FALL – SEMESTER

Name: Nidhi Singh

Course Code: MCSE502P

Reg. No:22MAI0015

Course-Title: – Design and Analysis of Algorithms

DIGITAL ASSIGNMENT - IV

(LAB)

Slot- L35+L36

Faculty: Dr. MOHAMMAD ARIF - SCOPE

---

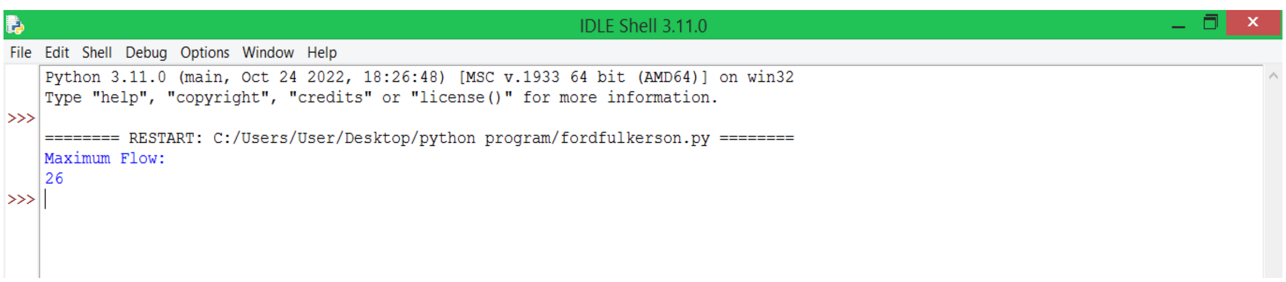
1. Implement Ford -Fulkerson and Edmond – Karp Algorithms

A) Ford-Fulkerson Algorithm

```
def BFS(graph, s, t, parent):
    # Return True if there is node that has not iterated.
    visited = [False] * len(graph)
    queue = []
    queue.append(s)
    visited[s] = True

    while queue:
        u = queue.pop(0)
        for ind in range(len(graph[u])):
            if visited[ind] is False and graph[u][ind] > 0:
                queue.append(ind)
                visited[ind] = True
                parent[ind] = u

    return True if visited[t] else False


def FordFulkerson(graph, source, sink):
    # This array is filled by BFS and to store path
    parent = [-1] * (len(graph))
    max_flow = 0
    while BFS(graph, source, sink, parent):
        path_flow = float("Inf")
        s = sink

        while s != source:
            # Find the minimum value in select path
            path_flow = min(path_flow, graph[parent[s]][s])
            s = parent[s]
```

```
        max_flow += path_flow
        v = sink

        while v != source:
            u = parent[v]
            graph[u][v] -= path_flow
            graph[v][u] += path_flow
            v = parent[v]
    return max_flow


graph = [
    [0, 15, 14, 0, 0, 0],
    [0, 0, 11, 13, 0, 0],
    [0, 4, 0, 0, 17, 0],
    [0, 0, 9, 0, 0, 21],
    [0, 0, 0, 8, 0, 5],
    [0, 0, 0, 0, 0, 0],
]

source, sink = 0, 5
print("Maximum Flow: ")
print(FordFulkerson(graph, source, sink))
```

Output :-



B). Edmonds-Karp Algorithm :-

Code :-

```
#Edmonds-Karp Algorithm
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, 1, 1, 1 ],  # 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, 7 ]]  # 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)
```

