```
#Submission for Technical Interview Questions
#5/12/13


# Question 1 ---------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#                    for loop.  However, since it is a "set" and not a string this
#                    function will run in O(n) time.
#
#Space Efficiency:  The will use up only the space needed for s and t.  O(n).
#
# Code Design:  Code was designed to run efficiently and is easy to understand.


def question1(s,t):
    #These must be converted to strings because the input could be a number.
    s = str(s)
    t = str(t)

    if t == "":
        return True

    sset = set(s)
    tset = set(t)

    if sset and tset is not None:
        for c in tset:
            if c not in sset:
                return False
            return True
    return False

#Testing the function
question1("udacity","udazx")
question1("udacity","ad")
question1(None,"ad")
question1("udacity",None)
question1("udacity","")
question1("udacity",5)
question1("uda55city",55)
question1("hello","hallo")


# Question 2 ---------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#               two for loops.  The first loop will not add a significant amount
#               of time to the function because it will only loop a few times.
#               This function will run in O(n^2) time.
#
#Space Efficiency:  The will use up only the space needed for a and the matrix created
#                   with the combinations function. O(2n)
#
# Code Design:  Using Dynamic programming to find the solution.

def question2(a):
    #Modified from:
    http://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/

    if a == "":
        return "(Empty String)"
    elif a is None:
        return None
```

```
    a = str(a)
    n = len(a)
    W = [[0 for x in range(n)] for x in range(n)]

    for i in range(n):
        W[i][i] = a[i]

    for cl in range(2, n+1):
        for i in range(n-cl+1):
            j = i+cl-1
            if a[i] == a[j] and cl == 2:
                W[i][j] = a[i] + a[j]
            elif a[i] == a[j]:
                W[i][j] = a[i] + W[i+1][j-1] + a[j]
            else:
                if len(W[i][j-1]) > len(W[i+1][j]):
                    W[i][j] = W[i][j-1]
                else:
                    W[i][j] = W[i+1][j]

    return W[0][n-1]



#Testing the function
question2("character")
question2("a")
question2("abc")
question2(None)
question2(5)
question2("")

# Question 3 ---------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#                 two for loops.  This function will run in O(n*m) time.  Where n
#                 is the number of nodes and m is the number of connections.
#
#Space Efficiency:  The will use up only the space needed for G.  O(1)
#
# Code Design:  The networkx libary functions were used to make the code easier
#                 to read and more effient.  They are towards the bottom of the page.
#                 Load Functions and classes below before running question3().
#

def question3(G):

    Gnx = Graph()

    for key , value in G.iteritems():
        Gnx.add_node(key)

        for val in value:
            Gnx.add_edge(key,val[0], key = val[1])

    mst = minimum_spanning_tree(Gnx)
    return mst


#Testing the function
val = question3({'A':[('B',2)],'B':[('A',2),('C',5)],'C':[('B',5)]})
print(sorted(val.edges(data=True)))

val = question3({'A':[('B',3)],'B':[('A',3),('C',10)],'C':[('B',10)]})
```

```python
print(sorted(val.edges(data=True)))

val = question3({'A':[('B',2)],'B':[('A',2)]})
print(sorted(val.edges(data=True)))


# Question 4 ---------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#                5 for loops.  This function will run in O(n*m) time.  Where n
#                is the number of nodes and m is the number of connections.  Each
#                will have to run twice.  Once to generate the tree and the second
#                time to fill in the information about where in the tree that node
#                is.
#
#Space Efficiency:  The will use up only the space needed for T,r, n1, and n2.
#                   G, n, and m will be assigned during the call.  This will
#                   result in O(2+n).  The n is for G and the 2 is for n and m.
#
# Code Design:   The networkx libary functions were used to make the code easier
#                to read and more effient.  They are towards the bottom of the page.
#                Load Functions and classes below before running question4().


def question4(T, r, n1, n2):

    G = Graph()
    G.add_node(r)

    n = len(T)
    m = len(T[0])

    for i in range(0,n):
        for j in range(0,m):
            if T[i][j] == 1:
                G.add_edge(i,j)

    for a in G.nodes():
        G.node[a]['Level']= None
    G.node[r]['Level']= 0

    m = G.number_of_nodes()
    for L in range (0,m):
        for n in G.nodes():
            if G.node[n]['Level'] == L:
                L1 = G[n]
                for a in L1.iteritems():
                    if G.node[a[0]]['Level'] == None:
                        G.node[a[0]]['Level'] = L+1

    path = shortest_path(G,n1,n2)
    path.pop(0)
    path.pop(len(path)-1)

    LCALevel = G.node[n1]['Level']
    for n in path:
        val = G.node[n]['Level']
        if val < LCALevel:
            LCALevel = G.node[n]['Level']
            LCA = n

    return LCA

#Testing the function
val = question4([[0,1,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[1,0,0,0,1],[0,0,0,0,0]],3,1,4)
```

```
#Correct Ans: 3
print val

val = question4([[0,0,0,0,0],[0,0,0,0,0],[0,1,0,1,1],[0,0,0,0,0],[1,0,0,0,0]],4,3,1)
#Correct Ans: 2
print val

val = question4([[0,0,0,0,1],[0,0,1,0,0],[0,1,0,1,1],[0,0,1,0,0],[1,0,1,0,0]],4,3,1)
#Correct Ans: 2
print val


# Question 5 ----------------------------------------------------

# Time Efficiency:   The main part of this code that might take a while to run is the
#                 while loop and the for loop.  This function will run in O(n) time.
#
#Space Efficiency:  The will use up only the space needed for ll and m.  O(1)
