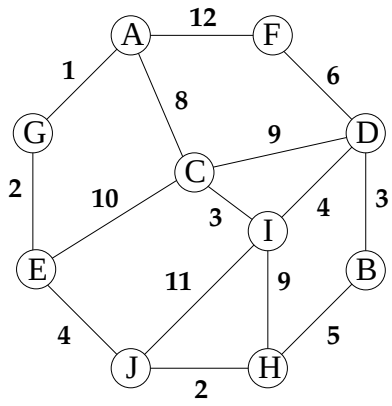
University of Nevada, Las Vegas Las Vegas Computer Science 477/677 Fall 2019

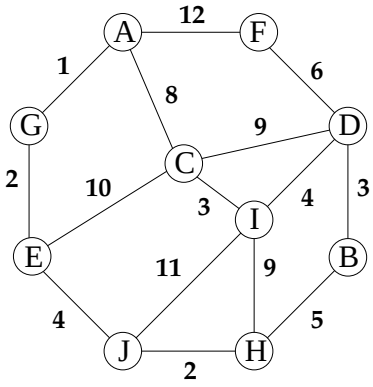
Answers to Assignment 6: Due Wednesday October 23, 2019

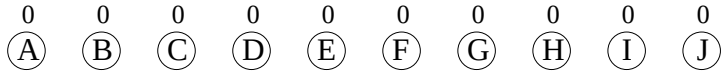
1.



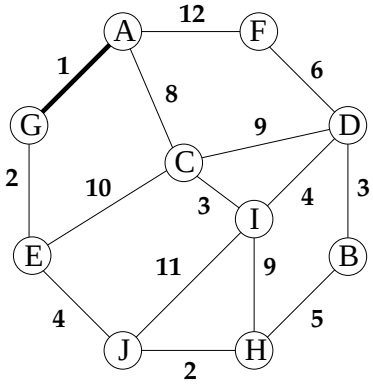
Use Kruskal's algorithm to find the minimum spanning tree of this weighted graph. Use the union/find data structure, and be sure to use path compression. Show the forest after each step. Union/find is explained in section 5.1.4 of the textbook.

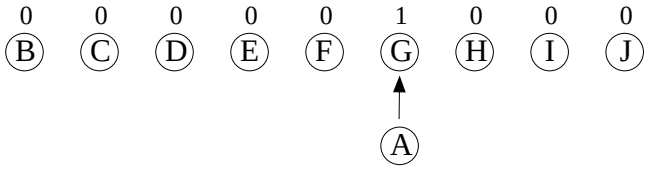
We use union/find to build a forest of all nodes. Initially, each node is a root of rank 0.



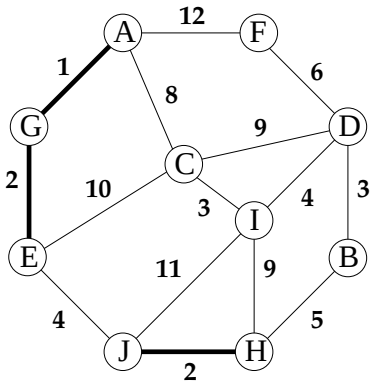


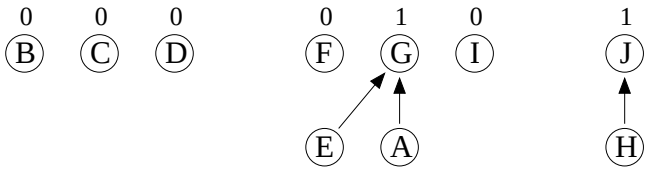
We examine the least weight edge, (A,G). We first compute find(A) = A and find(G) = G. We then execute union(A,G). The roots have equal rank, and we follow the rule from the textbook that, in that case, the alphabetically first root, A, attaches to the alphabetically second, G. G now has rank 1.



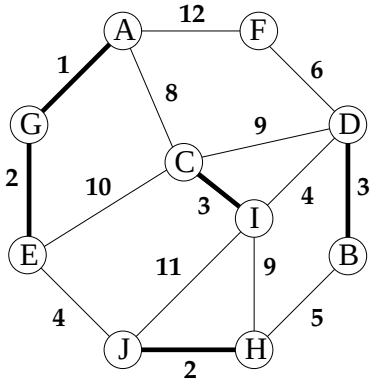


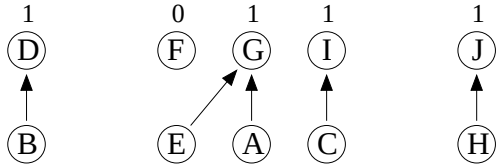
We will show two steps at once, since they don't interact with each other. The two smallese edges are (E,G) and (J,H), and find(E) = E, find(G) = G, find(J) = J, and find(H) = H. E has lower rank then G, and thus E attaches to G, which then still has rank 1. J and H have equal rank, and thus H attaches to J, which then has rank 1.



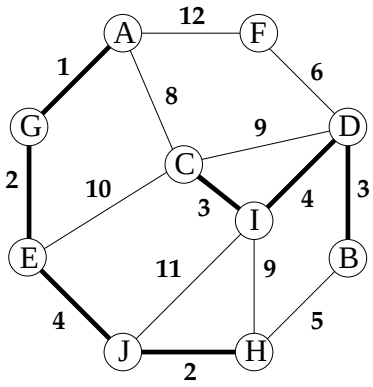


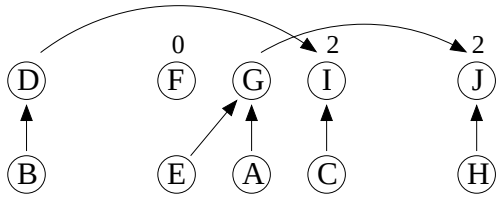
We show the next two steps at once. The two smallest edges are (B,D) and (C,I). Attach B to D, which then has rank 1, and attach C to I, which then has rank 1.