Output :-

```
IDLE Shell 3.11.0
File  Edit  Shell  Debug  Options  Window  Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================== RESTART: C:/Users/User/karp.py ========================
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)], 4: [(0, 4)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)], 4: [(0, 4)], 5: [(0, 5)]}
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)], 4: [(0, 4)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)], 4: [(0, 4)], 5: [(0, 3), (3, 5)]}
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 4: [(0, 4)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 4: [(0, 4)], 3: [(0, 1), (1, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 4: [(0, 4)], 3: [(0, 1), (1, 3)], 5: [(0, 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, 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)]}
Ln: 1  Col: 0
```

```
IDLE Shell 3.11.0
File  Edit  Shell  Debug  Options  Window  Help
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)], 4: [(0, 4)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)], 4: [(0, 4)], 5: [(0, 5)]}
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)], 4: [(0, 4)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 3: [(0, 3)], 4: [(0, 4)], 5: [(0, 3), (3, 5)]}
{0: [], 1: [(0, 1)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 4: [(0, 4)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 4: [(0, 4)], 3: [(0, 1), (1, 3)]}
{0: [], 1: [(0, 1)], 2: [(0, 2)], 4: [(0, 4)], 3: [(0, 1), (1, 3)], 5: [(0, 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, 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, 2), (2, 4)]}
Edmonds-Karp algorithm
max_flow_value is:  5
>>>
Ln: 36  Col: 0
```

OR

Code 2:-

```
class FlowNetwork:
    def __init__(self, graph, sources, sinks):
        self.sourceIndex = None
        self.sinkIndex = None
        self.graph = graph

        self._normalizeGraph(sources, sinks)
        self.verticesCount = len(graph)
        self.maximumFlowAlgorithm = None

    # make only one source and one sink
    def _normalizeGraph(self, sources, sinks):
        if sources is int:
            sources = [sources]
        if sinks is int:
            sinks = [sinks]

        if len(sources) == 0 or len(sinks) == 0:
            return

        self.sourceIndex = sources[0]
        self.sinkIndex = sinks[0]

        # make fake vertex if there are more
        # than one source or sink
        if len(sources) > 1 or len(sinks) > 1:
            maxInputFlow = 0
            for i in sources:
                maxInputFlow += sum(self.graph[i])

            size = len(self.graph) + 1
            for room in self.graph:
                room.insert(0, 0)
            self.graph.insert(0, [0] * size)
            for i in sources:
                self.graph[0][i + 1] = maxInputFlow
            self.sourceIndex = 0

            size = len(self.graph) + 1
```

