Assignment 4

Srinithish K

Graph base class that is used for all problems

In [3]:

```
class GraphNode():


    def __init__(self,graphNodeId):
        self.ID = graphNodeId
        self.visited = False
        self.distance = 0
        self.neighbours = []
        self.weights = []
        self.parent = None
        self.colour = 'White'
        self.startTime = 0
        self.endTime = 0
        self.set = None ## A or B

    def addNeighbour(self,graphNodeId):
        self.neighbours.append(graphNodeId)

    def getNeighbours(self):

        return self.neighbours

    def addWeight(self,weight):
        self.weights.append(weight)
```

In [4]:

```
def buildGraph(numNodes,listOfStrings):
    graphDict = {}
    for i in range(numNodes):
        graphDict[i] = GraphNode(i)

    for graphComb,weight in listOfStrings:

        graphDict[graphComb[0]].addNeighbour(graphComb[1])
        graphDict[graphComb[0]].addWeight(weight)

    return graphDict
```

Question 1

Description

1. Do a topological sort on DAG which is of complexity O(V+E)
2. In the order of topologically sorted array S check if node S_{i+1} is a child of node S_i where i is the index position of the sorted list.

   Say [A,C,B,D]

   - check if C is a child of A
   - check if B is child of C
   - check if D is child of B
   - So on till the last element of the sorted nodes Break if any of the above condition is not satisified
3. if S_{i+1} is a child of S_i for all i then there exists a path which passes through all vertices else there is no such path.
4. The above checking takes at most $O(E)$ hence the algorithm is at most O(V+E)

Pseudo code ¶

```
### bulid a DAG graph
graph = buildGraph(6,listOfConnections)

### topologically sort the graph
sortedNodes = topologicalSort(graph)

def checkExistantPath(sortedNodes,graphDict):

    ##check if (i+1) is child of (i)

    for i in range(len(sortedNodes)-1):

        fromNode = sortedNodes[i]
        toNode = sortedNodes[i+1]

        if toNode not in graphDict[fromNode].neighbours:
            return False

    return True

checkExistantPath(sortedNodes,graph)
```

Code

In [7]:

```
listOfConnections = [((0,1),1),((1,2),2),((1,5),5),((2,4),4),((2,3),1),((4,5),3)]   ### ((N
sortedNodes = []
currTime = 0

def DFSVisit(graphDict,currNodeID):
    global sortedNodes
    global currTime
    currTime +=1
#    print(currNodeID," start time ",currTime)
    graphDict[currNodeID].visited = True
    graphDict[currNodeID].startTime = currTime

    for neighNodeID in graphDict[currNodeID].neighbours:
        neigNode = graphDict[neighNodeID]

        if neigNode.visited == False:
#            print("Visiting ",neigNode.getID())
            neigNode.visited = True

            DFSVisit(graphDict,neighNodeID)


    currTime +=1
#    print(currNodeID," end time ",currTime)
    graphDict[currNodeID].endTime = currTime
    sortedNodes.append(currNodeID)


def DFS(graphDict):

    global currTime

    for nodeID,node in graphDict.items():

        if node.visited == False:


            DFSVisit(graphDict,node.ID)

def topologicalSort(graphDict):
    DFS(graphDict)
    return sortedNodes
```

In [5]:

```
graph = buildGraph(6,listOfConnections)
topologicalSort(graph)
sortedNodes = list(reversed(sortedNodes))

def checkExistantPath(sortedNodes,graphDict):

    for i in range(len(sortedNodes)-1):

        fromNode = sortedNodes[i]
        toNode = sortedNodes[i+1]
        ###may need to change this
        if toNode not in graphDict[fromNode].neighbours:
            return False

    return True

print("The nodes in topological order are ", sortedNodes)
checkExistantPath(sortedNodes,graph)
```

```
The nodes in topological order are  [0, 1, 2, 3, 4, 5]
```

