**CHAROTAR UNIVERSITY OF SCIENCE &**

**TECHNOLOGYDEVANG PATEL INSTITUTE OF ADVANCE TECHNOLOGY & RESEARCH**

**Computer Science & Engineering**

**NAME: PARTH NITESHKUMAR PATEL**

**ID: 19DCS098**

**SUBJECT: ARTIFICIAL INTELLIGENCE**

**CODE: CS 341**

**PRACTICAL-1**

Write Programs to demonstrate knowledge of Prolog Basics.

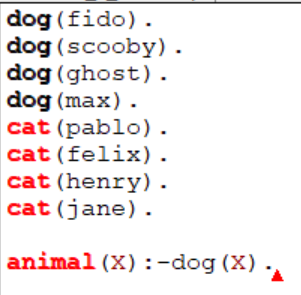
**PRACTICAL-1.1**

**AIM:**

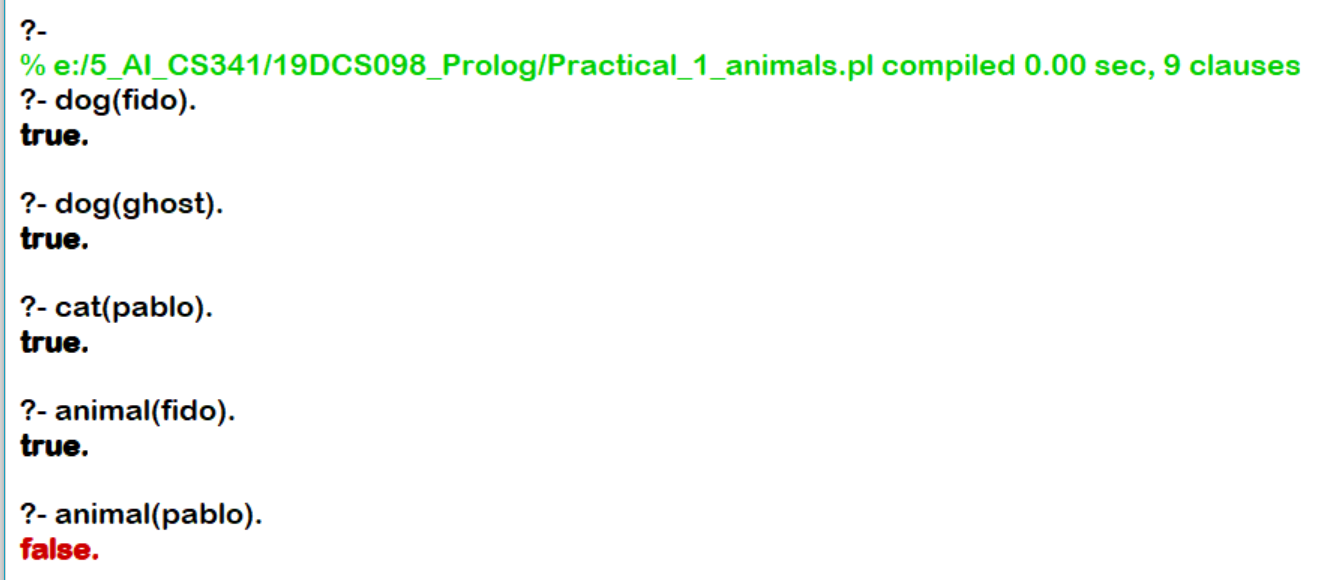
Write a program in prolog to implement simple facts and Queries.

**PROGRAM CODE:**

* **If anything X is animal if it is dog**

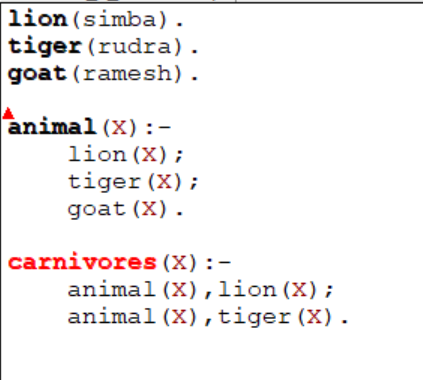


**OUTPUT:**

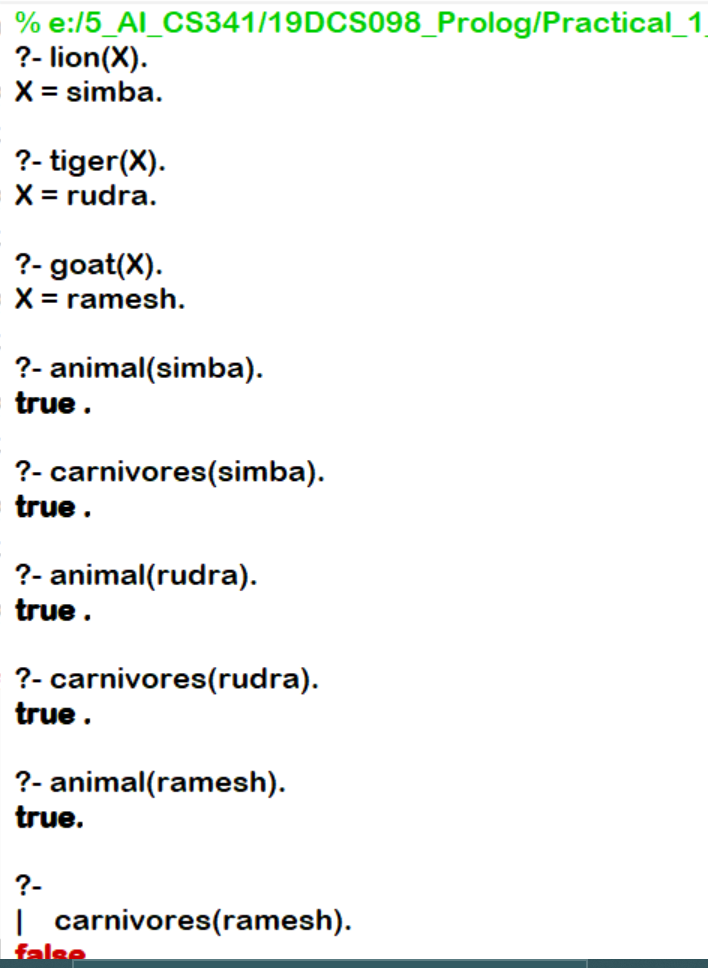


* **Lion, Tiger and goat are animals. Lion and tiger are carnivores but goat is not.**

