```
#Taylor Heilman
#an undirected simple graph
#Feb 11, 2016

import copy


class Graph(object):

    def __init__(self):
        #Graph constructor
        self.Graph = {}


    def add_vertices(self, vertices):
        #Add a list of vertices to the graph
        length = len(vertices)
        for i in range (length):
            if vertices[i] not in self.Graph:   #check if vertex exists in dictionary alr
eady
                self.Graph[vertices[i]] = []


    def delete_vertex(self, v):
        #Delete a vertex from the graph.
        if v in self.Graph:
            del self.Graph[v]           #delete key from dictionary
            for item in self.Graph:
                if v in self.Graph[item]:       #delete vertex from other keys' values
                    self.Graph[item].remove(v)




    def contract_edge(self, e):
        vertex1=e[0]        #first element of edge
        vertex2=e[1]        #second element of edge

        if vertex1 < vertex2:
            for item in (self.Graph[vertex2]):
                if item not in self.Graph[vertex1] and item != vertex1:
                    self.Graph[vertex1].append(item)    #copy vertex2's edges into vertex
1's

            for item in self.Graph:
                if vertex2 in self.Graph[item]:
                    self.Graph[item].remove(vertex2)    #remove vertex2 from other keys'
values

            del (self.Graph[vertex2])   #delete vertex2 from dictionary


        else:
            for item in (self.Graph[vertex1]):
                if item not in self.Graph[vertex2] and item != vertex2:
                    self.Graph[vertex2].append(item) #copy vertex1's edges into vertex2's

            for item in self.Graph:
                if vertex1 in self.Graph[item]:
                    self.Graph[item].remove(vertex1) #remove vertex1 from other keys' val
ues

            del (self.Graph[vertex1])  #delete vertex1 from dictionary




    def delete_edge(self, e):
        vertex1=e[0]        #first element of edge
        vertex2=e[1]        #second element of edge
```

```
        if vertex1 in self.Graph and vertex2 in self.Graph:
            if vertex1 in self.Graph[vertex2]:          #check if element1 is in element2
’s values
                    self.Graph[vertex2].remove(vertex1)     #remove element1 from values
if true
            if vertex2 in self.Graph[vertex1]:          #check if element2 is in element1
’s values
                    self.Graph[vertex1].remove(vertex2) #remove element2 from values if t
rue


    def vertices(self):
        #Return a list of nodes in the graph.
        return list(self.Graph.keys())

    def add_edges(self, edges):
        #Add a list of edges to the graph
        edges = list(edges)
        length1 = len(edges)
        for i in range (length1):
            temp = edges[i]
            first = temp[0]     #first vertex of pair
            second = temp[1]    #second vertex of pair

            if first not in self.Graph:         #Vertex1 is not in dictionary
                self.Graph[first] = [second]
            else:
                if second not in self.Graph[first]:     #Vertex1 in dictionary but Vertex
 2 isn’t
                    self.Graph[first].append(second)

            if second not in self.Graph:        #Vertex2 is not in dictionary
                self.Graph[second] = [first]

            else:
                if first not in self.Graph[second]:     #Vertex2 is in dictionary but Ver
tex1 isn’t
                    self.Graph[second].append(first)



    def edges(self):
        #Return a list of edges in the graph
        edge = []
        for vertex1 in self.Graph:
            for vertex2 in self.Graph[vertex1]:
                if (vertex2, vertex1) not in edge:      #check if inverse of edge is alre
ady in the list
                    edge.append((vertex1, vertex2))     #ex. if (u,v) is in list (v,u) wo
n’t be appended


        return edge


    #Breadth First Search
    def BFS(self,v):
        if v in self.Graph:
            #step 1
            graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
            S = [v]                           #list S
            label = []                         #list to hold labels, index refers to label
            T = []                              #empty list of edges
            #step 2
            C = graph[v]            #list of v’s neighbors
            C.sort()
            #step 3
            label.append(v)  #label v as 0
            bstar = v        #bstar = vertex we’re branching to
```

```
            for item in graph:
                if bstar in graph[item]:          #remove bstar from adjacency lists
                    graph[item].remove(bstar)

            #step 4
            while(True):
                neighbors = graph[bstar]        #list of adjacent vertices

                for vert in label:
                    if vert in neighbors:       #neighbors = adjacent vertices that haven
’t been visited
                        neighbors.remove(vert)

                for item in neighbors:          #iterate through neighbors
                    S.append(item)          #add vertex to S
                    S.sort()
                    T.append((bstar,item))  #add edge to T
                    label.append(item)      #label item

                graph2 = copy.deepcopy(graph)       #create copy of graph bc can’t edit s
omething you’re iterating

                for thing in C:
                    for item in graph:
                        if thing in graph2[item]:       #remove bstar from adjacency list
s
                            graph2[item].remove(thing)


                graph = graph2          #update graph

                #step 5
                if (len(C) == 0):           #halting condition
                    return T

