**1.WAP to implement Tic-Tac\_toe game problem.**

import random

class TicTacToe:

def \_\_init\_\_(self):

self.board = []

def create\_board(self):

for i in range(3):

row = []

for j in range(3):

row.append('-')

self.board.append(row)

def get\_random\_first\_player(self):

return random.randint(0, 1)

def fix\_spot(self, row, col, player):

self.board[row][col] = player

def is\_player\_win(self, player):

win = None

n = len(self.board)

# checking rows

for i in range(n):

win = True

for j in range(n):

if self.board[i][j] != player:

win = False

break

if win:

return win

# checking columns

for i in range(n):

win = True

for j in range(n):

if self.board[j][i] != player:

win = False

break

if win:

return win

# checking diagonals

win = True

for i in range(n):

if self.board[i][i] != player:

win = False

break

if win:

return win

win = True

for i in range(n):

if self.board[i][n - 1 - i] != player:

win = False

break

if win:

return win

return False

for row in self.board:

for item in row:

if item == '-':

return False

return True

def is\_board\_filled(self):

for row in self.board:

for item in row:

if item == '-':

return False

return True

def swap\_player\_turn(self, player):

return 'X' if player == 'O' else 'O'

def show\_board(self):

for row in self.board:

for item in row:

print(item, end=" ")

print()

def start(self):

self.create\_board()

player = 'X' if self.get\_random\_first\_player() == 1 else 'O'

while True:

print(f"Player {player} turn")

self.show\_board()

# taking user input

row, col = list(

map(int, input("Enter row and column numbers to fix spot: ").split()))

print()

# fixing the spot

self.fix\_spot(row - 1, col - 1, player)

# checking whether current player is won or not

if self.is\_player\_win(player):

print(f"Player {player} wins the game!")

break

# checking whether the game is draw or not

if self.is\_board\_filled():

print("Match Draw!")

break

# swapping the turn

player = self.swap\_player\_turn(player)

# showing the final view of board

print()

self.show\_board()

# starting the game

tic\_tac\_toe = TicTacToe()

tic\_tac\_toe.start()

Output:

Player X turn

- - -

- - -

- - -

Enter row and column numbers to fix spot: 1 1

Player O turn

X - -

- - -

- - -

Enter row and column numbers to fix spot: 2 1

Player X turn

X - -

O - -

- - -

Enter row and column numbers to fix spot: 1 2

Player O turn

X X -

O - -

- - -

Enter row and column numbers to fix spot: 1 3

Player X turn

X X O

O - -

- - -

Enter row and column numbers to fix spot: 2 2

Player O turn

X X O

O X -

- - -

Enter row and column numbers to fix spot: 3 3

Player X turn

X X O

O X -

- - O

Enter row and column numbers to fix spot: 3 2

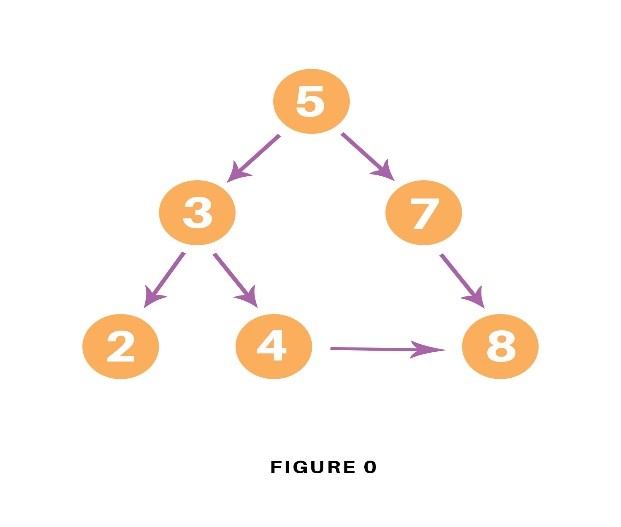
Player X wins the game!

X X O

O X -

- X O

**2.WAP to implement BFS**



### BFS pseudocode

The pseudocode for BFS in python goes as below:

**create a queue Q**

**mark v as visited and put v into Q**

**while Q is non-empty**

**remove the head u of Q**

**mark and enqueue all (unvisited) neighbors of u**

Program:

graph = {

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

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

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] *# List for visited nodes.*

queue = [] *#Initialize a queue*

**def** bfs(visited, graph, node): *#function for BFS*

visited.append(node)

queue.append(node)

**while** queue: *# Creating loop to visit each node*

m = queue.pop(0)

**print** (m, end = " ")

**for** neighbour **in** graph[m]:

**if** neighbour **not** **in** visited:

visited.append(neighbour)

queue.append(neighbour)

*# Driver Code*

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

bfs(visited, graph, '5') *# function calling*

In the above code, first, we will create the graph for which we will use the breadth-first search. After creation, we will create two lists, one to store the visited node of the graph and another one for storing the nodes in the queue.