#
# Code Design:  The linked list is cycled through to find the end.  It is then
#                 cycled through to the m position away from the end.

def question5(ll, m):

    next_item = ll.get_next()
    count = 1

    while next_item.get_next() is not None:
        next_item = next_item.get_next()
        count += 1

    selected = ll
    for i in range(0, count-m):
        selected = selected.get_next()

    if m < 0:
        raise ValueError("Verify Inputs")
        #return "Verify Inputs"
    m = count-m
    if m < 0:
        raise ValueError("Verify Inputs")
        #return "Verify Inputs"

    val = selected.get_data()

    return val

#Source for Node and LinkedList Classes:
https://www.codefellows.org/blog/implementing-a-singly-linked-list-in-python
class Node(object):

    def __init__(self, data=None, next_node=None):
        self.data = data
        self.next_node = next_node

    def get_data(self):
        return self.data

    def get_next(self):
        return self.next_node

    def set_next(self, new_next):
        self.next_node = new_next

class LinkedList(object):
```

```
    def __init__(self, head=None):
        self.head = head

    def insert(self, data):
        new_node = Node(data)
        new_node.set_next(self.head)
        self.head = new_node


#Create a Linked List to input into question 5
LL = LinkedList()
LL.insert("E")
LL.insert("D")
LL.insert("C")
LL.insert("B")
LL.insert("A")

#Testing the function
question5(LL.head, 0)
question5(LL.head, 1)
question5(LL.head, 2)
question5(LL.head, 3)
question5(LL.head, 4)
question5(LL.head, 5)
question5(LL.head, -1)
















#Source for everything below: https://github.com/networkx/networkx


"""
Algorithms for calculating min/max spanning trees/forests.
"""
#    Copyright (C) 2015 NetworkX Developers
#    Aric Hagberg <hagberg@lanl.gov>
#    Dan Schult <dschult@colgate.edu>
#    Pieter Swart <swart@lanl.gov>
#    Loïc Séguin-C. <loicseguin@gmail.com>
#    All rights reserved.
#    BSD license.

__all__ = [
    'minimum_spanning_edges', 'maximum_spanning_edges',
    'minimum_spanning_tree', 'maximum_spanning_tree',
]

from heapq import heappop, heappush
from itertools import count

#from networkx.utils import UnionFind, not_implemented_for



def boruvka_mst_edges(G, minimum=True, weight='weight', keys=False, data=True):
```

```
    opt = min if minimum else max

    forest = UnionFind(G)

    def best_edge(component):

        boundary = list(nx.edge_boundary(G, component, data=True))
        if not boundary:
            return None
        return opt(boundary, key=lambda e: e[-1][weight])


    best_edges = (best_edge(component) for component in forest.to_sets())
    best_edges = [edge for edge in best_edges if edge is not None]

    while best_edges:

        best_edges = (best_edge(component) for component in forest.to_sets())
        best_edges = [edge for edge in best_edges if edge is not None]

        for u, v, d in best_edges:
            if forest[u] != forest[v]:
                if data:
                    yield u, v, d
                else:
                    yield u, v
                forest.union(u, v)


def kruskal_mst_edges(G, minimum, weight='weight', keys=True, data=True):
    subtrees = UnionFind()
    if G.is_multigraph():
        edges = G.edges(keys=True, data=True)
    else:
        edges = G.edges(data=True)
    getweight = lambda t: t[-1].get(weight, 1)
    edges = sorted(edges, key=getweight, reverse=not minimum)
    is_multigraph = G.is_multigraph()
    # Multigraphs need to handle edge keys in addition to edge data.
    if is_multigraph:
        for u, v, k, d in edges:
            if subtrees[u] != subtrees[v]:
                if keys:
                    if data:
                        yield (u, v, k, d)
                    else:
                        yield (u, v, k)
                else:
                    if data:
                        yield (u, v, d)
                    else:
                        yield (u, v)
                subtrees.union(u, v)
    else:
        for u, v, d in edges:
            if subtrees[u] != subtrees[v]:
                if data:
                    yield (u, v, d)
                else:
                    yield (u, v)
                subtrees.union(u, v)


def prim_mst_edges(G, minimum, weight='weight', keys=True, data=True):
```

```
    is_multigraph = G.is_multigraph()
    push = heappush
    pop = heappop

    nodes = list(G)
    c = count()

    sign = 1
    if not minimum:
        sign = -1

    while nodes:
        u = nodes.pop(0)
        frontier = []
        visited = [u]
        if is_multigraph:
            for u, v, k, d in G.edges(u, keys=True, data=True):
                push(frontier, (d.get(weight, 1) * sign, next(c), u, v, k))
        else:
            for u, v, d in G.edges(u, data=True):
                push(frontier, (d.get(weight, 1) * sign, next(c), u, v))
        while frontier:
            if is_multigraph:
                W, _, u, v, k = pop(frontier)
            else:
                W, _, u, v = pop(frontier)
            if v in visited:
                continue
            visited.append(v)
            nodes.remove(v)
            if is_multigraph:
                for _, w, k2, d2 in G.edges(v, keys=True, data=True):
                    if w in visited:
                        continue
                    new_weight = d2.get(weight, 1) * sign
                    push(frontier, (new_weight, next(c), v, w, k2))
            else:
                for _, w, d2 in G.edges(v, data=True):
                    if w in visited:
                        continue
                    new_weight = d2.get(weight, 1) * sign
                    push(frontier, (new_weight, next(c), v, w))
