Exp 3

Uninformed

dls

Code:

graph={

'S':['A','B'],

'A':['C','D'],

'B':['I','J'],

'C':['E','F'],

'D':['G'],

'I':['H'],

'J':[]

}

def dls(start,goal,path,level,maxLimit):

print('\nCurrent level -->',level)

print('Goal node testing',start)

path.append(start)

if start==goal:

print('Test successfull goal found')

return path

print('Goal node test failed')

if level==maxLimit:

return False

print('Expanding current node:',start)

for child in graph[start]:

if dls(child, goal, path, level+1, maxLimit):

return path

return False

start='S'

goal=input('Enter goal:')

maxLimit=int(input("Enter max limit:"))

print()

path=list()

res=dls(start, goal, path, 0, maxLimit)

if(res):

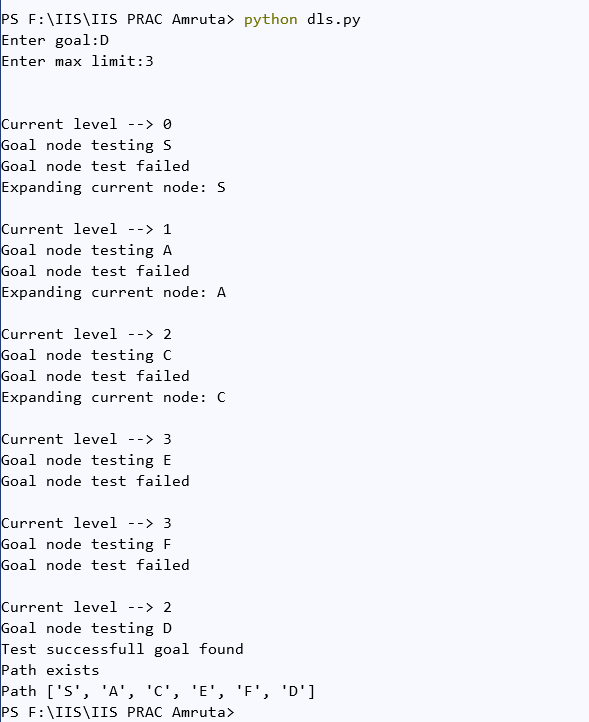
print('Path exists')

print('Path',path)

else:

print('Path doesnt exist')

Output



informed

A\*

Code:

from collections import deque

class Graph:

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

# heuristic function with equal values for all nodes

def h(self, n):

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

def a\_star\_algorithm(self, start\_node, stop\_node):

# open\_list is a list of nodes which have been visited, but who's neighbors

# haven't all been inspected, starts off with the start node

# closed\_list is a list of nodes which have been visited

# and who's neighbors have been inspected

open\_list = set([start\_node])

closed\_list = set([])

# g contains current distances from start\_node to all other nodes

# the default value (if it's not found in the map) is +infinity

g = {}

g[start\_node] = 0

# parents contains an adjacency map of all nodes

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

# find a node with the lowest value of f() - evaluation function

for v in open\_list:

if n == None or g[v] + self.h(v) < g[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

# for all neighbors of the current node do

for (m, weight) in self.get\_neighbors(n):

# if the current node isn't in both open\_list and closed\_list

# add it to open\_list and note n as it's parent

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

# otherwise, check if it's quicker to first visit n, then m

# and if it is, update parent data and g data

# and if the node was in the closed\_list, move it to open\_list

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

adjacency\_list = {

'A': [('B', 1), ('C', 3), ('D', 7)],

'B': [('D', 5)],

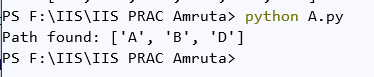
'C': [('D', 12)]

}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('A', 'D')

Output



Exp 4

Adversarial alpha beta pruning

Code:

# Initial values of Alpha and Beta

MAX, MIN = 1000, -1000

# Returns optimal value for current player

#(Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

# Terminating condition. i.e

# leaf node is reached

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

# Recur for left and right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

# Alpha Beta Pruning

if beta <= alpha:

break

return best

else:

best = MAX

# Recur for left and

# right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

# Alpha Beta Pruning

if beta <= alpha:

break

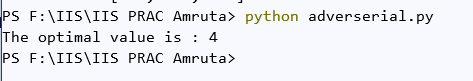
return best

if \_\_name\_\_ == "\_\_main\_\_":

values = [4, 2, 6, 19, 1, -2, 3, -1]

print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

Output



Exp 5

Genetic

Code:

import random

def score(parent1, parent2):

# doing crossover

for i in range(len(parent1)-1, len(parent1)-4, -1):

parent1[i], parent2[i] = parent2[i], parent1[i]