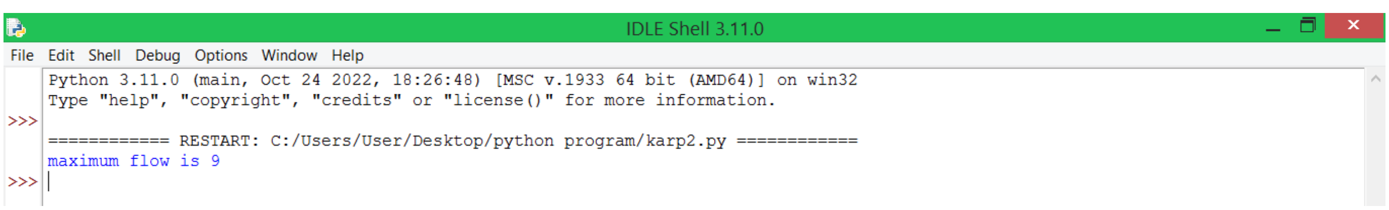
```
        for room in self.graph:
            room.append(0)
        self.graph.append([0] * size)
        for i in sinks:
            self.graph[i + 1][size - 1] = maxInputFlow
        self.sinkIndex = size - 1

    def findMaximumFlow(self):
        if self.maximumFlowAlgorithm is None:
            raise Exception("You need to set maximum flow algorithm before.")
        if self.sourceIndex is None or self.sinkIndex is None:
            return 0

        self.maximumFlowAlgorithm.execute()
        return self.maximumFlowAlgorithm.getMaximumFlow()

    def setMaximumFlowAlgorithm(self, Algorithm):
        self.maximumFlowAlgorithm = Algorithm(self)


class FlowNetworkAlgorithmExecutor:
    def __init__(self, flowNetwork):
        self.flowNetwork = flowNetwork
        self.verticesCount = flowNetwork.verticesCount
        self.sourceIndex = flowNetwork.sourceIndex
        self.sinkIndex = flowNetwork.sinkIndex
        # it's just a reference, so you shouldn't change
        # it in your algorithms, use deep copy before doing that
        self.graph = flowNetwork.graph
        self.executed = False

    def execute(self):
        if not self.executed:
            self._algorithm()
            self.executed = True

    # You should override it
    def _algorithm(self):
        pass
```

```
class MaximumFlowAlgorithmExecutor(FlowNetworkAlgorithmExecutor):
    def __init__(self, flowNetwork):
        super().__init__(flowNetwork)
        # use this to save your result
        self.maximumFlow = -1

    def getMaximumFlow(self):
        if not self.executed:
            raise Exception("You should execute algorithm before using its result!")

        return self.maximumFlow


class PushRelabelExecutor(MaximumFlowAlgorithmExecutor):
    def __init__(self, flowNetwork):
        super().__init__(flowNetwork)

        self.preflow = [[0] * self.verticesCount for i in range(self.verticesCount)]

        self.heights = [0] * self.verticesCount
        self.excesses = [0] * self.verticesCount

    def _algorithm(self):
        self.heights[self.sourceIndex] = self.verticesCount

        # push some substance to graph
        for nextVertexIndex, bandwidth in enumerate(self.graph[self.sourceIndex]):
            self.preflow[self.sourceIndex][nextVertexIndex] += bandwidth
            self.preflow[nextVertexIndex][self.sourceIndex] -= bandwidth
            self.excesses[nextVertexIndex] += bandwidth

        # Relabel-to-front selection rule
        verticesList = [
            i
            for i in range(self.verticesCount)
            if i != self.sourceIndex and i != self.sinkIndex
        ]

        # move through list
        i = 0
        while i < len(verticesList):
```

```
            vertexIndex = verticesList[i]
            previousHeight = self.heights[vertexIndex]
            self.processVertex(vertexIndex)
            if self.heights[vertexIndex] > previousHeight:
                # if it was relabeled, swap elements
                # and start from 0 index
                verticesList.insert(0, verticesList.pop(i))
                i = 0
            else:
                i += 1

        self.maximumFlow = sum(self.preflow[self.sourceIndex])

    def processVertex(self, vertexIndex):
        while self.excesses[vertexIndex] > 0:
            for neighbourIndex in range(self.verticesCount):
                # if it's neighbour and current vertex is higher
                if (
                    self.graph[vertexIndex][neighbourIndex]
                    - self.preflow[vertexIndex][neighbourIndex]
                    > 0
                    and self.heights[vertexIndex] > self.heights[neighbourIndex]
                ):
                    self.push(vertexIndex, neighbourIndex)

            self.relabel(vertexIndex)

    def push(self, fromIndex, toIndex):
        preflowDelta = min(
            self.excesses[fromIndex],
            self.graph[fromIndex][toIndex] - self.preflow[fromIndex][toIndex],
        )
        self.preflow[fromIndex][toIndex] += preflowDelta
        self.preflow[toIndex][fromIndex] -= preflowDelta
        self.excesses[fromIndex] -= preflowDelta
        self.excesses[toIndex] += preflowDelta

    def relabel(self, vertexIndex):
        minHeight = None
        for toIndex in range(self.verticesCount):
            if (
```

```
                self.graph[vertexIndex][toIndex] - self.preflow[vertexIndex][toIndex]
                > 0
            ):
                if minHeight is None or self.heights[toIndex] < minHeight:
                    minHeight = self.heights[toIndex]

        if minHeight is not None:
            self.heights[vertexIndex] = minHeight + 1


if __name__ == "__main__":
    entrances = [0]
    exits = [3]
    graph = [
        [0, 0, 8, 9, 0, 0],
        [0, 0, 5, 7, 0, 0],
        [0, 0, 0, 0, 6, 4],
        [0, 0, 0, 0, 9, 6],
        [0, 0, 5, 0, 0, 0],
        [0, 0, 0, 0, 0, 0],
    ]
    # graph = [[0, 7, 0, 0], [0, 0, 6, 0], [0, 0, 0, 8], [9, 0, 0, 0]]

    # prepare our network
    flowNetwork = FlowNetwork(graph, entrances, exits)
    # set algorithm
    flowNetwork.setMaximumFlowAlgorithm(PushRelabelExecutor)
    # and calculate
    maximumFlow = flowNetwork.findMaximumFlow()

    print(f"maximum flow is {maximumFlow}")
```