**PROGRAM CODE**:

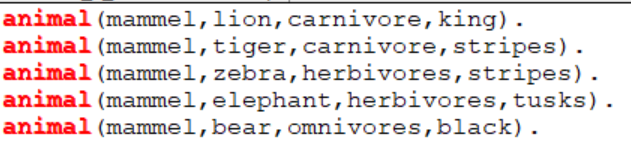


**OUTPUT:**

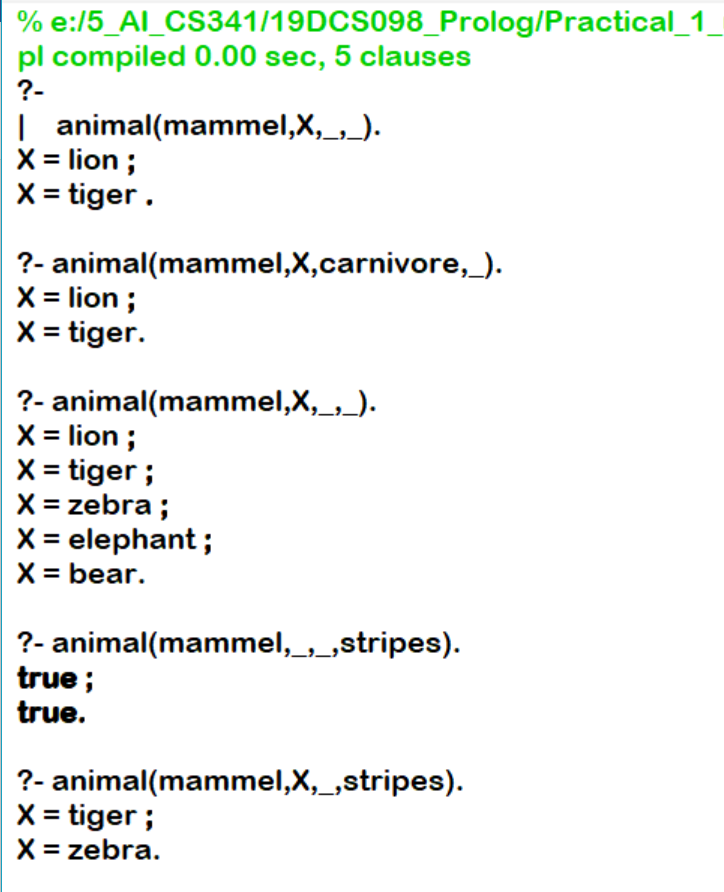


* **To show the use of unknown variable.**

**PROGRAM CODE:**

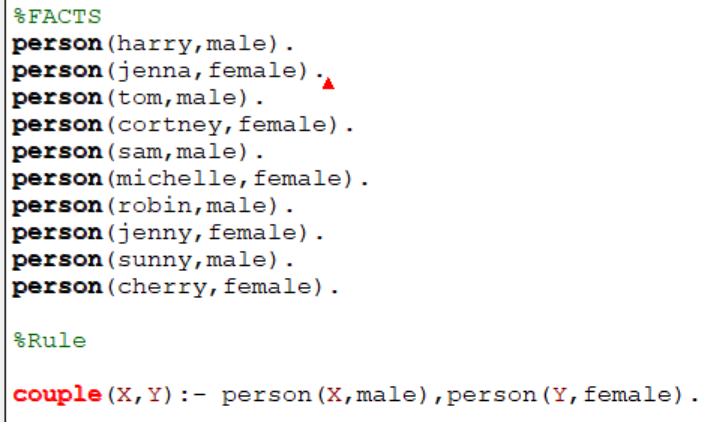


**OUTPUT:**

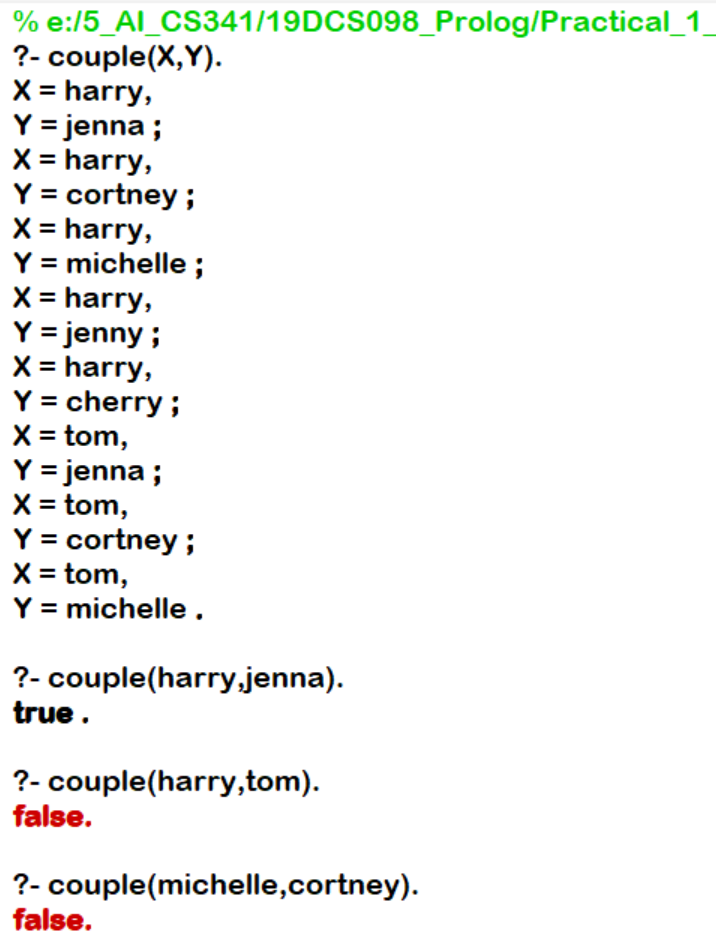


* **COUPLE PROBLEM**

**PROGRAM CODE:**

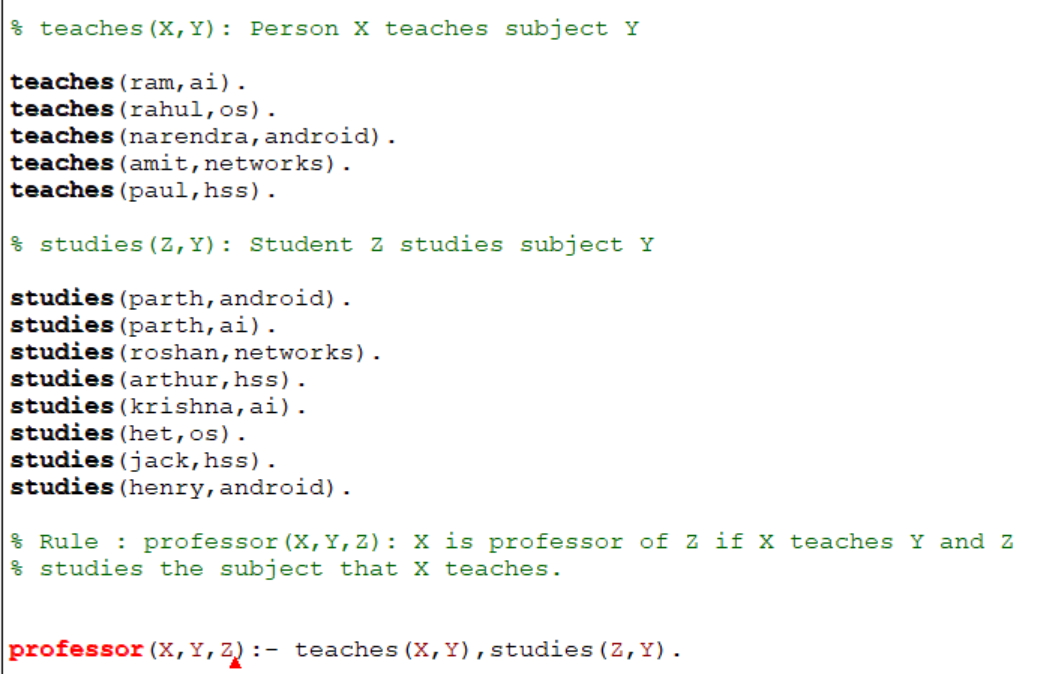


**OUTPUT:**

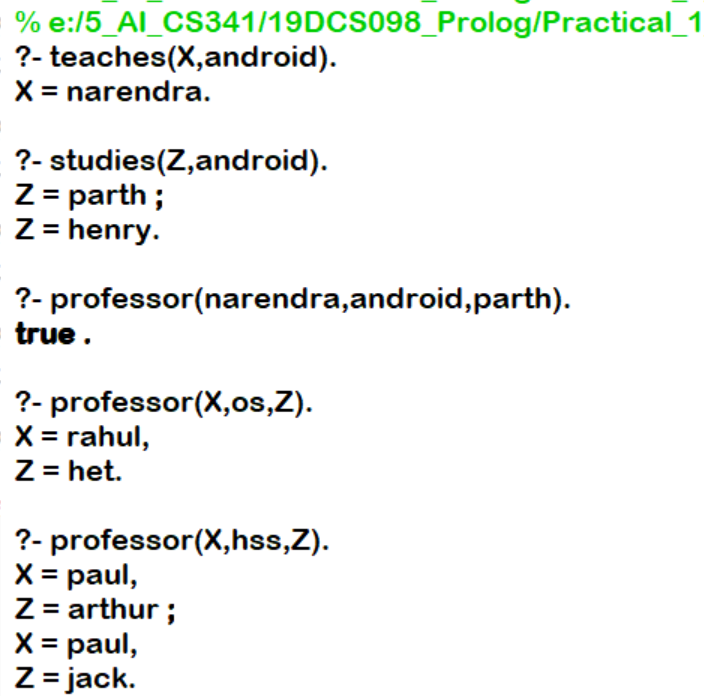


* **Person X teaches subject Y and student Z is studying the subject having subject Y.**

**PROGRAM CODE:**



**OUTPUT:**



**CONCLUSION:**

* By performing the above practicals, we came to know about the basics of Prolog.

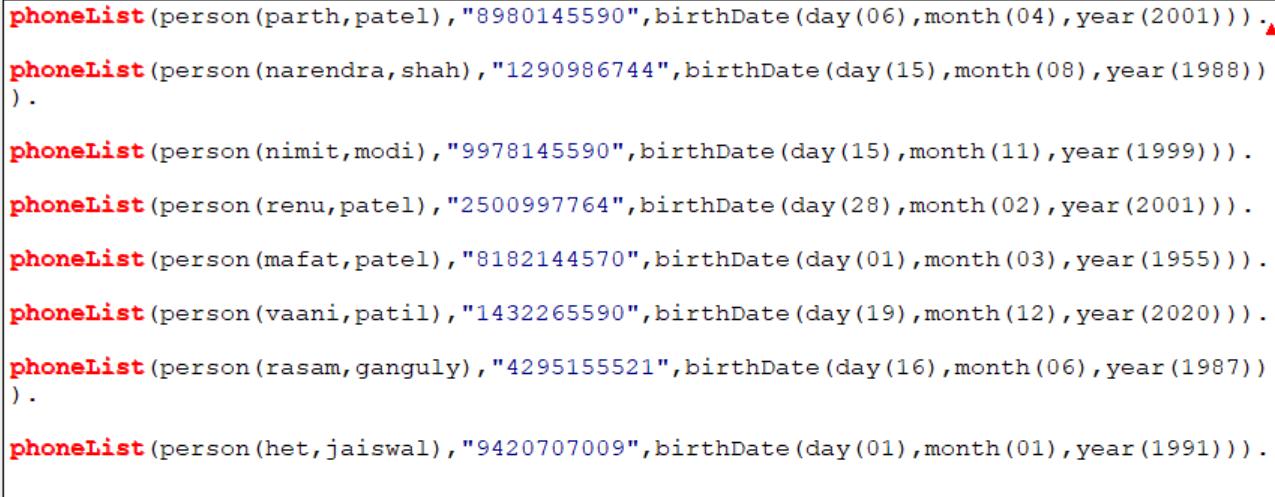
We also learnt about the facts and queries in Prolog.

**PRACTICAL-1.2**

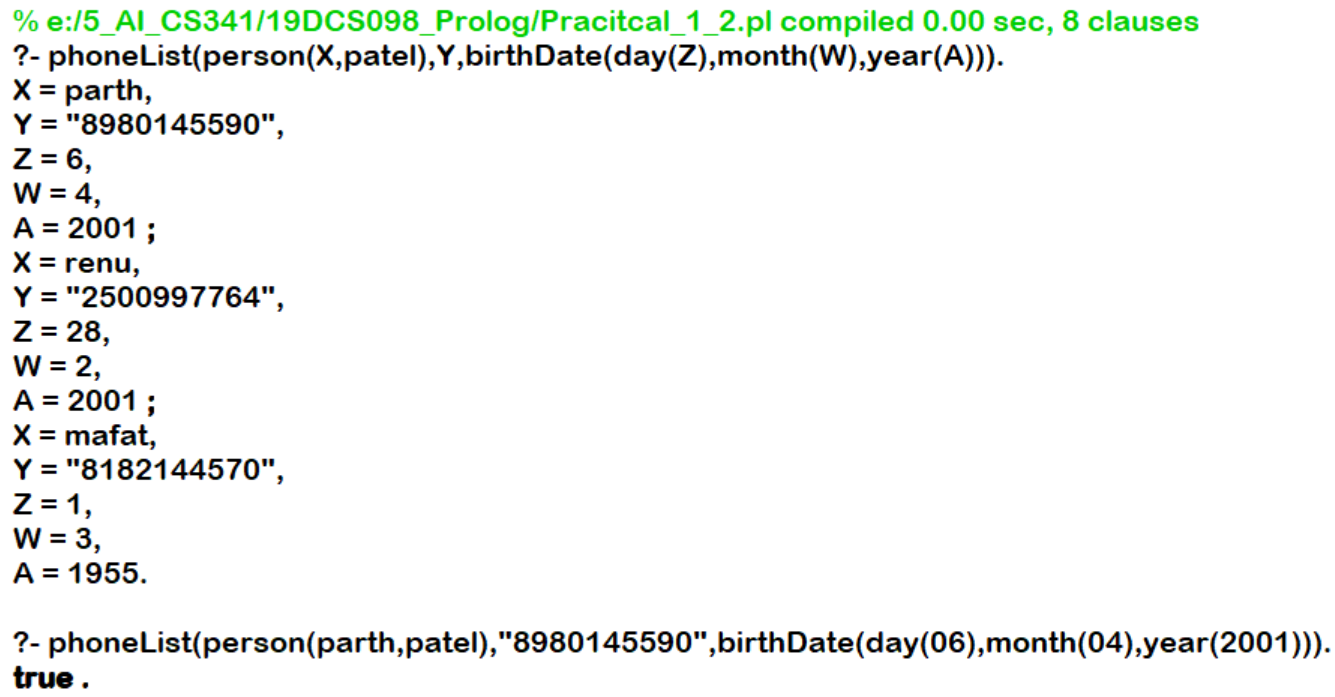
**AIM:**

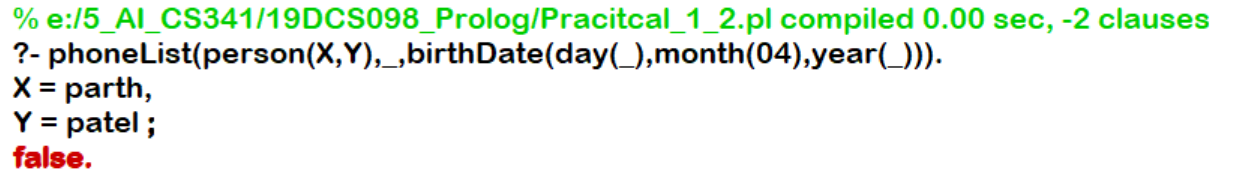
Write a program in prolog to implement phone list which stores name, phone number and birthdays of friends and family members. Write a query to get a list of people whose birthdays are in the current month.

**PROGRAM CODE:**



**OUTPUT:**





**CONCLUSION:**

* By performing the above practical, we learnt about how to write facts, queries and rules in prolog.
* We also learnt about how to execute a program in prolog.

**PRACTICE PROBLEM**

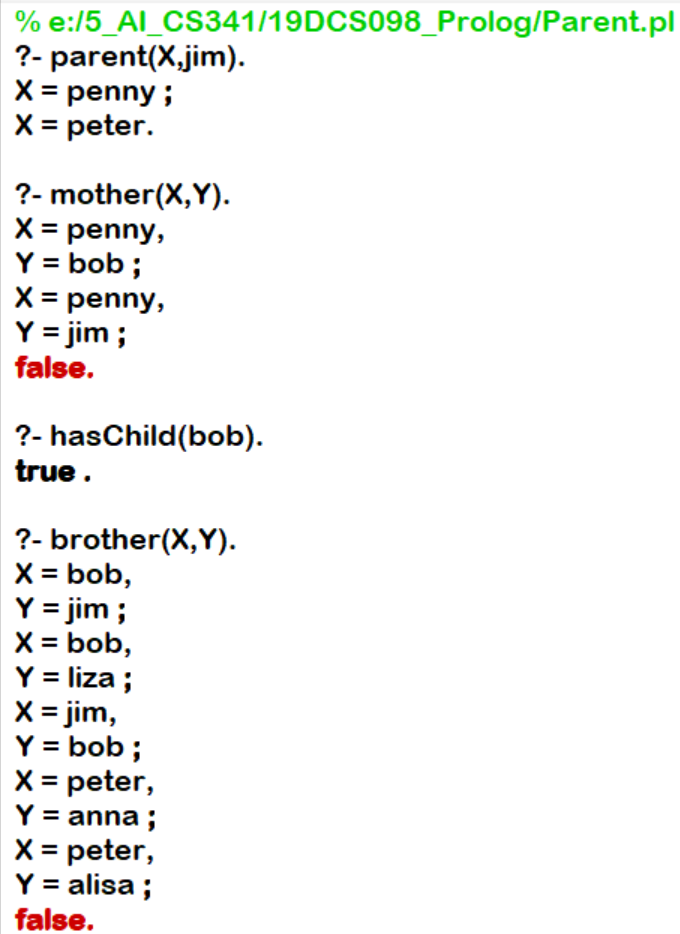
**AIM:**

The Family Problem

**PROGRAM CODE:**



**OUTPUT:**



**CONCLUSION:**

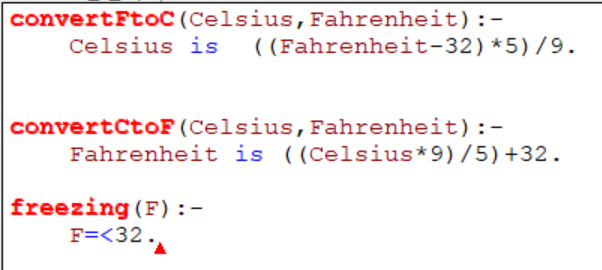
* By performing the above practical, we learnt about how to write facts, queries and rules in prolog.
* We also learnt about how to execute a program in prolog.