                else:
                    bstar = C[0]            # define a new bstar to be min vertex in C
                    C.remove(bstar)





    #Depth First Search
    def DFS(self,v):
        if v in self.Graph:     #check that vertex v is in the graph
            #step 1
            graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
            T = []                                  #empty list of edges
            bstar = v                               #initial branch vertex
            pstar = v
            history = [v]                           #history of vertices that are used as
 bstar
            label = []                              #list to hold labels
            label.append(v)                         #label v as 0
            U1 = graph.keys()                       #keys of dict
            U = []                                  #list of unlabeled vertices

            for item in U1:
                U.append(item)
            U.remove(v)         #remove v since it is labeled
            U.sort()

            neighbors = graph[v]        #list of vertices adjacent to v
            neighbors.sort()
            intersection = list(set(U) & set(neighbors))        #U intersect neighbors
            intersection.sort()                                 #finds unlabeled niehgbor
s

            #step 2
```

```
            while (True):
                while (intersection):
                    w = intersection[0]
                    label.append(w)          #label neighbor of bstar
                    T.append((bstar,w))      #add edge to T
                    U.remove(w)              #remove labeled vertex from unlabeled list
                    pstar = bstar            #help for backtracking
                    bstar = w
                    history.append(bstar)       #update history list
                    neighbors = graph[bstar]    #get neighbors
                    neighbors.sort()
                    intersection = list(set(U) & set(neighbors))    #get intersect of U a
nd neighbors
                    intersection.sort()

                neighbors = graph[bstar]            #update neighbors
                neighbors.sort()
                intersection = list(set(U) & set(neighbors))     #used to check for halti
ng condition
                intersection.sort()

                if (len(intersection) == 0):    #if U intersect neighbors = []
                    num = history.index(bstar)
                    bstar = history[num-1]      #bstar = pstar

                if (bstar == v and intersection == [] and len(U) == 0): #halting conditio
n
                    return T        #return spanning tree

                if (intersection != []):
                    history.append(bstar)       #update history list







    def MaxDegC(self):
        order = list()      #list of vertices with degrees
        sort = list()       #sorted list of veritces based off of degrees
        colors = {}         #dictionary to keep track of color labels
        for index in range (len(self.Graph)):       #creates dictionary of colors for wor
st case scenario
            colors[index+1] = []                    #worst case = (every vertex has own c
olor)

        for vertex in self.Graph:
            order.append((len(self.Graph[vertex]), vertex))     #make list with (degree,
vertex)

        order.sort()        #sort in ascending order
        order.reverse()     #reverse order

        for index in range (len(order)):
            sort.append(order[index][1])        #remove degrees from list

        colors[1] = [sort[0]]             #color first vertex 1
        x = sort[0]
        sort.remove(x)                   #remove vertex


        while len(sort) > 0:              #while not all vertices are labeled
            j=1                            #set color = 1
            found = False
            w = sort[0]                        #set w = vertex with max degree
            while found == False:
```

```
                for label in colors:
                    if found == True:           #if a vertex can be labeled stop the for
loop
                        break
                    spot = 0        #keep track of # of neighbors checked
                    for neighbor in self.Graph[w]:
                        spot = spot+1  #checked one more neighbor
                        if neighbor in colors[label]:   #a neighbor vertex is already lab
eled with a num = label
                            j = 0
                            j = label+1                         #set j = one more than label
of neighboring vertex

                        elif spot == len(self.Graph[w]):        #all neighbors have been
checked
                            found = True
                            colors[j].append(w)     #label w with color j
                            x = sort[0]
                            sort.remove(x)      #remove from U




        ret = list()
        for i in colors:
            if (len(colors[i]) != 0):       #don't print out empy lists
                ret.append(colors[i])

        return ret



    def SeqC(self):
        colors = {}                                 #dictionary to hold colors of vertices
        for index in range (len(self.Graph)):       #creates dictionary of colors for wor
st case scenario
            colors[index+1] = []                        #worst case = every vertex gets i
ts own color


        vertices = list()
        for vert in self.Graph:             #list of vertices in self.Graph
            vertices.append(vert)

        colors[1] = [vertices[0]]       #label first vertex 1

        labeled = list()                    #list of labeled vertices
        labeled.append(vertices[0])


        for vertex in self.Graph:           #iterate though all vertices in graph