#doint mutation by randomly selecting the genes

mutation\_index = [random.randint(0, len(parent1)-1) for i in range(len(parent1)//2)]

for i in mutation\_index:

if parent1[i] == '0':

parent1[i] = '1'

else:

parent1[i] = '0'

if parent2[i] == '0':

parent2[i] = '1'

else:

parent2[i] = '0'

score1 = parent1.count('1')

score2 = parent2.count('1')

#checking which child is better with more gene of type1

if score1 > score2:

return [''.join(parent1), score1]

else:

return [''.join(parent2), score2]

def genetic\_algo():

# Taking input as no. of parents

n = int(input('Enter the number of parents: '))

parents = []

#taking parents genes as input 1 by 1

for i in range(n):

parents.append(list(input(f'Enter the parent{i+1}: ')))

results = []

#finding the score and storing it in results

for i in range(len(parents)):

for j in range(i+1, len(parents)):

arr = [parents[i].copy(), parents[j].copy()]

scores = score(parents[i], parents[j])

results.append(scores + arr)

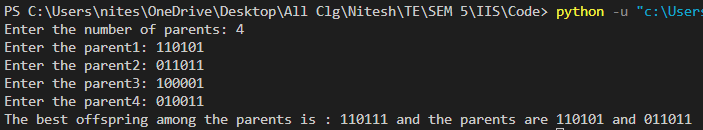
# finding the best score among all combination of parents

results.sort(key=lambda x: x[1], reverse=True)

print(f'The best offspring among the parents is : {results[0][0]} and the parents are {"".join(results[0][2])} and {"".join(results[0][3])}')

genetic\_algo()

Output:



Exp 6

Constraint

Code: graph coloring

colors = ['Red','Blue','Green']

states = ['Nagpur','Thane','Pune','Mumbai']

neighbors = {}

neighbors['Nagpur'] = ['Thane','Pune']

neighbors['Thane'] = ['Nagpur','Pune','Mumbai']

neighbors['Pune'] = ['Nagpur','Thane','Mumbai']

neighbors['Mumbai'] = ['Thane','Pune']

colors\_of\_states = {}

def promising(state, color):

for neighbor in neighbors.get(state):

color\_of\_neighbor = colors\_of\_states.get(neighbor)

if color\_of\_neighbor == color:

return False

return True

def get\_color\_for\_state(state):

for color in colors:

if promising(state, color):

return color

def main():

for state in states:

colors\_of\_states[state] = get\_color\_for\_state(state)

print(colors\_of\_states)

main()



Constraint

Code: N Queens

# Taking number of queens as input from user

N = int(input("Enter the number of queens: "))

# here we create a chessboard

# NxN matrix with all elements set to 0

board = [[0]\*N for \_ in range(N)]

def attack(i, j):

#checking vertically and horizontally if there are any queen placed

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

#checking diagonally if there are any queen placed

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queens(n):

if n==0:

return True

# here we are checking whether we can place queen at ith row and jth column

for i in range(0,N):

for j in range(0,N):

if (not(attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

if N\_queens(n-1)==True:

return True

board[i][j] = 0

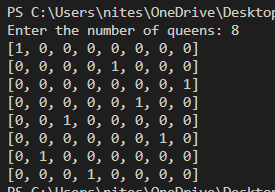
return False

N\_queens(N)

for i in board:

print (i)

Output



Exp 7

Local search

from random import \*

import random

import numpy

import copy

countCities = 20;

# 2D Array

cities = numpy.zeros(shape=(20,20))

# tour

hypothesis = [int]\*countCities

visitedCities = []

saveState = []

threshold = 2

lastFitness = 0

trials = 0

cityIndex = 1

# calculates fitness based on the difference between the distances

def getFitness(fitness, hypothesis, saveState, cities):

oldDistance = getDistance(cities, saveState)

print("Old Distance ",oldDistance,"km")

print("")

newDistance = getDistance(cities, hypothesis)

print("New Distance ",newDistance,"km")

print("")

if(oldDistance > newDistance):

fitness += 1

elif(oldDistance < newDistance):

fitness -= 1

return fitness

# choose random City at position cityIndex

def doRandomStep():

global visitedCities

global saveState

global hypothesis

if(len(visitedCities) >= countCities):

visitedCities.clear()

visitedCities.append(0)

randomNumbers = list(set(saveState) - set(visitedCities))

randomStep = random.choice(randomNumbers)

visitedCities.append(randomStep)

hypothesis.remove(randomStep)

hypothesis.insert(cityIndex,randomStep)

# next city

def increment():

global cityIndex

global visitedCities

if (cityIndex < countCities - 2):

cityIndex += 1

else:

visitedCities.clear()

cityIndex = 1

# calculates distance from tour

def getDistance(cities, hypothesis):

distance = 0

for i in range(countCities):

if (i < countCities-1):

distance += cities[hypothesis[i]][hypothesis[i+1]]

print("[",hypothesis[i],"]",distance,"km ",end="")

else:

print("[",hypothesis[i],"]")

return distance

if \_\_name\_\_ == '\_\_main\_\_':

for i in range(countCities):

hypothesis[i] = i

for j in range(countCities):

if (j > i):

cities[i][j] = randint(1,100)

elif(j < i):

cities[i][j] = cities[j][i]

print("=== START ===");

while(lastFitness < threshold):

print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

saveState = copy.deepcopy(hypothesis)

doRandomStep()

currentFitness = getFitness(lastFitness, hypothesis, saveState, cities)

print("Old fitness ",lastFitness)

print("Current fitness ",currentFitness)

if (currentFitness > lastFitness):

lastFitness = currentFitness

elif(currentFitness < lastFitness):

hypothesis = copy.deepcopy(saveState)

if(trials < 3):

increment()

else:

trials = 0

visitedCities.append(saveState[cityIndex])

Hill Climbing

Code:

import numpy as np

def find\_neighbours(state, landscape):

neighbours = []

dim = landscape.shape

# left neighbour

if state[0] != 0:

neighbours.append((state[0] - 1, state[1]))

# right neighbour

if state[0] != dim[0] - 1:

neighbours.append((state[0] + 1, state[1]))

# top neighbour

if state[1] != 0:

neighbours.append((state[0], state[1] - 1))

# bottom neighbour

if state[1] != dim[1] - 1:

neighbours.append((state[0], state[1] + 1))

# top left

if state[0] != 0 and state[1] != 0:

neighbours.append((state[0] - 1, state[1] - 1))

# bottom left

if state[0] != 0 and state[1] != dim[1] - 1:

neighbours.append((state[0] - 1, state[1] + 1))

# top right

if state[0] != dim[0] - 1 and state[1] != 0:

neighbours.append((state[0] + 1, state[1] - 1))

# bottom right

if state[0] != dim[0] - 1 and state[1] != dim[1] - 1:

neighbours.append((state[0] + 1, state[1] + 1))

return neighbours

# Current optimization objective: local/global maximum

def hill\_climb(curr\_state, landscape):

neighbours = find\_neighbours(curr\_state, landscape)

bool

ascended = False

next\_state = curr\_state

for neighbour in neighbours: #Find the neighbour with the greatest value

if landscape[neighbour[0]][neighbour[1]] > landscape[next\_state[0]][next\_state[1]]:

next\_state = neighbour

ascended = True

return ascended, next\_state

def \_\_main\_\_():

landscape = np.random.randint(1, high=50, size=(10, 10))

print(landscape)

start\_state = (3, 6) # matrix index coordinates

current\_state = start\_state

count = 1

ascending = True

while ascending:

print("\nStep #", count)

print("Current state coordinates: ", current\_state)

print("Current state value: ", landscape[current\_state[0]][current\_state[1]])

count += 1

ascending, current\_state = hill\_climb(current\_state, landscape)

print("\nStep #", count)

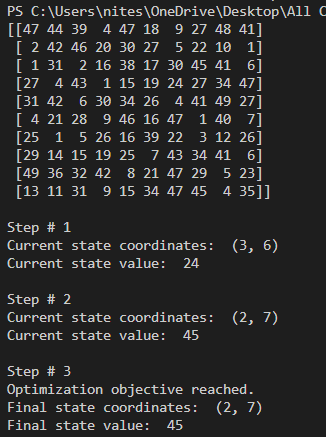
print("Optimization objective reached.")

print("Final state coordinates: ", current\_state)

print("Final state value: ", landscape[current\_state[0]][current\_state[1]])

\_\_main\_\_()

Output



MinMax

Code:

import math

def fun\_minmax(cd, node, maxt, scr, td):

if(cd == td):

return scr[node]

if(maxt):

return max(fun\_minmax(cd+1, node\*2, False, scr, td),fun\_minmax(cd+1, node\*2+1, False, scr, td))

else:

return min(fun\_minmax(cd+1, node\*2, True, scr, td),fun\_minmax(cd+1, node\*2+1, True, scr, td))

scr = []

x =int(input("Enter total number of leaf Node = "))

for i in range(x):

y = int(input("Enter leaf value: "))

scr.append(y)

td = math.log(len(scr), 2)

cd = int(input("Enter current depth value: "))

nodev = int(input("Enter node value: "))

maxt = True

print("The answer is: ", end=" ")

answer = fun\_minmax(cd, nodev, maxt, scr, td)

print(answer)

Output

