Experiment 2:

Code:

For all of the methods we use this graph

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

BFS

visited = []

queue = []

def bfs(graph,start):

if len(graph )== 0:

print("graph empty")

return

else:

# visited.append(start)

queue.append(start)

# while len(queue) > 0:

for i in queue:

print(i," - ")

if i in visited:

print(i,"is in graph but also visited")

print("visited: ",visited)

pass

else:

visited.append(i)

if i in graph:

print(i," is inside the graph")

el = graph[i]

for a in el:

queue.append(a)

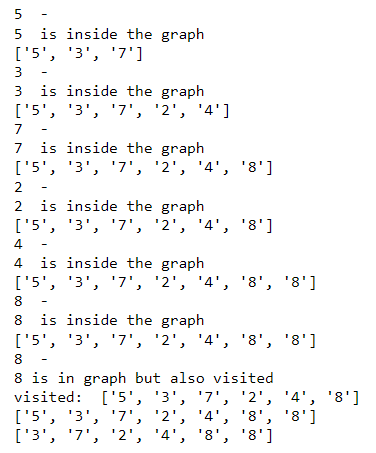
print(queue)

queue.pop(0)

print(queue)

bfs(graph, '5')

Output:



The BFS algorithm iteratively goes to different depths and first explores all the nodes at that depth

DFS:

visited=[]

queue=[]

closed\_list=[]

def dfs(visited, graph,node,goal):

path={}

path[node]=node

root=[]

queue.append(node)

print(f'Open List: {queue}\nClosed List: {closed\_list}\n')

while queue:

m=queue.pop()

closed\_list.append(m)

print(f'Open List: {queue}\nClosed List: {closed\_list}\n')

if(m==goal):

while path[m]!=m:

root.append(m)

m=path[m]

root.append(m)

root.reverse()

print(f"\nPath: {root}")

return

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

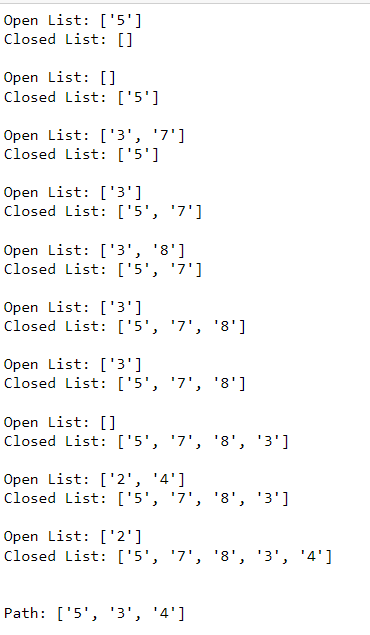
path[neighbour]=m

print(f'Open List: {queue}\nClosed List: {closed\_list}\n')

print("Path does not found")

dfs(visited,graph,'5','4')

output



In DFS we go to the leaf node for each branch by exploring the all the nodes till the max depth in a branch

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)

for i in range(2):

target = [5, 6]; maxDepth = [3,2]; src = 0

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

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

else :

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

Output:



Experiment 3:

Code:

import queue

class Node:

def \_\_init\_\_(self, state, grid, parent=None, cost=0, heuristic=0):

self.state = state

self.parent = parent

self.cost = cost

self.heuristic = heuristic

self.grid = grid

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

return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def manhattan\_distance(state, goal\_state):

x1, y1 = state

x2, y2 = goal\_state

return abs(x1 - x2) + abs(y1 - y2)

def get\_neighbors(state, grid):

x, y = state

neighbors = [(x+1, y), (x-1, y), (x, y+1), (x, y-1), (x+1,y+1), (x-1,y+1), (x+1,y-1), (x-1,y-1)] # Possible moves: right, left, up, down

valid\_neighbors = [(nx, ny) for nx, ny in neighbors if 0 <= nx < grid\_width and 0 <= ny < grid\_height]

valid\_neighbors = [(nx, ny) for nx, ny in valid\_neighbors if grid[nx][ny] == 0]

return [(neighbor, 1) for neighbor in valid\_neighbors] # Assuming uniform cost for all moves

def astar\_search(initial\_state, goal\_state, heuristic, grid):

open\_list = queue.PriorityQueue()

visited = set()

start\_node = Node(initial\_state, grid, None, 0, heuristic(initial\_state, goal\_state))

open\_list.put(start\_node)

while not open\_list.empty():

current\_node = open\_list.get()

if current\_node.state == goal\_state:

return reconstruct\_path(current\_node)

visited.add(current\_node.state)

for neighbor\_state, cost in get\_neighbors(current\_node.state, grid):

if neighbor\_state not in visited:

neighbor\_node = Node(neighbor\_state, grid, current\_node, current\_node.cost + cost, heuristic(neighbor\_state, goal\_state))

open\_list.put(neighbor\_node)

return None # If no path is found

def reconstruct\_path(node):

path = []

while node:

path.append(node.state)

node = node.parent

return path[::-1]