After the above process, we will declare a function with the parameters as visited nodes, the graph itself and the node respectively. And inside a function, we will keep appending the visited and queue lists.

Then we will run the while loop for the queue for visiting the nodes and then will remove the same node and print it as it is visited.

At last, we will run the for loop to check the not visited nodes and then append the same from the visited and queue list.

As the driver code, we will call the user to define the bfs function with the first node we wish to visit.

### Output

The output of the above code will be as follow:

 Following is the Breadth-First Search  
 5 3 7 2 4 8

**3. 2.WAP to implement DFS**

### **DFS pseudocode**

The pseudocode for Depth-First Search in python goes as below: In the init() function, notice that we run the DFS function on every node because many times, a graph may contain two different disconnected part and therefore to make sure that we have visited every vertex, we can also run the DFS algorithm at every node.

**DFS(G, u)**

**u.visited = true**

**for each v ∈ G.Adj[u]**

**if v.visited == false**

**DFS(G,v)**

**init() {**

**For each u ∈ G**

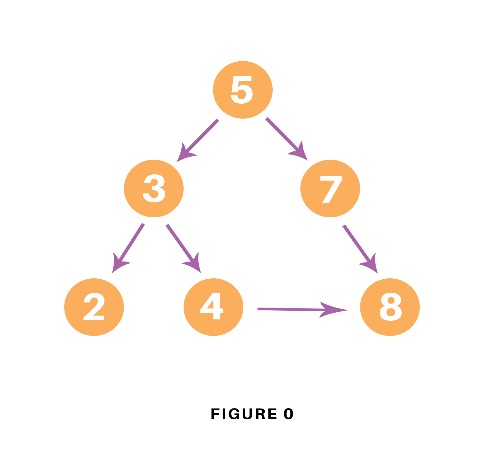
**u.visited = false**

**For each u ∈ G**

**DFS(G, u)**

**}**

## **DFS Implementation in Python (Source Code)**

Consider the following graph which is implemented in the code below:

# Using a Python dictionary to act as an adjacency list

graph = {

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

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

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

**def** **dfs**(visited, graph, node): #function for dfs

**if** node **not** **in** visited:

**print** (node)

visited.add(node)

**for** neighbour **in** graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

**print**("Following is the Depth-First Search")

dfs(visited, graph, '5')

In the above code, first, we will create the graph for which we will use the depth-first search. After creation, we will create a set for storing the value of the visited nodes to keep track of the visited nodes of the graph.

After the above process, we will declare a function with the parameters as visited nodes, the graph itself and the node respectively. And inside the function, we will check whether any node of the graph is visited or not using the “if” condition. If not, then we will print the node and add it to the visited set of nodes.

Then we will go to the neighboring node of the graph and again call the DFS function to use the neighbor parameter.

At last, we will run the driver code which prints the final result of DFS by calling the DFS the first time with the starting vertex of the graph.

### **Output**

The output of the above code is as follow:

 Following is the Depth-First Search  
5 3 2 4 8 7

**5.WAP to solve hill climbing problem**

def hill\_climbing(f, x0):

x = x0 # initial solution

while True:

neighbors = generate\_neighbors(x) # generate neighbors of x

# find the neighbor with the highest function value

best\_neighbor = max(neighbors, key=f)

if f(best\_neighbor) <= f(x): # if the best neighbor is not better than x, stop

return x

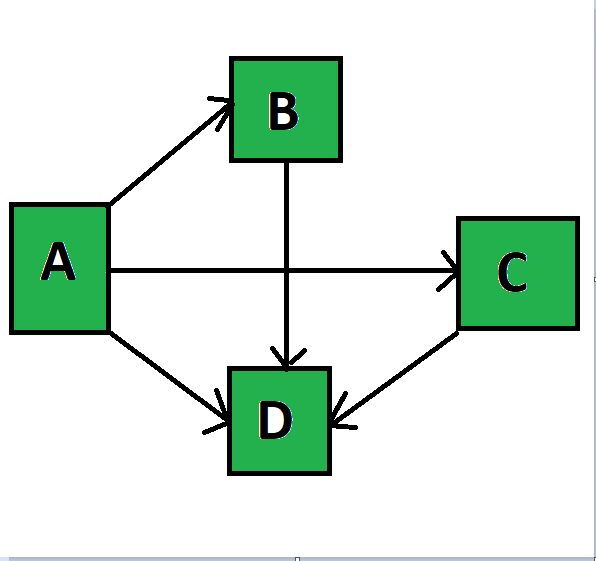
x = best\_neighbor # otherwise, continue with the best neighbor

