AA_Assignment4_Srinithish

April 8, 2019

1 Assignment 4

Graph base class that is used for every other problem

In [0]:
```
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 [0]:
```
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
```

1.1 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)

1.1.1 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)
```

1.1.2 Code

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

1.1.3 Question 2

```
For each edge v-t

  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 done tracking

  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 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
```

1.2 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)

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
```

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

graph = buildGraph(5,listOfConnections)
```

In [7]:
```python
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:
```

```python
                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
```

1.2.1 Question 4

Description

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 algorith takes O(V+E) 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])
```