Out[5]:

```
False
```

Question 2

```
For each edge v-t in G:

  1. Form a graph that is the same as G, except that edge v-t is removed.
  2. Record the shortest path dist(v, t) from v to t using dijkstra (this can be d
one tracking the parents from t till you arrive at v)

  3. If dist(v,t) is not Infinity then there is a cycle

          In the recorded shortest path,

           if there was edge e' with weight w(e') in the shortest path less than w
(v-t)
                - Then remove the edge w(e') from the graph
                - add e' to the set of edges in the feedback set
                - replace edge e(v-t) back into the graph

           else put e(v-t) in the feedback edge set

At the end we have Feedback set of edges that render the Graph Acyclic
```

Question 3

Description

1. Do a DFS on the nodes and assign alternatively Set A and Set B to parent and child
2. In the process if you encounter a node that is visited and has the following conditions,

```
if parentNode.set == 'A' and neighNode.set == 'A':

          return False

 if parentNode.set == 'B' and neighNode.set == 'B':

         return False
```

3. If there is no violation of the above rule *return True*

This algorithm runs with complexity O(V+E) as each edge is checked once

Pseudo Code

```
def DFSCheck(graphDict,parentNode):


    parentNode.visited = True

    for neighNode in parentNode.neighbours:


        ## assign sets if set is still unassigned
        if neighNode.set is None:
            if parentNode.set == 'A':
                neighNode.set = 'B'
            elif parentNode.set == 'B':
                neighNode.set = 'A'

        ###  check compliance
        elif neighNode.set is not None:
            if parentNode.set == 'A' and neighNode.set == 'A':
                return False
            if parentNode.set == 'B' and neighNode.set == 'B':
                return False


        ##if the node is already note visisted
        if neighNode.visited == False:

            neighNode.visited = True
            DFSCheck(graphDict,neighNode)
```

```
## main function which runs DFSCheck on all nodes once
def isPartitioon(graphDict):


  for parentNode in graphDict.items():


          if parentNode.visited == False:
              ##  if node not visited initialise with set A as the
              node.set = 'A'


          Flag = DFSCheck(graphDict,parentNode)

  if Flag == False:

      return False

   else:
        return True
```

Code

In [0]:

```
listOfConnections = [((0,1),1),((1,2),1),((0,3),1),((3,2),1)]

graph = buildGraph(5,listOfConnections)
```

In [7]:

```
def DFSCheck(graphDict,currNodeID):


    parentNode = graphDict[currNodeID]
    parentNode.visited = True

    for neighNodeID in parentNode.neighbours:
        neighNode = graphDict[neighNodeID]

        ## assign sets
        if neighNode.set is None:
            if parentNode.set == 'A':
                neighNode.set = 'B'
            elif parentNode.set == 'B':
                neighNode.set = 'A'

        ###  check compliance
        elif neighNode.set is not None:
            if parentNode.set == 'A' and neighNode.set == 'A':
                return False
            if parentNode.set == 'B' and neighNode.set == 'B':
                return False


        if neighNode.visited == False:

            neighNode.visited = True
            DFSCheck(graphDict,neighNodeID)

def isPartitioon(graphDict):


    for nodeID,node in graphDict.items():

        if node.visited == False:
            ## initialise with A
            node.set = 'A'


        Flag = DFSCheck(graphDict,node.ID)

    return False if Flag == False else True

print(isPartitioon(graph))
```

```
True
```

Question 4

Approach 1 with O(V+E) complexity

```
    Do a topological sort on the Graph

    initialise all nodes with distance as 0

    For each parentNode in topological order:

        For each neighbour in node.neighbours:
            ##moving each child as far as possible based on their prerequisites semest
    ers
            if neighbour.distance < parentNode.distance + 1
                neighbour.distance = parentNode.distance + 1

    Finally go through all the nodes and group them based on the their distances
```

This takes O(V+E) for topological sort and O(E) for the assigning distances

In [27]:

```
sortedNodes = []
listOfConnections = [((0,3),1),((1,2),1),((1,3),1),((2,3),1)]
graph = buildGraph(4,listOfConnections)
topologicalSort(graph)
sortedNodes = list(reversed(sortedNodes))

import collections as col
def assignSemesters(graph,sortedNodes):
    dictOfSemesters  = col.defaultdict(list)

    for parentNodeID in sortedNodes:

        parentNode = graph[parentNodeID]
        for childNodeID in parentNode.neighbours:
            childNode = graph[childNodeID]

            if childNode.distance < parentNode.distance + 1:
                childNode.distance = parentNode.distance + 1

    ### group them according the the semesters
    for nodeID in graph:
        dictOfSemesters[graph[nodeID].distance].append(nodeID)
    return dictOfSemesters

sems = assignSemesters(graph,sortedNodes)
print('number of semesters ', len(sems) ,' and grouped by ', sems)
```

```
number of semesters  3  and grouped by  defaultdict(<class 'list'>, {0: [0,
1], 1: [2], 2: [3]})
```