# Example usage:

grid\_width = 5

grid\_height = 5

initial\_state = (0, 0) # Initialize your initial state

goal\_state = (4, 4) # Initialize your goal state

grid = [

[0, 1, 0, 1, 1],

[1, 0, 1, 0, 1],

[0, 1, 1, 1, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 0, 0]

]

path = astar\_search(initial\_state, goal\_state, manhattan\_distance, grid)

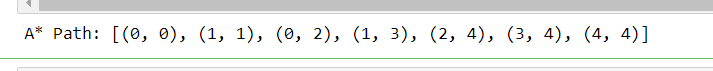
if path:

print("A\* Path:", path)

else:

print("No path found.")

Output:



Experiment 4:

CODE:

import copy

visited\_states = []

def heuristic(currentState, goalState):

goal = goalState[3]

val = 0

for i in range(len(currentState)):

checkValue = currentState[i]

if len(checkValue) > 0:

for j in range(len(checkValue)):

if checkValue[j] != goal[j]:

val -= j

else:

val += j

return val

def generate\_next(currentState, previousH, goalState):

global visited\_states

state = copy.deepcopy(currentState)

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:

currentH = heuristic(temp1, goalState)

if currentH > previousH:

child = copy.deepcopy(temp1)

return child

return 0

def solution(initialState, goalState):

global visited\_states

if initialState == goalState:

print(goalState)

print('Solution found.')

return

current\_state = copy.deepcopy(initialState)

while True:

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

previousH = heuristic(current\_state, goalState)

print(current\_state, ':', previousH)

child = generate\_next(current\_state, previousH, goalState)

if child == 0:

print('Final state is - ', current\_state)

return

current\_state = copy.deepcopy(child)

def solver():

global visited\_states

initialState = [[], [], [], ['B','C', 'D', 'A']]

goalState = [[], [], [], ['A', 'B','C', 'D']]

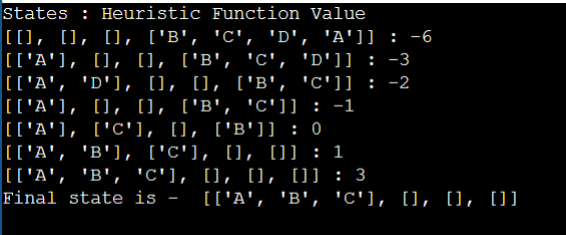
print('States : Heuristic Function Value')

solution(initialState, goalState)

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

solver()

Output:



Experiment 5:

Code:

from random import randint

def selection(li):

dec = list(map(lambda x: int(x, 2), li))

fit = list(map(lambda x: x \* x, dec))

s = sum(fit)

prob = list(map(lambda x: round(x / s, 3), fit))

avg = s / n

exe = list(map(lambda x: round(x / avg, 3), fit))

ac = list(map(lambda x: round(x), exe))

return dec, fit, prob, exe, ac

def pp(li, ac, n):

co = []

temp = []

index = []

for i in range(n):

if ac[i] == 1:

co.append(li[i])

elif ac[i] >= 2:

for j in range(ac[i] - 1):

temp.append(li[i])

co.append(li[i])

elif ac[i] == 0 and len(temp) != 0:

co.append(temp[0])

temp.pop(0)

elif ac[i] == 0 and len(temp) == 0:

index.append(i)

if len(index) != 0 and len(temp) != 0:

for i in index:

co.insert(i, temp[0])

temp.pop(0)

elif len(index) != 0 and len(temp) == 0:

co.insert(i, li[i])

return co

def cr(x):

s = 0

for i in x:

if i == '1':

s = s + 1

return s

def crossing(li, n):

crossed = []

for i in range(0, n, 2):

temp1 = li[i]

j = i + 1

temp2 = li[j]

crosspoint = cr(temp1)

print('The crossover for pair ' + str(i) + ' is ' + str(crosspoint))

temp3 = temp1[crosspoint:]

temp4 = temp2[crosspoint:]

temp1 = temp1[0:crosspoint] + temp4

temp2 = temp2[0:crosspoint] + temp3

crossed.append(temp1)

crossed.append(temp2)

return crossed

def mutation(li, n):

mut = []

for i in li:

j = randint(0, n - 1)

print('For gene ' + str(i) + ', the bit that will be changed is ' + str(j))

if i[j] == '1':

i = i[0:j] + '0' + i[j + 1:]

elif i[j] == '0':

i = i[0:j] + '1' + i[j + 1:]

mut.append(i)

return mut

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

sam = []

for i in range(n):

sam.append(input('Enter gene: '))