**PRACTICAL-1.3**

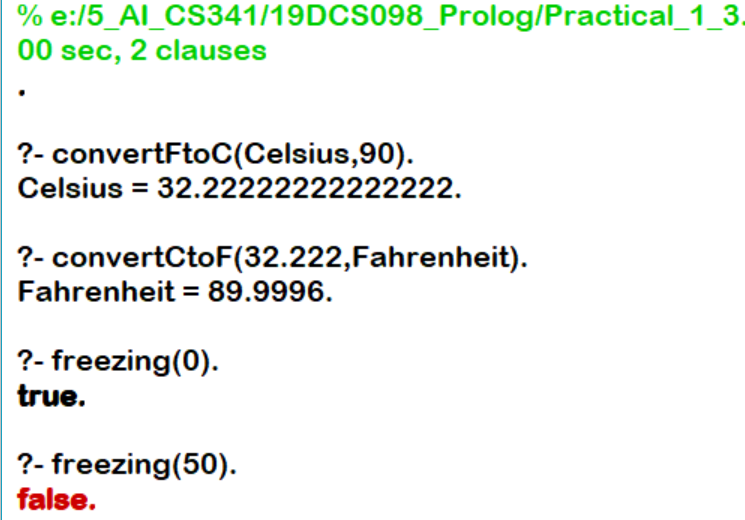
**AIM:**

Write predicates one converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

**PROGRAM CODE:**



**OUTPUT:**



**CONCLUSION:**

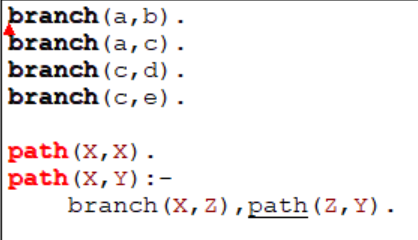
* By performing the above practical, we learnt about how the use of neck symbol.
* We also learnt about the basic procedure to create a prolog program.

**PRACTICAL-1.4**

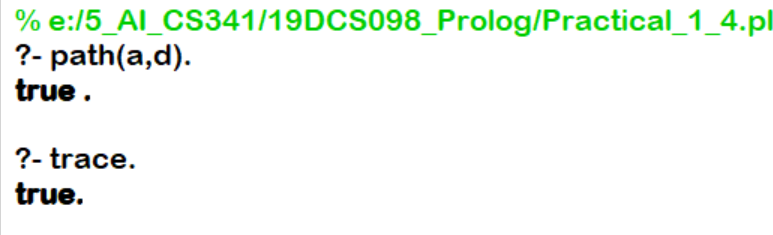
**AIM:**

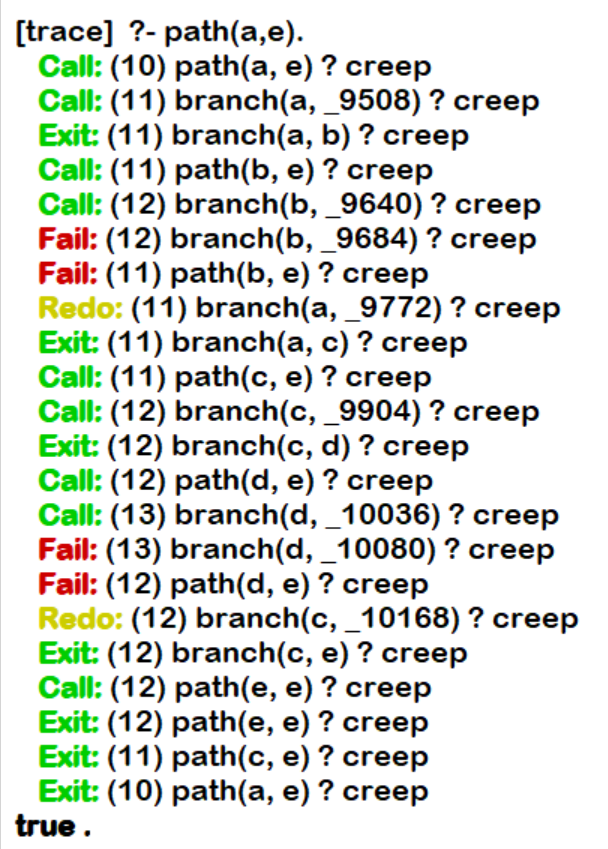
Demonstrate Backtracking in Prolog.

**PROGRAM CODE:**



**OUTPUT:**





**CONCLUSION:**

By performing the above practical, we learnt about the concept of backtracking in prolog.

**EXTRA PRACTICAL:**

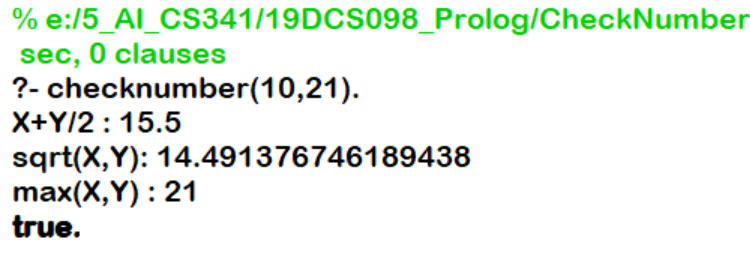
**AIM:**

Problem of checknumber.

**PROGRAM CODE:**



**OUTPUT:**



**CONCLUSION:**

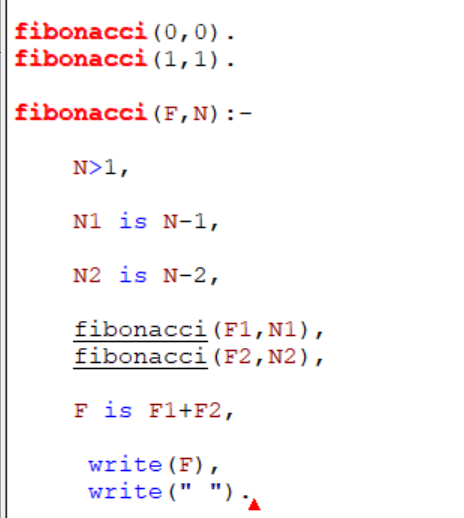
By performing the above practical, we learnt the use of write() in Prolog.

**PRACTICAL-2.1**

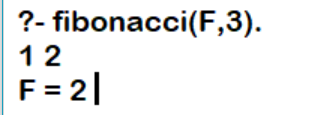
**AIM:**

Write a program to display Fibonacci series in prolog

**PROGRAM CODE:**



**OUTPUT:**



**CONCLUSION:**

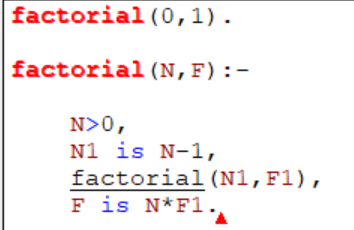
By performing the above practical, we learnt about the logic behind the Fibonacci series in Prolog.

**PRACTICAL-2.2**

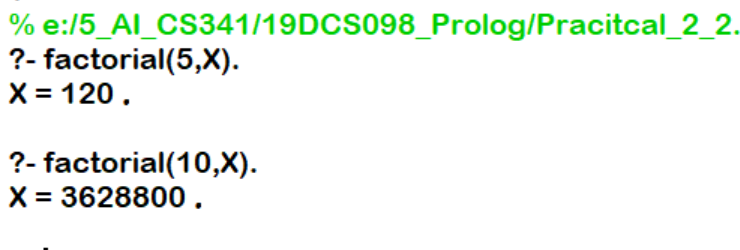
**AIM:**

Write a program to display Factorial in prolog

**PROGRAM CODE:**



**OUTPUT:**



**CONCLUSION:**

* By performing the above practical, we learnt the basic syntax of prolog and also about the arithmetic operators in Prolog.

**PRACTICAL-3**

**AIM:**

Write a prolog program for medical diagnosis system of childhood diseases.

**PROGRAM CODE:**

go :-

write('What is the patient''s name? '),

read(Patient),get\_single\_char(Code),

hypothesis(Patient,Disease),

write\_list([Patient,', probably has ',Disease,'.']),nl.

go :-

write('Sorry, I don''t seem to be able to'),nl,

write('diagnose the disease.'),nl.

symptom(Patient,fever) :-

verify(Patient," have a fever (y/n) ?").

symptom(Patient,rash) :-

verify(Patient," have a rash (y/n) ?").

symptom(Patient,headache) :-

verify(Patient," have a headache (y/n) ?").

symptom(Patient,runny\_nose) :-

verify(Patient," have a runny\_nose (y/n) ?").

symptom(Patient,conjunctivitis) :-

verify(Patient," have a conjunctivitis (y/n) ?").

symptom(Patient,cough) :-

verify(Patient," have a cough (y/n) ?").

symptom(Patient,body\_ache) :-

verify(Patient," have a body\_ache (y/n) ?").

symptom(Patient,chills) :-

verify(Patient," have a chills (y/n) ?").

symptom(Patient,sore\_throat) :-

verify(Patient," have a sore\_throat (y/n) ?").

symptom(Patient,sneezing) :-

verify(Patient," have a sneezing (y/n) ?").

symptom(Patient,swollen\_glands) :-

verify(Patient," have a swollen\_glands (y/n) ?").

ask(Patient,Question) :-

write(Patient),write(', do you'),write(Question),

read(N),

( (N == yes ; N == y)

->

assert(yes(Question)) ;

assert(no(Question)), fail).

:- dynamic yes/1,no/1.

verify(P,S) :-

(yes(S) -> true ;

(no(S) -> fail ;

ask(P,S))).

undo :- retract(yes(\_)),fail.

undo :- retract(no(\_)),fail.

undo.

hypothesis(Patient,german\_measles) :-

symptom(Patient,fever),

symptom(Patient,headache),

symptom(Patient,runny\_nose),

symptom(Patient,rash).

hypothesis(Patient,common\_cold) :-

symptom(Patient,headache),

symptom(Patient,sneezing),

symptom(Patient,sore\_throat),

symptom(Patient,runny\_nose),

symptom(Patient,chills).

hypothesis(Patient,measles) :-

symptom(Patient,cough),

symptom(Patient,sneezing),

symptom(Patient,runny\_nose).

hypothesis(Patient,flu) :-

symptom(Patient,fever),

symptom(Patient,headache),

symptom(Patient,body\_ache),

symptom(Patient,conjunctivitis),

symptom(Patient,chills),

symptom(Patient,sore\_throat),

symptom(Patient,runny\_nose),

symptom(Patient,cough).

hypothesis(Patient,mumps) :-

symptom(Patient,fever),

symptom(Patient,swollen\_glands).

hypothesis(Patient,chicken\_pox) :-

symptom(Patient,fever),

symptom(Patient,chills),

symptom(Patient,body\_ache),

symptom(Patient,rash).

write\_list([]).

write\_list([Term| Terms]) :-

write(Term),

write\_list(Terms).