            if vertex not in labeled:
                neighbors = self.Graph[vertex]      #list of adjacent vertices
                a = neighbors
                b = labeled                         #list of labeled vertices

                important = list(set(a) & set(b))   #list of labeled adjacent vertices

                if(len(important) == 0):        #no labeled adjacent vertices
                    colors[1].append(vertex)    #label vertex with 1

                else:
                    found = False
                    for label in colors:
                        if found == True:           #if a vertex can be labeled stop the
for loop
                            break
                        spot = 0        #keep track of # of neighbors checked
                        for neighbor in important:
                            spot = spot+1          #checked one more neighbor
```

```
                            if neighbor in colors[label]:   #a neighbor vertex is already
 labeled with a num = label
                                j = 0
                                j = label+1                     #set j = one more than la
bel of neighboring vertex

                            elif spot == len(important):        #all neighbors have been
checked
                                found = True
                                colors[j].append(vertex)     #label w with color j




        ret = list()
        for i in colors:
            if (len(colors[i]) != 0):       #don't print out empy lists
                ret.append(colors[i])

        return ret


    def isTree(self):       #modified DFS to check for cycles since we assume G is connec
ted
        keys = self.vertices()
        v = keys[0]             #arbitrarily pick a vertex for v
        #step 1
        graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
        S = [v]              #list
        T = []               #empty list of edges
        history = [v]        #history of vertices that are used as bstar
        bstar = v
        pstar = v
        label = []           #list to hold labels
        label.append(v)     #label v as 0
        U1 = graph.keys()   #keys of dict
        U = []              #list of unlabeled vertices
        for item in U1:
            U.append(item)
        U.remove(v)         #remove v since it is labeled
        U.sort()
        neighbors = graph[v]        #list of neighbors of v
        neighbors.sort()
        intersection = list(set(U) & set(neighbors))        #U intersect neighbors
        intersection.sort()
        #step 2
        while (True):
            while (intersection):
                w = intersection[0]
                label.append(w)          #label neighbor of bstar
                T.append((bstar,w))      #add edge to T
                U.remove(w)              #remove labeled vertex from unlabeled list
                pstar = bstar               #help for backtracking
                bstar = w
                history.append(bstar)       #update history list
                neighbors = graph[bstar]    #get niehgbors
                neighbors.sort()
                intersection = list(set(U) & set(neighbors)) #get intersect of U and neig
hbors
                intersection.sort()

        #MAIN EDIT
                #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                cycle = list(set(history) & set(neighbors)) #take the intersection of the
 bstar's neighbors and history(visited vertices)
                if (len(cycle) >= 2):                    # IF 2 OR MORE NEIGHBORS OF CURR
ENT VERTEX HAVE BEEN VISITED
                    return (False)              #A CYCLE HAS BEEN FOUND, RETURN FALSE
                #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
            neighbors = graph[bstar]
            neighbors.sort()
            intersection = list(set(U) & set(neighbors))     #used to check for halting c
ondition
            intersection.sort()


            if (len(intersection) == 0):    #if U intersect neighbors = []
                num = history.index(bstar)
                bstar = history[num-1]      #bstar = pstar

            if (bstar == v and intersection == [] and len(U) == 0): #halting condition
                return True        #IF DFS COMPLETES NORMALLY, RETURN TRUE

            if (intersection != []):
                history.append(bstar)       #update history list





    def Center(self):
        if (self.isTree()):     #isTree returned True
            graph = copy.deepcopy(self.Graph)   #make a copy to delete vertices without c
hanging real graph
            leaves = []             #make a list of leaves found
            x = True
            while (x == True):
                for vertex in graph:       #iterate through vertices in graph
                    if (len(graph[vertex]) == 1):       #leaf found (has degree = 1)
                        leaves.append(vertex)           #get a list of leaves to delete


                for i in range (len(leaves)):   #iterate through leaves
                    del graph[leaves[i]]               #delete leaf from dict
                    for item in graph:
                        if leaves[i] in graph[item]:       #delete leaf from other keys’
values
                                graph[item].remove(leaves[i])


                leaves = []             #reset list for next iteration

                if (len(graph) <= 2):  #check if center/ centers have been found
                    for item in graph:
                        leaves.append(item) #reuse leaves list to return center/centers
                    return(leaves)

        else:
            return(’Graph is not a Tree’)           #isTree returned False



def main():
    #call functions from here
    G = Graph()


main()
```