            # Multigraphs need to handle edge keys in addition to edge data.
            if is_multigraph and keys:
                if data:
                    yield u, v, k, G[u][v]
                else:
                    yield u, v, k
            else:
                if data:
                    yield u, v, G[u][v]
                else:
                    yield u, v

ALGORITHMS = {
    'boruvka': boruvka_mst_edges,
    u'borůvka': boruvka_mst_edges,
    'kruskal': kruskal_mst_edges,
    'prim': prim_mst_edges
}



def _spanning_edges(G, minimum, algorithm='kruskal', weight='weight',
                    keys=True, data=True):
```

```
    try:
        algo = ALGORITHMS[algorithm]
    except KeyError:
        msg = '{} is not a valid choice for an algorithm.'.format(algorithm)
        raise ValueError(msg)

    return algo(G, minimum=minimum, weight=weight, keys=keys, data=data)


def minimum_spanning_edges(G, algorithm='kruskal', weight='weight', keys=True,
                           data=True):

    return _spanning_edges(G, minimum=True, algorithm=algorithm,
                           weight=weight, keys=keys, data=data)


def maximum_spanning_edges(G, algorithm='kruskal', weight='weight', data=True):

    return _spanning_edges(G, minimum=False, algorithm=algorithm,
                           weight=weight, data=data)



def _optimum_spanning_tree(G, algorithm, minimum, weight='weight'):
    # When creating the spanning tree, we can ignore the key used to
    # identify multigraph edges, since a tree is guaranteed to have no
    # multiedges. This is why we use `keys=False`.
    edges = _spanning_edges(G, minimum, algorithm=algorithm, weight=weight,
                            keys=False, data=True)
    T = Graph(edges)

    # Add isolated nodes
    if len(T) != len(G):
        T.add_nodes_from(nx.isolates(G))

    # Add node and graph attributes as shallow copy
    for n in T:
        T.node[n] = G.node[n].copy()
    T.graph = G.graph.copy()

    return T


def minimum_spanning_tree(G, weight='weight', algorithm='kruskal'):

    return _optimum_spanning_tree(G, algorithm=algorithm, minimum=True,
                                  weight=weight)


def maximum_spanning_tree(G, weight='weight', algorithm='kruskal'):

    return _optimum_spanning_tree(G, algorithm=algorithm, minimum=False,
                                  weight=weight)


from itertools import groupby

class UnionFind:

    def __init__(self, elements=None):

        if elements is None:
            elements = ()
        self.parents = {}
        self.weights = {}
```

```
        for x in elements:
            self.weights[x] = 1
            self.parents[x] = x

    def __getitem__(self, object):

        # check for previously unknown object
        if object not in self.parents:
            self.parents[object] = object
            self.weights[object] = 1
            return object

        # find path of objects leading to the root
        path = [object]
        root = self.parents[object]
        while root != path[-1]:
            path.append(root)
            root = self.parents[root]

        # compress the path and return
        for ancestor in path:
            self.parents[ancestor] = root
        return root

    def __iter__(self):

        return iter(self.parents)

    def to_sets(self):

        for block in groups(self.parents).values():
            yield block

    def union(self, *objects):
        """Find the sets containing the objects and merge them all."""
        roots = [self[x] for x in objects]
        # Find the heaviest root according to its weight.
        heaviest = max(roots, key=lambda r: self.weights[r])
        for r in roots:
            if r != heaviest:
                self.weights[heaviest] += self.weights[r]
                self.parents[r] = heaviest


#    Copyright (C) 2004-2016 by
#    Aric Hagberg <hagberg@lanl.gov>
#    Dan Schult <dschult@colgate.edu>
#    Pieter Swart <swart@lanl.gov>
#    All rights reserved.
#    BSD license.
from __future__ import division
from copy import deepcopy
from networkx.exception import NetworkXError

__author__ = """\n""".join(['Aric Hagberg (hagberg@lanl.gov)',
                            'Pieter Swart (swart@lanl.gov)',
                            'Dan Schult(dschult@colgate.edu)'])


class Graph(object):

    node_dict_factory = dict
    adjlist_dict_factory = dict
    edge_attr_dict_factory = dict
```

```
    def __init__(self, data=None, **attr):

        self.node_dict_factory = ndf = self.node_dict_factory
        self.adjlist_dict_factory = self.adjlist_dict_factory
        self.edge_attr_dict_factory = self.edge_attr_dict_factory

        self.graph = {}   # dictionary for graph attributes
        self.node = ndf()  # empty node attribute dict
        self.adj = ndf()  # empty adjacency dict
        # attempt to load graph with data
        if data is not None:
            to_networkx_graph(data, create_using=self)
        # load graph attributes (must be after convert)
        self.graph.update(attr)
        self.edge = self.adj

    @property
    def name(self):
        return self.graph.get('name', '')

    @name.setter
    def name(self, s):
        self.graph['name'] = s

    def __str__(self):

        return self.name

    def __iter__(self):

        return iter(self.node)

    def __contains__(self, n):

        try:
            return n in self.node
        except TypeError:
            return False

    def __len__(self):

        return len(self.node)

    def __getitem__(self, n):

        return self.adj[n]

    def add_node(self, n, **attr):

        if n not in self.node:
            self.adj[n] = self.adjlist_dict_factory()
            self.node[n] = attr
        else:  # update attr even if node already exists
            self.node[n].update(attr)

    def add_nodes_from(self, nodes, **attr):

        for n in nodes:

            try:
                if n not in self.node:
                    self.adj[n] = self.adjlist_dict_factory()
                    self.node[n] = attr.copy()
                else:
                    self.node[n].update(attr)
```

```
        except TypeError:
            nn, ndict = n
            if nn not in self.node:
                self.adj[nn] = self.adjlist_dict_factory()
                newdict = attr.copy()
                newdict.update(ndict)
                self.node[nn] = newdict
            else:
                olddict = self.node[nn]
