus', 176]]
CHILD= [['Sibiu', 253], ['Bucharest', 0]]
Unsorted Frontier= [['Bucharest', 0], ['Rimnicu Vilcea', 193], ['Timisoara', 329], ['Zerind', 374], ['Oradea', 380]]
Sorted Frontier= [['Bucharest', 0], ['Rimnicu Vilcea', 193], ['Timisoara', 329], ['Zerind', 374], ['Oradea', 380]]
------------
N= ['Bucharest', 0]
EXPLORED= [['Arad', 366], ['Sibiu', 253], ['Fagarus', 176], ['Bucharest', 0]]
[['Arad', 366], ['Sibiu', 253], ['Fagarus', 176], ['Bucharest', 0]] True
```

Start= Rimnicu Vilcea, goal=Bucharest

```
------------
N= ['Rimnicu Vilcea', 366]
EXPLORED= [['Rimnicu Vilcea', 366]]
CHILD= [['Sibiu', 253], ['Pitesti', 100]]
Unsorted Frontier= [['Sibiu', 253], ['Pitesti', 100]]
Sorted Frontier= [['Pitesti', 100], ['Sibiu', 253]]
------------
N= ['Pitesti', 100]
EXPLORED= [['Rimnicu Vilcea', 366], ['Pitesti', 100]]
CHILD= [['Craiova', 160], ['Rimicu Vilcea', 193], ['Bucharest', 0]]
Unsorted Frontier= [['Craiova', 160], ['Rimicu Vilcea', 193], ['Bucharest', 0], ['Sibiu', 253]]
Sorted Frontier= [['Bucharest', 0], ['Craiova', 160], ['Rimicu Vilcea', 193], ['Sibiu', 253]]
------------
N= ['Bucharest', 0]
EXPLORED= [['Rimnicu Vilcea', 366], ['Pitesti', 100], ['Bucharest', 0]]
[['Rimnicu Vilcea', 366], ['Pitesti', 100], ['Bucharest', 0]] True
```

A* ALGORITHM (8 PUZZLE PROBLEM)

ALGORITHM

1. Import the necessary libraries:

   - Import `numpy` as `np` to work with arrays.

   - Import `deepcopy` from the `copy` module for creating deep copies of objects.

2. Define the `star_loop` function:

   - Takes the current puzzle state `step`, indices of the blank tile `idx1` and `idx2`, and a list of closed positions `closed`.

   - Initialize `val`, `swapped`, and `ind_list` lists to store values, possible next states, and swapped tile indices.

   - Loop over neighboring positions around the blank tile:

     - Check if the position is valid and not in the `closed` list.

     - If valid, and not diagonal, create a deep copy of the current state (`step2`).

     - Swap the blank tile with the neighboring tile in the deep copy.

     - Append the modified state to `swapped` and the swapped tile indices to `ind_list`.

   - Return the `swapped` states and their corresponding indices.

3. Define the initial and final puzzle configurations using NumPy arrays.

4. Determine the initial position of the blank tile using the `np.where` function.

5. Initialize variables:

   - Create a deep copy of the `initial` state as `step`.

   - Set `g` to 1 to represent the cost to reach the current state.

   - Set `closed` to the initial position of the blank tile.

6. Start a loop that iterates while `g` is less than 10:

   - Print the current indices of the blank tile (`idx1`, `idx2`).

7. Call the `star_loop` function to generate possible next states and their swapped tile indices.

8. Update the `closed` list with the current position of the blank tile.

9. Create an empty list `h_list` to store the total estimated costs for each possible next state.

10. Iterate over the possible next states using index `k`:

- Calculate the heuristic value `h` by counting the number of tiles out of place in the current state compared to the `final` state.

- Append the total estimated cost `g + h` to the `h_list`.

11. Print the list of total estimated costs for each possible next state.

12. If there are possible states in the `h_list`:

- Find the index of the state with the lowest total cost (`g + h`) using the `min` function.

- Update the current state `step` to the state with the lowest cost.

- Update the indices of the blank tile (`idx1`, `idx2`) using the corresponding indices from the `ind_list`.

13. Print the updated current state and the closed positions.

14. Check if the current state matches the `final` state. If true, break the loop as the puzzle is solved.

15. Increment the cost value `g` by 1 for the next iteration.

PROGRAM:

```
import numpy as np

from copy import deepcopy


def star_loop(step, idx1, idx2, closed):


    val = []

    swapped = []

    ind_list = []
```

```
    for i in  range(idx1-1, idx1+2):
        for j in range(idx2-1, idx2+ 2):
            if [i,j] != closed and i>=0 and j >=0 and i<=2 and j<=2:
                if [i,j] !=[idx1,idx2] and abs(i-idx1) != abs(j-idx2):
                    #print("i",i,"j",j,"idx1",idx1, "idx2",idx2)
                    step2 = deepcopy(step)
                    step2[idx1,idx2], step2[i,j] = step2[i,j], step2[idx1,idx2]
                    swapped.append(step2)
                    ind_list.append([i,j])
    return swapped, ind_list


initial = np.array([[2,8,3],
                    [1,6,4],
                    [7,0,5]])
step = deepcopy(initial)


final = np.array([[1,2,3],
                  [8,0,4],
                  [7,6,5]])

init = np.where(initial==0)
idx1=int(init[0])
idx2 =int(init[1])
g=1
closed = [idx1, idx2]
while g<10:
```

```
    print(idx1, idx2)


    swapped, indlist = star_loop(step, idx1, idx2, closed)
    closed = [idx1, idx2]
    h_list = []
    for k in range(len(swapped)):
        h = len(np.where(swapped[k]!=final)[0])
        h_list.append(g+h)
    print(h_list)
    if h_list:
        step = swapped[h_list.index(min(h_list))]
        idx1 = indlist[h_list.index(min(h_list))][0]
        idx2 = indlist[h_list.index(min(h_list))][1]
    print("step \n ",step, "\n closed :", closed)
    if (step == final).all():
        break
g +=1
```

OUTPUT

```
2 1
[4, 7, 7]
step
  [[2 8 3]
 [1 0 4]
 [7 6 5]]
 closed : [2, 1]
1 1
[6, 6, 7]
step
  [[2 0 3]
 [1 8 4]
 [7 6 5]]
 closed : [1, 1]
0 1
[6, 8]
step
  [[0 2 3]
 [1 8 4]
 [7 6 5]]
 closed : [0, 1]
    0 0
    [6]
    step
      [[1 2 3]
     [0 8 4]
     [7 6 5]]
     closed : [0, 0]
    1 0
    [5, 8]
    step
      [[1 2 3]
     [8 0 4]
     [7 6 5]]
     closed : [1, 0]
```

RESULT:

The informed search algorithms Best First Search and A* is implemented and the output is verified

EXPERIMENT 5

Implement Backtracking algorithms for CSP

AIM:

To implement backtracking algorithm in map coloring problem

ALGORITHM

1. Initialize Data:

- Define the `domain_colors` as a list of available colors.

- Define a list of `variable_states` representing the states that need to be colored.

- Create a `neighbors` dictionary to specify the neighboring relationships between states.

- Create an empty dictionary called `finalstateswithcolor` to store the assigned colors for each state.

2. Color Assignment Functions:

- Define two functions:

- `getthecolor(state)`: This function finds and returns a suitable color for a given state. It iterates through the available colors and calls the `assigncolor` function to check if each color is valid for the state.

- `assigncolor(state, color)`: This function checks if it's valid to assign a specific color to a state. It ensures that none of the neighboring states has the same color.

3. Main Function:

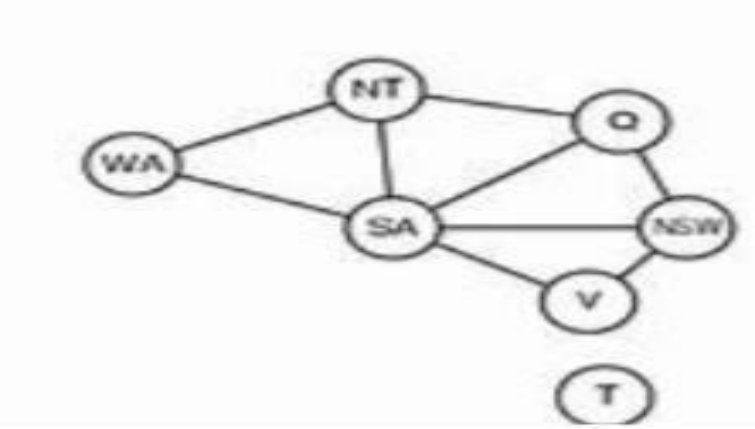
- In the `main` function:

- Use the degree heuristic to sort the states by the number of neighbors they have in descending order. This heuristic prioritizes states with more neighbors to be colored first.

- Iterate through the sorted list of states.

- For each state, call the `getthecolor` function to find a suitable color for the state, and store the assigned color in the `finalstateswithcolor` dictionary.

- Print the dictionary `finalstateswithcolor` to display the states with their assigned colors while ensuring that neighboring states have different colors.



4. .PROGRAM

