BFS:

visited = []

queue = []

found=0

def bfs(visited, graph, node, goal\_state):

  visited.append(node)

  queue.append(node)

  while queue:          # Creating loop to visit each node

    m = queue.pop(0)

    print (m, end = " ")

    if(m==goal\_state):

      print("GOAL", end= " ")

      global found

      found=1

      break

    for neighbour in graph[m]:

      if neighbour not in visited:

        visited.append(neighbour)

        queue.append(neighbour)

# graph = {

#   '5' : ['3','7'],

#   '3' : ['2', '4'],

#   '7' : ['8'],

#   '2' : [],

#   '4' : ['8'],

#   '8' : []

# }

graph = {

  '1' : ['2','3','4'],

  '2' : ['5', '6'],

  '3' : ['7'],

  '4' : ['8','9'],

  '5' : [],

  '6' : [],

  '7' : [],

  '8' : [],

  '9' : []

}

print("Following is the Breadth-First Search")

bfs(visited, graph, '1','9')

if found==0:

  print("Not found")

DFS:

def dfs(graph, node,path = []):

    if node not in path:

        path.append(node)

        for neighbour in graph[node]:

            path = dfs(graph, neighbour, path)

    return path

graph = {

    '5' : ['3','7'],

    '3' : ['2', '4'],

    '7' : ['8'],

    '2' : [],

    '4' : ['8'],

    '8' : []

}

#       5

#     /   \

#    3     7

#  /   \    \

# 2     4----8

print("Depth-First Search traversal sequence")

path = dfs(graph, "5")

goal\_state = '8'

new\_path=[]

for item in path:

    if(item==goal\_state):

        break

    else:

        new\_path.append(item)

new\_path.append("GOAL FOUND🎊")

print(" ".join(new\_path))

DFID:

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self,vertices):

        self.V = vertices # Number of vertices

        self.graph = defaultdict(list)

    def addEdge(self,u,v):

        self.graph[u].append(v)

    def DLS(self,src,target,maxDepth):

        if src == target : return True

        if maxDepth <= 0 : return False

        for i in self.graph[src]:

            if(self.DLS(i,target,maxDepth-1)):

                return True

        return False

    def IDDFS(self,src, target, maxDepth):

        for i in range(maxDepth):

            if (self.DLS(src, target, i)):

                return True

        return False

# Create a graph

#        0

#      /   \

#     1      2

#    / \     | \

#   3   4   5   6

g = Graph (7)

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

target = 3

maxDepth = 1

src = 0

if g.IDDFS(src, target, maxDepth) == True:

    print ("Target "+str(target) +" is reachable from source within max depth of "+str(maxDepth))

else :

    print ("Target "+str(target) +" is NOT reachable from source within max depth of "+str(maxDepth))

A-star:

class Graph:

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

        self.adjacency\_list = adjacency\_list

    def get\_neighbors(self, v):

        return self.adjacency\_list[v]

    def h(self, n):

        H={

            'A':1,

            'B':1,

            'C':1,

            'D':1

        }

        return H[n]

    def a\_star(self, start, stop):

        open = set([start])

        closed = set([])

        g = {}

        g[start] = 0

        parents = {}

        parents[start] = start

        while len(open)>0:

            n = None

            print("Open List: ",open,"Closed List: ",closed)

            for v in open:

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

                    n = v

            if n==None:

                print("not found")

                return None

            if n==stop:

                open.remove(stop)

                closed.add(stop)

                print("Open List: ",open,"Closed List: ",closed)

                path=[]

                while parents[n]!=n:

                    path.append(n)

                    n = parents[n]

                path.append(start)

                path.reverse()

                return path

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

                if m not in open and m not in closed:

                    open.add(m)

                    g[m] = g[n]+weight

                    parents[m] = n

                else:

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

                        g[m] = g[n]+weight

                        parents[m]=n

                        if m in closed:

                            open.add(m)

                            closed.remove(m)

            open.remove(n)

            closed.add(n)

        print("path doesnt exist")

        return None

adjacency\_list = {

    'A': [('B',1),('C',3),('D',7)],

    'B': [('D',5)],

    'C': [('D',12)],

}

g = Graph(adjacency\_list)

g.a\_star('A','D')

Hill Climbing:

import copy

visited\_states = []

def heuristic(curr\_state,goal\_state):

    goal\_=goal\_state[3]

    val=0

    for i in range(len(curr\_state)):

        check\_val=curr\_state[i]

        if len(check\_val)>0:

            for j in range(len(check\_val)):

                if check\_val[j]!=goal\_[j]:

                    val-=j

                else:

                    val+=j

    return val

def generate\_next(curr\_state,prev\_heu,goal\_state):

    global visited\_states

    state = copy.deepcopy(curr\_state)

    for i in range(len(state)):

        temp = copy.deepcopy(state)

        if len(temp[i]) > 0:

            elem = temp[i].pop()

            for j in range(len(temp)):

                temp1 = copy.deepcopy(temp)

                if j != i:

                    temp1[j] = temp1[j] + [elem]

                    if (temp1 not in visited\_states):