Output :-



2. Implement Cycle cancelling algorithm

Code :-

```
from sys import maxsize
from typing import List

# Stores the found edges
found = []

# Stores the number of nodes
N = 0

# Stores the capacity
# of each edge
cap = []

flow = []

# Stores the cost per
# unit flow of each edge
cost = []

# Stores the distance from each node
# and picked edges for each node
dad = []
dist = []
pi = []

INF = maxsize // 2 - 1

# Function to check if it is possible to
# have a flow from the src to sink
def search(src: int, sink: int) -> bool:

    # Initialise found[] to false
    found = [False for _ in range(N)]

    # Initialise the dist[] to INF
    dist = [INF for _ in range(N + 1)]
```

```
# Distance from the source node
dist[src] = 0

# Iterate until src reaches N
while (src != N):
    best = N
    found[src] = True

    for k in range(N):

        # If already found
        if (found[k]):
            continue

        # Evaluate while flow
        # is still in supply
        if (flow[k][src] != 0):

            # Obtain the total value
            val = (dist[src] + pi[src] -
                    pi[k] - cost[k][src])

            # If dist[k] is > minimum value
            if (dist[k] > val):

                # Update
                dist[k] = val
                dad[k] = src

        if (flow[src][k] < cap[src][k]):
            val = (dist[src] + pi[src] -
                    pi[k] + cost[src][k])

            # If dist[k] is > minimum value
            if (dist[k] > val):

                # Update
                dist[k] = val
                dad[k] = src
```

```
            if (dist[k] < dist[best]):
                best = k

        # Update src to best for
        # next iteration
        src = best

    for k in range(N):
        pi[k] = min(pi[k] + dist[k], INF)

    # Return the value obtained at sink
    return found[sink]

# Function to obtain the maximum Flow
def getMaxFlow(capi: List[List[int]],
            costi: List[List[int]],
            src: int, sink: int) -> List[int]:

    global cap, cost, found, dist, pi, N, flow, dad
    cap = capi
    cost = costi

    N = len(capi)
    found = [False for _ in range(N)]
    flow = [[0 for _ in range(N)]
            for _ in range(N)]
    dist = [INF for _ in range(N + 1)]
    dad = [0 for _ in range(N)]
    pi = [0 for _ in range(N)]

    totflow = 0
    totcost = 0

    # If a path exist from src to sink
    while (search(src, sink)):

        # Set the default amount
        amt = INF
        x = sink
```

```
        while x != src:
            amt = min(
                amt, flow[x][dad[x]] if
                (flow[x][dad[x]] != 0) else
                cap[dad[x]][x] - flow[dad[x]][x])
            x = dad[x]

        x = sink

        while x != src:
            if (flow[x][dad[x]] != 0):
                flow[x][dad[x]] -= amt
                totcost -= amt * cost[x][dad[x]]

            else:
                flow[dad[x]][x] += amt
                totcost += amt * cost[dad[x]][x]

            x = dad[x]

        totflow += amt

    # Return pair total cost and sink
    return [totflow, totcost]

# Driver Code
if __name__ == "__main__":

    s = 0
    t = 4

    cap = [ [ 0, 30, 10, 0, 33 ],
            [ 0, 0, 20, 0, 0 ],
            [ 0, 0, 0, 10, 16 ],
            [ 0, 0, 0, 0, 22 ],
            [ 0, 0, 0, 0, 0 ] ]

    cost = [ [ 0, 21, 5, 0, 20 ],
            [ 0, 0, 10, 3, 0 ],
```

```
        [ 0, 0, 0, 5, 10 ],
        [ 0, 0, 0, 0, 11],
        [ 0, 0, 0, 0, 0 ] ]

    ret = getMaxFlow(cap, cost, s, t)

    print("The Cost of this Flow: {} \nThe Cost before apply Cycle cancelling algorithm: {}".format(ret[0], ret[1]))
```

Code :-