Fundamental Algorithms Homework 11
MST, Prim, Kruskal, Single-Source Shortest Path, Bellman-Ford
Michael Lukiman mll469
Professor Amir Shpilka

1. Prim's MST

All weights the same.

```
# 1.1 Homogenous nodes.

def MST_homogeneous(G,w):

    Q = G.v

    u = dequeue(Q)
    u.key = 1

    while Q != []:
        recur(G, u, v, Q)

recur(G, u, v, Q):
    for v in G.adj[u]:
            if v in Q:
                v = dequeue(Q)
                v.pi = u
                v.key = w
                recur(G, u, v, Q)


# Way faster than Prim, at O(V) steps.
# Since all edges are the same, can make arbitrary tree as long as every next v is still in the Q and is connected to current u. We assume the graph is connected and has no double edges. This is basically DFS.
```

```
# 1.2 One weight that's not like the others.

def MST_odd_one_out(G,w):

    # if len(G.v) < len(G.e):
    #     adjacency_search = 'fib_heap'
    # else:
    #     adjacency_search = 'binary_heap'

    normal_weight, special_weight = determine_normal_weight(w)
    special_weight = determine_special_weight(w, special_weight)
    w = normal_weight
```

```
	if normal_weight => special_weight:
		Q = G.v
		u = dequeue(Q)
		u.key = 0

		while len(Q) > 1: # While there's more than one element queued
			if Q[0] in G.adj[u] and w(u,Q[0]) == normal_weight: # Assume Q dequeues from
front. If we prioritize the normal weight O(2V):
				v = dequeue(Q)
				v.pi = u
				v.key = w
				u = v
			else: # If it's not the normal weight, then the vertex gets moved to the back of the
queue.
				Q.append(dequeue(Q))

		v = dequeue(Q) # The last element is either unreachable by normal weights via the tree
we've made so far or it is simply the last element without any rearrangement, thus we see if there are
any normal adjacent weights:

		global key
		if any_adj_normals(G,v) != True: # ...if not true that means there is only one edge from
the vertex and it is the special weight.
			v.pi = key
			v.key = special_weight

	else: # If the normal isn’t => than the special, we prefer the special weight if we can see it.
		for u in G.v:
			extract_min(G.adj[u]): # O(ElogV) worst-case to find the special weight if graph
is strongly connected. O(V)
				if w(u,v) = special_weight: # Find that weight.
					v.pi = u
					v.key = special_weight
					key_G = G
					G.v.remove(u)
					G_prime = G
					G = key_G
					MST_homogeneous(G_prime) # Run the uniform w algorithm on
the rest of the vertices after making the first connection with w'. O(E)

def any_adj_normals(G,v): # O(V-1)
	for x in G.adj[v]:
		if w(x,v) == normal_weight: # If there is a normal weight we make the other vertex the
parent of v.
			v.pi = x
			v.key = w
			return True
		if w(x,v) == special_weight:
```

```
                global key
                key = x
                return False



def determine_special_weight(w, special_weight): O(V)

        if special_weight == 'none_yet':
            for weight in w:
                if weight != normal_weight:
                    special_weight = weight
                    return special_weight
        else:
            return special_weight
```

def determine_normal_weight(w): # returns normal weight (and special weight if found within the first two elements. else 'none_yet'). O(1)

```
    if len(w) < 3: # If less than three elements...
        return w[0], w[0] # Normalcy is arbitrary, can say first is normal and special.
```

    if w[0] == w[1]: # If the first two elements are the same, they share the normal weight (since there's only one element that will not have the normal weight).

```
        return w[0], 'none_yet'  # Return the normal.
```

    if w[1] == w[2]: # If the function continues, that means index 0 and 1 have different weights, therefore one of them is the special weight. If index 1 and 2 are the same, then index 0 is the special weight, and that's good news for the running time. We have an additional thing to return, which is the special weight.

```
        return w[1], w[0]
```

    else: # If the function still continues, then w[1] is the special one, because given the nature of the list, w[2] will be be the same as w[0]. (While this relies on the integrity of the input, this function is only for the given problem to minimize running time.)

```
        return w[2], w[1]
```

Runtime takes max ElogV or VlogV with the addition of the functions to determine the normal and special weight, taking maximum E additional steps to find the special weight, plus the last element in the queue, which can take an additional V-1 steps if it connects to all other nodes, and worst case if the special weight edge is the last in its adjacency list. Here we can use the heap search for efficiency in the single search, though the algorithm would still be dominated by XlogV.