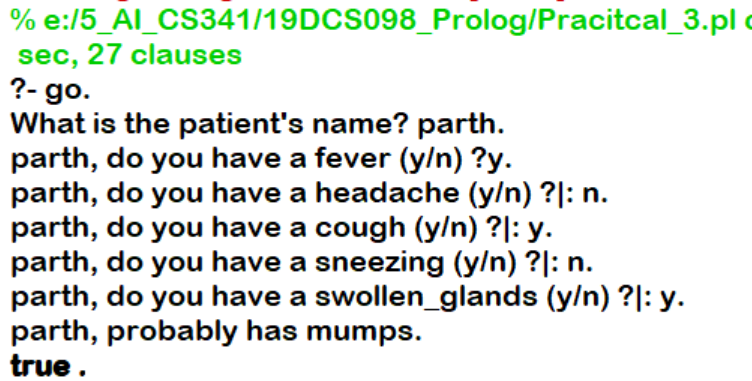
response(Reply) :-

get\_single\_char(Code),

put\_code(Code), nl,

char\_code(Reply, Code).

**OUTPUT:**



**CONCLUSION:**

* By performing the above practical, we can conclude that prolog is most useful in the areas related to AI research, such as problem solving, planning or naural language interpretation.

**PRACTICAL-4**

**AIM:**

Write a program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece, cousin.

**PROGRAM CODE:**

male(tom).

male(jerry).

male(harry).

male(sunny).

male(balmar).

female(anne).

female(jenna).

female(arthur).

female(jake).

female(granny).

male(barry).

male(goffy).

female(goffy).

parent(sunny,jerry).

parent(sunny,harry).

parent(sunny,anne).

parent(jenna,jerry).

parent(jenna,harry).

parent(jenna,anne).

parent(balmar,sunny).

parent(jake,sunny).

parent(arthur,jenna).

parent(arthur,tom).

parent(granny,arthur).

parent(jerry,barry).

male(milan).

male(tino).

parent(rahul,milan).

parent(rahul,tino).

indian(anne).

indian(X) :- ancestor(X,anne).

indian(X) :- ancestor(anne,X).

relation(X,Y) :- ancestor(A,X), ancestor(A,Y).

father(X,Y) :- male(X),parent(X,Y).

father(goffy, \_) :- male(goffy).

mother(X,Y) :- female(X),parent(X,Y).

son(X,Y) :- male(X),parent(Y,X).

daughter(X,Y) :- female(X),parent(Y,X).

grandfather(X,Y) :- male(X),parent(X,Somebody),parent(Somebody,Y).

aunt(X,Y) :- female(X),sister(X,Mom),mother(Mom,Y).

aunt(X,Y) :- female(X),sister(X,Dad),father(Dad,Y).

sister(X,Y) :- female(X),parent(Par,X),parent(Par,Y), X \= Y.

uncle(X,Y) :- brother(X,Par),parent(Par,Y).

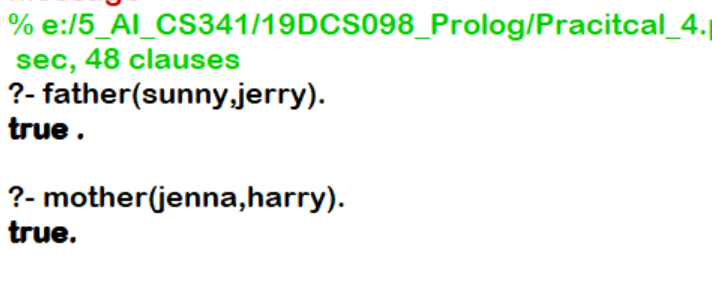
cousin(X,Y) :- uncle(Unc , X),father(Unc,Y).

ancestor(X,Y) :- parent(X,Y).

ancestor(X,Y) :- parent(X,Somebody),ancestor(Somebody,Y).

brother(X,Y) :- male(X),parent(Somebody,X),parent(Somebody,Y), X \= Y.

**OUTPUT:**



**CONCLUSION:**

* By performing the above practical, we came to know that Prolog is a declarative programming language where logic is expressed in terms of relations.

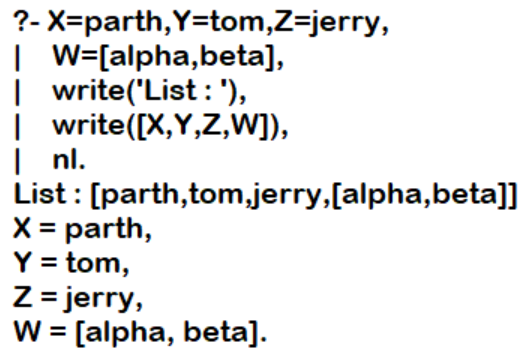
**PRACTICAL-5**

**AIM:**

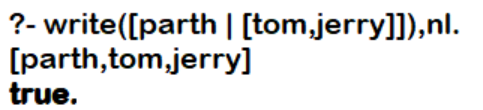
Write a program to perform following operations on lists in prolog.

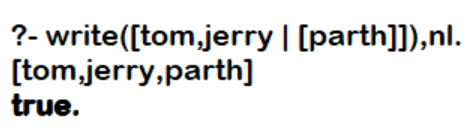
**PROGRAM CODE:**

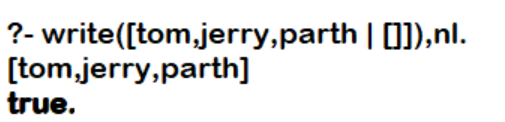
* **Create a list in Prolog:**

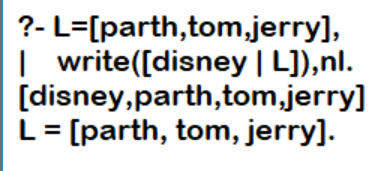


* **Cons Notation:**

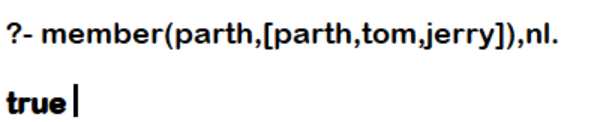


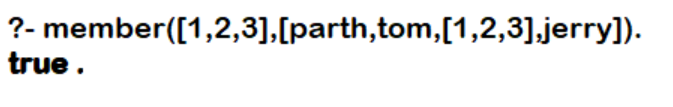


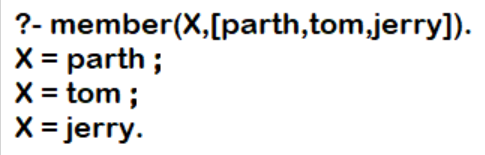




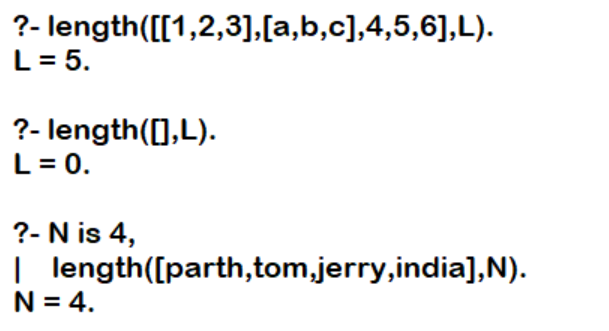
* **Membership in List:**



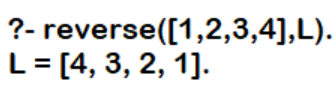


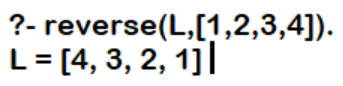


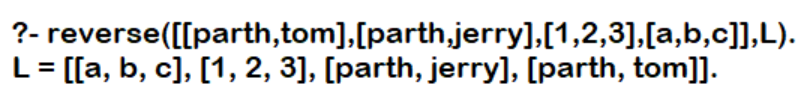
* **Length:**



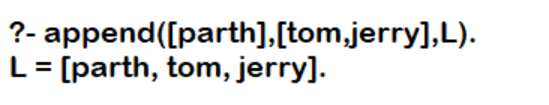
* **Reverse:**



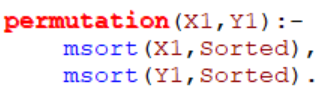




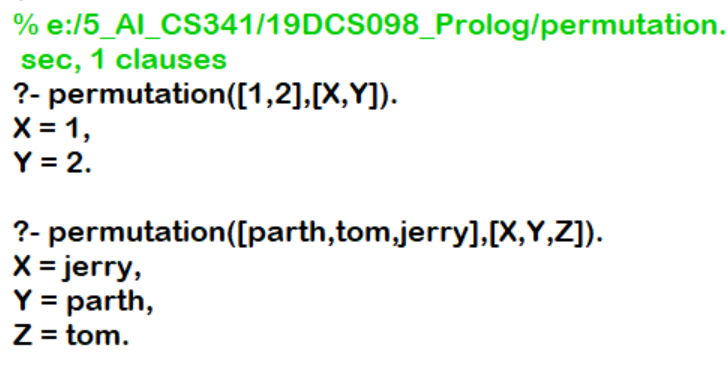
* **Append:**



* **Permutation**:



**OUTPUT:**



**CONCLUSION:**

* By performing the above practical, we learnt about the basic concept of lists in PROLOG.

**PRACTICAL-6**

**AIM:**

Write a program to demonstrate cut and fail in prolog.

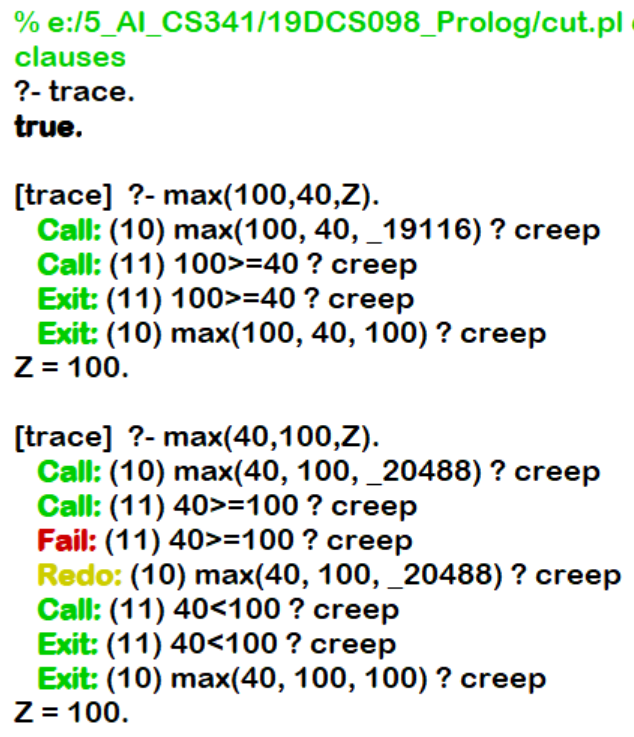
**PROGRAM CODE:**

**Program-1:**

max(X,Y,X):- X>=Y,!.

max(X,Y,Y):- X<Y.

**OUTPUT:**



**Program-2:**

animal(cobra).

animal(python).

animal(blackMamba).

snake(cobra).

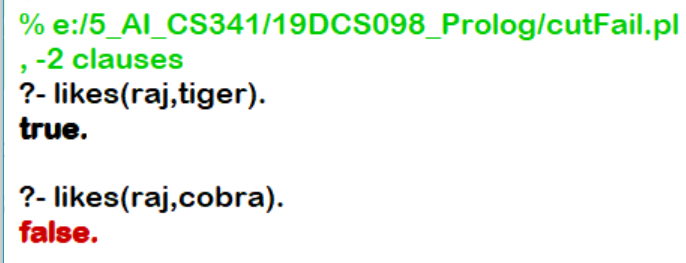
snake(python).

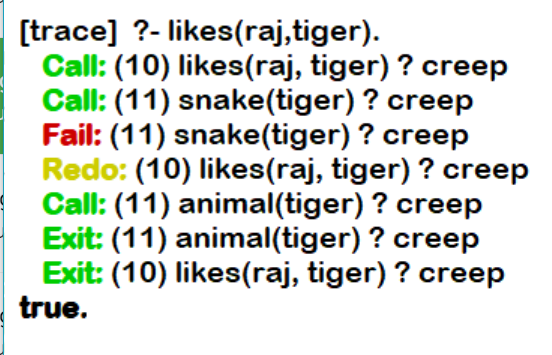
snake(blackMamba).

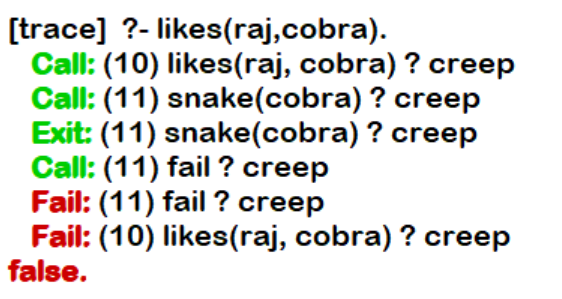
likes(raj,X):- snake(X),!,fail.

likes(raj,X):- animal(X).

