**Faculty of Engineering, Environment and Computing**

##### 5003CEM Advanced Algorithms

##### STUDENT ID: 9961812

##### Musharof husan munna

**Assignment Brief 2020/21**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Module Title  **Advanced Algorithms** | Ind/Group  **Individual** | | Cohort (Sept/Jan/May)  **Sept** | Module Code  **5003CEM** |
| Coursework Title  **CW: Code Submission and Report** | | | | Hand out date:  **18.01.2021** |
| Lecturer  **John Halloran / Beate Grawemeyer** | | | | Due date:  **09.04.2021** |
| Estimated Time (hrs): **50**  Word Limit\*: 3000 | | Coursework type:  **Code Submission / Report** | | % of Module Mark  **66.66% (10 of 15 credits)** |
| Submission arrangement online via Aula: **Upload is (a) a folder of code and a report; or (b) a report with a link to a repository. For (a) these should be submitted separately, not in the same zip folder.**  File types and method of recording: **Python and / or C++ files, plus Report**  Mark and Feedback date: **within 3 working weeks of submission.**  Mark and Feedback method: **Rubric marks and individualized comments** | | | | |

|  |
| --- |
| **Module Learning Outcomes Assessed:**  2 Design and implement algorithms and data structures for novel problems  3 Specify and implement methods to estimate solutions to intractable problems   1. Design and implement a basic concurrent application |
| **SUBMISSION**  You will submit your Assignment as a document with a separate codebase, as described above.  **The deadline is 9th April 2021: 18:00** |
| **ASSIGNMENT BRIEF**  CODE SUBMISSION  Each week on 5003CEM you will be given a range of programming tasks.  Some of these are defined as assessed, some as non-assessed.  Assessed tasks are required work, which you must submit by April 9th 2021 at 18:00.  There will be EIGHT assessed tasks to be completed. 5 of these will be at a standard level of difficulty; and 3 will be at an advanced level of difficulty. Each of the advanced tasks will relate to each of the 3 learning outcomes specified.  The tasks will cover a range of algorithms taught on the module, as well a basic concurrent application. You will also need to consider novel algorithms.  Assessed tasks will be evenly spaced across the module from Weeks 4 to 9.  Your assessed tasks will all need to be kept within the same folder with (a) nothing else in it; and (b) a clear structure with appropriately named files (and folders if used). If your code is on GitHub or Codio, the repository can be shared with relevant staff when the work is submitted for assessment purposes.  There will be a Moodle submission, which will have the pasted commented code for each solution, as well as a link to your GitHub repository.  MARKING CRITERIA  Over. |
| Notes   1. If relevant, you are expected to use the [CUHarvard](https://curve.coventry.ac.uk/open/file/bdfb947c-9d43-48d3-8ec8-f511682e1dd1/1/The%20CU%20Guide%20to%20Referencing%20in%20Harvard%20Style.pdf) referencing format. For support and advice on how this students can contact [Centre for Academic Writing (CAW)](http://www.coventry.ac.uk/study-at-coventry/student-support/academic-support/centre-for-academic-writing/?theme=main). 2. Please notify your registry course support team and module leader for disability support. 3. Any student requiring an extension or deferral should follow the university process as outlined [here](https://share.coventry.ac.uk/students/Registry/Pages/Deferrals-and-Extension.aspx). 4. The University cannot take responsibility for any coursework lost or corrupted on disks, laptops or personal computer. Students should therefore regularly back-up any work and are advised to save it on the University system. 5. If there are technical or performance issues that prevent students submitting coursework through the online coursework submission system on the day of a coursework deadline, an appropriate extension to the coursework submission deadline will be agreed. This extension will normally be 24 hours or the next working day if the deadline falls on a Friday or over the weekend period. This will be communicated via email and as a CUMoodle announcement. |