                olddict.update(attr)
                olddict.update(ndict)

def remove_node(self, n):

    adj = self.adj
    try:
        nbrs = list(adj[n].keys())  # keys handles self-loops (allow mutation later)
        del self.node[n]
    except KeyError:  # NetworkXError if n not in self
        raise NetworkXError("The node %s is not in the graph." % (n,))
    for u in nbrs:
        del adj[u][n]    # remove all edges n-u in graph
    del adj[n]           # now remove node

def remove_nodes_from(self, nodes):

    adj = self.adj
    for n in nodes:
        try:
            del self.node[n]
            for u in list(adj[n].keys()):   # keys() handles self-loops
                del adj[u][n]  # (allows mutation of dict in loop)
            del adj[n]
        except KeyError:
            pass

def nodes(self, data=False, default=None):

    if data is True:
        for n, ddict in self.node.items():
            yield (n, ddict)
    elif data is not False:
        for n, ddict in self.node.items():
            d = ddict[data] if data in ddict else default
            yield (n, d)
    else:
        for n in self.node:
            yield n

def number_of_nodes(self):

    return len(self.node)

def order(self):

    return len(self.node)

def has_node(self, n):

    try:
        return n in self.node
    except TypeError:
        return False

def add_edge(self, u, v, **attr):
```

```
        # add nodes
        if u not in self.node:
            self.adj[u] = self.adjlist_dict_factory()
            self.node[u] = {}
        if v not in self.node:
            self.adj[v] = self.adjlist_dict_factory()
            self.node[v] = {}
        # add the edge
        datadict = self.adj[u].get(v, self.edge_attr_dict_factory())
        datadict.update(attr)
        self.adj[u][v] = datadict
        self.adj[v][u] = datadict

    def add_edges_from(self, ebunch, **attr):

        # process ebunch
        for e in ebunch:
            ne = len(e)
            if ne == 3:
                u, v, dd = e
            elif ne == 2:
                u, v = e
                dd = {}  # doesnt need edge_attr_dict_factory
            else:
                raise NetworkXError(
                    "Edge tuple %s must be a 2-tuple or 3-tuple." % (e,))
            if u not in self.node:
                self.adj[u] = self.adjlist_dict_factory()
                self.node[u] = {}
            if v not in self.node:
                self.adj[v] = self.adjlist_dict_factory()
                self.node[v] = {}
            datadict = self.adj[u].get(v, self.edge_attr_dict_factory())
            datadict.update(attr)
            datadict.update(dd)
            self.adj[u][v] = datadict
            self.adj[v][u] = datadict

    def add_weighted_edges_from(self, ebunch, weight='weight', **attr):

        self.add_edges_from(((u, v, {weight: d}) for u, v, d in ebunch),
                            **attr)

    def remove_edge(self, u, v):

        try:
            del self.adj[u][v]
            if u != v:  # self-loop needs only one entry removed
                del self.adj[v][u]
        except KeyError:
            raise NetworkXError("The edge %s-%s is not in the graph" % (u, v))

    def remove_edges_from(self, ebunch):

        adj = self.adj
        for e in ebunch:
            u, v = e[:2]  # ignore edge data if present
            if u in adj and v in adj[u]:
                del adj[u][v]
                if u != v:  # self loop needs only one entry removed
                    del adj[v][u]

    def has_edge(self, u, v):
```

```
        try:
            return v in self.adj[u]
        except KeyError:
            return False


    def neighbors(self, n):

        try:
            return iter(self.adj[n])
        except KeyError:
            raise NetworkXError("The node %s is not in the graph." % (n,))


    def edges(self, nbunch=None, data=False, default=None):

        seen = {}     # helper dict to keep track of multiply stored edges
        if nbunch is None:
            nodes_nbrs = self.adj.items()
        else:
            nodes_nbrs = ((n, self.adj[n]) for n in self.nbunch_iter(nbunch))
        if data is True:
            for n, nbrs in nodes_nbrs:
                for nbr, ddict in nbrs.items():
                    if nbr not in seen:
                        yield (n, nbr, ddict)
                seen[n] = 1
        elif data is not False:
            for n, nbrs in nodes_nbrs:
                for nbr, ddict in nbrs.items():
                    if nbr not in seen:
                        d = ddict[data] if data in ddict else default
                        yield (n, nbr, d)
                seen[n] = 1
        else:  # data is False
            for n, nbrs in nodes_nbrs:
                for nbr in nbrs:
                    if nbr not in seen:
                        yield (n, nbr)
                seen[n] = 1
        del seen

    def get_edge_data(self, u, v, default=None):

        try:
            return self.adj[u][v]
        except KeyError:
            return default

    def adjacency(self):

        return iter(self.adj.items())

    def degree(self, nbunch=None, weight=None):

        if nbunch in self:
            nbrs = self.adj[nbunch]
            if weight is None:
                return len(nbrs) + (1 if nbunch in nbrs else 0) # handle self-loops
            return sum(dd.get(weight, 1) for nbr,dd in nbrs.items()) +\
                    (nbrs[nbunch].get(weight, 1) if nbunch in nbrs else 0)

        if nbunch is None:
            nodes_nbrs = self.adj.items()
        else:
```

```
            nodes_nbrs = ((n, self.adj[n]) for n in self.nbunch_iter(nbunch))
        if weight is None:
            def d_iter():
                for n, nbrs in nodes_nbrs:
                    yield (n, len(nbrs) + (1 if n in nbrs else 0))  # return tuple (n,degree)
        else:
            def d_iter():
                for n, nbrs in nodes_nbrs:
                    yield (n, sum((nbrs[nbr].get(weight, 1) for nbr in nbrs)) +
                          (nbrs[n].get(weight, 1) if n in nbrs else 0))
        return d_iter()

    def clear(self):

        self.name = ''
        self.adj.clear()
        self.node.clear()
        self.graph.clear()

    def copy(self, with_data=True):

        if with_data:
            return deepcopy(self)
        return self.subgraph(self)

    def is_multigraph(self):
        """Return True if graph is a multigraph, False otherwise."""
