```python
#!/usr/bin/env python

from __future__ import division
import math
import random
import networkx as nx

"""
Implementations of d-Heaps and Prim's MST following Tarjan. Includes testing
and visualization code for both.
"""

ARITY = 3  # the branching factor of the d-Heaps

#=========================================================================
# d-Heap
#=========================================================================

class HeapItem(object):
    """Represents an item in the heap"""
    def __init__(self, key, item):
        self.key = key
        self.item = item
        self.pos = None

def makeheap(S):
    """Create a heap from set S, which should be a list of pairs (key, item)."""
    heap = list(HeapItem(k,i) for k,i in S)
    for pos in xrange(len(heap)-1, -1, -1):
        siftdown(heap[pos], pos, heap)
    return heap

def findmin(heap):
    """Return element with smallest key, or None if heap is empty"""
    return heap[0] if len(heap) > 0 else None

def deletemin(heap):
    """Delete the smallest item"""
    if len(heap) == 0: return None
    i = heap[0]
    last = heap[-1]
    del heap[-1]
    if len(heap) > 0:
        siftdown(last, 0, heap)
    return i

def heapinsert(key, item, heap):
    """Insert an item into the heap"""
    heap.append(None)
    hi = HeapItem(key,item)
    siftup(hi, len(heap)-1, heap)
    return hi

def heap_decreasekey(hi, newkey, heap):
    """Decrease the key of hi to newkey"""
    hi.key = newkey
    siftup(hi, hi.pos, heap)

def siftup(hi, pos, heap):
    """Move hi up in heap until it's parent is smaller than hi.key"""
    p = parent(pos)
    while p is not None and heap[p].key > hi.key:
        heap[pos] = heap[p]
        heap[pos].pos = pos
        pos = p

        p = parent(p)
    heap[pos] = hi
    hi.pos = pos

def siftdown(hi, pos, heap):
```

```
    """Move hi down in heap until its smallest child is bigger than hi's key"""
    c = minchild(pos, heap)
    while c != None and heap[c].key < hi.key:
        heap[pos] = heap[c]
        heap[pos].pos = pos
        pos = c
        c = minchild(c, heap)
    heap[pos] = hi
    hi.pos = pos

def parent(pos):
    """Return the position of the parent of pos"""
    if pos == 0: return None
    return int(math.ceil(pos / ARITY) - 1)

def children(pos, heap):
    """Return a list of children of pos"""
    return xrange(ARITY * pos + 1, min(ARITY * (pos + 1) + 1, len(heap)))

def minchild(pos, heap):
    """Return the child of pos with the smallest key"""
    minpos = minkey = None
    for c in children(pos, heap):
        if minkey == None or heap[c].key < minkey:
            minkey, minpos = heap[c].key, c
    return minpos


#=========================================================================
# Heap Testing and Visualization Code
#=========================================================================

def bfs_tree_layout(G, root, rowheight = 0.02, nodeskip = 0.6):
    """Return node position dictionary, layingout the graph in BFS order."""