———

2.

Weights are integer values between 1 and W, inclusive.

```
def MST_known_integers():
	sorted_w = counting_sort(w,max_w)

	Qw = sorted_w # O(E)
	Qv = G.v
	Qe = G.e

	while Qv != []:  # O(V)
		weight_to_find = dequeue(Qw) # Already sorted so can just dequeue.
		find_edge(G,Qe,weight_to_find)

def find_edge(G,Qe,weight_to_find):
	for edge in Qe: # O(W)
		if w(edge) = weight_to_find and edge[1] in Qv: # Find the lightest edge connecting to a separated vertex.
			edge[1].pi = edge[0] # v.pi of edge is u
			edge[1].key = w(edge)
			Qv.remove(v) # v is accounted for in the tree with the smallest weight connected to it
			return

# O(E) + O(V)*O(U)



# Counting Sort Base Algorithm
def counting_sort(array, maxval):
   """in-place counting sort"""
   m = maxval + 1
   count = [0] * m              # init with zeros
   for a in array:
      count[a] += 1             # count occurences
   i = 0
   for a in range(m):            # emit
      for c in range(count[a]):
         array[i] = a
         i += 1
   return array # and count, if needed.
```

3.

MakeSet, FindSet, Union

Hint: connecting root of shallower tree to deeper tree.

Maybe heap/binary tree

```
def MST_Kruskal(G,w):
	A = make_set(G)

	w = quicksort(w) # non-decreasing order

	# m log n for m edges

	for e in G.e:
		if find_set(e.u) != find_set(e.v): # at most 2m finds, each log n
			A = union(A, e) # upper bound of n - 1 vertices

	return A

# dominated by O(2m log n)

def make_set(G):
	all = []
	for vertex in G.v: # Initialize set to be a one node tree for each vertex.
		vertex.pi = NIL
		all.append(vertex)
	return all

def union(x,y):
	if len(x) < len(y):
		y[0].pi = x[len(x)-1] # The root of y is attached to the tree of x.
		for element in y:
			x.append(element)
	else:
		x[0].pi = y[len(x)-1] # The root of x is attached to the tree of y.
		for element in x:
			y.append(element)

def find_set(tree, node):
	return binary_tree_search(tree, i)

	# O(ln(n))
	# Depth of a node is the amount of times set was unioned/reassigned.
	# Since it's always merged with a larger, sets at least double with each redirection.
	# Total algorithm is dominated by O(2m log n)
```

4.

Relax edges in order: (t,x), (t,y), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y). Show d and parent values at each pass.

4.1 z is source with distance zero. The rest are distance infinity.

Pass 1:

s=inf, pi=nil
t=inf, pi=nil
x=inf, pi=nil
y=inf, pi=nil
z=0, pi=nil

t to x is inf + 5 => x.d = inf, pi=t
t to y is inf + 8 => y.d = inf, pi=t
t to z is inf + -4 => unchanged, z.d = 0, pi=nil
x to t is inf + -2 => t.d = inf, pi=t
y to x is inf + -3 => x.d = inf, pi=y
y to z is inf + 9 => unchanged, z.d = 0, pi=nil
z to x is 0 + 7 => x.d = 7, pi=z
z to s is 0 + 2 => s.d = 2, pi=z
s to t is 2 + 6 => t.d = 8, pi=s
s to y is 2 + 7 => y.d = 9, pi=s

Pass 2:

s=2, pi=z
t=8, pi=s
x=7, pi=z
y=9, pi=s
z=0, pi=nil

t to x is 8 + 5 => unchanged, x.d = 7, pi=z #
t to y is 8 + 8 => unchanged, y.d = 9, pi=s #
t to z is 8 + -4 => unchanged, z.d = 0, pi=nil #
x to t is 7 + -2 => t.d = 5, pi=x #
y to x is 9 + -3 => x.d = 6, pi=y #
y to z is 9 + 9 => unchanged, z.d = 0, pi=nil #
z to x is 0 + 7 => unchanged, x.d = 6, pi=y #
z to s is 0 + 2 => unchanged, s.d = 2, pi=z #
s to t is 2 + 6 => unchanged, t.d = 5, pi=x #
s to y is 2 + 7 => unchanged, y.d = 9, pi=s #