Approach 2 with $O(V^2)$ complexity

Description (uses adjacency matrix)

1. Imagine a Graph to be represented in Adjacency matrix

2. Collect all the nodes {V'} that have zero incoming edges i.e in-degree 0 which would be scheduled the first. Basiclly columns in the Adjacency Matrix that are all zero
3. Remove the collected nodes from Graph G since they can all be scheduled in a semester
4. Now the adjacency matrix reduces to smaller matrix without these nodes
5. Repeat 1 to 4 on the reduced adjacency matrix till there are no more vertices left

This algorithm takes $O(V^2)$ complexity

pseudo code

```
def assignSemesters(adjacencyMatrix):

    ### set of all columns
    listOfColumnNames = list(range(len(adjacencyMatrix[0])))

    listOfSems = []

    while (len(listOfColumnNames)>0): ### runs equal to max V times

        semCourses = []
        columnIndices = []


        for column in range(len(adjacencyMatrix)):

            if sum(adjacencyMatrix[:,column]) == 0: ## no incoming edges
                semCourses.append(listOfColumnNames[column])
                columnIndices.append(column)


        ### remove the nodes with in-degree zero
        ### remove correspingin columns and rows and update the adjacencyMatrix
        ### ~ is negation
        adjacencyMatrix = adjacencyMatrix[~columnIndices,~columnIndices]


        ## remove the nodes that are scheduled in this semster
        listOfColumnNames = listOfColumnNames - semCourses


        ###grouped courses
        listOfSems.append(semCourses)




    return listOfSems
```

Code

In [8]:

```
import numpy as np
adjacencyMatrix = np.array([[0,0,0,1],
                            [0,0,1,1],
                            [0,0,0,1],
                            [0,0,0,0]])


def assignSemesters(adjacencyMatrix):

    listOfColumnNames = list(range(len(adjacencyMatrix[0]))) ### set of all columns

    listOfSems = []
    while (len(listOfColumnNames)>0): ### runs equal to max V times

        semCourses = []
        columnIndices = []
        for column in range(len(adjacencyMatrix)):
            if sum(adjacencyMatrix[:,column]) == 0: ## no incoming edges
                semCourses.append(listOfColumnNames[column])
                columnIndices.append(column)

        copyAdj = np.delete(adjacencyMatrix,columnIndices,0) ### delete correspingin columr
        copyAdj = np.delete(copyAdj,columnIndices,1) ## delete coreesonponding rows


        ## remove rest of the columns
        listOfColumnNames = np.delete(listOfColumnNames, columnIndices)


        adjacencyMatrix = copyAdj

        listOfSems.append(semCourses)




    return listOfSems

listOfSems = assignSemesters(adjacencyMatrix)

print("the semester grouping for the above configuration is :" , listOfSems)
```

```
the semester grouping for the above configuration is : [[0, 1], [2], [3]]
```

