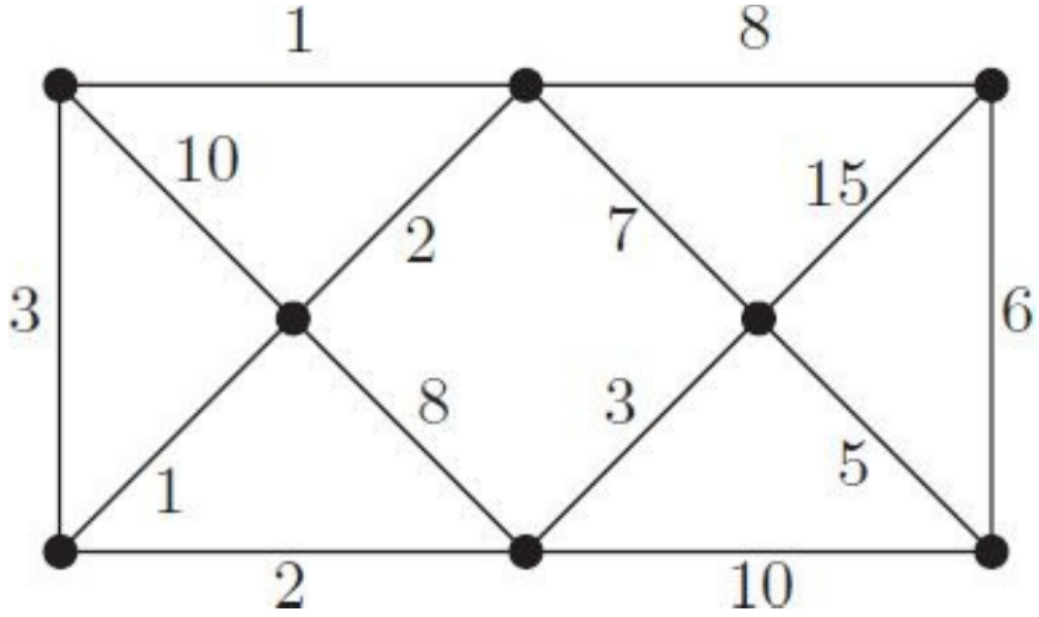
Math 15 - Spring 2017 - Homework 5.2 Solutions

1. (5.2 # 14 (not assigned)) Use Prim's algorithm to construct a minimal spanning tree for the following network. Draw the minimal tree and compute its total weight.



Prim's algorithm has

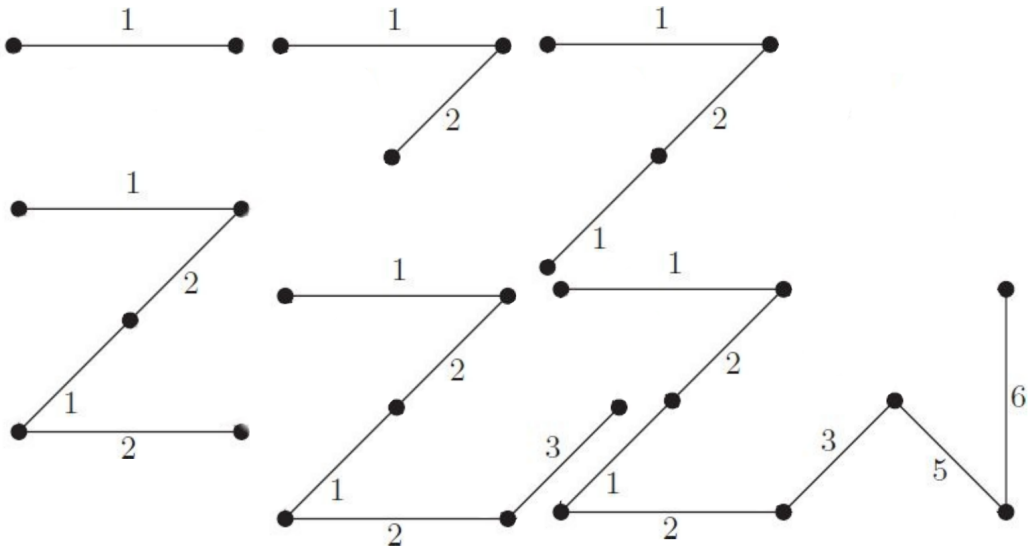
`precondition:` N is a connected network.

`postcondition:` T is a minimal spanning tree of N.

The algorithm is:

```
T <- e_1, where e_1 is the shortest edge of N
while T does not contain all of N's vertices
   e <- the shortest edge between a vertex in T and a vertex not in T.
   Add edge e and the new vertex to T.
```

There are two choices for e_1, so we pick the one on top, though it doesn't seem to matter–in the end you get this minimal spanning tree, as illustrated by the sequence below, reading roughly left to right and down:



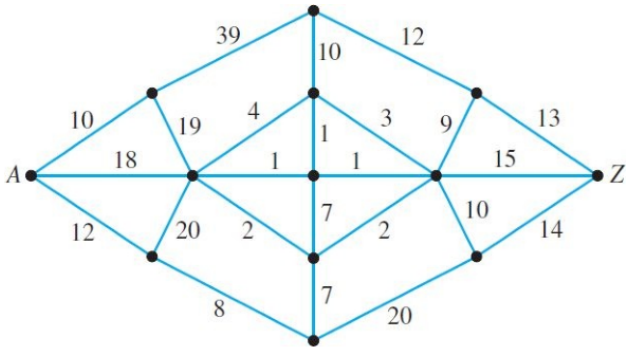
2. (5.2 # 16) Kruskal's algorithm gives another way to find a minimal spanning tree in a network.