        return False

    def is_directed(self):
        """Return True if graph is directed, False otherwise."""
        return False

    def to_directed(self):

        from networkx import DiGraph
        G = DiGraph()
        G.name = self.name
        G.add_nodes_from(self)
        G.add_edges_from(((u, v, deepcopy(data))
            for u, nbrs in self.adjacency()
            for v, data in nbrs.items()))
        G.graph = deepcopy(self.graph)
        G.node = deepcopy(self.node)
        return G

    def to_undirected(self):

        return deepcopy(self)

    def subgraph(self, nbunch):

        bunch = self.nbunch_iter(nbunch)
        # create new graph and copy subgraph into it
        H = self.__class__()
        # copy node and attribute dictionaries
        for n in bunch:
            H.node[n] = self.node[n]
        # namespace shortcuts for speed
        H_adj = H.adj
        self_adj = self.adj
        # add nodes and edges (undirected method)
        # Note that changing this may affect the deep-ness of self.copy()
        for n in H.node:
            Hnbrs = H.adjlist_dict_factory()
```

```
            H_adj[n] = Hnbrs
            for nbr, d in self_adj[n].items():
                if nbr in H_adj:
                    # add both representations of edge: n-nbr and nbr-n
                    Hnbrs[nbr] = d
                    H_adj[nbr][n] = d
        H.graph = self.graph
        return H

    def edge_subgraph(self, edges):

        H = self.__class__()
        adj = self.adj
        # Filter out edges that don't correspond to nodes in the graph.
        edges = ((u, v) for u, v in edges if u in adj and v in adj[u])
        for u, v in edges:
            # Copy the node attributes if they haven't been copied
            # already.
            if u not in H.node:
                H.node[u] = self.node[u]
            if v not in H.node:
                H.node[v] = self.node[v]
            # Create an entry in the adjacency dictionary for the
            # nodes u and v if they don't exist yet.
            if u not in H.adj:
                H.adj[u] = H.adjlist_dict_factory()
            if v not in H.adj:
                H.adj[v] = H.adjlist_dict_factory()
            # Copy the edge attributes.
            H.edge[u][v] = self.edge[u][v]
            H.edge[v][u] = self.edge[v][u]
        H.graph = self.graph
        return H

    def nodes_with_selfloops(self):

        return (n for n, nbrs in self.adj.items() if n in nbrs)

    def selfloop_edges(self, data=False, default=None):

        if data is True:
            return ((n, n, nbrs[n])
                    for n, nbrs in self.adj.items() if n in nbrs)
        elif data is not False:
            return ((n, n, nbrs[n].get(data, default))
                    for n, nbrs in self.adj.items() if n in nbrs)
        else:
            return ((n, n)
                    for n, nbrs in self.adj.items() if n in nbrs)

    def number_of_selfloops(self):

        return sum(1 for _ in self.selfloop_edges())

    def size(self, weight=None):

        s = sum(d for v, d in self.degree(weight=weight))

        return s // 2 if weight is None else s / 2

    def number_of_edges(self, u=None, v=None):

        if u is None: return int(self.size())
        if v in self.adj[u]:
            return 1
```

```
        else:
            return 0

    def nbunch_iter(self, nbunch=None):

        if nbunch is None:   # include all nodes via iterator
            bunch = iter(self.adj)
        elif nbunch in self:  # if nbunch is a single node
            bunch = iter([nbunch])
        else:                # if nbunch is a sequence of nodes
            def bunch_iter(nlist, adj):
                try:
                    for n in nlist:
                        if n in adj:
                            yield n
                except TypeError as e:
                    message = e.args[0]
                    # capture error for non-sequence/iterator nbunch.
                    if 'iter' in message:
                        raise NetworkXError(
                            "nbunch is not a node or a sequence of nodes.")
                    # capture error for unhashable node.
                    elif 'hashable' in message:
                        raise NetworkXError(
                            "Node {} in the sequence nbunch is not a valid node.".format(n))
                    else:
                        raise
            bunch = bunch_iter(nbunch, self.adj)
        return bunch


#    Copyright (C) 2006-2013 by
#    Aric Hagberg <hagberg@lanl.gov>
#    Dan Schult <dschult@colgate.edu>
#    Pieter Swart <swart@lanl.gov>
#    All rights reserved.
#    BSD license.
import warnings

__author__ = """\n""".join(['Aric Hagberg <aric.hagberg@gmail.com>',
                           'Pieter Swart (swart@lanl.gov)',
                           'Dan Schult(dschult@colgate.edu)'])
__all__ = ['to_networkx_graph',
           'from_dict_of_dicts', 'to_dict_of_dicts',
           'from_dict_of_lists', 'to_dict_of_lists',
           'from_edgelist', 'to_edgelist']

def _prep_create_using(create_using):

    if create_using is None:
        return Graph()
    try:
        create_using.clear()
    except:
        raise TypeError("Input graph is not a networkx graph type")
    return create_using

def to_networkx_graph(data,create_using=None,multigraph_input=False):

    # NX graph
    if hasattr(data,"adj"):
        try:
            result= from_dict_of_dicts(data.adj,\
                    create_using=create_using,\
                    multigraph_input=data.is_multigraph())
```

```
            if hasattr(data,'graph'): # data.graph should be dict-like
                result.graph.update(data.graph)
            if hasattr(data,'node'): # data.node should be dict-like
                result.node.update( (n,dd.copy()) for n,dd in data.node.items() )
            return result
        except:
            raise NetworkXError("Input is not a correct NetworkX graph.")