Question 5

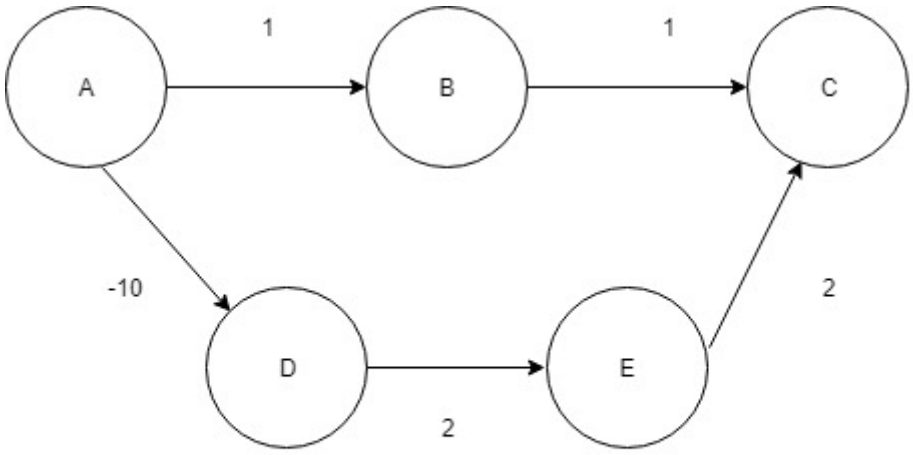

Fig 1

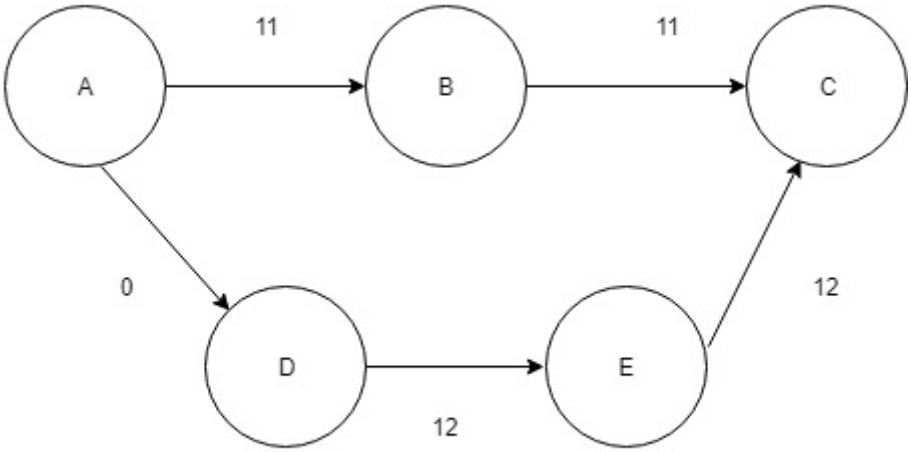

Fig 2

This is wrong and proved by the following above example

In the ***fig1*** with negative edges,

- For the path A -> B -> C the path length is 2.
- For the path A -> D -> E -> C the path length is -6.

  The shortest path is A -> D -> E -> C

Now lets add +10 to all the edges so that every edge is now positive which results in ***fig2***

- For the path A -> B -> C the path length is 22.
- For the path A -> D -> E -> C the path length is 24.

The shortest path is now A -> B -> C not as before hence the solutions are not the same if you offset the paths by large positive number

Question 6

```
for each node in graph as source node (s):

      1. Run Dijkstra on the Graph with source as 's'
      2. We would have now computed distances  d(s,t) from the source node to all
 other target nodes (t)
      3. Compute the all the cycle distances starting at 's' and ending at 's'
            cycleLength = d(s,t) + weight(t,v)
            Note: if edge E(t -> v) is not existant add weight(t,v) as infinity

      4. Append all the cycleLenghts in a list

Finally find:
minCycle = min(cycleLengths)

if min(cycleLengths) == Infinty
      return 'NoCycle'
else:
    return minCycle
```

1. The outer loop in the **findShortestCycle** runs on every vertex in graph
2. Inside this loop **dijkstra** runs , which takes $O(V^2)$ when prioprty queue is implemented as a simple array
3. Inner loop also runs on each vertex in the graph hence $O(V)$ Hence total complexity is $V * O(V^2) + O(V^2)$ ie $O(V^3)$