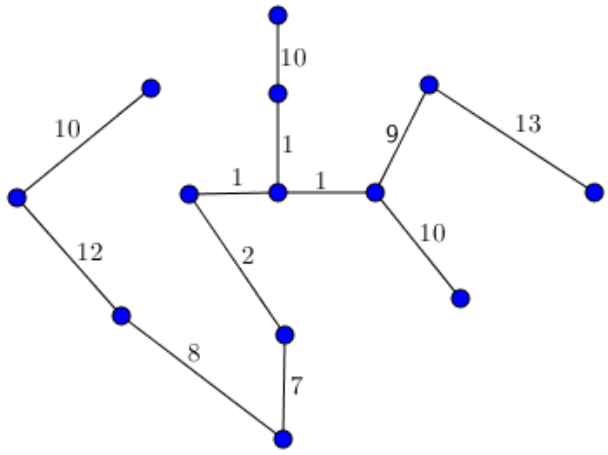
```
Algorithm 5.13 Kruskal's Algorithm for constructing a minimal spanning tree.
Preconditions:  N is a connected network with n > 2 vertices.
Postconditions:  T is a minimal spanning tree of N.
Append e_1 and its vertices to T, where e_1 is the shortest edge of N.
for i ∈ {2, ..., n − 1} do
    e_i ←  the shortest edge whose addition to T will not form a circuit.
    Add edge  e_i (and its vertices) to T.
```

   (a) Is Kruskal's algorithm greedy? Explain.
   Yes, it's greed: it's grabbing the smallest things first without regard to anything else.

   (b) Use Kruskal's algorithm to construct a minimal spanning tree of the network in Figure 5.10.
   ANS: You end up with the same spanning tree as in Prim's algo of exercise 14 above.

   (c) Use Kruskal's algorithm to construct a minimal spanning tree of the network in Figure 5.11.



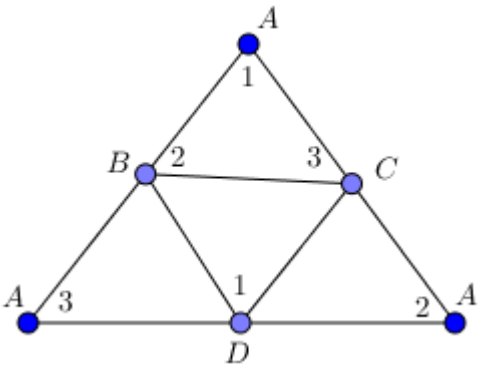


3. (5.2 # 18 (not assigned)) Recall that a coloring of a graph is an assignment of colors to the vertices such that no two vertices of the same color are connected by an edge. The following algorithm attempts to produce a coloring for the vertices of a graph G using the fewest number of colors possible.

```
C <- Null
while (G has vertices left to color) do
    Pick a new color x not in C.
    C <- C union {x}
    Assign color x to as many vertices of G as possible, such that no edge
    connects vertices of the same color.
```

   (a) Which type of algorithm is this?
   ANS: This algorithms attempts to accomplish the long-term goal of coloring the graph by doing the most obvious short-term task at every opportunity. As such, it's a greedy algorithm.

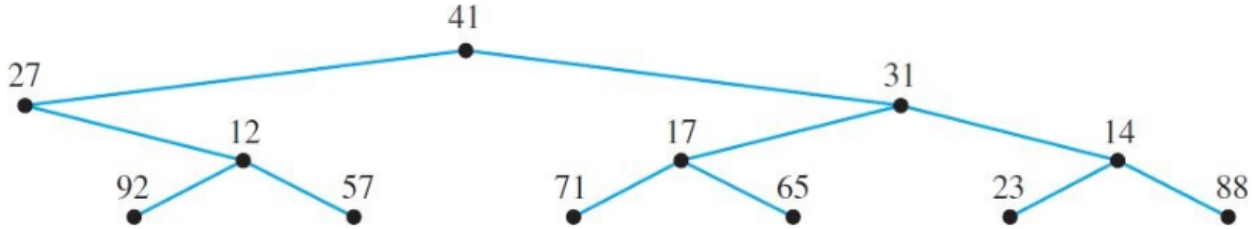
   (b) Find a graph for which this algorithm fails to produce a coloring with the fewest possible number of colors.
   ANS: The first vertices colored would be the three labeled 'A', then the algorithm fails to three-color (see numbers) the following graph:



4. (5.2 # 20) Let T be a binary tree whose nodes are elements of some set U. The following algorithm searches T for a target value $t \in U$ and returns true if and only if t is a node in T.

```
function Search (t ∈ U, T ∈ { binary trees})
if T is empty then
     return false
else
if t = the root of T then
     return true
else
     return (Search(t, left subtree of T) ∨ Search (t, right subtree of T))
```

(a) Write a top-down evaluation of `Search` $(17, T)$, where T is the tree in below.



$$\begin{aligned}
\texttt{Search}(17,T) &= \texttt{Search}(17,T_{27}) \vee \texttt{Search}(17,T_{31}) \\
&= \texttt{Search}(17,\emptyset) \vee \texttt{Search}(17,T_{12}) \vee \texttt{Search}(17,T_{17}) \vee \texttt{Search}(17,T_{14}) \\
&= \texttt{false} \vee \texttt{Search}(17,T_{92}) \vee \texttt{Search}(17,T_{57}) \vee \texttt{true} \vee \texttt{Search}(17,T_{23}) \vee \texttt{Search}(17,T_{88}) \\
&= \texttt{false} \vee \texttt{Search}(17,\emptyset) \vee \texttt{Search}(17,\emptyset) \vee \texttt{Search}(17,\emptyset) \vee \texttt{Search}(17,\emptyset) \\
&\quad \vee \texttt{true} \vee \texttt{Search}(17,\emptyset) \vee \texttt{Search}(17,\emptyset) \vee \texttt{Search}(17,\emptyset \vee \texttt{Search}(17,\emptyset) \\
&= \texttt{false} \vee \texttt{false} \vee \texttt{false} \vee \texttt{false} \vee \texttt{false} \vee \texttt{true} \vee \texttt{false} \vee \texttt{false} \vee \texttt{false} \vee \texttt{false} \\
&= \texttt{true}
\end{aligned}$$

(b) Which type of algorithm is this? Explain.
This is an `inorder traversal` algorithm. You can implement this in Python as follows. First, create your tree (the simple way, with a list of lists.)

```
T = [41,    #root
       [21,[],  #left subtree
        [12,
         [92,[],[]],
         [57,[],[]]]],
       [31,
        [17,
         [71,[],[]],
         [65,[],[]]],
        [14,
         [23,[],[]],
         [88,[],[]]]]]
```

Here, then is the `inorder traversal`:

```
def InSearch(t,T):
    if len(T)==0:
        print('Null')
        return False
```

```
    elif InSearch(t,T[1]) or t==T[0]:
        return True
    else:
        print('InSearch(',t,T[2],')')
        return InSearch(t,T[2])
print(InSearch(17,T))
```

The print statement produces this output:

```
Null
InSearch( 17 [12, [92, [], []], [57, [], []]] )
Null
InSearch( 17 [] )
Null
InSearch( 17 [57, [], []] )
Null
InSearch( 17 [] )
Null
InSearch( 17 [31, [17, [71, [], []], [65, [], []]], [14, [23, [], []], [88, [], []]]] )
Null
InSearch( 17 [] )
Null
True
```

By contrast, searching for 88 produces this output:

```
Null
InSearch( 88 [12, [92, [], []], [57, [], []]] )
Null
InSearch( 88 [] )
Null
InSearch( 88 [57, [], []] )
Null
InSearch( 88 [] )
Null
InSearch( 88 [31, [17, [71, [], []], [65, [], []]], [14, [23, [], []], [88, [], []]]] )
Null
InSearch( 88 [] )
Null
InSearch( 88 [65, [], []] )
Null
InSearch( 88 [] )
Null
InSearch( 88 [14, [23, [], []], [88, [], []]] )
Null
InSearch( 88 [] )
Null
InSearch( 88 [88, [], []] )
Null
True
```

Contrast this with the Python code for PreOrder and PostOrder traversals:

```
def PreSearch(t,T):
    if len(T)==0:
        return False
    elif t==T[0]:
        return True
    else:
        return PreSearch(t,T[1]) or PreSearch(t,T[2])
```

```
def PostSearch(t,T):
    if len(T)==0:
        return False
    elif PostSearch(t,T[1]) or PostSearch(t,T[2]):
        return True
    else:
        return t==T[0]
```

5. (5.2 # 24) Write a divide-and-conquer algorithm that computes the sum of all elements of a finite set $K = \{k_1, k_2, \ldots, k_n\}$ of integers.
   ANS: The pseudocode would look something like this:

```
function DCSum(L)
    if L = {x} then
        return x
    else
        X1 <- {x_1,x_2,...,x_floor(n/2)}
        X2 <- {x_floor(n/2)+1,...,x_n}
    return DCSum(X1) + DCSum(X2)
```

   In Python, we'd have something like this:

```
## List comprehension for X = [0,1,2,...,100]
X = [i for i in range(101)]

def DCSum(X):
    if len(X) == 1:
        return X[0]
    else:
        X1 = X[:len(X)//2]
        X2 = X[len(X)//2:]
    return DCSum(X1)+DCSum(X2)

# Test
print(DCSum(X))
```

   Which returns, as expected, 5050