**Matrix marking scheme (marking criteria by degree band)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **CODE SUBMISSION**  **40 MARKS** | **REPORT**  **60 MARKS** | | |
| Marking criterion | Code quality  40 | Knowledge of code  40 | Critical reflection  10 | Clarity of communication  10 |
| 1st (70+) | Code is fully working, bug-free, well-commented, possibly with some advanced features. | Student shows sophisticated and detailed knowledge of how the code works at every level. Student can explain the design and implementation decisions. | Able to critique the implementation and offer a range of alternative possibilities; will be able to explain implementation decisions at a sophisticated level, including complexity analysis. | Able to write clearly and coherently in a well-structured way with good academic style, and appropriate referencing. Authoritative on the material. |
| 2.1 (60-69) | Code works, may have one or two non-important bugs. Commented. Does not need any advanced features. | Good knowledge of how major sections of the code work. Is able to explain design decisions and why things were implemented in particular ways. | Able to critique the implementation against possible alternatives (which may be limited to one). Can broadly explain implementation decisions, but less effective on detail. May be less effective on complexity analysis. | Generally able to write clearly and coherently although there may be some repetition or gaps. Appropriate academic style and referencing. There may be one or two unimportant errors. Clear ability to communicate what has been done. Clear insight into the material. |
| 2.2 (50-59) | Code generally works, may have bugs, may not cover all the required features even at basic level. Has some comments. | Some knowledge of how the code works but there may be areas of confusion. Ability to explain basic decisions about the design and implementation but again there may be misconceptions. | Less able to be critical about the code, and may not be able to discuss alternative implementations except in broad terms. May show confusion on complexity analysis. | Writing will broadly cover the work done and describe it, but with some gaps and errors. Academic style may be less evident, with less referencing. However, insight shown. The structure may not be entirely clear. |
| 3rd (40-49) | Code may not work; may not cover basic required features, may have significant bugs; but shows potential to work. May have some comments. | Limited knowledge of how the code works which is unlikely to show insight at detailed levels. Aware of the design of the code but not much insight into different design decisions that could be made. | Unlikely to be critical about the code beyond describing what it does. Little awareness of possible alternatives or complexity. | May show some elementary insight but unable to write authoritatively or with academic style. May be few references. Basic insight shown but there may be many gaps and errors. The structure of the report may be unclear. |
| Fail (0-39) | At this level some or all of the following may apply: Code is incomplete, does not work, is not commented, does not show clear potential. At the zero end, nothing has been done. | Little if any ability to explain how the code works. At the zero level the explanation (if any) will have no value. | Largely unable to be critical about the code or implementation. At the zero end, student will be completely unable to evaluate the code. | Little knowledge or engagement with the module and the writing performance shows this. Possibly poor academic style, and may be no references. Little insight, many gaps and errors. At the zero end, student may appear to be largely unaware of the problems set. |

* 1. **BST ITERATIVE AND RECURSIVE FIND METHODS**

1. **Commented Code**

import math

""" Node class

"""

class Node:

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

self.data = data

self.left = None

self.right = None

""" BST class with insert and display methods. display pretty prints the tree

"""

class BinaryTree:

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

self.root = None

def insert(self, data):

if self.root is None:

self.root = Node(data)

else:

self.\_insert(data, self.root)

def \_insert(self, data, cur\_node):

if data < cur\_node.data:

if cur\_node.left is None:

cur\_node.left = Node(data)

else:

self.\_insert(data, cur\_node.left)

elif data > cur\_node.data:

if cur\_node.right is None:

cur\_node.right = Node(data)

else:

self.\_insert(data, cur\_node.right)

else:

print("Value already present in tree")

def display(self, cur\_node):

lines, \_, \_, \_ = self.\_display(cur\_node)

for line in lines:

print(line)

def \_display(self, cur\_node):

if cur\_node.right is None and cur\_node.left is None:

line = '%s' % cur\_node.data

width = len(line)

height = 1

middle = width // 2

return [line], width, height, middle

if cur\_node.right is None:

lines, n, p, x = self.\_display(cur\_node.left)

s = '%s' % cur\_node.data

u = len(s)

first\_line = (x + 1) \* ' ' + (n - x - 1) \* '\_' + s

second\_line = x \* ' ' + '/' + (n - x - 1 + u) \* ' '

shifted\_lines = [line + u \* ' ' for line in lines]

return [first\_line, second\_line] + shifted\_lines, n + u, p + 2, n + u // 2

if cur\_node.left is None:

lines, n, p, x = self.\_display(cur\_node.right)

s = '%s' % cur\_node.data

u = len(s)

first\_line = s + x \* '\_' + (n - x) \* ' '

second\_line = (u + x) \* ' ' + '\\' + (n - x - 1) \* ' '

shifted\_lines = [u \* ' ' + line for line in lines]

return [first\_line, second\_line] + shifted\_lines, n + u, p + 2, u // 2

left, n, p, x = self.\_display(cur\_node.left)

right, m, q, y = self.\_display(cur\_node.right)

s = '%s' % cur\_node.data

u = len(s)

first\_line = (x + 1) \* ' ' + (n - x - 1) \* '\_' + s + y \* '\_' + (m - y) \* ' '

second\_line = x \* ' ' + '/' + (n - x - 1 + u + y) \* ' ' + '\\' + (m - y - 1) \* ' '

if p < q:

left += [n \* ' '] \* (q - p)

elif q < p:

right += [m \* ' '] \* (p - q)

zipped\_lines = zip(left, right)

lines = [first\_line, second\_line] + [a + u \* ' ' + b for a, b in zipped\_lines]

return lines, n + m + u, max(p, q) + 2, n + u // 2

""" Iterative Method

Input: any integers

Output: True, False, None

Implement Iterative method in BST code """

**def** **find\_i**(self, target):

cur\_node = self.root # set root value to cur\_node

**while** cur\_node != **None**: # check if cur\_node is not none

# compare cur\_node value with target

**if** cur\_node.data == target:

**return** cur\_node.data # if true, return the value

# check if cur\_node's value is bigger than the target

**elif** cur\_node.data > target:

# if True, set cur\_node.left to cur\_node/follow left node

cur\_node = cur\_node.left

**else**:

# if all are false above, follow the right node

cur\_node = cur\_node.right

**return** **False**

""" Recursive Method

Input: any integers

Output: True, False, None

Implement Recursive method in BST code"""

**def** **find\_r**(self, target):

**if** self.root: # if self.root exists, continue the function

# put target and self.root as the argument of \_find\_r method

**if** self.\_find\_r(target, self.root):

**return** **True** # if the function is true, return true

**return** **False** # or else, return false

**else**:

**return** **None**

**def** **\_find\_r**(self, target, cur\_node): # take three argument

# compare target value with cur\_node value and the right leaf's

**if** target > cur\_node.data **and** cur\_node.right:

#if false, call the function for same target but new cur\_node

**return** self.\_find\_r(target, cur\_node.right)

**elif** target < cur\_node.data **and** cur\_node.left:

# if true, call function with new cur\_node and same target

**return** self.\_find\_r(target, cur\_node.left)

**if** target == cur\_node.data: # check if target is found in any cur\_node

**return** **True** # if True, return True

#example calls, which construct and display the tree

bst = BinaryTree()

bst.insert(4)

bst.insert(2)

bst.insert(6)

bst.insert(1)

bst.insert(3)

bst.insert(5)

bst.insert(7)

bst.insert(8)

bst.insert(9)

##bst.insert(10)

##bst.insert(11)

##bst.insert(12)

##bst.insert(13)

##bst.insert(14)

##bst.insert(15)

##bst.insert(100)

##bst.insert(200)

bst.display(bst.root)

print(bst.find\_i(**21**)) # call function

recursive = bst.find\_r(**21**) # recursive function call

print(recursive)

Recursive Output:



Iterative Output:

Text

Description automatically generated

**(b) Eplanation of code**

This code implements iterative searching method and recursive searching method.

Iterative searching method works by iterating the whole fuction repeatedly while looking for the targeted value in the BST code. We start from the root node and compare targeted value with it. If the target is bigger than the current node, we move to the right node, or else we follow the left node. Same process keeps happening until we find the target. It returns true when we find the target, else, false/None.

Recursive searching method works by making two methods. We call the first method, which then calls the second method with two different arguments (target and root). Then, second method takes these arguments and compare the target with current node’s value and it’s right node. If the target is bigger, it calls the second method again with the new current node (left). Or else, it checks if the target is lower than the current node and it’s left node, if true, it calls the method with new current node (right).

Once the target matches with the current node, it returns true, else, false/None.

**(C) Critical Comments**

**1.1 Selection Sort**

**(a) Commented Code**

def selection\_sort(A): # defining a function

for i in range(len(A)): # iterating through the number of list elements

min = i # set i to min

for j in range(i+1, len(A)): # To iterate through the next element of i

if A[j] < A[min]: # compare i with the next element j.

min = j # set j to min when j is smaller than i

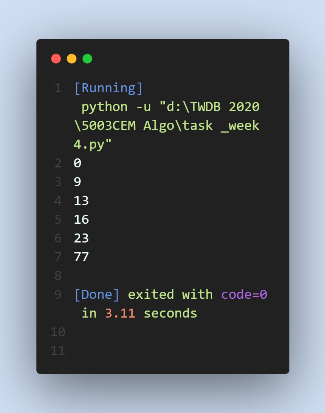
A[i], A[min] = A[min], A[i] #swapping the places of i and min to sort the array

for i in range(len(A)): # check the result of sorted array

print(A[i])

selection\_sort([13, 77, 16, 0, 23, 9]) # calling the function

Output:



**(b) Explanation of Code**

Selection sort works with two sub arrays. It takes the minimum value from the unsorted array and put it at the end of the sorted array. It keeps doing the same thing until there is no value left in the unsorted array.

We start by defining a function. Then we loop through the length of given list. We get i for each loop, set i to min. For each loop, there is an inner loop which iterates with the range of (i+1) to the length of the list. For each inner iteration, we get a value j. Which we put as an index of the given list and compare it with the A[min]. If A[j] is smaller than the A[min], assign j to min. When the loop breaks, we swap the unsorted list A[i] with the sorted list A[min] and we get the sorted array.

**(C) Critical Comments**

The code is an implementation of selection sort. However, there are few things to consider, the implementation of swap could be done using a buffer. Which is not required by python in the first place. So we do, A[i], A[min] = A[min], A[i]

Second issue to be considered is the language. I have used python to code. Python has the most easy syntax which will be easily readable by any beginners. It is short so it takes less time to code in python. On the other hand, if the same code is written in C++, it will look more complex and less readable than the python. C++ is strongly typed code. That means, it requires coders to use proper data types. Which python simply ignores. Also, using array to represent the input sequence in C++ doesn’t let you print the list as you simply can do in python.

To summarize, the code written in python is simple and readable. It takes less time and less amount of codes. Prioritising all the above benefits, I chose to write it in python instead of C++.

**1.4 Implement Prim’s Algorithm**

**(a) Commented Code**

import sys #needed for maxsize

class Graph():

def \_\_init\_\_(self, vertices): #implements graph as adjacency matrix

self.V = vertices #number of vertices

self.graph = [[0 for column in range(vertices)] #adjacency matrix with no edges (all connections set to zero)

for row in range(vertices)]

def printMST(self, parent):

print ("Edge \t Weight")

for i in range(1, self.V):

print (parent[i], "-", i, "\t", self.graph[i][ parent[i] ])

#from reached nodes find the unreached node with the minimum cost

def minKey(self, key, mstSet):

min = sys.maxsize #set min to infinity (use maxsize which is next best thing!)

for v in range(self.V): #count through number of vertices

if key[v] < min and mstSet[v] == False:#if vertex is less than min and unreached

min = key[v] #assign to min

min\_index = v #min\_index is position of min in key

return min\_index #return min\_index

#find MST

def primMST(self):

key = [sys.maxsize] \* self.V #initialise key to list of values all set to infinity;

same length as self.V

parent = [None] \* self.V #list for constructed MST

key[0] = 0 #set first element of key to

zero (this is where we start)

mstSet = [False] \* self.V #mstSet is list of booleans

set to False

parent[0] = -1 #first element of parent list

set to -1

for vertex in range(self.V): #go through all vertices

u = self.minKey(key, mstSet) #call minKey; minKey returns

u (index of unreached node)

mstSet[u] = True #mstSet at index of node is

set to True

for v in range(self.V): #go through all vertices

if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:

#if edge from u to connected node v is > 0 (if there is an edge)

#and mstSet[v] is unreached

#and key[v] is greater than the edge cost

#(only if the current edge cost is greater will need to change)

key[v] = self.graph[u][v] #key[v] takes edge cost

parent[v] = u #parent[v] is index of node;

so list of parents (nodes) is the MST

self.printMST(parent) #print the list of parents, i.e. the MST

g = Graph(5)

g.graph = [ [0, 2, 0, 6, 0],

[2, 0, 3, 8, 5],

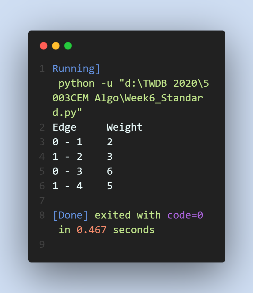
[0, 3, 0, 0, 7],

[6, 8, 0, 0, 9],

[0, 5, 7, 9, 0]]

g.primMST();

Output:



**(b) Code Explanation:**

Prim's Algorithm is used to get the minimum spanning tree/MST from a graph. Prim's algorithm obtains the subset of edges that includes all the vertices of the graph such that the overall sum of all the edge’s weight can be reduced.

It starts with the single node and go through all the connected nodes with all the connecting edges at every step. The edges which have the minimal weights and doesn’t start a cycle in the graph gets selected.

In this code, we start with implementing the graph as adjecency matrix. In the method – minKey, we take two arguments, one of them is the targeted value. We assign the maximum value to the min variable. We loop through the length of the vertices and for each loop, we compare the vertex with minimum value and the unreached node. If it is smaller than them, then we set the vertex to min. Then we find out the MST. We again loop through all the vertices in the range of vertice’s size. We call a function that returns the index of unreached node. We make an inner loop of v. Inside that loop, we give few conditions to the program such that if edge from u to v is greater than 0 and the node v is unreached, and key[v] is greater than the edge cost, then we set edge cost to key[v]. At last by printing, the list of parents, we get the MST.

**(b ) Critical Comments**

2. 3

**import** **newspaper**

**from** **newspaper** **import** Article

**import** **concurrent.futures**

**import** **urllib.request**

URLs = ['http://www.foxnews.com/',

#the URLs to get the headers from

'http://www.cnn.com/',

'http://europe.wsj.com/',

'http://www.bbc.co.uk/',

'https://theguardian.com',]

**def** **load\_url**(url, timeout):

**with** urllib.request.urlopen(url, timeout=timeout) **as** conn: #assigning the time out request to conn

**return** conn.read()

**def** **con\_URLS**():

**with** concurrent.futures.ThreadPoolExecutor(max\_workers=**5**) **as** executor: #assign the concurrent thread workers as the executor

url\_future = {executor.submit(load\_url, url, **60**): url **for** url **in** URLs}

current\_url = url\_future[future] #set the url future to the current url

**try**:

data = future.result() #set the results to the variable data

result = newspaper.build(url, memoize\_articles=**False**) #build the result

**except** **Exception** **as** exc: #creating an exception

print('%r made an exception: %s' % (url, exc))

**else**:

print('**\n**''The headlines from %s are' % url, '**\n**')

**for** i **in** range(**1**,**6**): # run for loop in the range of 1 to 6

article = result.articles[i] #set the result to article

article.download() #empty list of downloaded results

article.parse() #empty list of parsed results

print(article.title)

**def** **get\_headlines**():

**for** url **in** URLs: #for loop of the URLs

result = newspaper.build(url, memoize\_articles=**False**) #building the results

print('**\n**''The headlines from %s are' % url, '**\n**') #printing the headlines

**for** i **in** range(**1**,**6**): #for loop of the first 5 articles

art = result.articles[i] #assigning the arcticle results

art.download() #make a empty list of downloaded results

art.parse() #make a empty list of parsed results

print(art.title)

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

**import** **timeit**

#elapsed\_time = timeit.timeit("concurrent\_URLs\_example()", setup="from \_\_main\_\_ import concurrent\_URLs\_example", number=2)/2

elapsed\_time = timeit.timeit("get\_headlines()", setup="from \_\_main\_\_ import get\_headlines", number=**2**)/**2**#the time when using concurrency

print(elapsed\_time)

**(b ) Code Explanation:**

We start importing 4 libraries. Then we load the given URLS by using one of concurrent.futures library. Then we parse the article using a function named parse. We take the headings from the parsed articles. Then we print the titles of the articles. Then we print elapsed times using concurrency.

**(c ) Critical Comments:**  
The program we wrote is a web scrapping script. We used python to write this code because python has the most efficient way of solving this task. It is fast and it has bunch of libraries that can be used to do these types of tasks. The issue would happen if we chose some other language like C++. C++ doesn’t have any library for web scrapping. Which would make it quite impossible to get the result.

**2.2 Implement Dijkstra’s Algorithm**

**(a ) Commented Code:**

**from** **collections** **import** defaultdict

**import** **sys**

**class** **Graph**(): # Graph generation

**def** **\_\_init\_\_**(self, size):

self.edges = defaultdict(list)

self.weights = {}

self.size = size #assign the size

self.dist = [] #empty list of the distances

**for** i **in** range(size):

self.dist.append(sys.maxsize) #add max to distance

self.previous = []