Pass 3:

s=2, pi=z
t=5, pi=x
x=6, pi=y
y=9, pi=s
z=0, pi=nil

t to x is 5 + 5 => unchanged, x.d = 7, pi=z #
t to y is 5 + 8 => unchanged, y.d = 9, pi=s #
t to z is 5 + -4 => unchanged, z.d = 0, pi=nil #
x to t is 6 + -2 => t.d = 4, pi=x #
y to x is 9 + -3 => unchanged, x.d = 6, pi=y #
y to z is 9 + 9 => unchanged, z.d = 0, pi=nil #
z to x is 0 + 7 => unchanged, x.d = 6, pi=y #
z to s is 0 + 2 => unchanged, s.d = 2, pi=z #
s to t is 2 + 6 => unchanged, t.d = 5, pi=x #
s to y is 2 + 7 => unchanged, y.d = 9, pi=s #

Pass 4:

s=2, pi=z
t=4. pi=x
x=6, pi=y
y=9, pi=s
z=0, pi=nil

t to x is 4 + 5 => unchanged, x.d = 7, pi=z #
t to y is 4 + 8 => unchanged, y.d = 9, pi=s #
t to z is 4 + -4 => unchanged, z.d = 0, pi=nil #
x to t is 6 + -2 => unchanged, t.d = 4, pi=x #
y to x is 9 + -3 => unchanged, x.d = 6, pi=y #
y to z is 9 + 9 => unchanged, z.d = 0, pi=nil #
z to x is 0 + 7 => unchanged, x.d = 6, pi=y #
z to s is 0 + 2 => unchanged, s.d = 2, pi=z #
s to t is 2 + 6 => unchanged, t.d = 5, pi=x #
s to y is 2 + 7 => unchanged, y.d = 9, pi=s #

Bellman-Ford complete!

s=2, pi=z
t=4. pi=x
x=6, pi=y
y=9, pi=s
z=0, pi=nil

4.2

s is the source with distance zero. The rest are distance infinity, and (z,x) is now 4.

Pass 1:

s=0, pi=nil
t=inf, pi=nil

x=inf, pi=nil
y=inf, pi=nil
z=inf, pi=nil

t to x is inf + 5 => x.d = inf, pi=t.
t to y is inf + 8 => y.d = inf, pi=t
t to z is inf + -4 => z.d = inf, pi=t
x to t is inf + -2 => t.d = inf, pi=t
y to x is inf + -3 => x.d = inf, pi=y
y to z is inf + 9 => z.d = inf, pi=y
z to x is inf + 4 => x.d = inf, pi=z
z to s is inf + 2 => s.d = 0, pi=nil
s to t is 0 + 6 => t.d = 6, pi=s
s to y is 0 + 7 => y.d = 7, pi=s

Pass 2:

s=0, pi=nil
t=6, pi=s
x=inf, pi=z
y=7, pi=s
z=inf, pi=y

t to x is 6 + 5 => x.d = 11, pi=t
t to y is 6 + 8 => unchanged, y.d = 7, pi=s
t to z is 6 + -4 => z.d = 2, pi=t
x to t is 11 + -2 => unchanged, t.d = 6, pi=s
y to x is 7 + -3 => x.d = 4, pi=y
y to z is 7 + 9 => unchanged, z.d = 2, pi=t
z to x is 2 + 4 => unchanged, x.d = 4, pi=y
z to s is 2 + 2 => unchanged, s.d = 0, pi=nil
s to t is 0 + 6 => unchanged, t.d = 6, pi=s
s to y is 0 + 7 => unchanged, y.d = 7, pi=s

Pass 3:

s=0, pi=nil
t=6, pi=s
x=4, pi=y
y=7, pi=s
z=2, pi=t

t to x is 6 + 5 => unchanged, x.d = 4, pi=y
t to y is 6 + 8 => unchanged, y.d = 7, pi=s
t to z is 6 + -4 => unchanged, z.d = 2, pi=t
x to t is 4 + -2 => unchanged, t.d = 6, pi=s
y to x is 7 + -3 => unchanged, x.d = 4, pi=y
y to z is 7 + 9 => unchanged, z.d = 2, pi=t
z to x is 2 + 4 => unchanged, x.d = 4, pi=y

z to s is 2 + 2 => unchanged, s.d = 0, pi=nil
s to t is 0 + 6 => unchanged, t.d = 6, pi=s
s to y is 0 + 7 => unchanged, y.d = 7, pi=s

Bellman-Ford complete!