    def width(T, u, W):
        """Returns the width of the subtree of T rooted at u; returns in W the
        width of every node under u"""
        W[u] = sum(width(T, c, W)
            for c in T.successors(u)) if len(T.successors(u))>0 else 1.0
        return W[u]

    T = nx.bfs_tree(G, root)
    W = {}
    width(T, root, W)

    pos = {}
    left = {}
    queue = [root]
    while len(queue):
        c = queue[0]
        del queue[0]  # pop

        left[c] = 0.0  # amt of child space used up

        # posn is computed relative to the parent
        if c == root:
            pos[c] = (0,0)
        else:
            p = T.predecessors(c)[0]
            pos[c] = (
                pos[p][0] - W[p]*nodeskip/2.0 + left[p] + W[c]*nodeskip/2.0,
                pos[p][1] - rowheight
            )
            left[p] += W[c]*nodeskip

        # add the children to the queue
        for i,u in enumerate(G.successors(c)):
            queue.append(u)
    return pos

def snapshot_heap(heap):
```

```
    draw_heap(heap, pdffile)

def draw_heap(heap, outfile=None):
    """Draw the heap using matplotlib and networkx"""
    import matplotlib.pyplot as plt
    G = nx.DiGraph()
    for i in xrange(1, len(heap)):
        G.add_edge(parent(i), i)

    labels = dict((u, "%d" % (heap[u].key)) for u in G.nodes())

    plt.figure(facecolor="w", dpi=80)
    plt.margins(0,0)
    plt.xticks([])
    plt.yticks([])
    plt.box(False)
    nx.draw_networkx(G,
        labels=labels,
        node_size = 700,
        node_color = "white",
        pos=bfs_tree_layout(G, 0))
    if outfile is not None:
        plt.savefig(outfile, format="pdf", dpi=150, bbox_inches='tight', pad_inches=0.0)
        plt.close()
    else:
        plt.show()

def test_heap():
    """Generate a random heap"""
    global pdffile
    pdffile = start_pdf("mst.pdf")
    draw_heap(makeheap((random.randint(0,100), 'a') for i in xrange(40)))


#==========================================================================
# Prim's minimum spanning tree algorithm
#==========================================================================

def prim_mst(G):
    """Compute the minimum spanning tree of G. Assumes each edge has an
    attribute 'length' giving it's length. Returns a dictionary P such
    that P[u] gives the parent of u in the MST."""

    for u in G.nodes():
        G.node[u]['distto'] = float("inf")  # key stores the Prim key
        G.node[u]['heap'] = None         # heap = pointer to node's HeapItem
    parent = {}

    heap = makeheap([])
    v = G.nodes()[0]

    # go through vertices in order of closest to current tree
    while v != None:
        G.node[v]['distto'] = float("-inf") # v now in the tree

        snapshot_mst(G, parent)

        # update the estimated distance to each of v's neighbors
        for w in G.neighbors(v):
            # if new length is smaller that old length, update
            if G[v][w]['length'] < G.node[w]['distto']:
                # closest tree node to w is v
                G.node[w]['distto'] = G[v][w]['length']
                parent[w] = v

                # add to heap or decreae key if already in heap
                hi = G.node[w]['heap']
                if hi is None:
                    G.node[w]['heap'] = heapinsert(G.node[w]['distto'], w, heap)
                else:
                    heap_decreasekey(hi, G.node[w]['distto'], heap)
        # get the next vertex closest to the tree
```

```
        # get the next vertex closest to the tree
        v = deletemin(heap)
        v = v.item if v is not None else None
    return parent

#==========================================================================
# Union-Find
#==========================================================================

class ArrayUnionFind:
    """Holds the three "arrays" for union find"""
    def __init__(self, S):
        self.group = dict((s,s) for s in S) # group[s] = id of its set
        self.size = dict((s,1) for s in S) # size[s] = size of set s
        self.items = dict((s,[s]) for s in S) # item[s] = list of items in set s

def make_union_find(S):
    """Create a union-find data structure"""
    return ArrayUnionFind(S)

def find(UF, s):
    """Return the id for the group containing s"""
    return UF.group[s]

def union(UF, a,b):
    """Union the two sets a and b"""
    assert a in UF.items and b in UF.items
    # make a be the smaller set
    if UF.size[a] > UF.size[b]:
        a,b = b,a
    # put the items in a into the larger set b
    for s in UF.items[a]:
        UF.group[s] = b
        UF.items[b].append(s)
    # the new size of b is increased by the size of a
    UF.size[b] += UF.size[a]
    # remove the set a (to save memory)
    del UF.size[a]
    del UF.items[a]

#==========================================================================
# Kruskal MST
#==========================================================================

def kruskal_mst(G):
    """Return a minimum spanning tree using kruskal's algorithm"""
    # sort the list of edges in G by their length
    Edges = [(u, v, G[u][v]['length']) for u,v in G.edges()]
    Edges.sort(cmp=lambda x,y: cmp(x[2],y[2]))

    UF = make_union_find(G.nodes())  # union-find data structure

    # for edges in increasing weight
    mst = [] # list of edges in the mst
    for u,v,d in Edges:
        setu = find(UF, u)
        setv = find(UF, v)
        # if u,v are in different components
        if setu != setv:
            mst.append((u,v))
            union(UF, setu, setv)
            snapshot_kruskal(G, mst)
    return mst

#==========================================================================
# MST Testing and Visualization Code
#==========================================================================

def dist(xy1, xy2):
    """Euclidean distance"""
    return math.sqrt((xy1[0] - xy2[0])**2 + (xy1[1] - xy2[1])**2)
```

```
def random_mst_graph(n, k=4):
    """Make a random graph by choosing n nodes in the [0,1.0] by [0,1]
    square. The 'length' of each edge is the euclidean distance between
    them. Edges connect to the k nearest neighbors of each node."""