**7.WAP to implement A\* algorithm**

## Pseudo-code of A\* algorithm

let openList equal empty list of nodes let closedList equal empty list of nodes put startNode on the openList (leave it's f at zero) while openList is not empty let currentNode equal the node with the least f value remove currentNode from the openList add currentNode to the closedList if currentNode is the goal You've found the exit! let children of the currentNode equal the adjacent nodes for each child in the children if child is in the closedList continue to beginning of for loop child.g = currentNode.g + distance b/w child and current child.h = distance from child to end child.f = child.g + child.h if child.position is in the openList's nodes positions if child.g is higher than the openList node's g continue to beginning of for loop add the child to the openList

## A\* Algorithm code for Graph



Code:

from collections import deque

class Graph:

    def \_\_init\_\_(self, adjac\_lis):

        self.adjac\_lis = adjac\_lis

    def get\_neighbors(self, v):

        return self.adjac\_lis[v]

    # This is heuristic function which is having 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, stop):

        # In this open\_lst is a lisy of nodes which have been visited, but who's

        # neighbours haven't all been always inspected, It starts off with the start

  #node

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

        # and who's neighbors have been always inspected

        open\_lst = set([start])

        closed\_lst = set([])

        # poo has present distances from start to all other nodes

        # the default value is +infinity

        poo = {}

        poo[start] = 0

        # par contains an adjac mapping of all nodes

        par = {}

        par[start] = start

        while len(open\_lst) > 0:

            n = None

            # it will find a node with the lowest value of f() -

            for v in open\_lst:

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

                    n = v;

            if n == None:

                print('Path does not exist!')

                return None

            # if the current node is the stop

            # then we start again from start

            if n == stop:

                reconst\_path = []

                while par[n] != n:

                    reconst\_path.append(n)

                    n = par[n]

                reconst\_path.append(start)

                reconst\_path.reverse()

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

                return reconst\_path

            # for all the neighbors of the current node do

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

              # if the current node is not presentin both open\_lst and closed\_lst

                # add it to open\_lst and note n as it's par

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

                    open\_lst.add(m)

                    par[m] = n

                    poo[m] = poo[n] + weight

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

                # and if it is, update par data and poo data

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

                else:

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

                        poo[m] = poo[n] + weight

                        par[m] = n

                        if m in closed\_lst:

                            closed\_lst.remove(m)

                            open\_lst.add(m)

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

            # because all of his neighbors were inspected

            open\_lst.remove(n)

            closed\_lst.add(n)

        print('Path does not exist!')

        return None

**INPUT:**

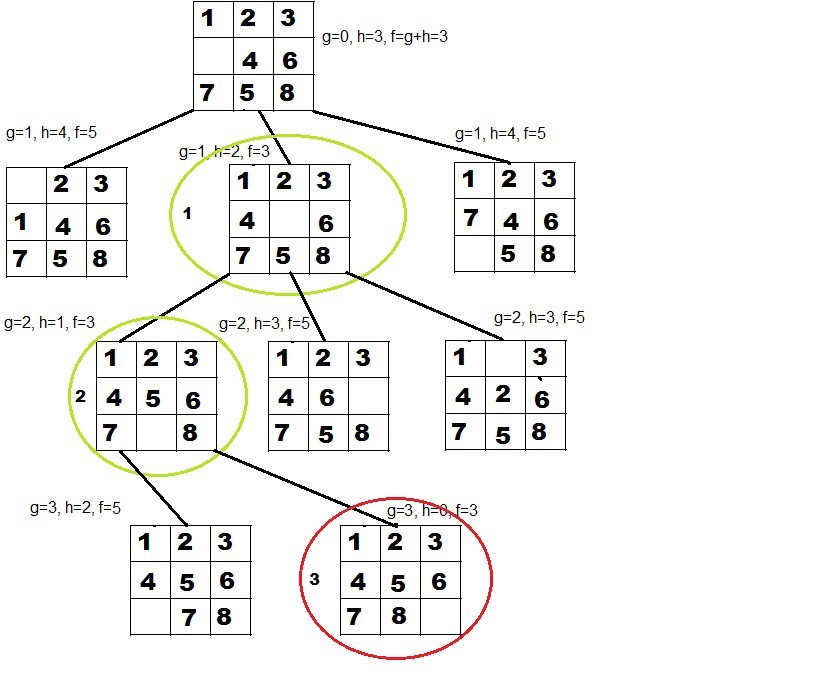
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | adjac\_lis = {      'A': [('B', 1), ('C', 3), ('D', 7)],      'B': [('D', 5)],      'C': [('D', 12)]  }  graph1 = Graph(adjac\_lis)  graph1.a\_star\_algorithm('A', 'D') |

**OUTPUT:**

Path found: ['A', 'B', 'D']

['A', 'B', 'D']

**10.program to solve 8 puzzle problem**



class Node:

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

""" Initialize the node with the data, level of the node and the calculated fvalue """

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):

""" Copy function to create a similar matrix of the given node"""

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):

""" Accepts the puzzle from the user """

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(" \\\'/ \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)

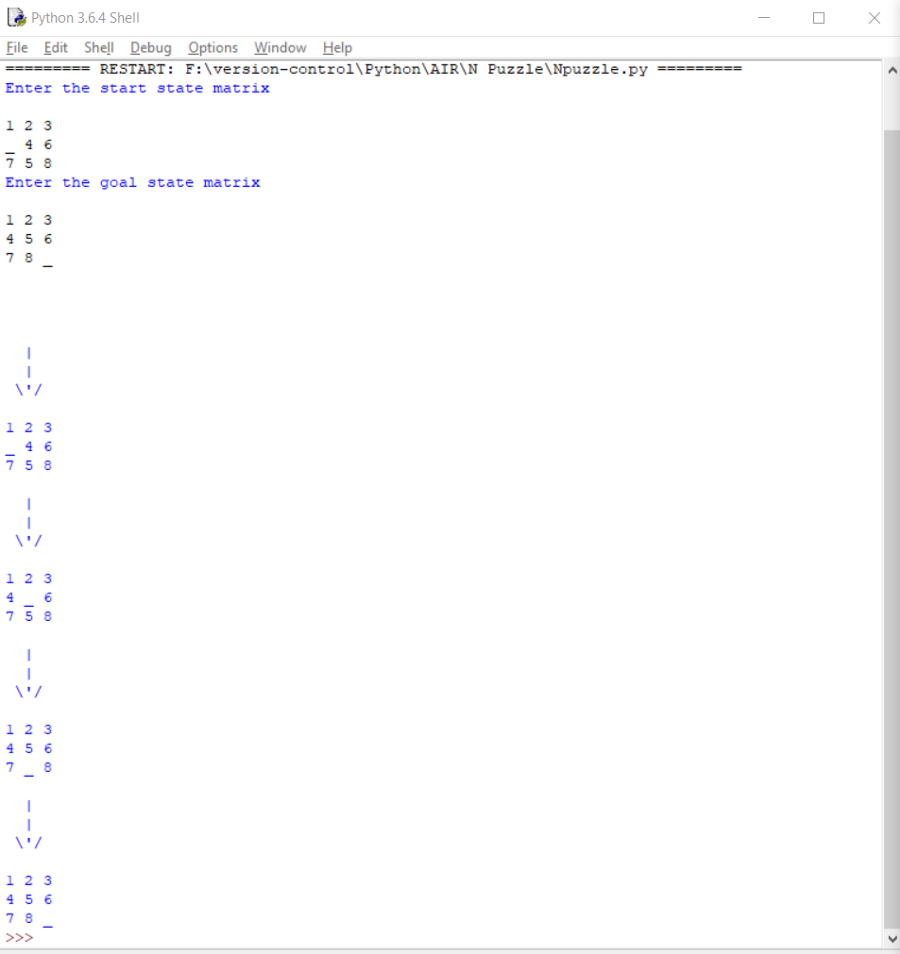
del self.open[0]