In [6]:

```
listOfConnections = [((0,1),2),((1,2),1),((0,3),1),((3,2),1),((2,0),2),((4,5),1)]

graph = buildGraph(6,listOfConnections)

def initialiseSourceAndGraph(sourceID,graph):

    for nodeID in graph:
        if nodeID == sourceID:
            graph[nodeID].distance = 0
        else:
            graph[nodeID].distance = float('inf')
    return graph

def relax(parentNode,childNode,weight):


    if childNode.distance > (parentNode.distance + weight):
       childNode.distance = parentNode.distance + weight

def dijkstra(sourceNodeID,graph):

    initialiseSourceAndGraph(sourceNodeID,graph) ## makes source 0 and allother distances a

    prioityQueue = graph.copy() ### not a deep copy

    while len(prioityQueue) > 0:

        ### extract min
        ## prirority queue implemented as array
        minDistNodeID = min(prioityQueue, key = lambda i:  prioityQueue[i].distance)
        del prioityQueue[minDistNodeID]

        for childNodeID,weight in zip(graph[minDistNodeID].neighbours,graph[minDistNodeID].

            parentNode = graph[minDistNodeID]
            childNode = graph[childNodeID]
            relax(parentNode,childNode,weight)
```

In [7]:

```
def findShortestCycle(graph):
    cycleLenghts = []
    for startVertex in graph: ## on every vertex as source run dijkstra

        dijkstra(startVertex,graph)

        for endVertex in graph:
            weight = float('inf')

            if startVertex in graph[endVertex].neighbours:
                weightIndex = graph[endVertex].neighbours.index(startVertex)
                weight =  graph[endVertex].weights[weightIndex]

            cycleLength = weight + graph[endVertex].distance ## record cycle lengths
            cycleLenghts.append(cycleLength)

    shortest = min(cycleLenghts) ## get the minimum possible cycle lengths if any
    if shortest == float('inf'):
        return 'No Cycle'
    else:
        return shortest

print("the shortest cycle is ",findShortestCycle(graph))
```

```
the shortest cycle is  4
```

Question 7

Part A

Description

1. Start a BFS from starting at node "s"
2. Put 's' in Queue Q
3. Extract a element from Q and and expand its neighbours
4. From the neighbours put only those nodes into the queue whose edge weights $L_e \leq L$
5. After Dequeuing, first check if this node was the target city "t" and else continue with the BFS

```
Input: G=(V,E), edge weights "L(e)", initial node "s", final node "t"
    and L fuel tank capacity
Output: True/False
```

Pseudocode

In [0]:

```
def checkPossibility(graph,startNode,terminalNode,FuelTankCapacity):
    q = queue.Queue()
    q.put(startNode) # Push "startNode" to the queue

    while q.qsize > 0: # queue is not empty
        nextNode = q.get() # extract an element from the queue
        nextNode.visited = True

        if nextNode == terminalNode: # Target city reached
            return True

        for  neigh in graph[nextNode].neighbours: # exploring neighbors of nextNode

            ## if distance betweeen cities is less that the fuel tank capacity and the node
            if EdgeWeight[nextNode,neigh] <= FuelTankCapacity and nextNode.visited == False

                q.put(neigh) # Push "neigh" to the queue

    return False
```

Question 7

Part B

The problem here is to minimize the inter cities distance instead of optimizing the total distance from source "s" to a terminal node 't',

1. We can use Dijkstra's algorithm but instead of basing the priority in the queue on the distance from the source , we would rank it accorifing to Edge weights from parent to neighbours while putting into the queue.
2. This ensures that we are minimsing the segemental length (city distance) at every stage

This is exactly same as Dijsktra hence the complexity is O(V+E)log(V)

In [2]:

```
def getMinTankCapacity(Graph,startNode,terminalNode):

    # elements in priority-queue => (priority,node,max_edge)
    priorQ = queue.PriorityQueue()

    # Push (edge-weight as priority,node,max_edge along the path) to the priority-queue
    priorQ.put((0,startNode,0))

    while not priorQ.empty():  # till queue is not empty

        # extract an element with minimum edge length
        (priority,fromCity,max_edge_till_now) = priorQ.get()

        fromCity.visited = True

        if fromCity == terminalNode: # goal city state reached

            # The max edge weight along the path would be the minimum fuel tank capacity ne
            return max_edge_till_now

        for nextCity in graph[fromCity].neighbours: # exploring neighbors of fromCity
            distFtoN = E(fromCity,nextCity) ##city distances
            max_edge_till_now = max(distFtoN,max_edge_till_now)

            if nextCity.visited == False:

            # Push (edge-weight as priority,node,max_edge along the path)  to the priority-
                priorQ.put((distFtoN,nextCity,max_edge_till_now))

    return False
```