    # pygraphviz  agraph
    if hasattr(data,"is_strict"):
        try:
            return nx_agraph.from_agraph(data,create_using=create_using)
        except:
            raise NetworkXError("Input is not a correct pygraphviz graph.")

    # dict of dicts/lists
    if isinstance(data,dict):
        try:
            return from_dict_of_dicts(data,create_using=create_using,\
                    multigraph_input=multigraph_input)
        except:
            try:
                return from_dict_of_lists(data,create_using=create_using)
            except:
                raise TypeError("Input is not known type.")

    # list or generator of edges
    if (isinstance(data,list)
        or isinstance(data,tuple)
        or hasattr(data,'next')
        or hasattr(data, '__next__')):
        try:
            return from_edgelist(data,create_using=create_using)
        except:
            raise NetworkXError("Input is not a valid edge list")

    # Pandas DataFrame
    try:
        import pandas as pd
        if isinstance(data, pd.DataFrame):
            try:
                return from_pandas_dataframe(data, create_using=create_using)
            except:
                msg = "Input is not a correct Pandas DataFrame."
                raise NetworkXError(msg)
    except ImportError:
        msg = 'pandas not found, skipping conversion test.'
        warnings.warn(msg, ImportWarning)

    # numpy matrix or ndarray
    try:
        import numpy
        if isinstance(data,numpy.matrix) or \
               isinstance(data,numpy.ndarray):
            try:
                return from_numpy_matrix(data,create_using=create_using)
            except:
                raise NetworkXError(\
                  "Input is not a correct numpy matrix or array.")
    except ImportError:
        warnings.warn('numpy not found, skipping conversion test.',
                      ImportWarning)

    # scipy sparse matrix - any format
    try:
```

```
        import scipy
        if hasattr(data,"format"):
            try:
                return from_scipy_sparse_matrix(data,create_using=create_using)
            except:
                raise NetworkXError(\
                      "Input is not a correct scipy sparse matrix type.")
    except ImportError:
        warnings.warn('scipy not found, skipping conversion test.',
                      ImportWarning)


    raise NetworkXError(\
          "Input is not a known data type for conversion.")

    return


def convert_to_undirected(G):
    """Return a new undirected representation of the graph G."""
    return G.to_undirected()


def convert_to_directed(G):
    """Return a new directed representation of the graph G."""
    return G.to_directed()


def to_dict_of_lists(G,nodelist=None):

    if nodelist is None:
        nodelist=G

    d = {}
    for n in nodelist:
        d[n]=[nbr for nbr in G.neighbors(n) if nbr in nodelist]
    return d

def from_dict_of_lists(d,create_using=None):

    G=_prep_create_using(create_using)
    G.add_nodes_from(d)
    if G.is_multigraph() and not G.is_directed():
        # a dict_of_lists can't show multiedges.  BUT for undirected graphs,
        # each edge shows up twice in the dict_of_lists.
        # So we need to treat this case separately.
        seen={}
        for node,nbrlist in d.items():
            for nbr in nbrlist:
                if nbr not in seen:
                    G.add_edge(node,nbr)
            seen[node]=1  # don't allow reverse edge to show up
    else:
        G.add_edges_from( ((node,nbr) for node,nbrlist in d.items()
                              for nbr in nbrlist) )
    return G


def to_dict_of_dicts(G,nodelist=None,edge_data=None):

    dod={}
    if nodelist is None:
        if edge_data is None:
            for u,nbrdict in G.adjacency():
                dod[u]=nbrdict.copy()
```

```
        else: # edge_data is not None
            for u,nbrdict in G.adjacency():
                dod[u]=dod.fromkeys(nbrdict, edge_data)
    else: # nodelist is not None
        if edge_data is None:
            for u in nodelist:
                dod[u]={}
                for v,data in ((v,data) for v,data in G[u].items() if v in nodelist):
                    dod[u][v]=data
        else: # nodelist and edge_data are not None
            for u in nodelist:
                dod[u]={}
                for v in ( v for v in G[u] if v in nodelist):
                    dod[u][v]=edge_data
    return dod

def from_dict_of_dicts(d,create_using=None,multigraph_input=False):

    G=_prep_create_using(create_using)
    G.add_nodes_from(d)
    # is dict a MultiGraph or MultiDiGraph?
    if multigraph_input:
        # make a copy of the list of edge data (but not the edge data)
        if G.is_directed():
            if G.is_multigraph():
                G.add_edges_from( (u,v,key,data)
                                  for u,nbrs in d.items()
                                  for v,datadict in nbrs.items()
                                  for key,data in datadict.items()
                                )
            else:
                G.add_edges_from( (u,v,data)
                                  for u,nbrs in d.items()
                                  for v,datadict in nbrs.items()
                                  for key,data in datadict.items()
                                )
        else: # Undirected
            if G.is_multigraph():
                seen=set()   # don't add both directions of undirected graph
                for u,nbrs in d.items():
                    for v,datadict in nbrs.items():
                        if (u,v) not in seen:
                            G.add_edges_from( (u,v,key,data)
                                              for key,data in datadict.items()
                                            )
                            seen.add((v,u))
            else:
                seen=set()   # don't add both directions of undirected graph
                for u,nbrs in d.items():
                    for v,datadict in nbrs.items():
                        if (u,v) not in seen:
                            G.add_edges_from( (u,v,data)
                                        for key,data in datadict.items() )
                            seen.add((v,u))

    else: # not a multigraph to multigraph transfer
        if G.is_multigraph() and not G.is_directed():
            # d can have both representations u-v, v-u in dict.  Only add one.
            # We don't need this check for digraphs since we add both directions,
            # or for Graph() since it is done implicitly (parallel edges not allowed)
            seen=set()
            for u,nbrs in d.items():
                for v,data in nbrs.items():
                    if (u,v) not in seen:
                        G.add_edge(u,v,key=0)
```

```
                        G[u][v][0].update(data)
                    seen.add((v,u))
        else:
            G.add_edges_from( ( (u,v,data)
                                for u,nbrs in d.items()
                                for v,data in nbrs.items()) )
    return G

def to_edgelist(G,nodelist=None):

    if nodelist is None:
        return G.edges(data=True)
    else:
        return G.edges(nodelist,data=True)

def from_edgelist(edgelist,create_using=None):

    G=_prep_create_using(create_using)
    G.add_edges_from(edgelist)
    return G

__author__ = """Aric Hagberg (hagberg@lanl.gov)\nPieter Swart (swart@lanl.gov)\nDan
Schult(dschult@colgate.edu)\nLoïc Séguin-C. <loicseguin@gmail.com>"""
#    Copyright (C) 2004-2016 by
#    Aric Hagberg <hagberg@lanl.gov>
#    Dan Schult <dschult@colgate.edu>
#    Pieter Swart <swart@lanl.gov>
#    All rights reserved.
#    BSD license.
#

# Exception handling

# the root of all Exceptions
class NetworkXException(Exception):
    """Base class for exceptions in NetworkX."""

class NetworkXError(NetworkXException):
    """Exception for a serious error in NetworkX"""

class NetworkXPointlessConcept(NetworkXException):
    """Harary, F. and Read, R. "Is the Null Graph a Pointless Concept?"
In Graphs and Combinatorics Conference, George Washington University.
New York: Springer-Verlag, 1973.
"""

class NetworkXAlgorithmError(NetworkXException):
    """Exception for unexpected termination of algorithms."""

class NetworkXUnfeasible(NetworkXAlgorithmError):
    """Exception raised by algorithms trying to solve a problem
    instance that has no feasible solution."""

class NetworkXNoPath(NetworkXUnfeasible):
    """Exception for algorithms that should return a path when running
    on graphs where such a path does not exist."""

class NetworkXNoCycle(NetworkXUnfeasible):
    """Exception for algorithms that should return a cycle when running
    on graphs where such a cycle does not exist."""

class NetworkXUnbounded(NetworkXAlgorithmError):
    """Exception raised by algorithms trying to solve a maximization
    or a minimization problem instance that is unbounded."""
```

```
class NetworkXNotImplemented(NetworkXException):
    """Exception raised by algorithms not implemented for a type of graph."""

class NodeNotFound(NetworkXException):
    """Exception raised if requested node is not present in the graph"""

__all__ = ['shortest_path', 'all_shortest_paths',
           'shortest_path_length', 'average_shortest_path_length',
           'has_path']


def has_path(G, source, target):

    try:
        sp = shortest_path(G, source, target)
    except NetworkXNoPath:
        return False
    return True


def shortest_path(G, source=None, target=None, weight=None):

    if source is None:
        if target is None:
            # Find paths between all pairs.
            if weight is None:
                paths = all_pairs_shortest_path(G)
            else:
                paths = all_pairs_dijkstra_path(G, weight=weight)
        else:
            # Find paths from all nodes co-accessible to the target.
            with utils.reversed(G):
                if weight is None:
                    paths = single_source_shortest_path(G, target)
                else:
                    paths = single_source_dijkstra_path(G, target,
                                                        weight=weight)
                # Now flip the paths so they go from a source to the target.
                for target in paths:
                    paths[target] = list(reversed(paths[target]))

    else:
        if target is None:
            # Find paths to all nodes accessible from the source.
            if weight is None:
                paths = single_source_shortest_path(G, source)
            else:
                paths = single_source_dijkstra_path(G, source,
                                                    weight=weight)
        else:
            # Find shortest source-target path.
            if weight is None:
                paths = bidirectional_shortest_path(G, source, target)
            else:
                paths = dijkstra_path(G, source, target, weight)

    return paths



def single_source_shortest_path(G,source,cutoff=None):


    level=0                  # the current level
    nextlevel={source:1}       # list of nodes to check at next level
```

```
    paths={source:[source]}  # paths dictionary  (paths to key from source)
    if cutoff==0:
        return paths
    while nextlevel:
        thislevel=nextlevel
        nextlevel={}
        for v in thislevel:
            for w in G[v]:
                if w not in paths:
                    paths[w]=paths[v]+[w]
                    nextlevel[w]=1
        level=level+1
        if (cutoff is not None and cutoff <= level):  break
    return paths


def multi_source_dijkstra_path(G, sources, cutoff=None, weight='weight'):

    length, path = multi_source_dijkstra(G, sources, cutoff=cutoff,
                                         weight=weight)
    return path

def multi_source_dijkstra(G, sources, target=None, cutoff=None,
                          weight='weight'):

    if not sources:
        raise ValueError('sources must not be empty')
    if target in sources:
        return ({target: 0}, {target: [target]})
    weight = _weight_function(G, weight)
    paths = {source: [source] for source in sources}  # dictionary of paths
    dist = _dijkstra_multisource(G, sources, weight, paths=paths,
                                 cutoff=cutoff, target=target)
    return (dist, paths)


def _weight_function(G, weight):

    if callable(weight):
        return weight
    # If the weight keyword argument is not callable, we assume it is a
    # string representing the edge attribute containing the weight of
    # the edge.
    if G.is_multigraph():
        return lambda u, v, d: min(attr.get(weight, 1) for attr in d.values())
    return lambda u, v, data: data.get(weight, 1)

def dijkstra_path(G, source, target, weight='weight'):

    (length, path) = single_source_dijkstra(G, source, target=target,
                                            weight=weight)
    try:
        return path[target]
    except KeyError:
        raise NetworkXNoPath(
            "node %s not reachable from %s" % (source, target))

def _dijkstra_multisource(G, sources, weight, pred=None, paths=None,
                          cutoff=None, target=None):

    G_succ = G.succ if G.is_directed() else G.adj

    push = heappush
    pop = heappop
    dist = {}  # dictionary of final distances
```

```
    seen = {}
    # fringe is heapq with 3-tuples (distance,c,node)
    # use the count c to avoid comparing nodes (may not be able to)
    c = count()
    fringe = []
    for source in sources:
        seen[source] = 0
        push(fringe, (0, next(c), source))
    while fringe:
        (d, _, v) = pop(fringe)
        if v in dist:
            continue  # already searched this node.
        dist[v] = d
        if v == target:
            break
        for u, e in G_succ[v].items():
            cost = weight(v, u, e)
            if cost is None:
                continue
            vu_dist = dist[v] + cost
            if cutoff is not None:
                if vu_dist > cutoff:
                    continue
            if u in dist:
                if vu_dist < dist[u]:
                    raise ValueError('Contradictory paths found:',
                                     'negative weights?')
            elif u not in seen or vu_dist < seen[u]:
                seen[u] = vu_dist
                push(fringe, (vu_dist, next(c), u))
                if paths is not None:
                    paths[u] = paths[v] + [u]
                if pred is not None:
                    pred[u] = [v]
            elif vu_dist == seen[u]:
                if pred is not None:
                    pred[u].append(v)

    # The optional predecessor and path dictionaries can be accessed
    # by the caller via the pred and paths objects passed as arguments.
    return dist


def single_source_dijkstra(G, source, target=None, cutoff=None,
                           weight='weight'):

    return multi_source_dijkstra(G, {source}, cutoff=cutoff, target=target,
                                 weight=weight)


def single_source_dijkstra_path(G, source, cutoff=None, weight='weight'):

    return multi_source_dijkstra_path(G, {source}, cutoff=cutoff,
                                      weight=weight)

def single_source_shortest_path_length(G,source,cutoff=None):

    seen = {}                  # level (number of hops) when seen in BFS
    level = 0                  # the current level
    nextlevel = {source:1}  # dict of nodes to check at next level

    while nextlevel:
        thislevel = nextlevel  # advance to next level
        nextlevel = {}         # and start a new list (fringe)
        for v in thislevel:
```

```
            if v not in seen:
                seen[v] = level # set the level of vertex v
                nextlevel.update(G[v]) # add neighbors of v
                yield (v, level)
        if (cutoff is not None and cutoff <= level):  break
        level=level+1
    del seen


def all_pairs_shortest_path_length(G, cutoff=None):

    length = single_source_shortest_path_length
    # TODO This can be trivially parallelized.
    for n in G:
        yield (n, dict(length(G, n, cutoff=cutoff)))


def bidirectional_shortest_path(G,source,target):


    # call helper to do the real work
    results=_bidirectional_pred_succ(G,source,target)
    pred,succ,w=results

    # build path from pred+w+succ
    path=[]
    # from source to w
    while w is not None:
        path.append(w)
        w=pred[w]
    path.reverse()
    # from w to target
    w=succ[path[-1]]
    while w is not None:
        path.append(w)
        w=succ[w]

    return path

def _bidirectional_pred_succ(G, source, target):

    # does BFS from both source and target and meets in the middle
    if target == source:
        return ({target:None},{source:None},source)

    # handle either directed or undirected
    if G.is_directed():
        Gpred=G.predecessors
        Gsucc=G.successors
    else:
        Gpred=G.neighbors
        Gsucc=G.neighbors

    # predecesssor and successors in search
    pred={source:None}
    succ={target:None}

    # initialize fringes, start with forward
    forward_fringe=[source]
    reverse_fringe=[target]

    while forward_fringe and reverse_fringe:
        if len(forward_fringe) <= len(reverse_fringe):
            this_level=forward_fringe
            forward_fringe=[]
```

```
            for v in this_level:
                for w in Gsucc(v):
                    if w not in pred:
                        forward_fringe.append(w)
                        pred[w]=v
                    if w in succ:  return pred,succ,w # found path
        else:
            this_level=reverse_fringe
            reverse_fringe=[]
            for v in this_level:
                for w in Gpred(v):
                    if w not in succ:
                        succ[w]=v
                        reverse_fringe.append(w)
                    if w in pred:  return pred,succ,w # found path

    raise NetworkXNoPath("No path between %s and %s." % (source, target))




def all_pairs_shortest_path(G, cutoff=None):

    # TODO This can be trivially parallelized.
    return {n: single_source_shortest_path(G, n, cutoff=cutoff) for n in G}


def predecessor(G,source,target=None,cutoff=None,return_seen=None):


    level=0                  # the current level
    nextlevel=[source]       # list of nodes to check at next level
    seen={source:level}      # level (number of hops) when seen in BFS
    pred={source:[]}         # predecessor dictionary
    while nextlevel:
        level=level+1
        thislevel=nextlevel
        nextlevel=[]
        for v in thislevel:
            for w in G[v]:
                if w not in seen:
                    pred[w]=[v]
                    seen[w]=level
                    nextlevel.append(w)
                elif (seen[w]==level):# add v to predecessor list if it
                    pred[w].append(v) # is at the correct level
        if (cutoff and cutoff <= level):
            break

    if target is not None:
        if return_seen:
            if not target in pred: return ([],-1)  # No predecessor
            return (pred[target],seen[target])
        else:
            if not target in pred: return []  # No predecessor
            return pred[target]
    else:
        if return_seen:
            return (pred,seen)
        else:
            return pred
```