                        curr\_heu=heuristic(temp1,goal\_state)

                        if curr\_heu>prev\_heu:

                            child = copy.deepcopy(temp1)

                            return child

    return 0

def solution\_(init\_state,goal\_state):

    global visited\_states

    if (init\_state == goal\_state):

        print (goal\_state)

        print("solution found!")

        return

    current\_state = copy.deepcopy(init\_state)

    while(True):

        visited\_states.append(copy.deepcopy(current\_state))

        print(current\_state)

        prev\_heu=heuristic(current\_state,goal\_state)

        child = generate\_next(current\_state,prev\_heu,goal\_state)

        if child==0:

            print("Final state - ",current\_state)

            return

        current\_state = copy.deepcopy(child)

def solver():

    global visited\_states

    init\_state = [[],[],[],['B','C','D','A']]

    goal\_state = [[],[],[],['A','B','C','D']]

    solution\_(init\_state,goal\_state)

solver()

Genetic:

import random

def generate\_population(population\_size, gene\_length):

    return [''.join(random.choice('01') for \_ in range(gene\_length)) for \_ in range(population\_size)]

def calculate\_fitness(individual, target):

    return sum(1 for a, b in zip(individual, target) if a == b)

def select\_parents(population, target):

    fitness\_scores = [calculate\_fitness(individual, target) for individual in population]

    total\_fitness = sum(fitness\_scores)

    if total\_fitness == 0:

        return random.sample(population, 2)

    probabilities = [score / total\_fitness for score in fitness\_scores]

    parents = random.choices(population, probabilities, k=2)

    return parents

def crossover(parent1, parent2):

    crossover\_point = random.randint(0, len(parent1) - 1)

    child1 = parent1[:crossover\_point] + parent2[crossover\_point:]

    child2 = parent2[:crossover\_point] + parent1[crossover\_point:]

    return child1, child2

def mutate(individual, mutation\_rate):

    mutated\_individual = ''.join(

        bit if random.random() > mutation\_rate else random.choice('01')

        for bit in individual

    )

    return mutated\_individual

def genetic\_algorithm(target, population\_size, mutation\_rate, generations):

    population = generate\_population(population\_size, len(target))

    for generation in range(generations):

        population = sorted(population, key=lambda x: calculate\_fitness(x, target), reverse=True)

        print(f"Generation {generation}: {population[0]} (Fitness: {calculate\_fitness(population[0], target)})")

        if calculate\_fitness(population[0], target) == len(target):

            print("Target achieved!")

            break

        new\_population = []

        for \_ in range(population\_size // 2):

            parent1, parent2 = select\_parents(population, target)

            child1, child2 = crossover(parent1, parent2)

            child1 = mutate(child1, mutation\_rate)

            child2 = mutate(child2, mutation\_rate)

            new\_population.extend([child1, child2])

        population = new\_population

    if calculate\_fitness(population[0], target) != len(target):

        print("Nearest match reached is: ", population[0])

        binary\_string = str(population[0])

        decimal\_number\_again = int(binary\_string, 2)

        print(decimal\_number\_again)

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

    num = int(input("Enter a number: "))

    binNum = bin(num).replace("0b","")

    print(binNum)

    target\_string = str(binNum)

    population\_size = 100

    mutation\_rate = 0.01

    generations = 1000

    genetic\_algorithm(target\_string, population\_size, mutation\_rate, generations)

Perceptron

import numpy as np

X1 = np.array([1, -2, 0, -1])

X2 = np.array([0, 1.5, -0.5, -1])

X3 = np.array([-1, 1, 0.5, -1])

X = np.array([X1, X2, X3])

W = np.array([1, -1, 0, 0.5])

d = np.array([-1, -1, 1])

c = 0.1

epochs = 1

for i in range(epochs):

    print("Iteration ", i+1)

    for j in range(len(X)):

        net = np.dot(X[j], W)

        if (net <= 0):

            op = -1

        elif net > 0:

            op = 1

        error = d[j] - op

        dW = c\*error\*X[j]

        W += dW

        print("W", j,  W)

    print("\nW after ", i+1, " epochs ", W)

    # c=c/2

print("Final W after ", epochs, "epochs:")

print(W)

prolog:

parent(sunil, prerna).

parent(sunil, diksha).

parent(sunil, krishna).

parent(swati, prerna).

parent(swati, diksha).

parent(swati, krishna).

parent(sujit, kajal).

parent(sujit, yash).

parent(malti, kajal).

parent(malti, yash).

parent(sulochana, sunil).

parent(sulochana, sujit).

parent(ram, sunil).

parent(ram, sujit).

female(prerna).

female(diksha).

female(kajal).

female(swati).

female(malti).

female(sulochana).

male(ram).

male(sunil).

male(sujit).

male(yash).

male(krishna).

/\*Rules\*/

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

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

sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X).

brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X).

grandmother(X,Y) :- parent(Z,Y), parent(X,Z), female(X).

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

A screenshot of a computer

Description automatically generated