**OUTPUT:**







**CONCLUSION:**

By performing the above practical, we learned about cut and fail.

**CUT:**

* Represented by !.
* It always succeeds, but cannot be backtracked. It is best used to prevent unwanted backtracking,

**fail** is a special symbol that will immediately fail when Prolog encounters it as a goal

**PRACTICAL-7.1**

**AIM:**

Design Depth First Search Tree and Breadth First Search Tree for Water-Jug Problem in python

**PROGRAM CODE:**

BFS APPROACH:

#BFS APPROACH

print("------------------------------------------------------------------------------------")

print("SOLUTION OF WATER JUG PROBLEM WITH BFS APPROACH")

print("------------------------------------------------------------------------------------")

#INPUT FOR JUG-1

capacity\_of\_X = int ( input ( "ENTER THE MAXIMUM CAPACITY OF JUG-1 : " ))

print("------------------------------------------------------------------------------------")

#INPUT FOR JUG-2

capacity\_of\_Y = int ( input ( "ENTER THE MAXIMUM CAPACITY OF JUG-2 : " ))

print("------------------------------------------------------------------------------------")

#INPUT FOR GOAL

end = int ( input ( "ENTER THE DESIRED VOLUME (GOAL) : " ))

print("------------------------------------------------------------------------------------")

#MAIN LOGIC

def bfs(begin, end, capacity\_of\_X, capacity\_of\_Y):

traversal\_path = []

front = []

front.append(begin)

visited = []

while ( not ( not front)):

current = front.pop()

x = current[ 0 ]

y = current[ 1 ]

traversal\_path.append(current)

if x == end or y == end:

print ( "PATH FOUND" )

print("------------------------------------------------------------------------------------")

return traversal\_path

# RULE 1

if current[ 0 ] < capacity\_of\_X and ([capacity\_of\_X, current[ 1 ]] not in visited):

front.append([capacity\_of\_X, current[ 1 ]])

visited.append([capacity\_of\_X, current[ 1 ]])

# RULE 2

if current[ 1 ] < capacity\_of\_Y and ([current[ 0 ], capacity\_of\_Y] not in visited):

front.append([current[ 0 ], capacity\_of\_Y])

visited.append([current[ 0 ], capacity\_of\_Y])

# RULE 3

if current[ 0 ] > capacity\_of\_X and ([ 0 , current[ 1 ]] not in visited):

front.append([ 0 , current[ 1 ]])

visited.append([ 0 , current[ 1 ]])

# RULE 4

if current[ 1 ] > capacity\_of\_Y and ([capacity\_of\_X, 0 ] not in visited):

front.append([capacity\_of\_X, 0 ])

visited.append([capacity\_of\_X, 0 ])

# RULE 5

#(x, y) -> (min(x + y, capacity\_of\_X), max(0, x + y - capacity\_of\_X))

if current[ 1 ] > 0 and ([ min (x + y, capacity\_of\_X), max ( 0 , x + y - capacity\_of\_X)] not in visited):

front.append([ min (x + y, capacity\_of\_X), max ( 0 , x + y - capacity\_of\_X)])

visited.append([ min (x + y, capacity\_of\_X), max ( 0 , x + y - capacity\_of\_X)])

if current[ 0 ] > 0 and ([ max ( 0 , x + y - capacity\_of\_Y), min (x + y, capacity\_of\_Y)] not in visited):

front.append([ max ( 0 , x + y - capacity\_of\_Y), min (x + y,capacity\_of\_Y)])

visited.append([ max ( 0 , x + y - capacity\_of\_Y), min (x + y,capacity\_of\_Y)])

return "PATH NOT FOUND"

def hcf(a,b):

if a==0:

return b

return hcf(b%a,a)

begin = [ 0 , 0 ]

if end % hcf(capacity\_of\_X,capacity\_of\_Y) == 0 :

print (bfs(begin, end, capacity\_of\_X, capacity\_of\_Y))

else :

print("------------------------------------------------------------------------------------")

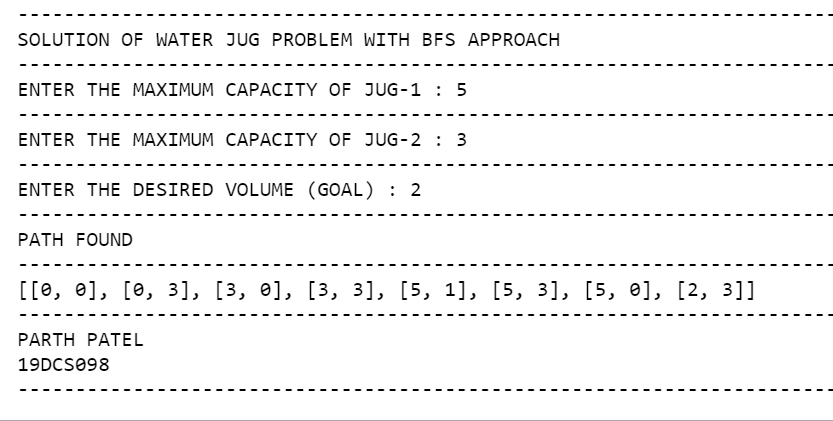
print ( " CANNOT FIND THE SOLUTION FOR THE GIVEN PROBLEM/STATE" )

print("------------------------------------------------------------------------------------")

print("PARTH PATEL\n19DCS098")

print("------------------------------------------------------------------------------------")

**OUTPUT:**

****

DFS APPROACH:

#DFS APPROACH

print("------------------------------------------------------------------------------------")

print("SOLUTION OF WATER JUG PROBLEM WITH DFS APPROACH")

print("------------------------------------------------------------------------------------")

#capacity = (10, 9, 8)

#INPUT FOR JUG-1

x = int ( input ( "ENTER THE MAXIMUM CAPACITY OF JUG-1 : " ))

print("------------------------------------------------------------------------------------")

#INPUT FOR JUG-2

y = int ( input ( "ENTER THE MAXIMUM CAPACITY OF JUG-2 : " ))

print("------------------------------------------------------------------------------------")

#INPUT FOR GOAL

z = int ( input ( "ENTER THE DESIRED VOLUME (GOAL) : " ))

print("------------------------------------------------------------------------------------")

#x = capacity[0]

#y = capacity[1]

#z = capacity[2]

data = {}

path = []

def find\_states(state):

a = state[0]

b = state[1]

c = state[2]

if a == 6 and b == 6:

path.append(state)

return True

if (a, b, c) in data:

return False

data[(a, b, c)] = 1

if a > 0:

if a + b <= y:

if find\_states((0, a + b, c)):

path.append(state)

return True

else:

if find\_states((a - (y - b), y, c)):

path.append(state)

return True

if a + c <= z:

if find\_states((0, b, a + c)):

path.append(state)

return True

else:

if find\_states((a - (z - c), b, z)):

path.append(state)

return True

if b > 0:

if a + b <= x:

if find\_states((a + b, 0, c)):

path.append(state)

return True

else:

if find\_states((x, b - (x - a), c)):

path.append(state)

return True

if b + c <= z:

if find\_states((a, 0, b + c)):

path.append(state)

return True

else:

if find\_states((a, b - (z - c), z)):

path.append(state)

return True

if c > 0:

if a + c <= x:

if find\_states((a + c, b, 0)):

path.append(state)

return True

else:

if find\_states((x, b, c - (x - a))):

path.append(state)

return True

if b + c <= y:

if find\_states((a, b + c, 0)):

path.append(state)

return True

else:

if find\_states((a, y, c - (y - b))):

path.append(state)

return True

return False

initial\_state = (12, 0, 0)

find\_states(initial\_state)

path.reverse()

for i in path:

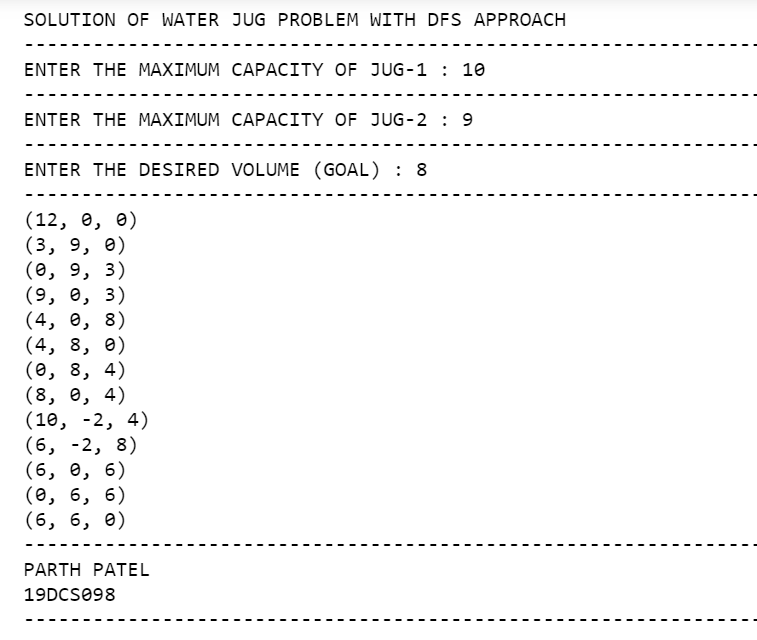
print(i)

print("------------------------------------------------------------------------------------")

print("PARTH PATEL\n19DCS098")

print("------------------------------------------------------------------------------------")

**OUTPUT:**



**CONCLUSION:**

* By performing the above practical, we learned how to solve water jug problem.

**PRACTICAL-7.2**

**AIM:**

Write a program to solve The N queen Problem using Hill Climbing with random Neighbor

**PROGRAM CODE:**

#include <iostream>

#include <math.h>

#include<ctime>

#define N 4

using namespace std;

void configureRandomly(int board[][N], int\* state)

{

srand(time(0));

for (int i = 0; i < N; i++) {

state[i] = rand() % N;

board[state[i]][i] = 1;

}

}

void printBoard(int board[][N])

{

for (int i = 0; i < N; i++) {

cout << " ";

for (int j = 0; j < N; j++) {

cout << board[i][j] << " ";

}

cout << "\n";

}

}

void printState(int\* state)

{

for (int i = 0; i < N; i++) {

cout << " " << state[i] << " ";

}

cout << endl;

}

bool compareStates(int\* state1, int\* state2)

{

for (int i = 0; i < N; i++) {

if (state1[i] != state2[i]) {

return false;

}

}

return true;

}

void fill(int board[][N], int value)

{

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

board[i][j] = value;

}

}

}

int calculateObjective(int board[][N], int\* state)

{

int attacking = 0;

int row, col;

for (int i = 0; i < N; i++) {

row = state[i], col = i - 1;

while (col >= 0&& board[row][col] != 1) {

col--;

}

if (col >= 0&& board[row][col] == 1) {

attacking++;

}

row = state[i], col = i + 1;

while (col < N&& board[row][col] != 1) {

col++;

}

if (col < N&& board[row][col] == 1) {

attacking++;

}

row = state[i] - 1, col = i - 1;

while (col >= 0 && row >= 0&& board[row][col] != 1) {

col--;

row--;

}

if (col >= 0 && row >= 0&& board[row][col] == 1) {

attacking++;

}

row = state[i] + 1, col = i + 1;

while (col < N && row < N&& board[row][col] != 1) {

col++;

row++;

}

if (col < N && row < N&& board[row][col] == 1) {

attacking++;

}

row = state[i] + 1, col = i - 1;

while (col >= 0 && row < N&& board[row][col] != 1) {

col--;

row++;

}

if (col >= 0 && row < N&& board[row][col] == 1) {

attacking++;

}

row = state[i] - 1, col = i + 1;

while (col < N && row >= 0 && board[row][col] != 1) {

col++;

row--;

}

if (col < N && row >= 0&& board[row][col] == 1) {

attacking++;

}

}

return (int)(attacking / 2);

}

void generateBoard(int board[][N], int\* state)

{

fill(board, 0);

for (int i = 0; i < N; i++) {

board[state[i]][i] = 1;

}

}

void copyState(int\* state1, int\* state2)