m = int(input('Enter the number of generations: '))

crossed = sam.copy()

for i in range(m):

dec, fit, prob, exe, ac = selection(crossed)

s = sum(ac)

if s < n:

maxi = max(ac)

k = ac.index(maxi-1)

ac[k] += 1

if s > n:

maxi = max(ac)

k = ac.index(maxi)

ac[k] -= 1

print('---------------------------------------- GENERATION', i, ' ----------------------------------------')

print("Initial Population\tX Value\tFitness Value\tPopulation\tExpectedCount\tActual Count")

for j in range(n):

print(crossed[j], '\t\t', dec[j], '\t\t', fit[j], '\t', prob[j], '\t\t', exe[j], '\t\t', ac[j])

co = pp(crossed, ac, n)

print('\nThe selected genes for crossover:\n', co)

crossed = crossing(co, n)

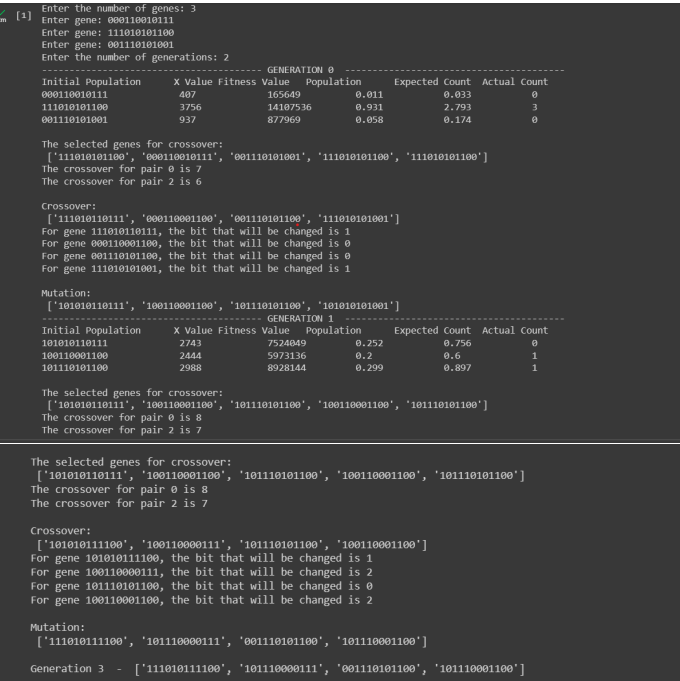
print('\nCrossover:\n', crossed)

crossed = mutation(crossed, n)

print('\nMutation:\n', crossed)

print('\nGeneration', (m + 1), ' - ', crossed)

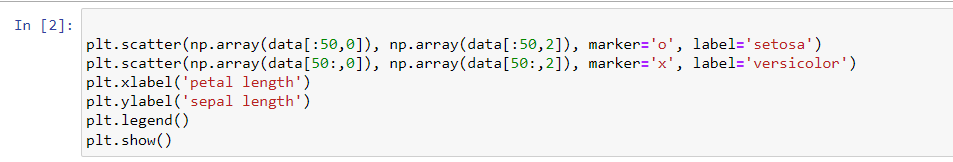
Output:

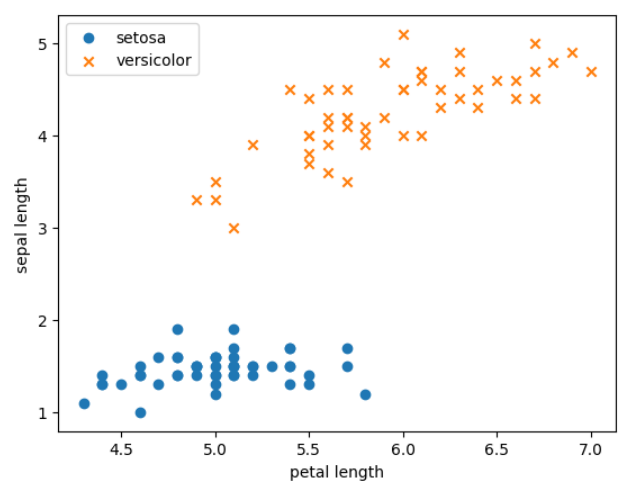


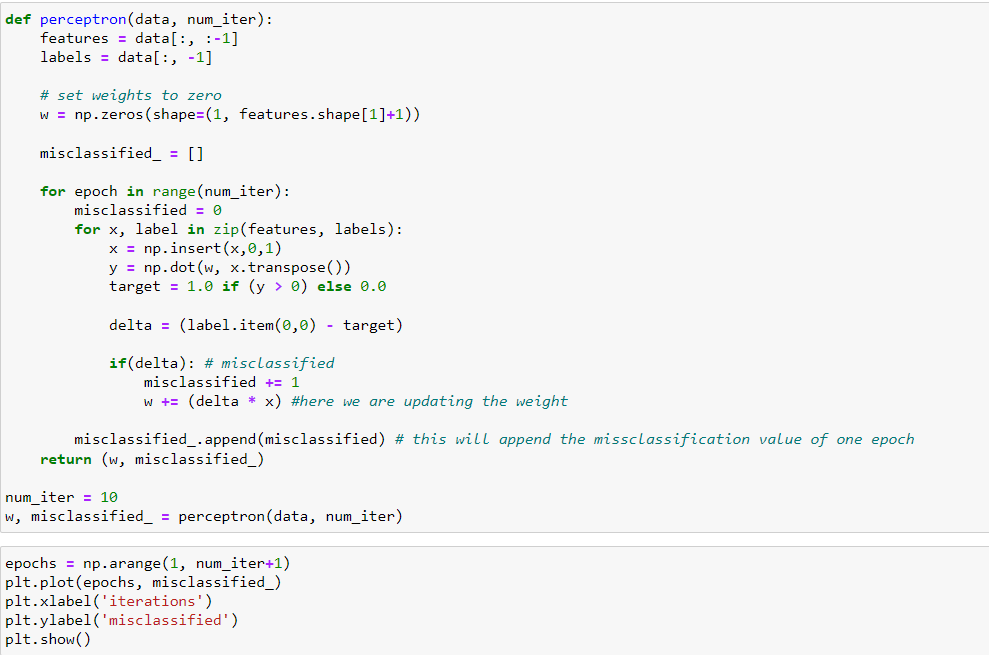
Experiment 6:

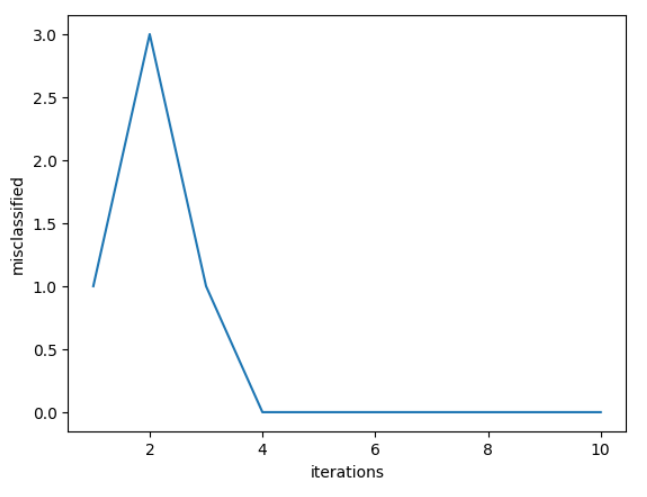
Code:











Experiment7:

**Code:**

#include<stdio.h>

#include<math.h>

#include<stdlib.h>

#include<stdbool.h>

int x[100],n;

bool isSafe(int k, int i)

{

int j;

for(j=1; j<=k-1; j++)

{

if(x[j] == i || abs(x[j] - i) == abs(j-k) )

{

return false;

}

}

return true;

}

void nQueen(int k)

{

int i,j;

for(i=1; i<= n; i++)

{

if(isSafe(k,i))

{

x[k] = i;

if(k == n)

{

for(j=1; j<=n; j++)

{

printf("%d\t",x[j]);

}

printf("\n");

}

else

{

nQueen(k+1);

}

}

}

}

void main()

{

int i;

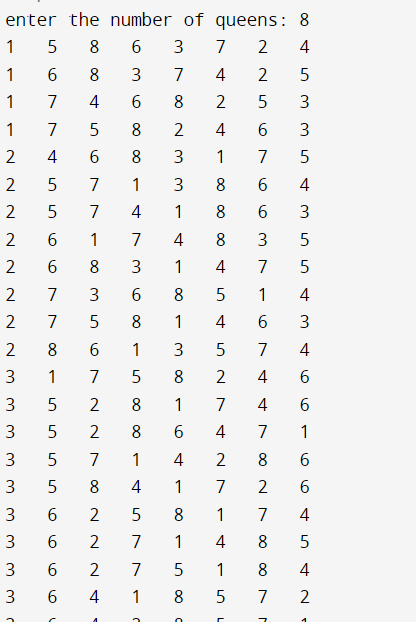
printf("enter the number of queens: ");

scanf("%d",&n);

nQueen(1);

}

**Output:**

****

**Experiment 8:**

**Code:**

female(pam).

female(liz).

female(pat).

female(ann).

male(jim).

male(bob).

male(tom).

male(peter).

parent(pam,bob).

parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,jim).

parent(bob,peter).

parent(peter,jim).

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

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

haschild(X):- parent(X,\_).

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

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

Output:

