**Practical No. 01**

**Title :- Implement Depth First Search algorithm and Breadth First Search algorithm , use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or a tree data structure.**

# Depth First Search (DFS) algorithm

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)

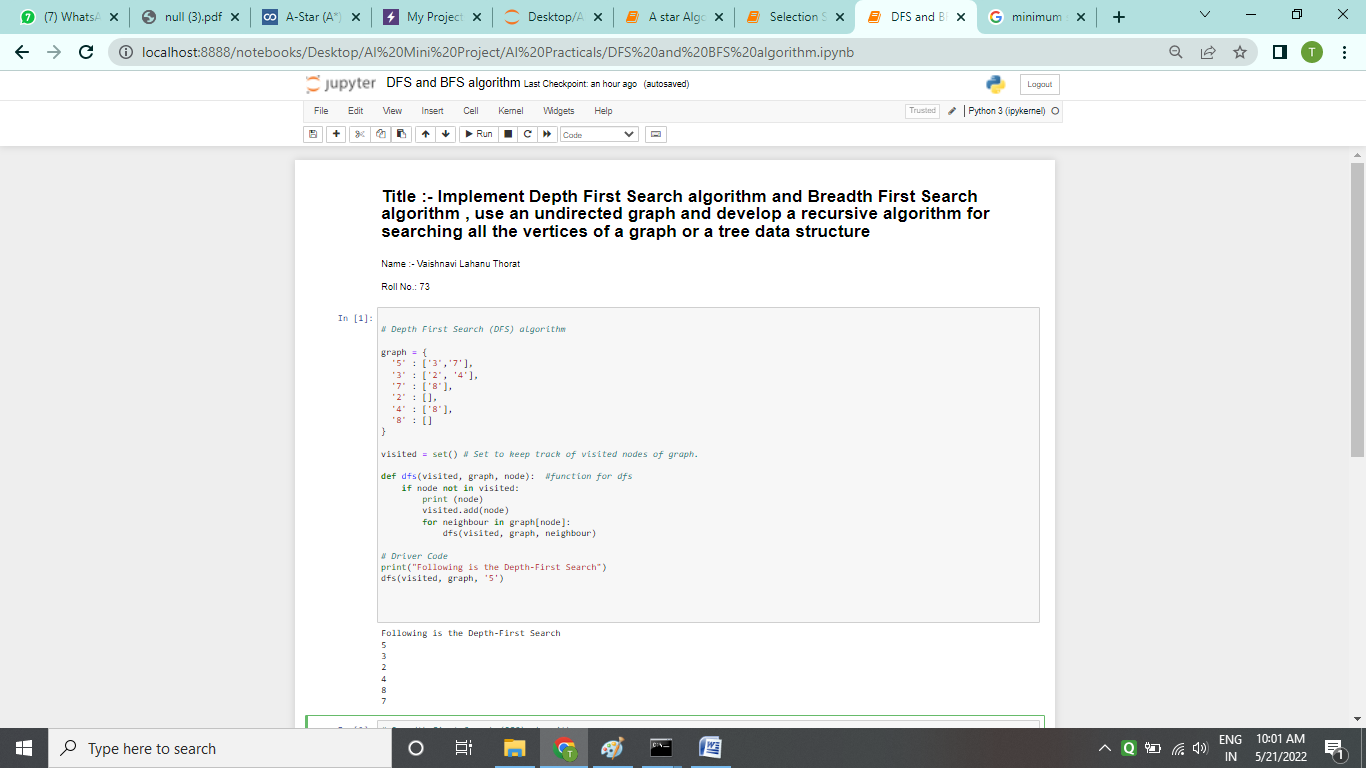
for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

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

dfs(visited, graph, '5')



# Breadth First Search (BFS) algorithm

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:

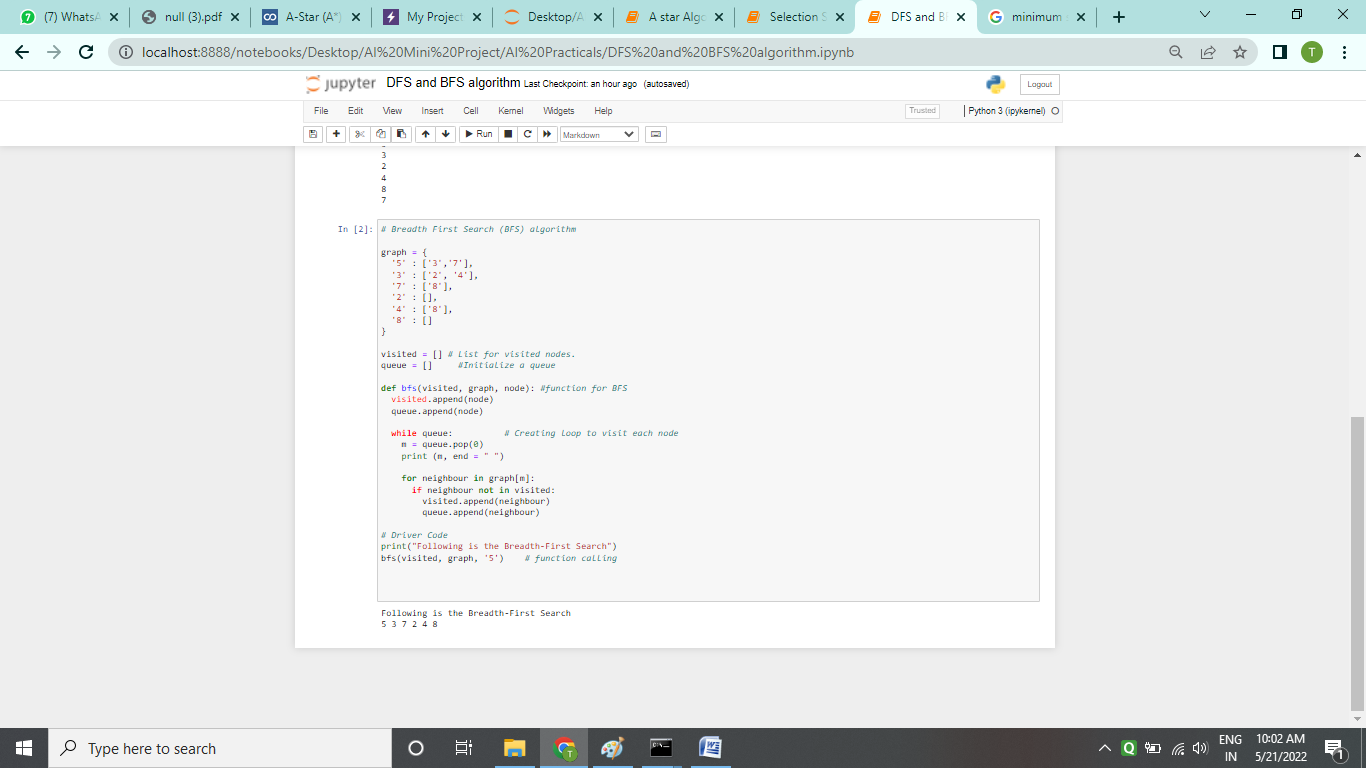
visited.append(neighbour)

queue.append(neighbour)

# Driver Code

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

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



**Practical No. 02**

# Title :- Implement A\* algorithm for any game search problem

# from pyamaze import maze,agent,textLabel

# from queue import PriorityQueue

# def h(cell1,cell2):

# x1,y1=cell1

# x2,y2=cell2

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

# def aStar(m):

# start=(m.rows,m.cols)

# g\_score={cell:float('inf') for cell in m.grid}

# g\_score[start]=0

# f\_score={cell:float('inf') for cell in m.grid}

# f\_score[start]=h(start,(1,1))

# open=PriorityQueue()

# open.put((h(start,(1,1)),h(start,(1,1)),start))

# aPath={}

# while not open.empty():

# currCell=open.get()[2]

# if currCell==(1,1):

# break

# for d in 'ESNW':

# if m.maze\_map[currCell][d]==True:

# if d=='E':

# childCell=(currCell[0],currCell[1]+1)

# if d=='W':

# childCell=(currCell[0],currCell[1]-1)

# if d=='N':

# childCell=(currCell[0]-1,currCell[1])

# if d=='S':

# childCell=(currCell[0]+1,currCell[1])

# temp\_g\_score=g\_score[currCell]+1

# temp\_f\_score=temp\_g\_score+h(childCell,(1,1))

# if temp\_f\_score < f\_score[childCell]:

# g\_score[childCell]= temp\_g\_score

# f\_score[childCell]= temp\_f\_score

# open.put((temp\_f\_score,h(childCell,(1,1)),childCell))

# aPath[childCell]=currCell

# fwdPath={}

# cell=(1,1)

# while cell!=start:

# fwdPath[aPath[cell]]=cell

# cell=aPath[cell]

# return fwdPath

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

# m=maze(5,5)

# m.CreateMaze()

# path=aStar(m)

# a=agent(m,footprints=True)

# m.tracePath({a:path})

# l=textLabel(m,'A Star Path Length',len(path)+1)

# m.run()

# A star1.png

# A star2.png

# Output :

# A star_output.png

# Practical No. 03

# Title : Implement a greedy search algorithm for any of the following application-

# Selection sort

# Minimum Spanning Tree

# Single-Source Shortest Path algorithm

# Job Scheduling Problem

# Prim’s Minimum Spanning Tree Algorithm

# Kruskal’s Minimum Spanning Tree Algorithm

# Dijktra’s Minimum Spanning Tree Algorithm

# SELECTION SORT:

# # Python program for implementation of Selection Sort

# import sys

# A = [64, 25, 12, 22, 11]

# # Traverse through all array elements

# for i in range(len(A)):

# # Find the minimum element in remaining unsorted array

# min\_idx = i

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

# if A[min\_idx] > A[j]:

# min\_idx = j

# 

# # Swap the found minimum element with the first element

# A[i], A[min\_idx] = A[min\_idx], A[i]

# # Driver code to test above

# print ("Sorted array")

# for i in range(len(A)):

# print("%d" %A[i],end=" ")

# 

# Single-Source Shortest Path algorithm (SSSP):

#### Implementation of BFS Traversal for SSSP

The BFS Traversal algorithm for SSSP is based on the following steps:

* Insert the graph’s source vertex at the back of a queue.
* Retrieve the first item of the queue and mark it as visited.
* Created a list of the nodes adjacent to the current node. Traverse the unvisited nodes and insert them to the back of queue.
* Repeat the steps continuously until the queue is empty or the destination node is reached.

#Initializing the Graph Class

class Graph:

def \_\_init\_\_(self, gdict=None):

if gdict is None:

gdict = {}

self.gdict = gdict

def addEdge(self, vertex, edge):

self.gdict[vertex].append(edge)

#Function to implement BFS Traversal for SSSP

def bfs(self, start, end):

queue = []

queue.append([start])

while queue:

path = queue.pop(0)

node = path[-1]

if node == end:

return path

for adjacent in self.gdict.get(node, []):

new\_path = list(path)

new\_path.append(adjacent)

queue.append(new\_path)

customDict = { "a" : ["b", "c"],

"b" : ["d", "g"],

"c" : ["d", "e"],

"d" : ["f"],

"e" : ["f"],

"g" : ["f"]

}

g = Graph(customDict)

print(g.bfs("a", "e"))

# 

# Job Scheduling Problem

# def printJobScheduling(arr, t):

# # length of array

# n = len(arr)

# # Sort all jobs according to

# # decreasing order of profit

# for i in range(n):

# for j in range(n - 1 - i):

# if arr[j][2] < arr[j + 1][2]:

# arr[j], arr[j + 1] = arr[j + 1], arr[j]

# # To keep track of free time slots

# result = [False] \* t

# # To store result (Sequence of jobs)

# job = ['-1'] \* t

# # Iterate through all given jobs

# for i in range(len(arr)):

# # Find a free slot for this job

# # (Note that we start from the

# # last possible slot)

# for j in range(min(t - 1, arr[i][1] - 1), -1, -1):

# # Free slot found

# if result[j] is False:

# result[j] = True

# job[j] = arr[i][0]

# break

# # print the sequence

# print(job)

# # Driver COde

# arr = [['a', 2, 100], # Job Array

# ['b', 1, 19],

# ['c', 2, 27],

# ['d', 1, 25],

# ['e', 3, 15]]

# print("Following is maximum profit sequence of jobs")

# # Function Call

# printJobScheduling(arr, 3)

# Job_scheduling_problem-output1.png

# Job_scheduling_problem-output2.png

# Prim’s Minimum Spanning Tree Algorithm

# # Prim's Algorithm in Python

# INF = 9999999

# # number of vertices in graph

# V = 5

# # create a 2d array of size 5x5

# # for adjacency matrix to represent graph

# G = [[0, 9, 75, 0, 0],

# [9, 0, 95, 19, 42],

# [75, 95, 0, 51, 66],

# [0, 19, 51, 0, 31],

# [0, 42, 66, 31, 0]]

# # create a array to track selected vertex

# # selected will become true otherwise false

# selected = [0, 0, 0, 0, 0]

# # set number of edge to 0

# no\_edge = 0

# # the number of egde in minimum spanning tree will be

# # always less than(V - 1), where V is number of vertices in

# # graph

# # choose 0th vertex and make it true

# selected[0] = True

# # print for edge and weight

# print("Edge : Weight\n")

# while (no\_edge < V - 1):

# # For every vertex in the set S, find the all adjacent vertices

# #, calculate the distance from the vertex selected at step 1.

# # if the vertex is already in the set S, discard it otherwise

# # choose another vertex nearest to selected vertex at step 1.

# minimum = INF

# x = 0

# y = 0

# for i in range(V):

# if selected[i]:

# for j in range(V):

# if ((not selected[j]) and G[i][j]):

# # not in selected and there is an edge

# if minimum > G[i][j]:

# minimum = G[i][j]

# x = i

# y = j

# print(str(x) + "-" + str(y) + ":" + str(G[x][y]))

# selected[y] = True

# no\_edge += 1

# Prim's Algo..png

# Prim's Algo.2.png

# Kruskal’s Minimum Spanning Tree Algorithm

# # Python program for Kruskal's algorithm to find

# # Minimum Spanning Tree of a given connected,

# # undirected and weighted graph

# from collections import defaultdict

# # Class to represent a graph

# class Graph:

# def \_\_init\_\_(self, vertices):

# self.V = vertices # No. of vertices

# self.graph = [] # default dictionary

# # to store graph

# # function to add an edge to graph

# def addEdge(self, u, v, w):

# self.graph.append([u, v, w])

# # A utility function to find set of an element i

# # (uses path compression technique)

# def find(self, parent, i):

# if parent[i] == i:

# return i

# return self.find(parent, parent[i])

# # A function that does union of two sets of x and y

# # (uses union by rank)

# def union(self, parent, rank, x, y):

# xroot = self.find(parent, x)

# yroot = self.find(parent, y)

# # Attach smaller rank tree under root of

# # high rank tree (Union by Rank)

# if rank[xroot] < rank[yroot]:

# parent[xroot] = yroot

# elif rank[xroot] > rank[yroot]:

# parent[yroot] = xroot

# # If ranks are same, then make one as root

# # and increment its rank by one

# else:

# parent[yroot] = xroot

# rank[xroot] += 1

# # The main function to construct MST using Kruskal's

# # algorithm

# def KruskalMST(self):

# result = [] # This will store the resultant MST

# # An index variable, used for sorted edges

# i = 0

# # An index variable, used for result[]

# e = 0

# # Step 1: Sort all the edges in

# # non-decreasing order of their

# # weight. If we are not allowed to change the

# # given graph, we can create a copy of graph

# self.graph = sorted(self.graph,

# key=lambda item: item[2])

# parent = []

# rank = []

# # Create V subsets with single elements

# for node in range(self.V):

# parent.append(node)

# rank.append(0)

# # Number of edges to be taken is equal to V-1

# while e < self.V - 1:

# # Step 2: Pick the smallest edge and increment

# # the index for next iteration

# u, v, w = self.graph[i]

# i = i + 1

# x = self.find(parent, u)

# y = self.find(parent, v)

# # If including this edge doesn't

# # cause cycle, include it in result

# # and increment the indexof result

# # for next edge

# if x != y:

# e = e + 1

# result.append([u, v, w])

# self.union(parent, rank, x, y)

# # Else discard the edge

# minimumCost = 0

# print ("Edges in the constructed MST")

# for u, v, weight in result:

# minimumCost += weight

# print("%d -- %d == %d" % (u, v, weight))

# print("Minimum Spanning Tree" , minimumCost)

# # Driver code

# g = Graph(4)

# g.addEdge(0, 1, 10)

# g.addEdge(0, 2, 6)

# g.addEdge(0, 3, 5)

# g.addEdge(1, 3, 15)

# g.addEdge(2, 3, 4)

# # Function call

# g.KruskalMST()

# Kruskal's algo.1.png

# Kruskal's algo.2.png

# Kruskal's algo.3.png

# Dijktra’s Minimum Spanning Tree Algorithm:

# # Python program for Dijkstra's single

# # source shortest path algorithm. The program is

# # for adjacency matrix representation of the graph

# # Library for INT\_MAX

# import sys

# class Graph():

# def \_\_init\_\_(self, vertices):

# self.V = vertices

# self.graph = [[0 for column in range(vertices)]

# for row in range(vertices)]

# def printSolution(self, dist):

# print("Vertex \tDistance from Source")

# for node in range(self.V):

# print(node, "\t", dist[node])

# # A utility function to find the vertex with

# # minimum distance value, from the set of vertices

# # not yet included in shortest path tree

# def minDistance(self, dist, sptSet):

# # Initialize minimum distance for next node

# min = sys.maxsize

# # Search not nearest vertex not in the

# # shortest path tree

# for u in range(self.V):

# if dist[u] < min and sptSet[u] == False:

# min = dist[u]

# min\_index = u

# return min\_index

# # Function that implements Dijkstra's single source

# # shortest path algorithm for a graph represented

# # using adjacency matrix representation

# def dijkstra(self, src):

# dist = [sys.maxsize] \* self.V

# dist[src] = 0

# sptSet = [False] \* self.V

# for cout in range(self.V):

# # Pick the minimum distance vertex from

# # the set of vertices not yet processed.

# # x is always equal to src in first iteration

# x = self.minDistance(dist, sptSet)

# # Put the minimum distance vertex in the

# # shortest path tree

# sptSet[x] = True

# # Update dist value of the adjacent vertices

# # of the picked vertex only if the current

# # distance is greater than new distance and

# # the vertex in not in the shortest path tree

# for y in range(self.V):

# if self.graph[x][y] > 0 and sptSet[y] == False and \

# dist[y] > dist[x] + self.graph[x][y]:

# dist[y] = dist[x] + self.graph[x][y]

# self.printSolution(dist)

# # Driver program

# g = Graph(9)

# g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],

# [4, 0, 8, 0, 0, 0, 0, 11, 0],

# [0, 8, 0, 7, 0, 4, 0, 0, 2],

# [0, 0, 7, 0, 9, 14, 0, 0, 0],

# [0, 0, 0, 9, 0, 10, 0, 0, 0],

# [0, 0, 4, 14, 10, 0, 2, 0, 0],

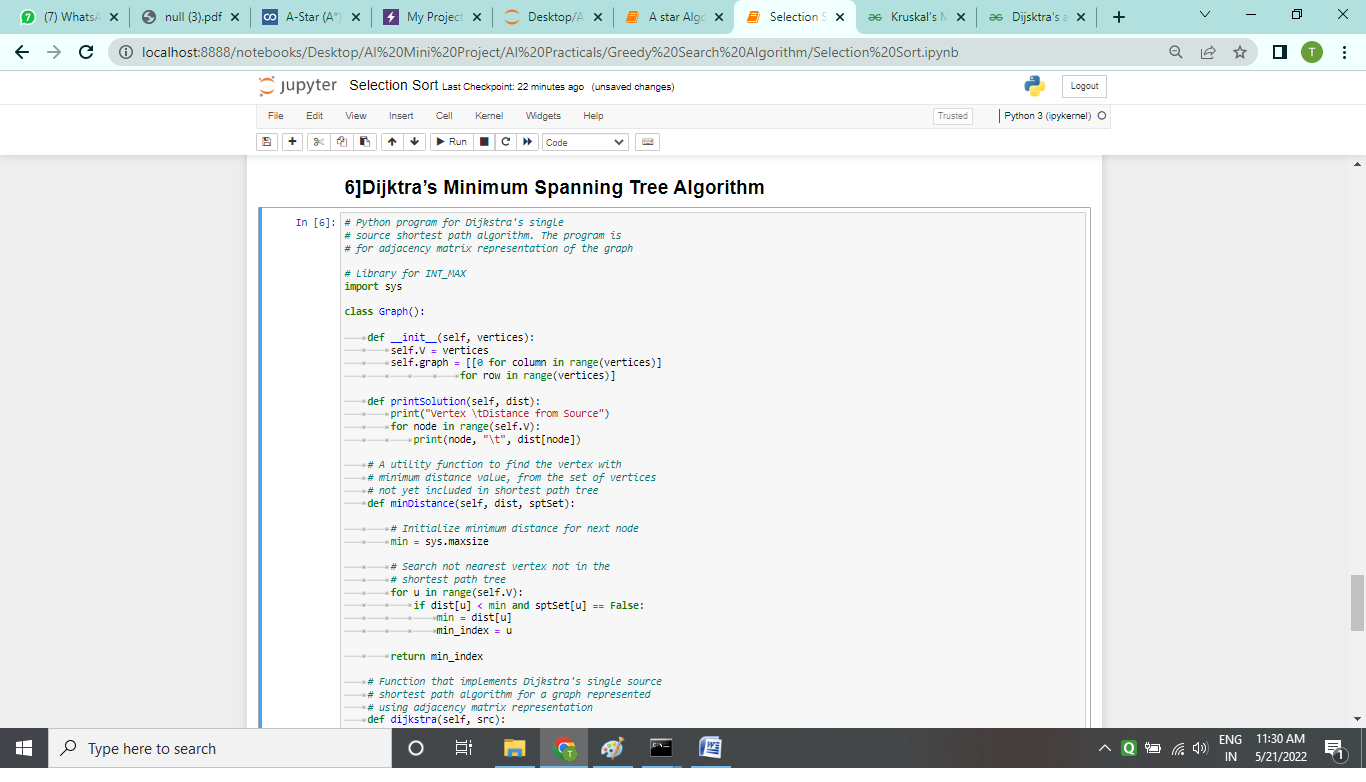
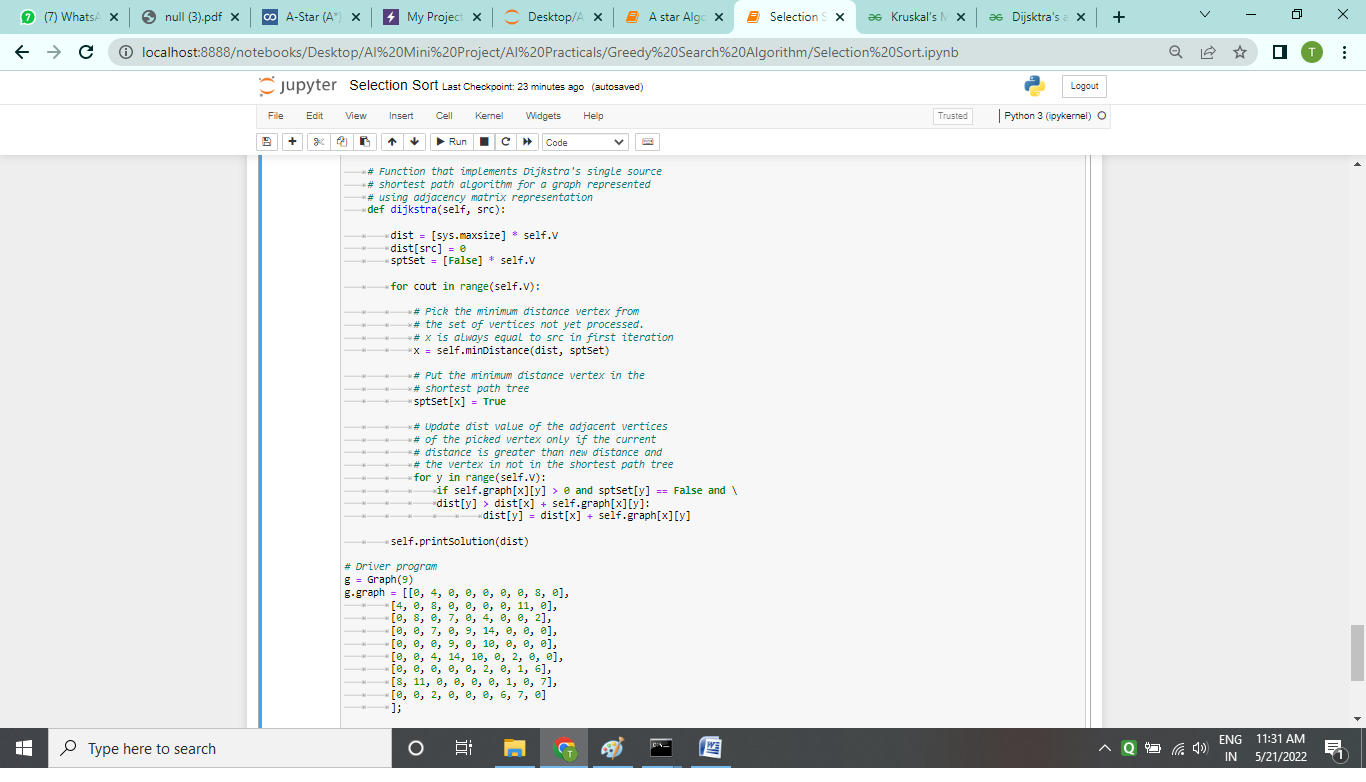
# [0, 0, 0, 0, 0, 2, 0, 1, 6],

# [8, 11, 0, 0, 0, 0, 1, 0, 7],

# [0, 0, 2, 0, 0, 0, 6, 7, 0]

# ];

# g.dijkstra(0);

 ****

# 