{

for (int i = 0; i < N; i++) {

state1[i] = state2[i];

}

}

void getNeighbour(int board[][N], int\* state)

{

int opBoard[N][N];

int opState[N];

copyState(opState, state);

generateBoard(opBoard, opState);

int opObjective = calculateObjective(opBoard, opState);

int NeighbourBoard[N][N];

int NeighbourState[N];

copyState(NeighbourState, state);

generateBoard(NeighbourBoard, NeighbourState);

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

if (j != state[i]) {

NeighbourState[i] = j;

NeighbourBoard[NeighbourState[i]][i]= 1;

NeighbourBoard[state[i]][i]= 0;

int temp= calculateObjective(NeighbourBoard,NeighbourState);

if (temp <= opObjective) {

opObjective = temp;

copyState(opState,NeighbourState);

generateBoard(opBoard,opState);

}

NeighbourBoard[NeighbourState[i]][i]= 0;

NeighbourState[i] = state[i];

NeighbourBoard[state[i]][i] = 1;

}

}

}

copyState(state, opState);

fill(board, 0);

generateBoard(board, state);

}

void hillClimbing(int board[][N], int\* state)

{

int neighbourBoard[N][N] = {};

int neighbourState[N];

copyState(neighbourState, state);

generateBoard(neighbourBoard,neighbourState);

do {

copyState(state, neighbourState);

generateBoard(board, state);

getNeighbour(neighbourBoard, neighbourState);

if (compareStates(state, neighbourState)) {

printBoard(board);

break;

}

else if (calculateObjective(board, state) == calculateObjective(neighbourBoard,neighbourState)) {

neighbourState[rand() % N]= rand() % N;

generateBoard(neighbourBoard,neighbourState);

}

}

while (true);

}

int main()

{

int state[N] = {};

int board[N][N] = {};

configureRandomly(board, state);

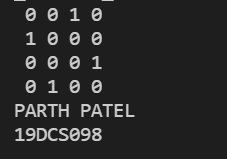
hillClimbing(board, state);

cout<<"PARTH PATEL\n19DCS098"<<endl;

return 0;

}

**OUTPUT:**



**CONCLUSION:**

* By performing the above practical, we learnt the concept of hill climbing by solving N-Queen problem.

**PRACTICAL-7.3**

**AIM:**

Implement Travelling Salesman Problem using Simulated Annealing Algorithm in Python.

**PROGRAM CODE:**

#TSP USING SIMULATED ANNEALING

# Import libraries

import sys

import random

import copy

import numpy as np

# This class represent a state

class State:

# Create a new state

def \_\_init\_\_(self, route:[], distance:int=0):

self.route = route

self.distance = distance

# Compare states

def \_\_eq\_\_(self, other):

for i in range(len(self.route)):

if(self.route[i] != other.route[i]):

return False

return True

# Sort states

def \_\_lt\_\_(self, other):

return self.distance < other.distance

# Print a state

def \_\_repr\_\_(self):

return ('({0},{1})\n'.format(self.route, self.distance))

# Create a shallow copy

def copy(self):

return State(self.route, self.distance)

# Create a deep copy

def deepcopy(self):

return State(copy.deepcopy(self.route), copy.deepcopy(self.distance))

# Update distance

def update\_distance(self, matrix, home):

# Reset distance

self.distance = 0

# Keep track of departing city

from\_index = home

# Loop all cities in the current route

for i in range(len(self.route)):

self.distance += matrix[from\_index][self.route[i]]

from\_index = self.route[i]

# Add the distance back to home

self.distance += matrix[from\_index][home]

# This class represent a city (used when we need to delete cities)

class City:

# Create a new city

def \_\_init\_\_(self, index:int, distance:int):

self.index = index

self.distance = distance

# Sort cities

def \_\_lt\_\_(self, other):

return self.distance < other.distance

# Return true with probability p

def probability(p):

return p > random.uniform(0.0, 1.0)

# Schedule function for simulated annealing

def exp\_schedule(k=20, lam=0.005, limit=1000):

return lambda t: (k \* np.exp(-lam \* t) if t < limit else 0)

# Get best solution by distance

def get\_best\_solution\_by\_distance(matrix:[], home:int):

# Variables

route = []

from\_index = home

length = len(matrix) - 1

# Loop until route is complete

while len(route) < length:

# Get a matrix row

row = matrix[from\_index]

# Create a list with cities

cities = {}

for i in range(len(row)):

cities[i] = City(i, row[i])

# Remove cities that already is assigned to the route

del cities[home]

for i in route:

del cities[i]

# Sort cities

sorted = list(cities.values())

sorted.sort()

# Add the city with the shortest distance

from\_index = sorted[0].index

route.append(from\_index)

# Create a new state and update the distance

state = State(route)

state.update\_distance(matrix, home)

# Return a state

return state

# Mutate a solution

def mutate(matrix:[], home:int, state:State, mutation\_rate:float=0.01):

# Create a copy of the state

mutated\_state = state.deepcopy()

# Loop all the states in a route

for i in range(len(mutated\_state.route)):

# Check if we should do a mutation

if(random.random() < mutation\_rate):

# Swap two cities

j = int(random.random() \* len(state.route))

city\_1 = mutated\_state.route[i]

city\_2 = mutated\_state.route[j]

mutated\_state.route[i] = city\_2

mutated\_state.route[j] = city\_1

# Update the distance

mutated\_state.update\_distance(matrix, home)

# Return a mutated state

return mutated\_state

# Simulated annealing

def simulated\_annealing(matrix:[], home:int, initial\_state:State, mutation\_rate:float=0.01, schedule=exp\_schedule()):

# Keep track of the best state

best\_state = initial\_state

# Loop a large number of times (int.max)

for t in range(sys.maxsize):

# Get a temperature

T = schedule(t)

# Return if temperature is 0

if T == 0:

return best\_state

# Mutate the best state

neighbor = mutate(matrix, home, best\_state, mutation\_rate)

# Calculate the change in e

delta\_e = best\_state.distance - neighbor.distance

# Check if we should update the best state

if delta\_e > 0 or probability(np.exp(delta\_e / T)):

best\_state = neighbor

# The main entry point for this module

def main():

# Cities to travel

cities = ['AHMEDABAD', 'BHOPAL', 'CHANDIGARH', 'DELHI', 'BENGALURU', 'CHENNAI', 'SURAT', 'VADODARA', 'MUMBAI', 'LUCKNOW', 'PUNE', 'HYDERABAD', 'NOIDA']

city\_indexes = [0,1,2,3,4,5,6,7,8,9,10,11,12]

# Index of start location

home = 2 # Chicago

# Distances in kiometres between cities, same indexes (i, j) as in the cities array

matrix = [[0, 1451, 700, 1000, 1600, 1300, 2000, 200, 2500, 800, 100, 2100, 1900],

[2400, 0, 1750, 1500, 850, 1250, 950, 2600, 400, 1600, 1350, 300, 600],

[700, 1700, 0, 350, 900, 800, 1700, 850, 1800, 200, 950, 1400, 1200],

[1100, 1500, 350, 0, 700, 800, 1400, 1100, 1500, 400, 1050, 1000, 900],

[1600, 800, 900, 700, 0, 600, 1000, 1700, 950, 800, 800, 500, 350],

[1300, 1200, 800, 800, 600, 0, 1600, 1500, 1700, 500, 200, 800, 1000],

[2400, 900, 1700, 1300, 1000, 1600, 0, 2400, 600, 1700, 1890, 1110, 700],

[213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099],

[2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600],

[875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162],

[1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200],

[2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504],

[1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0]]

# Get the best route by distance

state = get\_best\_solution\_by\_distance(matrix, home)

print('-- Best solution by distance --')

print(cities[home], end='')

for i in range(0, len(state.route)):

print(' -> ' + cities[state.route[i]], end='')

print(' -> ' + cities[home], end='')

print('\n\nTotal distance: {0} KM'.format(state.distance))

print()

# Run simulated annealing to find a better solution

state = get\_best\_solution\_by\_distance(matrix, home)

state = simulated\_annealing(matrix, home, state, 0.1)

print('-- Simulated annealing solution --')

print(cities[home], end='')

for i in range(0, len(state.route)):

print(' -> ' + cities[state.route[i]], end='')

print(' -> ' + cities[home], end='')

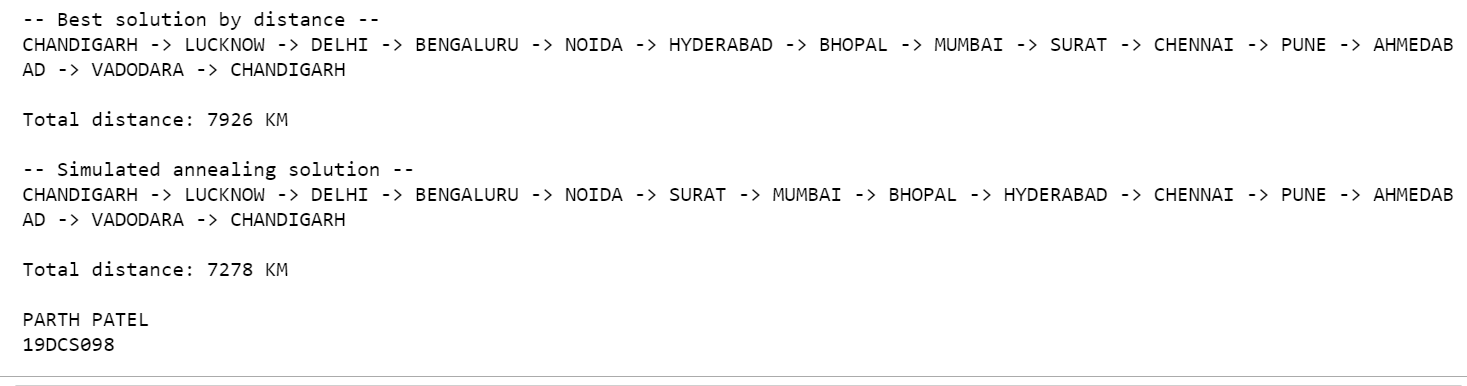
print('\n\nTotal distance: {0} KM'.format(state.distance))

print()

# Tell python to run main method

if \_\_name\_\_ == "\_\_main\_\_": main()

**OUTPUT:**



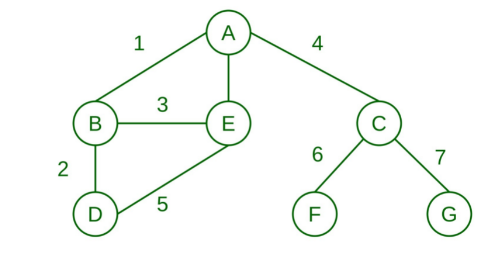
**CONCLUSION:**

By performing the above practical, we learnt the logic and concept of solving the problem of **TRAVELING SALESMAN PROBLEM** using **SIMULATED ANNEALING**.

**PRACTICAL-7.4**

**AIM:**

Write a Program to find Shortest Path using Best First Search Algorithm.