    # build random nodes
    G = nx.Graph()
    for i in xrange(n):
        G.add_node(i, pos=(0.9*random.random()+0.05,0.9*random.random()+0.05))

    # add edges
    for i in G.nodes():
        near = [(u, dist(G.node[i]['pos'],G.node[u]['pos']))
                    for u in G.nodes() if u != i]
        near.sort(cmp=lambda x,y: cmp(x[1],y[1]))
        for u,d in near[0:k]:
            G.add_edge(i, u, length=d)

    # ensure it's connected
    CC = nx.connected_components(G)
    for i in xrange(len(CC)-1):
        u = random.choice(CC[i])
        v = random.choice(CC[i+1])
        G.add_edge(u,v, length=dist(G.node[u]['pos'], G.node[v]['pos']))
    return G


def draw_mst_graph(G, T={}, outfile=None):
    """Draw the MST graph, highlight edges given by the MST parent dictionary
    T. T should be in the same format as returned by prim_mst()."""

    import matplotlib.pyplot as plt

    # construct the attributes for the edges
    pos = dict((u,G.node[u]['pos']) for u in G.nodes())
    labels = dict((u,str(u)) for u in G.nodes())
    colors = []
    width = []
    for u,v in G.edges():
        if isinstance(T, dict):
            inmst = (u in T and v == T[u]) or (v in T and u == T[v])
        elif isinstance(T, nx.Graph):
            inmst = T.has_edge(u,v)
        colors.append(1 if inmst else 255)
        width.append(5 if inmst else 1)

    # draw it
    plt.figure(facecolor="w", dpi=80)
    plt.margins(0,0)
    plt.xticks([])
    plt.yticks([])
    plt.ylim(0.0,1.0)
    plt.xlim(0.0,1.0)
    plt.box(False)
    nx.draw_networkx(G,
        labels=labels,
        node_size = 500,
        node_color = "white",
        edge_color = colors,
        width=width,
        pos=pos)
    if outfile is not None:
        plt.savefig(outfile, format="pdf", dpi=150, bbox_inches='tight', pad_inches=0.0)
        plt.close()
    else:
        plt.show()

def snapshot_kruskal(G, edges, pdf=True):
    T = nx.Graph()
    for u,v in edges: T.add_edge(u,v)
    draw_mst_graph(G, T, pdffile if pdf else None)
```

```
def snapshot_mst(G, parent):
    tree = dict((u,parent[u])
        for u in G.nodes()
            if G.node[u]['distto'] == float("-inf") and u in parent)
    draw_mst_graph(G, tree, pdffile)

def test_mst():
    """Draw the MST for a random graph."""
    global pdffile
    pdffile = start_pdf("mst.pdf")
    N = random_mst_graph(20)
    draw_mst_graph(N, prim_mst(N))
    close_pdf(pdffile)

def test_kruskal():
    """Draw the MST for a random graph."""
    global pdffile
    pdffile = start_pdf("kruskal.pdf")
    N = random_mst_graph(20)
    snapshot_kruskal(N, kruskal_mst(N), False)
    close_pdf(pdffile)

def start_pdf(outfile):
    from matplotlib.backends.backend_pdf import PdfPages
    pp = PdfPages(outfile)
    return pp

def close_pdf(pp):
    pp.close()

def main():
    import sys
    if len(sys.argv) >= 2:
        if sys.argv[1] == "heap": test_heap()
        if sys.argv[1] == "mst": test_mst()
        if sys.argv[1] == "kruskal": test_kruskal()
    else:
        print "Usage: mstprim.py [heap|mst|kruskal]"

if __name__ == "__main__": main()
```