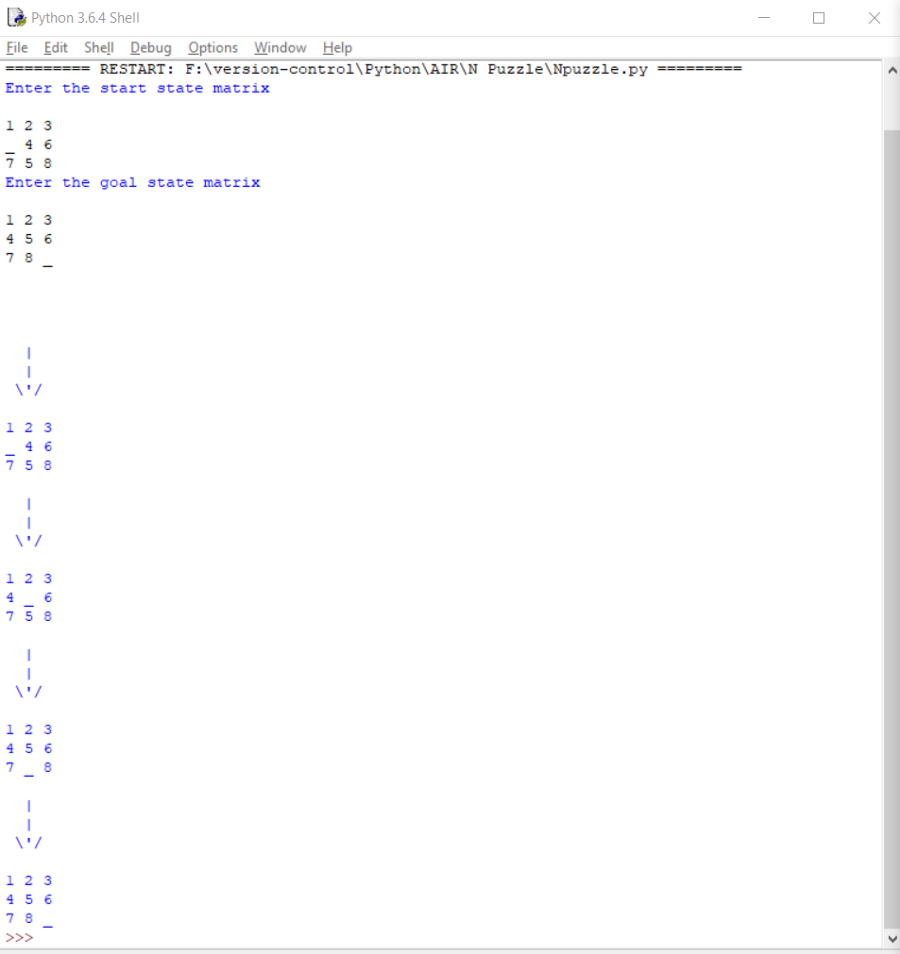
""" sort the opne list based on f value """

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

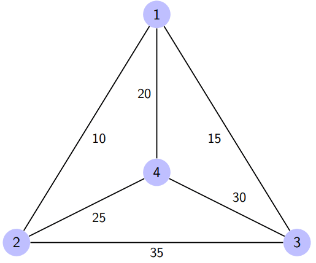
puz = Puzzle(3)

puz.process()





11.Travelling salesman program



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| dist(i,j) | 1 | 2 | 3 | 4 |
| 1 | 0 | 10 | 15 | 20 |
| 2 | 10 | 0 | 35 | 25 |
| 3 | 15 | 35 | 0 | 30 |
| 4 | 20 | 25 | 30 | 0 |

**Pseudo-code**

Algorithm: Traveling-Salesman-Problem

Cost (1, {}, 1) = 0

for s = 2 to n do

for all subsets S belongs to {1, 2, 3, … , n} of size s

Cost (s, S, 1) = Infinity

for all i Є S and i ≠ 1

Cost (i, S, j) = min {Cost (i, S – {i}, j) + dist(i, j) for j Є S and i ≠ j}

Return min(i) Cost (i, {1, 2, 3, …, n}, j) + d(j, i)

## Implementation in Python

from sys import maxsize

from itertools, import permutations

V = 4

def tsp(graph, s):

vertex = []

for i in range(V):

if i != s:

vertex.append(i)

min\_cost = maxsize

next\_permutation=permutations(vertex)

for i in next\_permutation:

current\_cost = 0

k = s

for j in i:

current\_cost += graph[k][j]

k = j

current\_cost += graph[k][s]

min\_cost = min(min\_cost, current\_cost)

return min\_cost

graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]

s = 0

print(tsp(graph, s))

**Output:**

80