**PROGRAM CODE:**

#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> pi;

vector<vector<pi> > graph;

void addedge(int x, int y, int cost)

{

graph[x].push\_back(make\_pair(cost, y));

graph[y].push\_back(make\_pair(cost, x));

}

void best\_first\_search(int source, int target, int n)

{

vector<bool> visited(n, false);

priority\_queue<pi, vector<pi>, greater<pi> > pq;

// sorting in pq gets done by first value of pair

pq.push(make\_pair(0, source));

int s = source;

visited[s] = true;

while (!pq.empty()) {

int x = pq.top().second;

cout << x << " ";

pq.pop();

if (x == target)

break;

for (int i = 0; i < graph[x].size(); i++) {

if (!visited[graph[x][i].second]) {

visited[graph[x][i].second] = true;

pq.push(make\_pair(graph[x][i].first,graph[x][i].second));

}

}

}

}

int main()

{

int v; //NUMBER OF NODES

cout<<"ENTER THE SIZE OF THE GRAPH: ";

cin>>v;

cout<<"----------------------------------"<<endl;

graph.resize(v);

int start,end,cost;

for(int i=1;i<=v/2;i++){

cout<<"ENTER THE STARTING POSITION,ENDING POSITION, AND COST TO REACH : ";

cin>>start>>end>>cost;

addedge(start,end,cost);

cout<<"-------------------------------------------------------------------"<<endl;

}

int source;

int destination;

cout<<"ENTER THE SOURCE : ";

cin>>source;

cout<<"-------------------"<<endl;

cout<<"ENTER THE DESTINATION : ";

cin>>destination;

cout<<"--------------------------"<<endl; best\_first\_search(source, destination, v);

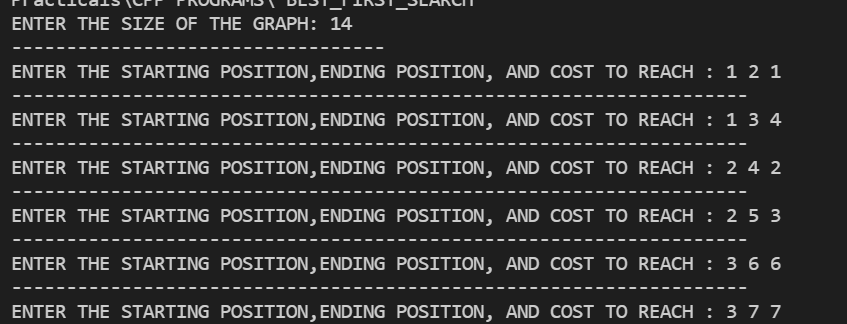
cout<<endl;

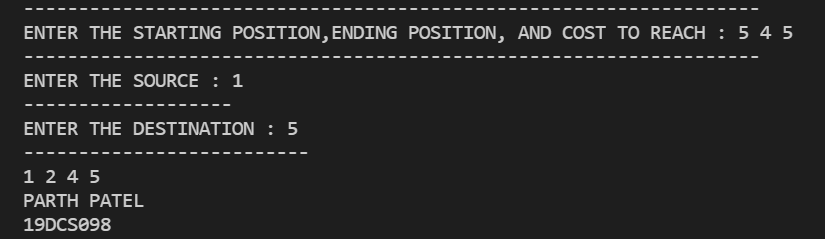
cout<<"PARTH PATEL\n19DCS098"<<endl;

return 0;

}

**OUTPUT:**





**CONCLUSION:**

By performing the above practical, we learnt the concept of **BEST FIRST SEARCH.**

**PRACTICAL-7.5**

**AIM:**

Write a program to solve 8 puzzle problem using A\*Algorithm in python

**PROGRAM CODE:**

class Node:

def \_\_init\_\_(self,data,level,fval):

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

""" Generate child nodes from the given node by moving the blank space

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 in 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 position value are out

of limits the return None """

if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

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):

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 closed lists to empty """

self.n = size

self.open = []

self.closed = []

def accept(self):

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(" \\\'/ \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 have 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]

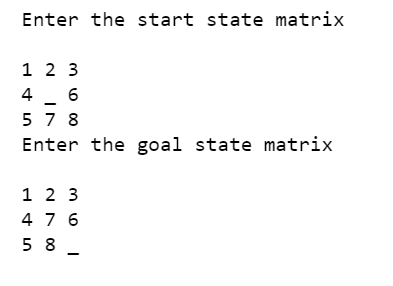
""" sort the opne list based on f value """

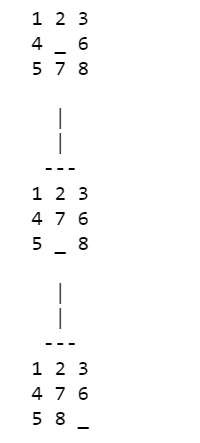
self.open.sort(key = lambda x:x.fval,reverse=False)

puz = Puzzle(3)

puz.process()

**OUTPUT:**





**PRACTICAL-8**

**AIM:**

Write a program for game Tic-Tac-Toe using MINIMAX Algorithm in python.

**PROGRAM CODE:**

from math import inf as infinity

from random import choice

import platform

import time

from os import system

HUMAN = -1

AI = +1

board = [

[0, 0, 0],

[0, 0, 0],

[0, 0, 0],

]

def evaluate(state):

# Function to heuristic evaluation of state.

# :param state: the state of the current board

# :return: +1 if the AIuter wins; -1 if the human wins; 0 draw

if wins(state, AI):

score = +1

elif wins(state, HUMAN):

score = -1

else:

score = 0

return score

def wins(state, player):

# This function tests if a specific player wins. Possibilities:

# \* Three rows [X X X] or [O O O]

# \* Three cols [X X X] or [O O O]

# \* Two diagonals [X X X] or [O O O]

# :param state: the state of the current board

# :param player: a human or a AIuter

# :return: True if the player wins

win\_state = [

[state[0][0], state[0][1], state[0][2]],

[state[1][0], state[1][1], state[1][2]],

[state[2][0], state[2][1], state[2][2]],

[state[0][0], state[1][0], state[2][0]],

[state[0][1], state[1][1], state[2][1]],

[state[0][2], state[1][2], state[2][2]],

[state[0][0], state[1][1], state[2][2]],

[state[2][0], state[1][1], state[0][2]],

]

if [player, player, player] in win\_state:

return True

else:

return False

def game\_over(state):

# This function test if the human or AIuter wins

# :param state: the state of the current board

# :return: True if the human or AIuter wins

return wins(state, HUMAN) or wins(state, AI)

def empty\_cells(state):

# Each empty cell will be added into cells' list

# :param state: the state of the current board

# :return: a list of empty cells

cells = []

for x, row in enumerate(state):

for y, cell in enumerate(row):

if cell == 0:

cells.append([x, y])

return cells

def valid\_move(x, y):

# A move is valid if the chosen cell is empty

# :param x: X coordinate

# :param y: Y coordinate

# :return: True if the board[x][y] is empty

if [x, y] in empty\_cells(board):

return True

else:

return False

def set\_move(x, y, player):

# Set the move on board, if the coordinates are valid

# :param x: X coordinate

# :param y: Y coordinate

# :param player: the current player

if valid\_move(x, y):

board[x][y] = player

return True

else:

return False

def minimax(state, depth, player):

# AI function that choice the best move

# :param state: current state of the board

# :param depth: node index in the tree (0 <= depth <= 9),

# but never nine in this case (see iaturn() function)

# :param player: an human or a AIuter

# :return: a list with [the best row, best col, best score]

if player == AI:

best = [-1, -1, -infinity]

else:

best = [-1, -1, +infinity]

if depth == 0 or game\_over(state):

score = evaluate(state)

return [-1, -1, score]

for cell in empty\_cells(state):

x, y = cell[0], cell[1]

state[x][y] = player

score = minimax(state, depth - 1, -player)

state[x][y] = 0

score[0], score[1] = x, y

if player == AI:

if score[2] > best[2]:

best = score # max value

else:

if score[2] < best[2]:

best = score # min value

return best

def clean():

# Clears the console

os\_name = platform.system().lower()

if 'windows' in os\_name:

system('cls')

else:

system('clear')

def render(state, c\_choice, h\_choice):

# Print the board on console

# :param state: current state of the board

chars = {

-1: h\_choice,

+1: c\_choice,

0: ' '

}

str\_line = '---------------'

print('\n' + str\_line)

for row in state:

for cell in row:

symbol = chars[cell]

print(f'| {symbol} |', end='')

print('\n' + str\_line)

def ai\_turn(c\_choice, h\_choice):

# It calls the minimax function if the depth < 9,

# else it choices a random coordinate.

# :param c\_choice: AIuter's choice X or O

# :param h\_choice: human's choice X or O

depth = len(empty\_cells(board))

if depth == 0 or game\_over(board):

return

clean()

print(f'AIuter turn [{c\_choice}]')

render(board, c\_choice, h\_choice)

if depth == 9:

x = choice([0, 1, 2])

y = choice([0, 1, 2])

else:

move = minimax(board, depth, AI)

x, y = move[0], move[1]

set\_move(x, y, AI)

time.sleep(1)

def human\_turn(c\_choice, h\_choice):

# The Human plays choosing a valid move.

# :param c\_choice: AIuter's choice X or O

# :param h\_choice: human's choice X or O

# :return:

depth = len(empty\_cells(board))

if depth == 0 or game\_over(board):

return

# Dictionary of valid moves

move = -1

moves = {

1: [0, 0], 2: [0, 1], 3: [0, 2],

4: [1, 0], 5: [1, 1], 6: [1, 2],

7: [2, 0], 8: [2, 1], 9: [2, 2],

}

clean()

print(f'Human turn [{h\_choice}]')

render(board, c\_choice, h\_choice)

while move < 1 or move > 9:

try:

move = int(input('Use numpad (1..9): '))

coord = moves[move]

can\_move = set\_move(coord[0], coord[1], HUMAN)

if not can\_move:

print('Bad move')

move = -1

except (EOFError, KeyboardInterrupt):

print('Bye')

exit()

except (KeyError, ValueError):

print('Bad choice')

def play():

# Main function that calls all functions

clean()

h\_choice = '' # X or O

c\_choice = '' # X or O

first = '' # if human is the first

# Human chooses X or O to play

while h\_choice != 'O' and h\_choice != 'X':

try:

print('')

h\_choice = input('Choose X or O\nChosen: ').upper()

except (EOFError, KeyboardInterrupt):

print('Bye')

exit()

except (KeyError, ValueError):

print('Bad choice')

# Setting AIuter's choice

if h\_choice == 'X':

c\_choice = 'O'

else:

c\_choice = 'X'

# Human may starts first

clean()

while first != 'Y' and first != 'N':

try:

first = input('First to start?[y/n]: ').upper()

except (EOFError, KeyboardInterrupt):

print('Bye')

exit()

except (KeyError, ValueError):

print('Bad choice')

# Main loop of this game

while len(empty\_cells(board)) > 0 and not game\_over(board):

if first == 'N':

ai\_turn(c\_choice, h\_choice)

first = ''

human\_turn(c\_choice, h\_choice)

ai\_turn(c\_choice, h\_choice)

# Game over message

if wins(board, HUMAN):

clean()

print(f'Human turn [{h\_choice}]')

render(board, c\_choice, h\_choice)

print('YOU WIN!')

elif wins(board, AI):

clean()

print(f'AIuter turn [{c\_choice}]')

render(board, c\_choice, h\_choice)

print('YOU LOSE!')

else:

clean()

render(board, c\_choice, h\_choice)

print('DRAW!')

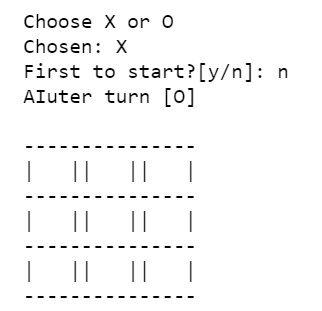
exit()

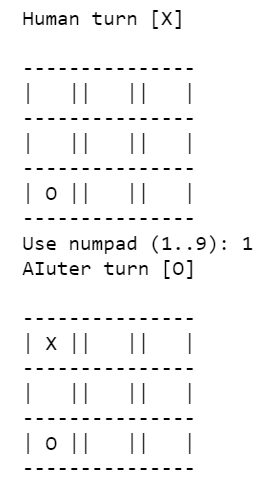
play()

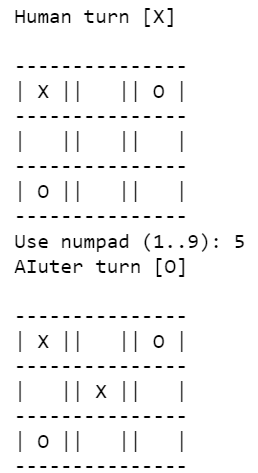
print()

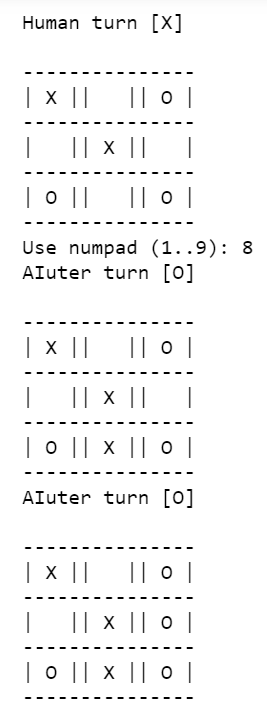
print("PARTH PATEL\n19DCS098")

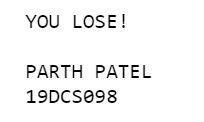
**OUTPUT:**











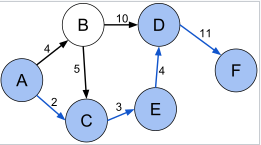
**CONCLUSION:**

By performing the above practical, we learnt the concept of minimax algorithm.

**PRACTICAL-9**

**AIM:**

Find out shortest path using Ant Colony Optimization for given Graph.



**PROGRAM CODE:**

import random as rn

import numpy as np

from numpy.random import choice as np\_choice

class AntColony(object):

def \_\_init\_\_(self, distances, n\_ants, n\_best, n\_iterations, decay, alpha=1, beta=1):

self.distances = distances

self.pheromone = np.ones(self.distances.shape) / len(distances)

self.all\_inds = range(len(distances))

self.n\_ants = n\_ants

self.n\_best = n\_best

self.n\_iterations = n\_iterations

self.decay = decay

self.alpha = alpha

self.beta = beta

def run(self):

shortest\_path = None

all\_time\_shortest\_path = ("placeholder", np.inf)

for i in range(self.n\_iterations):

all\_paths = self.gen\_all\_paths()

self.spread\_pheronome(all\_paths, self.n\_best, shortest\_path=shortest\_path)

shortest\_path = min(all\_paths, key=lambda x: x[1])

#print (shortest\_path)

if shortest\_path[1] < all\_time\_shortest\_path[1]:

all\_time\_shortest\_path = shortest\_path

self.pheromone = self.pheromone \* self.decay

return all\_time\_shortest\_path

def spread\_pheronome(self, all\_paths, n\_best, shortest\_path):

sorted\_paths = sorted(all\_paths, key=lambda x: x[1])

for path, dist in sorted\_paths[:n\_best]:

for move in path:

self.pheromone[move] += 1.0 / self.distances[move]

def gen\_path\_dist(self, path):

total\_dist = 0

for ele in path:

total\_dist += self.distances[ele]

return total\_dist

def gen\_all\_paths(self):

all\_paths = []

for i in range(self.n\_ants):

path = self.gen\_path(0)

all\_paths.append((path, self.gen\_path\_dist(path)))

return all\_paths

def gen\_path(self, start):

path = []

visited = set()

visited.add(start)

prev = start

for i in range(len(self.distances) - 1):

move = self.pick\_move(self.pheromone[prev], self.distances[prev], visited)

path.append((prev, move))

prev = move

visited.add(move)

path.append((prev, start)) # going back to where we started

return path

def pick\_move(self, pheromone, dist, visited):

pheromone = np.copy(pheromone)

pheromone[list(visited)] = 0

row = pheromone \*\* self.alpha \* (( 1.0 / dist) \*\* self.beta)

norm\_row = row / row.sum()

move = np\_choice(self.all\_inds, 1, p=norm\_row)[0]

return move

distances = np.array([[np.inf,4,2,14,5,20],

[4,np.inf,5,10,8,21],

[2,5,np.inf,7,3,18],

[9,10,7,np.inf,4,11],

[5,8,3,4,np.inf,15],

[20,21,18,11,15,np.inf]])

ant\_colony = AntColony(distances, 1, 1, 100, 0.95, alpha=1, beta=1)

shortest\_path = ant\_colony.run()

print ("shorted\_path: {}".format(shortest\_path))

**OUTPUT:**



**CONCLUSION:**

By performing the above practical, we learnt about solving to find Shortest Path using Best First Search Algorithm

**PRACTICAL-10**

**AIM:**

Write a program to solve Multi Objective Optimization problem Using Particle Swarm Optimization

**PROGRAM CODE:**

import numpy as np

from numpy import matlib

import matplotlib.pyplot as plt

import random as random

import math

def deleteOneRepositoryMember(rep , gamma):

gridindices = [item.gridIndex for item in rep]

OCells = np.unique(gridindices)

N = np.zeros(len(OCells))

for k in range(len(OCells)):

N[k] = gridindices.count(OCells[k])

# selection probablity

p = [math.exp(gamma\*item) for item in N]

p = np.array(p)/sum(p)

# select cell index

sci = roulettewheelSelection(p)

SelectedCell = OCells[sci]

#selected Cell members

selectedCellmembers = [item for item in gridindices if item == SelectedCell]

selectedmemberindex = np.random.randint(0,len(selectedCellmembers))

#selectedmember = selectedCellmembers[selectedmemberindex]

# delete memeber

#rep[selectedmemberindex] = []

rep = np.delete(rep, selectedmemberindex)

return rep.tolist()

def SelectLeader(rep , beta):

gridindices = [item.gridIndex for item in rep]

OCells = np.unique(gridindices) # ocupied cells

N = np.zeros(len(OCells))

for k in range(len(OCells)):

N[k] = gridindices.count(OCells[k])

# selection probablity

p = [math.exp(-beta\*item) for item in N]

p = np.array(p)/sum(p)

# select cell index

sci = roulettewheelSelection(p)

SelectedCell = OCells[sci]

#selected Cell members

selectedCellmembers = [item for item in gridindices if item == SelectedCell]

selectedmemberindex = np.random.randint(0,len(selectedCellmembers))

# selectedmember = selectedCellmembers[selectedmemberindex]

return rep[selectedmemberindex]

def roulettewheelSelection(p):

r = random.random()

cumsum = np.cumsum(p)

y = (cumsum<r)

x= [i for i in y if i==True]

return len(x)

def FindGridIndex(particle, grid):

nObj = len(particle.cost)

NGrid = len(grid[0].LowerBounds)

particle.gridSubIndex = np.zeros((1,nObj))[0]

for j in range(nObj):

index\_in\_Dim = len( [item for item in grid[j].UpperBounds if particle.cost[j]>item])

particle.gridSubIndex[j] = index\_in\_Dim

particle.gridIndex = particle.gridSubIndex[0]

for j in range(1,nObj):

particle.gridIndex = particle.gridIndex

particle.gridIndex = NGrid\*particle.gridIndex

particle.gridIndex = particle.gridIndex + particle.gridSubIndex[j]

return particle

def CreateGrid(pop,nGrid,alpha,nobj):

costs = [item.cost for item in pop]

Cmin = np.min(costs,axis=0)

Cmax = np.max(costs,axis=0)

deltaC = Cmax - Cmin

Cmin = Cmin - alpha\*deltaC

Cmax = Cmax + alpha\*deltaC

grid = [GridDim() for p in range(nobj)]

for i in range(nobj):

dimValues = np.linspace(Cmin[i],Cmax[i],nGrid+1).tolist()

grid[i].LowerBounds = [-float('inf')] + dimValues

grid[i].UpperBounds = dimValues + [float('inf')]

return grid

def Dominates(x,y):

x=np.array(x)

y=np.array(y)

x\_dominate\_y = all(x<=y) and any(x<y)

return x\_dominate\_y

def DetermineDomination(pop):

pop\_len= len(pop)

for i in range(pop\_len):

pop[i].IsDominated = False

for i in range(pop\_len-1):

for j in range(i+1,pop\_len):

if Dominates(pop[i].cost,pop[j].cost):

pop[j].IsDominated = True

if Dominates(pop[j].cost,pop[i].cost):

pop[i].IsDominated = True

return pop

# problem definition

def MOP2(x):

x = np.array(x)

n= len(x)

z1 = 1 - math.exp(-sum((x-1/math.sqrt(n))\*\*2))

z2 = 1 - math.exp(-sum((x+1/math.sqrt(n))\*\*2))

return [z1,z2]

costfunction = lambda x: MOP2(x)

nVar = 5 # number of decision vars

varMin = -4

varMax = 4

maxIt = 100

nPop = 200 # population size

nRep = 50 # size of repository

w = 0.5 # inertia wieght

c1 = 2 # personal learning coefficient

c2 = 2 # global learning coefficient

wdamping = 0.99

# ################ constriction coefficients

# phi1 = 2.05

# phi2 = 2.05

# phi = phi1+phi2

# chi = 2/(phi - 2 + np.sqrt(phi\*\*2 - 4\*phi))

# w = chi # inertia wieght

# c1 = chi\*phi1 # personal learning coefficient

# c2 = chi\*phi2 # global learning coefficient

# wdamping = 1

# #################

beta = 1 # leader selection pressure

gamma = 1 # deletion selection pressure

NoGrid = 5

alpha=0.1 # nerkhe tavarrom grid

# initialization

class Particle:

position = []

cost = []

velocity = []

best\_position = []

best\_cost = []

IsDominated = []

gridIndex = []

gridSubIndex = []

# for each objective a grid items is division of values of objective cost

class GridDim:

LowerBounds = []

UpperBounds = []

#Particles = np.matlib.repmat(Particle,nPop,1)

Particles = [Particle() for p in range(nPop)]

for i in range(nPop):

Particles[i].position = np.random.uniform(varMin,varMax,nVar)

Particles[i].velocity = np.zeros(nVar)

Particles[i].cost = costfunction(Particles[i].position)

# update best personal Best

Particles[i].best\_position = Particles[i].position

Particles[i].best\_cost = Particles[i].cost

Particles[i].IsDominated = False

Particles = DetermineDomination(Particles)

Repos = [item for item in Particles if item.IsDominated == False ]

nObj =len( Repos[0].cost)

grid = CreateGrid(Repos,NoGrid,alpha=0.1,nobj=nObj)

for r in range(len(Repos)):

Repos[r] = FindGridIndex(Repos[0],grid)

# MOPSO main loop

for it in range(maxIt):

for i in range(nPop):

leader = SelectLeader(Repos,beta)

# update velocity

Particles[i].velocity = w\*Particles[i].velocity \

+ c1\*np.random.rand(1,nVar)[0]\*(Particles[i].best\_position - Particles[i].position) \

+ c2\*np.random.rand(1,nVar)[0]\*(leader.position - Particles[i].position)

# update position

Particles[i].position = Particles[i].position + Particles[i].velocity

# evaluation

Particles[i].cost = costfunction(Particles[i].position)

if Dominates(Particles[i].cost,Particles[i].best\_cost):

Particles[i].best\_position = Particles[i].position

Particles[i].best\_cost = Particles[i].cost

else:

if np.random.rand() > 0.5:

Particles[i].best\_position = Particles[i].position

Particles[i].best\_cost = Particles[i].cost

Repos = Repos + Particles

Repos = DetermineDomination(Repos)

Repos = [item for item in Repos if item.IsDominated == False ]

grid = CreateGrid(Repos,NoGrid,alpha=0.1,nobj=nObj)

for r in range(len(Repos)):

Repos[r] = FindGridIndex(Repos[r],grid)

# check if repository is full

if len(Repos) > nRep :

extra = len(Repos) - nRep

for e in range(extra):

Repos = deleteOneRepositoryMember(Repos,gamma)

########## show figure ##########

plt.clf()

particlesCost = np.reshape( [item.cost for item in Particles ],newshape=(nPop,2))

repositoryCost = [item.cost for item in Repos]

repositoryCost = np.reshape( repositoryCost, newshape=(len(repositoryCost),2))

plt.plot(particlesCost[:,0], particlesCost[:,1], 'o' ,mfc='none')

plt.plot(repositoryCost[:,0], repositoryCost[:,1], 'r\*')

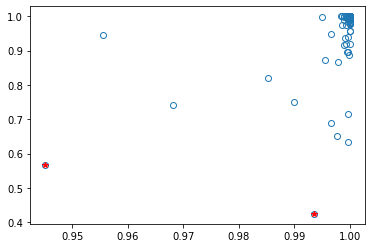
plt.draw()

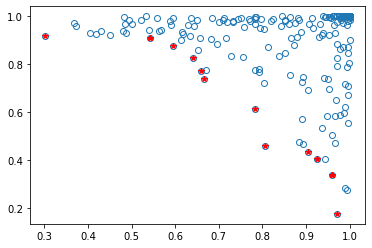
plt.pause(0.00000000001)

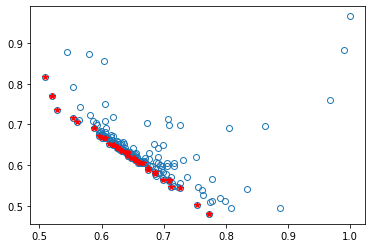
w=w\*wdamping

plt.show()

**OUTPUT:**







**CONCLUSION:**

By performing the above practical, we learnt about how to solve Multi Objective Optimization problem Using Particle Swarm Optimization