```
domain_colors = ['Red', 'Blue', 'Green']

#wa-west australia,nt-northern territory,sa-south australia,q-queensland,nsw-new south wales,v-victoria,t-tansmania

variable_states = ['wa', 'nt', 'sa', 'q', 'nsw', 'v','t']

 neighbors = {}

neighbors['wa'] = ['nt', 'sa']

neighbors['nt'] = ['wa', 'sa', 'q']

neighbors['sa'] = ['wa', 'nt', 'q', 'nsw', 'v']

neighbors['q'] = ['nt', 'sa', 'snw']

neighbors['nsw'] = ['q', 'sa', 'v']

neighbors['v'] = ['sa', 'nsw']

neighbors['t'] = []


finalstateswithcolor = {}


def getthecolor(state):
    for color in colors:
        if assigncolor(state, color):
            return color


def assigncolor(state, color):
```

```
    for neighbor in neighbors.get(state):
        color_of_neighbor = finalstateswithcolor.get(neighbor)
        if color_of_neighbor == color:
            return False
     return True
 def main():
   #use degree heuristics to find the state with largest number first
    sorted_states = sorted(neighbors.keys(), key=lambda state: len(neighbors[state]),
reverse=True)


    for state in sorted_states:
        finalstateswithcolor[state] = getthecolor(state)
     print("The states with colors are",finalstateswithcolor)
 main()
```

OUTPUT

```
The states with colors are {'sa': 'Red', 'nt': 'Blue', 'q': 'Green', 'nsw': 'Blue', 'wa': 'Green', 'v': 'Green', 't': 'Red'}
```

RESULT:

The backtracking algorithm in map coloring problem is implemented and the output is verified

EXPERIMENT 6

Implement Local Search Algorithm

AIM:

To implement local search algorithm in map colouring

ALGORITHM

1. Create a recursive function that takes the current index, number of vertices and output color array
2. If the current index is equal to number of vertices. Check if the output color configuration is safe, i.e check if the adjacent vertices do not have same color. If the conditions are met, print the configuration and break
3. Assign a color to a vertex (1 to m)
4. For every assigned color recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and returns true.

PROGRAM

```
def isSafe(graph, color):

    # check for every edge
    for i in range(4):
        for j in range(i + 1, 4):
            if (graph[i][j] and color[j] == color[i]):
                return False
    return True

# /* This function solves the m Coloring
# problem using recursion. It returns
# false if the m colours cannot be assigned,
# otherwise, return true and prints
# assignments of colours to all vertices.
# Please note that there may be more than
# one solutions, this function prints one
# of the feasible solutions.*/


def graphColoring(graph, m, i, color):

    # if current index reached end
    if (i == 4):

        # if coloring is safe
        if (isSafe(graph, color)):
```

```
            # Print the solution
            printSolution(color)
            return True
        return False

    # Assign each color from 1 to m
    for j in range(1, m + 1):
        color[i] = j

        # Recur of the rest vertices
        if (graphColoring(graph, m, i + 1, color)):
            return True
        color[i] = 0
    return False

# /* A utility function to print solution */


def printSolution(color):
    print("Solution Exists:" " Following are the assigned colors ")
    for i in range(4):
        print(color[i], end=" ")


# Driver code
if __name__ == '__main__':

    # /* Create following graph and
    # test whether it is 3 colorable
    # (3)---(2)
    # | / |
    # | / |
    # | / |
    # (0)---(1)
    # */
    graph = [
        [0, 1, 1, 1],
        [1, 0, 1, 0],
        [1, 1, 0, 1],
        [1, 0, 1, 0],
    ]
    m = 3  # Number of colors

    # Initialize all color values as 0.
    # This initialization is needed
    # correct functioning of isSafe()
    color = [0 for i in range(4)]
```

```
# Function call
if (not graphColoring(graph, m, 0, color)):
    print("Solution does not exist")
```

OUTPUT

Solution Exists: Following are the assigned colors

>

1 2 3 2 >

RESULT:

The local search algorithm for map colouring is done and the output is verified.

EXPERIMENT-7

PROPOSITIONAL LOGIC INFERENCES FOR AI TASKS

AIM:

To implement propositional logic inferences for AI tasks

ALGORITHM:

1. Print the header for the prepositional logic inferences.

2. Define the `negation` function that takes a proposition `p` and returns the negation of `p`.

3. Print the header for the "NEGATION" section.

4. Print the column headers for the truth table of the `negation` function.

5. Iterate over each value of `p` (True, False):

   - Calculate the result of negation using the `negation` function.

   - Print the values of `p` and the calculated result.

6. Define the `conjunction` function that takes propositions `p` and `q` and returns the conjunction (AND) of `p` and `q`.

7. Print the header for the "CONJUNCTION(AND OPERATION)" section.

8. Print the column headers for the truth table of the `conjunction` function.

9. Iterate over each value of `p` (True, False):

   - Iterate over each value of `q` (True, False):

     - Calculate the result of conjunction using the `conjunction` function.

     - Print the values of `p`, `q`, and the calculated result.

10. Define the `disjunction` function that takes propositions `p` and `q` and returns the disjunction (OR) of `p` and `q`.

11. Print the header for the "DISJUNCTION(OR OPERATION)" section.

12. Print the column headers for the truth table of the `disjunction` function.

13. Iterate over each value of `p` (True, False):

    - Iterate over each value of `q` (True, False):

      - Calculate the result of disjunction using the `disjunction` function.

      - Print the values of `p`, `q`, and the calculated result.

14. Define the `exclusive_disjunction` function that takes propositions `p` and `q` and returns the exclusive disjunction (XOR) of `p` and `q`.

15. Print the header for the "EXCLUSIVE DISJUNCTION(XOR OPERATION)" section.

16. Print the column headers for the truth table of the `exclusive_disjunction` function.

17. Iterate over each value of `p` (True, False):

- Iterate over each value of `q` (True, False):

  - Calculate the result of exclusive disjunction using the `exclusive_disjunction` function.

  - Print the values of `p`, `q`, and the calculated result.

18. Define the `implication` function that takes propositions `p` and `q` and returns the implication of `p` implies `q`.

19. Print the header for the "IMPLICATION" section.

20. Print the column headers for the truth table of the `implication` function.

21. Iterate over each value of `p` (True, False):

- Iterate over each value of `q` (True, False):

  - Calculate the result of implication using the `implication` function.

  - Print the values of `p`, `q`, and the calculated result.

PROGRAM:

```
print("******PREPOSITIONAL LOGIC INFERENCES FOR AI TASKS********")
def negation(p):
    return not p
print("\u0332".join("NEGATION"))
print("p    result")
for p in [True, False]:
    a = negation(p)
    print(p, a)


def conjunction(p, q):
    return p and q
print("\u0332".join("CONJUCTION(AND OPERATION)"))
print("p    q    result")
```

```
for p in [True, False]:
    for q in [True, False]:
        a = conjunction(p, q)
        print(p, q, a)

def disjunction(p, q):
    return p or q
print("\u0332".join("DISJUNCTION(OR OPERATION)"))
print("p     q     result")
for p in [True, False]:
    for q in [True, False]:
        a = disjunction(p, q)
        print(p, q, a)

def exclusive_disjunction(p,  q):
    return (p and not q) or (not p and q)
print("\u0332".join("EXCLUSIVE DISJUNCTION(XOR OPERATION)"))
print("p     q     result")
for p in [True, False]:
    for q in [True, False]:
        a = exclusive_disjunction(p, q)
        print(p, q, a)

def implication(p, q):
    return #FIX ME#
print("\u0332".join("IMPLICATION"))
print("p     q     result")
for p in [True, False]:
    for q in [True, False]:
```

```
a = not(p) or q
print(p, q, a)
```

OUTPUT:

```
******PREPOSITIONAL LOGIC INFERENCES FOR AI TASKS********
NEGATION
p    result
True False
False True
CONJUCTION(AND OPERATION)
p    q    result
True True True
True False False
False True False
False False False
DISJUNCTION(OR OPERATION)
p    q    result
True True True
True False True
False True True
False False False
```

```
EXCLUSIVE DISJUNCTION(XOR OPERATION)
p    q    result
True True False
True False True
False True True
False False False
IMPLICATION
p    q    RESULT
True True True
True False False
False True True
False False True
```

RESULT:

The propotional logic inferences for AI tasks are successfully implemented and the output is verified.

EXPERIMENT 8

Implement Knowledge Representation Schemes

AIM:

To implement Knowledge representation schemes in expert medical diagnosis system.

ALGORITHM

1. Define Symptom-Checking Functions:

- Define four functions: `measles(a, b, c, d, e)`, `flu(a, b, c, d, e, f, g, h)`, `cold(c, j, k, d, h)`, and `chickenpox(a, h, g, b)`.

- Each function takes as input a set of symptoms as 'y' (yes) or 'n' (no) and checks for specific combinations of symptoms.

- If a combination of symptoms matches a known illness, the function returns the name of the illness (e.g., "measles," "flu"). If not, it 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 and a series of yes/no responses for various symptoms.

- Store the user's responses in variables (e.g., a, b, c, etc.), ensuring all responses are in lowercase.

3. Diagnosis Initialization:

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

4. Diagnosis for Each Illness:

- Use each symptom-checking function to determine if the user's symptoms match any known illnesses.

- If a function returns a diagnosis (i.e., not `None`), add the diagnosis to the `diagnosis` list.

5. Final Diagnosis:

- If there are any diagnoses in the `diagnosis` list, print a message to the user, including their name and the list of possible diagnoses.

- If no diagnoses were found (the 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.

4..PROGRAM

```
def measles(a, b, c, d, e):
    if a == 'y' and b == 'y' and c == 'y' and d == 'y' and e == 'y':
        return "measles"
    else:
        return None


def flu(a, b, c, d, e, f, g, h):
    if a == 'y' and b == 'y' and c == 'y' and d == 'y' and e == 'y' and f == 'y' and g == 'y' and h == 'y':
        return "flu"
    else:
        return None


def cold(c, j, k, d, h):
    if c == 'y' and j == 'y' and k == 'y' and d == 'y' and h == 'y':
        return "cold"
    else:
        return None


def chickenpox(a, h, g, b):
    if a == 'y' and h == 'y' and g == 'y' and b == 'y':
        return "chickenpox"
    else:
        return None
```

```
def run_diagnosis():
    name = input("Please enter your name: ")
    a = input("Do you have fever? (y/n): ").lower()
    b = input("Do you have rashes? (y/n): ").lower()
    c = input("Do you have headache? (y/n): ").lower()
    d = input("Do you have running nose? (y/n): ").lower()
    e = input("Do you have conjunctivitis? (y/n): ").lower()
    f = input("Do you have cough? (y/n): ").lower()
    g = input("Do you have ache? (y/n): ").lower()
    h = input("Do you have chills? (y/n): ").lower()
    i = input("Do you have swollen glands? (y/n): ").lower()
    j = input("Do you have sneezing? (y/n): ").lower()
    k = input("Do you have sore throat? (y/n): ").lower()

    diagnosis = []

    result = measles(a, f, e, d, b)
    if result:
        diagnosis.append(result)

    result = flu(a, c, g, e, h, k, f, d)
    if result:
        diagnosis.append(result)

    result = cold(c, j, k, d, h)
    if result:
        diagnosis.append(result)
```

```
    result = chickenpox(a, h, g, b)
    if result:
        diagnosis.append(result)

    if len(diagnosis) > 0:
        print(f"{name}, based on the symptoms provided, you may have: {', '.join(diagnosis)}.")
    else:
        print("No diagnosis could be made based on the given symptoms.")

run_diagnosis()
```

OUTPUT

```
Please enter your name:  Nikhil
Do you have fever? (y/n):  y
Do you have rashes? (y/n):  y
Do you have headache? (y/n):  n
Do you have running nose? (y/n):  y
Do you have conjunctivitis? (y/n):  n
Do you have cough? (y/n):  y
Do you have ache? (y/n):  y
Do you have chills? (y/n):  y
Do you have swollen glands? (y/n):  n
Do you have sneezing? (y/n):  y
Do you have sore throat? (y/n):  y
Nikhil, based on the symptoms provided, you may have: chickenpox.
```

RESULT:

Knowledge representation schemes in expert medical diagnosis system is implemented and the output is verified.

EXPERIMENT 9

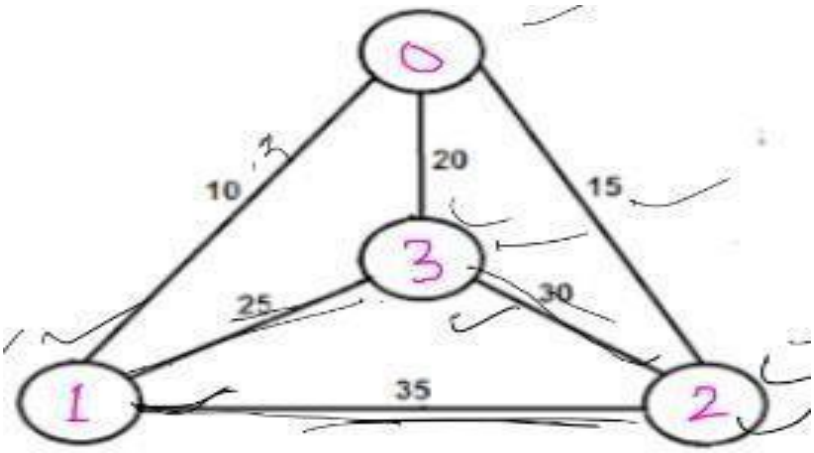
Implementing Travelling Salesman Problem

AIM:

To implement Travelling Salesman Problem

ALGORITHM

1. Import necessary libraries:
   - Import `maxsize` from `sys` to use as an initial value for the minimum path.
   - Import `permutations` from `itertools` to generate all possible permutations of the vertices.
2. Initialize graph and variables:
   - Define the number of vertices (`V`) and the graph representing the distances between them.
   - Set the starting vertex (`s`) to 0.
3. Define the traveling salesman function:
   - Create a list of vertices excluding the starting vertex (`s`).
   - Initialize variables for the minimum path (`min_path`) and the best tour (`best_tour`).
   - Generate all permutations of the remaining vertices.
4. Iterate through permutations:
   - For each permutation, calculate the total weight of the path.
   - Update the minimum path and the corresponding tour if the current path is shorter.
5. Return the result:
   - Return the minimum path and the best tour.
6. Invoke the function and print the result:
   - Call the `travellingSalesmanProblem` function with the given graph and starting vertex.
   - Print the minimum path and the best tour.



PROGRAM

```
from sys import maxsize
from itertools import permutations
V = 4
graph = [[0, 10, 15, 20], [10, 0, 35, 25],
         [15, 35, 0, 30], [20, 25, 30, 0]]
s = 0
def travellingSalesmanProblem(graph, s):
    vertex = []
    for i in range(V):

        if i != s:
            #print (i)
            vertex.append(i)
            #print(vertex)

    min_path = maxsize
    best_tour= None
```

```
    next_permutation=permutations(vertex)

    for i in next_permutation:
        #print(i)

        current_pathweight = 0
        k = s

        for j in i:

            current_pathweight += graph[k][j]
            #print(current_pathweight)
            k = j

        current_pathweight += graph[k][s]
        #print("weight=",current_pathweight)
        if current_pathweight < min_path:
            min_path = min(min_path, current_pathweight)

            best_tour = [s] + list(i) + [s]

    return min_path,best_tour
min_path,best_tour=(travellingSalesmanProblem(graph, s))
print('min_path=',min_path)
print('best_tour=',best_tour)
```

OUTPUT

```
min_path= 80
best_tour= [0, 1, 3, 2, 0]
```

RESULT:

Travelling Salesman problem is implemented and the output is verified.

EXPERIMENT 10

Implement Backtracking algorithms for CSP

AIM:

To construct a pruned game tree using Alpha-Beta pruning. Take the sequence, [10,5,7,11,12,8,9,8,5,12,11,12,9,8,7,10] of MINIMAX values for the nodes at the cutoff depth of 4 plies. Assume that branching factor is 2, MIN makes the first move, and nodes are generated from left to right.

ALGORITHM

1. Define two constants, `MAX` and `MIN`, to represent the maximum and minimum values for initialization.

2. Create the `minimax` function, which takes the following parameters:

- `depth`: The current depth in the search tree.

- `nodeIndex`: The index of the current node.

- `maximizingPlayer`: A boolean indicating whether the current player is maximizing (True) or minimizing (False).

- `values`: A list of values representing the nodes in the tree.

- `alpha`: The best value found for the maximizing player.

- `beta`: The best value found for the minimizing player.

3. The function uses a recursive approach to explore the search tree. The termination condition is when the maximum depth (`depth == 4`) is reached. In this case, the function returns the value of the current node.

4. If it's a maximizing player's turn, the function initializes `best` as the minimum value (`MIN`). It then iterates through the left and right children of the current node, calling `minimax` recursively for each child. It updates `best` with the maximum value found and also updates `alpha` with the maximum value. Alpha-beta pruning is applied by checking if `beta` is less than or equal to `alpha`. If this condition is met, the loop breaks early. The best value for the maximizing player is returned.

5. If it's a minimizing player's turn, the function initializes `best` as the maximum value (`MAX`). It follows a similar process to the maximizing player, but this time it seeks the minimum value. It updates `best` and `beta` accordingly and applies alpha-beta pruning if `beta` is less than or equal to `alpha`.

6. The driver code initializes the `values` list, representing the values in the search tree, and calls the `minimax` function with the initial parameters. The optimal value is printed as the result.

PROGRAM

```
# Define constants for maximum and minimum values
MAX, MIN = 1000, -1000


# Minimax function for finding the optimal value in a search tree
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    # Terminating condition: If the maximum depth is reached, return the value of the current node
    if depth == 4:
        return values[nodeIndex]


    if maximizingPlayer:
        best = MIN
        # For maximizing player, initialize best as the minimum value
        for i in range(0, 2):
            # Recur for left and right children
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)  # Update best with the maximum value
            alpha = max(alpha, best)  # Update alpha with the maximum value
            if beta <= alpha:
                break  # Alpha-Beta Pruning: If beta is less than or equal to alpha, break the loop
        print("best value of max player-->",best)
        return best # Return the best value found so far for the maximizing player
    else:
        best = MAX
        # For minimizing player, initialize best as the maximum value
        for i in range(0, 2):
            # Recur for left and right children
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
```

```
            best = min(best, val) # Update best with the minimum value
            beta = min(beta, best) # Update beta with the minimum value
            if beta <= alpha:
                break  # Alpha-Beta Pruning: If beta is less than or equal to alpha, break the loop
        print("best value of min player-->",best)
        return best  # Return the best value found so far for the minimizing player


# Driver code
if__name___== "_main_":
    #values = [3, 5, 6, 9, 1, 2, 0, -1]
    values= [10,5,7,11,12,8,9,8,5,12,11,12,9,8,7,10]
    # Call the minimax function with initial parameters and print the result
    print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

OUTPUT

```
best value of min player--> 5
best value of min player--> 7
best value of max player--> 7
best value of min player--> 8
best value of max player--> 8
best value of min player--> 7
best value of min player--> 5
best value of min player--> 11
best value of max player--> 11
best value of min player--> 8
best value of min player--> 7
best value of max player--> 8
best value of min player--> 8
best value of max player--> 8
The optimal value is: 8
```

RESULT:

The backtracking algorithm MINIMAX in MIN MAX playing game using Alpha beta pruning is implemented and the output is verified.

CONTENT BEYOND

Integrate Knowledge Base With Supervised Learning

AIM:

Program that integrates a knowledge base with supervised learning involves combining symbolic knowledge representation with machine learning technique.

ALGORITHM

1) Create an instance of the KnowledgeBase class

1.1) Add rules to the knowledge base using the add_rule method

2) Generate training data

2.1) Create lists of features and corresponding labels

3) Train a supervised learning model

3.1) Train the model using the train_supervised_model function with the generated training data

4) Prompt user for input for a new data point

4.1) Print a message asking the user to enter features separated by commas

4.2) Accept user input as a string

5) Process user input

5.1) Split the user input string into a list of features, trimming whitespace

6) Classify using the knowledge base

6.1) Use the classify method of the knowledge base for classification

6.2) Print the result of knowledge base classification

7) Classify using the supervised learning model

7.1) Use the predict method of the trained model for classification

7.2) Print the result of supervised learning model classification

8)Print the results of knowledge base and supervised learning model

Classification

PROGRAM

```
def main():
    # Create a knowledge base
    knowledge_base = KnowledgeBase()
    knowledge_base.add_rule(["Color: Red", "Shape: Round"], "Apple")
    knowledge_base.add_rule(["Color: Yellow", "Shape: Oblong"], "Banana")

    # Generate some training data
    features = [["Color: Red", "Shape: Round"], ["Color: Yellow", "Shape:
    Oblong"]]
    labels = ["Apple", "Banana"]

    # Train a supervised learning model
    model = train_supervised_model(features, labels)

    # Prompt user for input for new data
    print("Enter features for a new data point:")
    new_data_input = input("Separate features with commas (e.g., Color: Red,
      Shape: Round): ")

    # Process user input into a list of features
    new_data_features = [feature.strip() for feature in new_data_input.split(',')]

    # Use the knowledge base for classification
    kb_classification = knowledge_base.classify(new_data_features)
```

```
    # Use the supervised learning model for classification
    model_classification = model.predict([new_data_features])[0]

    print("\nKnowledge Base Classification:", kb_classification)
    print("Supervised Learning Model Classification:", model_classification)

if __name__ == "__main__":
    main()
```

OUTPUT

```
Enter features for a new data point:
Separate features with commas (e.g., Color: Red, Shape: Round): Color: Yellow, Shape: Oblong
Knowledge Base Classification: Banana
Supervised Learning Model Classification: Banana
```