MANAV RACHNA
vidyanatariksha

Practical File
For
Advance Algorithm
Lab
(PC-CS-M-211)

Submitted to:
Ms. Rachna Bahl
(**A**ssociate Professor)
Dept. Of Computer Science & Engineering

Submitted By:
Uttam Raj
M.tech (CSE)
1/19/FET/MCN/003
2nd Semester

INDEX

S/ No	Lab Activity Name	Lab Date	Evaluation Date	Remarks
1a	Implement BFS procedures to search a node in a graph.	17-01-2020		
1b	Implement DFS procedures to search a node in a graph.	17-01-2020		
2	Write a program to implement insertion sort	17-01-2020		
3	Write a program to implement heap sort	17-01-2020		
4a	Implement Prim's algorithm to find minimum spanning tree.	24-01-2020		
4b	Implement Kruskal's algorithm to find minimum spanning tree.	24-01-2020		
5a	Program to Implement Edmond Karps algorithm in Python.	13-02-2020		
5b	Program to Implement **Ford-Fulkerson** algorithm in Python.	13-02-2020		
6	Program to calculate inverse of a triangular matrix	30-03-2020		
7	Program to calculate GCD	01-04-2020		
8	Program to implement floyd warshall algorithm	9-04-2020		

Practical-1a

Program Implement BFS procedures to search a node in a graph.

Source Code

```
# Initialising a graph
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

#creating 2 empty array for keeping track of visited nodes of graph and the queue for the sequence of access
visited = []
queue = []

def bfs(visited, graph, node, elementToSearch):
  visited.append(node)   #when a node is visited , it is added to visited array
  queue.append(node)    #node added to queue
  flag=0                         #setting up flag variable to tell when the element is found

  while queue:
    s = queue.pop(0)     #getting first element of queue
    print (s, end = " ")
    if s==elementToSearch:
        print ('element found')
        flag=1
        break
    else:                        #if neighbours of the element not visited then it is added to the queue

        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)
  if flag==0:
      print ('not found')

elementToSearch= input ("Enter the element to search: ")
bfs(visited, graph, 'A',elementToSearch)
```

Complexity

O(V + E)

Output

```
Enter the element to search: D
A B C D element found

In [2]:
```

Practical-1b

Program Implement DFS procedures to search a node in a graph.

Source Code

```
import sys
graph = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}

#creating 2 empty array for keeping track of visited nodes of graph and the queue for the sequence of access

visited = []   #when a node is visited , it is added to visited array
flag=0         #setting up flag variable to tell when the element is found


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

    if node not in visited:
        print (node)
        if node==elementToSearch:
            print ('element found')
            h= input ("Enter 1 to exit")
            sys.exit()

        else:                    #if neighbours of the element not visited then it is added to the stack
            visited.append(node)
            for neighbour in graph[node]:
                dfs(visited, graph, neighbour,elementToSearch)
                dfs(visited, graph, 'A',elementToSearch)

elementToSearch= input ("Enter the element to search: ")
dfs(visited, graph, 'A',elementToSearch)
print ('element not found')
```

Complexity

O(V + E)

Output

```
Enter the element to search: E
A
B
D
E
element found

Enter 1 to exit
```

Practical-2

Program to Implement Insertion sort in Python. This program can sort a given list of integer elements

Source Code

```
def insertionsort(mylist):   #function definition
    for i in range(1,len(mylist)):
        a=mylist[i]
        b=i-1
        while b>0 and a<mylist[b]:
            mylist[b+1]=mylist[b]
            b -= 1
        mylist[b+1]=a


mylist = [1,2,3,4,8,5,6]
insertionsort(mylist)
for i in range (len(mylist)):   #to print the final list
    print(mylist[i])
```

Complexity

O(n^2)

Output

```
1
2
3
4
5
6
8

In [2]:
```

Practical-3

Program Implement heap sort in Python.

Source Code

```
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)
```

```
    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)

# Driver code to test above
arr = [ 12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("Sorted array is")
for i in range(n):
    print ("%d" %arr[i]),
```

Complexity

O(nLogn)

Output

```
Sorted array is
5
6
7
11
12
13

In [3]:
```

Practical-4a

Program to Implement PRIM's minimum spanning tree algorithm in Python.