**for** i **in** range(size):

self.previous.append(**None**) #adding nothing to the previous

**def** **add\_edge**(self, from\_node, to\_node, weight):

self.edges[from\_node].append(to\_node) #add the node to the edge

self.edges[to\_node].append(from\_node) #add the previous node to the edge

self.weights[(from\_node, to\_node)] = weight #assign the nodes to the weight

self.weights[(to\_node, from\_node)] = weight

**def** **findSmallestNode**(self): #Find the smallest node

smallest = self.dist[self.getIndex(self.Q[**0**])] #set the shortest distance node

result = self.getIndex(self.Q[**0**]) #set the shortest node

node = self.dist #set the distance

**for** i **in** range(len(self.dist)):

**if** self.dist[i] < smallest: #if the distance is less than smallest

**if** node **in** self.Q: #if the node is part of Q

smallest = self.dist[i] #set the dist to smallest

result = self.getIndex(node)

**return** result

**def** **getIndex**(self, neighbour):

**for** i **in** range(len(self.unpoppedQ)):

**if** neighbour == self.unpoppedQ[i]:

**return** i

**def** **getPopPosition**(self, uNode):

result = **0** #creat an empty result

**for** i **in** range(len(self.Q)):

**if** self.Q[i] == uNode:

**return** i

**return** result

**def** **getUnvisitedNodes**(self, uNode): #Getting unvisited node

resultList = [] #empty list of results

allNeighbours = self.edges[uNode]

**if** neighbour **in** self.Q: #if the neighbour is part of Q

resultList.append(neighbour) #adding the neighbour to the result list

**return** resultList

**def** **dijsktra**(self, start, end): # starting Dijsktra algorithm

self.Q = [] #empty list for Q

**for** key **in** self.edges:

self.Q.append(key) #add the key to Q

**for** i **in** range(len(self.Q)):

**if** self.Q[i] == start:

self.dist[i] = **0**

self.unpoppedQ = self.Q[**0**:] #assigning the first Q to unpopped

**while** self.Q:

u = self.findSmallestNode() #assign the smallest node to u

**if** self.dist[u] == sys.maxsize: #if the fistance of u is equal to the maximum

**break**

**if** self.unpoppedQ[u] == end: #if the unpopped of u is equal to the end

**break**

uNode = self.unpoppedQ[u] #assign the unpopped Q to uNode

**for** i **in** self.edges[uNode]: #for loop of edges of uNode

**if** i **in** self.Q: #if edge in Q

**if** i **in** self.getUnvisitedNodes(uNode): #if the edge is part of univisited nodes

**if** int (self.dist[self.unpoppedQ.index(uNode)]) + self.weights[(uNode,i)] < self.dist[self.unpoppedQ.index(i)]:#if the weight and unpopped is less than the distance of unpopped

self.dist[self.unpoppedQ.index(i)] = int (self.dist[self.unpoppedQ.index(uNode)]) + self.weights[(uNode,i)]#assigning the distance and weight to distance

self.previous[self.unpoppedQ.index(i)] = uNode #assigning the previous to uNode

self.Q.pop(self.Q.index(uNode))

weight = self.dist[self.unpoppedQ.index(uNode)+**1**]

shortest\_path = [] #empty list of shortest path

shortest\_path.insert(**0**, end) #insert the start and end to the shortest path

u = self.getIndex(end) #assign the index to u

**while** self.previous[u] != **None**: #while loop of the previous not being empty

shortest\_path.insert(**0**, self.previous[u]) #adding the previous of u to the shortest path

u = self.getIndex(self.previous[u]) #assigning the previous u of index to u

output = (shortest\_path,weight) #assigning the weight of shortest path to output

**return** output

graph = Graph(**8**) #the number of nodes of the graph

edges =[ #the weights of the edges in a list of tuples in a list

('O', 'A', **2**),

('O', 'B', **5**),

('O', 'C', **4**),

('A', 'B', **2**),

('A', 'D', **7**),

('A', 'F', **12**),

('B', 'C', **1**),

('B', 'D', **4**),

('B', 'E', **3**),

('C', 'E', **4**),

('D', 'E', **1**),

('D', 'T', **5**),

('E', 'T', **7**),

('F', 'T', **3**),

]

**for** edge **in** edges: #for loop of the edges

graph.add\_edge(\*edge) #adding the eddges to the graph size

print(graph.dijsktra('O', 'T')) #printing the path

**( b ) Code explanation**

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph. We generate a*shortest path tree* with given source as root. Two sets are maintained, one set has the vertices included in shortest path tree, other set has the vertices not yet included in shortest path tree. At every step, we get a vertex which is in the other set (set of not yet included) and has a lowest distance from the root. This is Dijkstra’s shortest path algorithm.