Source Code

```
import sys # Library for INT_MAX

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # A utility function to print the constructed MST stored in parent[]
    def printMST(self, parent):
        print ("Edge \tWeight")
        for i in range(1, self.V):
            print (parent[i], "-", i, "\t", self.graph[i][ parent[i] ])

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initilaize min value
        min = int(sys.maxsize)

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
```

```
            min = key[v]
            min_index = v

    return min_index

# Function to construct and print MST for a graph
# represented using adjacency matrix representation
def primMST(self):

    # Key values used to pick minimum weight edge in cut
    key = [int(sys.maxsize)] * self.V
    parent = [None] * self.V # Array to store constructed MST
    # Make key 0 so that this vertex is picked as first vertex
    key[0] = 0
    mstSet = [False] * self.V

    parent[0] = -1 # First node is always the root of

    for cout in range(self.V):

        # Pick the minimum distance vertex from
        # the set of vertices not yet processed.
        # u is always equal to src in first iteration
        u = self.minKey(key, mstSet)

        # Put the minimum distance vertex in
        # the shortest path tree
        mstSet[u] = 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 shotest path tree
            for v in range(self.V):
                # graph[u][v] is non zero only for adjacent vertices of m
                # mstSet[v] is false for vertices not yet included in MST
                # Update the key only if graph[u][v] is smaller than key[v]
                if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
                        key[v] = self.graph[u][v]
                        parent[v] = u

        self.printMST(parent)

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();
```

Complexity

O ((V + E) l o g V)

Output

```
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5

In [5]:
```

Practical-4b

Program to Implement Kruskal's minimum spanning tree algorithm in Python.

Source Code

```
class Graph:
    def __init__(self,vertices):
        self.V= vertices
        self.graph = []
    def addEdge(self,u,v,w):
        self.graph.append([u,v,w])
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else :
            parent[yroot] = xroot
            rank[xroot] += 1
    def KruskalMST(self):

        result =[]
        i = 0
        e = 0
        self.graph = sorted(self.graph,key=lambda item: item[2])
        parent = [] ; rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V -1 :
            u,v,w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent ,v)
            if x != y:
                e = e + 1
                result.append([u,v,w])
                self.union(parent, rank, x, y)
        print ("Following are the edges in the constructed MST")
        for u,v,weight in result:
            print ("%d -- %d == %d" % (u,v,weight))
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)

g.KruskalMST()
```

Complexity

$O(E \log V)$

Output

```
In [4]: runfile('C:/Users/Uttam Raj/.spyder-py3/spanni
Raj/.spyder-py3')
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Practical-5a

Program to Implement Edmond Karps algorithm in Python.

Source Code

```
def max_flow(C, s, t):
        n = len(C) # C is the capacity matrix
        F = [[0] * n for i in range(n)]
        path = bfs(C, F, s, t)
      #  print path
        while path != None:
            flow = min(C[u][v] - F[u][v] for u,v in path)
            for u,v in path:
                F[u][v] += flow
                F[v][u] -= flow
            path = bfs(C, F, s, t)
        return sum(F[s][i] for i in range(n))

#find path by using BFS
def bfs(C, F, s, t):
        queue = [s]
        paths = {s:[]}
        if s == t:
            return paths[s]
        while queue:
            u = queue.pop(0)
            for v in range(len(C)):
                    if(C[u][v]-F[u][v]>0) and v not in paths:
                        paths[v] = paths[u]+[(u,v)]
                        print (paths)
                        if v == t:
                            return paths[v]
                        queue.append(v)
        return None

# make a capacity graph
# node s  o  p  q  r  t
C = [[ 0, 3, 3, 0, 0, 0 ],  # s
     [ 0, 0, 2, 3, 0, 0 ],  # o
     [ 0, 0, 0, 0, 2, 0 ],  # p
     [ 0, 0, 0, 0, 4, 2 ],  # q
     [ 0, 0, 0, 0, 0, 2 ],  # r
     [ 0, 0, 0, 0, 0, 3 ]]  # t

source = 0  # A
sink = 5    # F
max_flow_value = max_flow(C, source, sink)
```

```
print ("Edmonds-Karp algorithm")
print ("max_flow_value is: ", max_flow_value)
```

Complexity

$O(VE2)$

Output

```
2nd SEM/AAL/LAB '
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 1), (1, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 1), (1, 3)], 4: [(0, 2), (2, 4)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 1), (1, 3)], 4: [(0, 2), (2, 4)], 5:
[(0, 1), (1, 3), (3, 5)]}
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 1), (1, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 1), (1, 3)], 4: [(0, 2), (2, 4)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 1), (1, 3)], 4: [(0, 2), (2, 4)], 5:
[(0, 2), (2, 4), (4, 5)]}
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 1), (1, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 1), (1, 3)], 4: [(0, 1), (1, 3), (3,
4)]}
Edmonds-Karp algorithm
max_flow_value is:  4

In [11]:
```

Practical-5b

Program to Implement **Ford-Fulkerson** algorithm in Python.

Source Code

```
#Python program for implementation of Ford Fulkerson algorithm

from collections import defaultdict

#This class represents a directed graph using adjacency matrix representation
class Graph:

    def __init__(self,graph):
        self.graph = graph # residual graph
        self. ROW = len(graph)
        #self.COL = len(gr[0])


    '''Returns true if there is a path from source 's' to sink 't' in
    residual graph. Also fills parent[] to store the path '''
    def BFS(self,s, t, parent):

        # Mark all the vertices as not visited
        visited =[False]*(self.ROW)

        # Create a queue for BFS
        queue=[]

        # Mark the source node as visited and enqueue it
        queue.append(s)
```

```
    visited[s] = True

     # Standard BFS Loop
    while queue:

        #Dequeue a vertex from queue and print it
        u = queue.pop(0)

        # Get all adjacent vertices of the dequeued vertex u
        # If a adjacent has not been visited, then mark it
        # visited and enqueue it
        for ind, val in enumerate(self.graph[u]):
            if visited[ind] == False and val > 0 :
                queue.append(ind)
                visited[ind] = True
                parent[ind] = u

    # If we reached sink in BFS starting from source, then return
    # true, else false
    return True if visited[t] else False



# Returns tne maximum flow from s to t in the given graph
def FordFulkerson(self, source, sink):

    # This array is filled by BFS and to store path
    parent = [-1]*(self.ROW)

    max_flow = 0 # There is no flow initially

    # Augment the flow while there is path from source to sink
```

```
        while self.BFS(source, sink, parent) :

            # Find minimum residual capacity of the edges along the
            # path filled by BFS. Or we can say find the maximum flow
            # through the path found.
            path_flow = float("Inf")
            s = sink
            while(s !=  source):
                path_flow = min (path_flow, self.graph[parent[s]][s])
                s = parent[s]

            # Add path flow to overall flow
            max_flow +=  path_flow

            # update residual capacities of the edges and reverse edges
            # along the path
            v = sink
            while(v !=  source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow



# Create a graph given in the above diagram

graph = [[0, 16, 13, 0, 0, 0],
        [0, 0, 10, 12, 0, 0],
        [0, 4, 0, 0, 14, 0],
```

```
        [0, 0, 9, 0, 0, 20],
        [0, 0, 0, 7, 0, 4],
        [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0; sink = 5

print ("The maximum possible flow is %d " % g.FordFulkerson(source, sink))
```

Complexity

O (F*E), F is the maximum flow

Output

```
The maximum possible flow is 23

In [6]:
```

Practical-6

Program to Calculate inverse of a triangular matrix. Take input of 3X3 triangular matrix and calculate its inverse

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 31 21:28:37 2020

@author: krishanbhadana
"""
def entermatrix(size):
    T=[[0 for w in range(size)] for t in range(size)]
    for i in range(size):
        for j in range(size):
            T[i][j]=int(input("Enter "+str(i)+" row and "+str(j)+" column "))
    return T

def caluppertriangularmatrix(T):
    if (T[1][0]==0 & T[2][0]==0 & T[2][1]==0):
        return(1)
    else:
        return(0)

def callowertriangularmatrix(T):
    if (T[0][2]==0 & T[0][1]==0 & T[1][2]==0):
        return(1)
    else:
        return(0)

def inverse(X):
    I=X
    uppert=caluppertriangularmatrix(I)
    lowert=callowertriangularmatrix(I)
    if(uppert==1 & lowert==1):
        return(X)
    elif(uppert==0 & lowert==0):
        print("Inverse cannot be calculated using this method")
    elif(uppert==1 & lowert==0):
        I[0][1]=0-X[0][1]
        I[1][2]=0-X[1][2]
        I[0][2]=0-X[0][2]-(X[0][1]*I[1][2])
        I[0][0]=1/X[0][0]
        I[1][1]=1/X[1][1]
        I[2][2]=1/X[2][2]
        I[1][0]=0
        I[2][0]=0
        I[2][1]=0
    elif(uppert==0 & lowert==1):
```

```
        I[1][0]=0-X[1][0]
        I[2][0]=0-X[2][0]-(X[2][1]*I[1][0])
        I[2][1]=0-X[2][1]
        I[0][0]=1/X[0][0]
        I[1][1]=1/X[1][1]
        I[2][2]=1/X[2][2]
        I[0][2]=0
        I[0][1]=0
        I[1][2]=0
    return(I)

print("\nEnter a 3X3 matrix")
X=entermatrix(3)
Y=inverse(X)
print("Inverse of the matrix is",Y)
```

OUTPUT

```
Enter a 3X3 matrix

Enter 0 row and 0 column 1

Enter 0 row and 1 column 3

Enter 0 row and 2 column 5

Enter 1 row and 0 column 0

Enter 1 row and 1 column 1

Enter 1 row and 2 column 6

Enter 2 row and 0 column 0

Enter 2 row and 1 column 0

Enter 2 row and 2 column 1
Inverse of the matrix is [[1, 3, 5], [0, 1, 6], [0, 0, 1]]
```

Practical-7

Program to Calculate GCD(greatest common divisor) in Python.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu May 28 16:20:30 2020

@author: krishanbhadana
"""
def GCD(x,y):
    if(y==0):
        return (x)
    else:
        return (GCD(y,x%y))
x=int(input("Program to calculate GCD\nEnter 1st number"))
y=int(input("Enter 2nd number"))
result=GCD(x,y)
print("GCD is ",result)
```

OUTPUT

```
Program to calculate GCD
Enter 1st number493

Enter 2nd number899
GCD is  29

In [10]:
```

Practical-8

Program to implement floyd-warshall algorithm in Python.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu May 28 16:56:20 2020

@author: krishanbhadana
"""

class Graph:
    def __init__(self):
        # dictionary containing keys that map to the corresponding vertex object
        self.vertices = {}

    def add_vertex(self, key):
        """Add a vertex with the given key to the graph."""
        vertex = Vertex(key)
        self.vertices[key] = vertex

    def get_vertex(self, key):
        """Return vertex object with the corresponding key."""
        return self.vertices[key]

    def __contains__(self, key):
        return key in self.vertices

    def add_edge(self, src_key, dest_key, weight=1):
        """Add edge from src_key to dest_key with given weight."""
        self.vertices[src_key].add_neighbour(self.vertices[dest_key], weight)

    def does_edge_exist(self, src_key, dest_key):
        """Return True if there is an edge from src_key to dest_key."""
        return self.vertices[src_key].does_it_point_to(self.vertices[dest_key])

    def __len__(self):
        return len(self.vertices)

    def __iter__(self):
        return iter(self.vertices.values())


class Vertex:
    def __init__(self, key):
        self.key = key
        self.points_to = {}

    def get_key(self):
```

```
        """Return key corresponding to this vertex object."""
        return self.key

    def add_neighbour(self, dest, weight):
        """Make this vertex point to dest with given edge weight."""
        self.points_to[dest] = weight

    def get_neighbours(self):
        """Return all vertices pointed to by this vertex."""
        return self.points_to.keys()

    def get_weight(self, dest):
        """Get weight of edge from this vertex to dest."""
        return self.points_to[dest]

    def does_it_point_to(self, dest):
        """Return True if this vertex points to dest."""
        return dest in self.points_to


def floyd_warshall(g):
    """Return dictionaries distance and next_v.

    distance[u][v] is the shortest distance from vertex u to v.
    next_v[u][v] is the next vertex after vertex v in the shortest path from u
    to v. It is None if there is no path between them. next_v[u][u] should be
    None for all u.

    g is a Graph object which can have negative edge weights.
    """
    distance = {v:dict.fromkeys(g, float('inf')) for v in g}
    next_v = {v:dict.fromkeys(g, None) for v in g}

    for v in g:
        for n in v.get_neighbours():
            distance[v][n] = v.get_weight(n)
            next_v[v][n] = n

    for v in g:
         distance[v][v] = 0
         next_v[v][v] = None

    for p in g:
        for v in g:
            for w in g:
                if distance[v][w] > distance[v][p] + distance[p][w]:
                    distance[v][w] = distance[v][p] + distance[p][w]
                    next_v[v][w] = next_v[v][p]

    return distance, next_v


def print_path(next_v, u, v):
```

```
    """Print shortest path from vertex u to v.

    next_v is a dictionary where next_v[u][v] is the next vertex after vertex u
    in the shortest path from u to v. It is None if there is no path between
    them. next_v[u][u] should be None for all u.

    u and v are Vertex objects.
    """
    p = u
    while (next_v[p][v]):
        print('{} -> '.format(p.get_key()), end='')
        p = next_v[p][v]
    print('{} '.format(v.get_key()), end='')


g = Graph()
print('Menu')
print('add vertex <key>')
print('add edge <src> <dest> <weight>')
print('floyd-warshall')
print('display')
print('quit')

while True:
    do = input('What would you like to do? ').split()

    operation = do[0]
    if operation == 'add':
        suboperation = do[1]
        if suboperation == 'vertex':
            key = int(do[2])
            if key not in g:
                g.add_vertex(key)
            else:
                print('Vertex already exists.')
        elif suboperation == 'edge':
            src = int(do[2])
            dest = int(do[3])
            weight = int(do[4])
            if src not in g:
                print('Vertex {} does not exist.'.format(src))
            elif dest not in g:
                print('Vertex {} does not exist.'.format(dest))
            else:
                if not g.does_edge_exist(src, dest):
                    g.add_edge(src, dest, weight)
                else:
                    print('Edge already exists.')

    elif operation == 'floyd-warshall':
        distance, next_v = floyd_warshall(g)
        print('Shortest distances:')
        for start in g:
```

```
        for end in g:
            if next_v[start][end]:
                print('From {} to {}: '.format(start.get_key(),
                                                end.get_key()),
                        end = '')
                print_path(next_v, start, end)
                print('(distance {})'.format(distance[start][end]))

    elif operation == 'display':
        print('Vertices: ', end='')
        for v in g:
            print(v.get_key(), end=' ')
        print()

        print('Edges: ')
        for v in g:
            for dest in v.get_neighbours():
                w = v.get_weight(dest)
                print('(src={}, dest={}, weight={}) '.format(v.get_key(),
                                                             dest.get_key(), w))
        print()

    elif operation == 'quit':
        break
```

COMPLEXITY

$O(n^3)$

Output

```
Menu
add vertex <key>
add edge <src> <dest> <weight>
floyd-warshall
display
quit

What would you like to do? add vertex 5

What would you like to do? add vertex 1

What would you like to do? add edge 5 1 15

What would you like to do? floyd-warshall
Shortest distances:
From 5 to 1: 5 -> 1 (distance 15)

What would you like to do?
```

s: **RW** End-of-lines: **LF** Encoding: **UTF-8** Line: **143** Column: **18** Memory: **73 %**