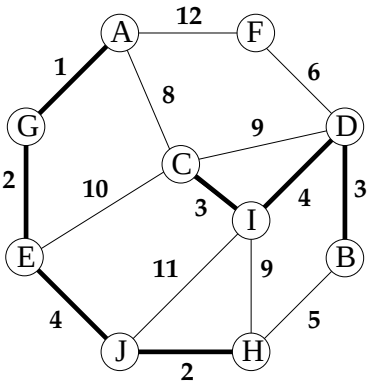


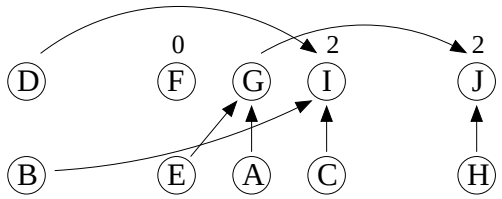
The next smallest edge is (D,I). We compute find(D) = H. and find(I) = I. We then attach H to I. Since those roots had equal rank 1, the new root, I, has rank 2.



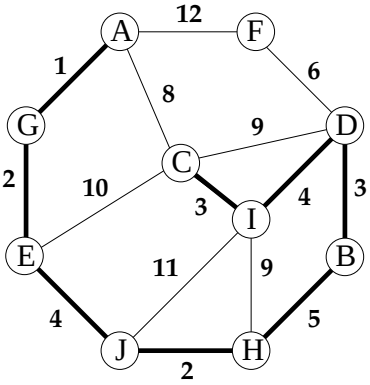


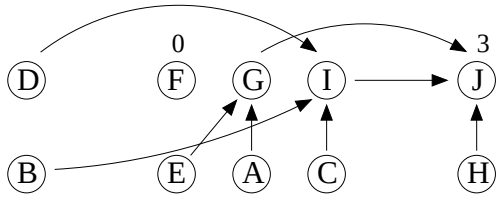
The next smallest edge is (B,H). When we compute find(B) = find(D) = I, path compression occurs. The new parent of B is I.



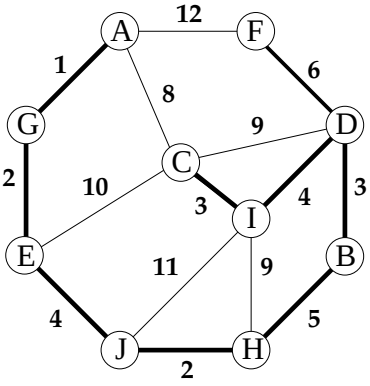


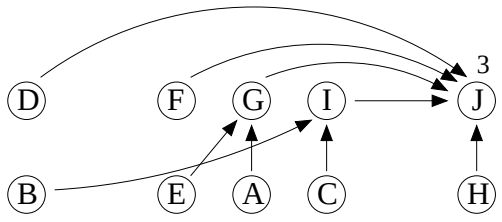
We are still working on edge (B,H); find(H) = J. We attach I to J, which then gets rank 3.





The next smallest edge is (D,F); find(F) = F. When we compute find(D) = J there is path compression; the new pointer of D is J. We can stop because we have selected $n - 1$ edges.





0

2. It is difficult to decide where to put the loop invariant of a **for** loop. However, a **for** loop can always be rewritten as a **while** loop.

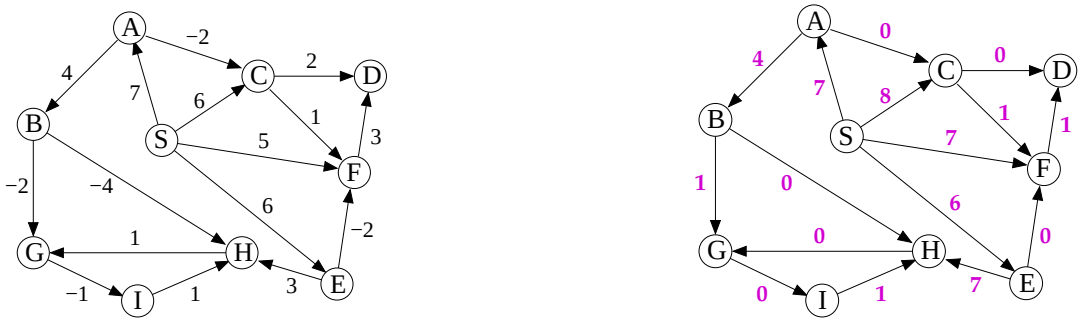
   Here is code for the Floyd-Warshall algorithm.

```
void main()
 {
  for(int u = 1; u <= n; u++)
   for(int v = 1; v <= n; v++)
    cost[u,v] = w(u,v);
  for(int u = 1; u <= n; u++)
   for(int v = 1; v <= n; v++)
    if(v != u) back[u,v] = u; // back[u,u] is undefined
  int j = 0;
  // Loop Invariant
  while(j < n)
   {
    // Loop Invariant
    for(int i = 1; i <= n; i++)
     for(int k = 1; k <= n; i++)
      if relax(cost[i,k],cost[i,j],cost[j,k])
       back[i,k] = back[j,k];
    j++;
    // Loop Invariant
   }
  // Loop Invariant
  return 1;
 }
```

```
bool relax(int&y,int x, int z)
 {
  if( x+z < y)
   {
    y = x+z;
    return true;
   }
  else return false;
 }
```

   Loop invariant: For any nodes $i \neq k$, $cost(i,k]$ is the minimum length of a path from i to k which does not have an interior node of index less than j, and $back[i,k]$ is the next-to-the last node on such a path.

3. Let G be the directed graph given on page 120 of your textbook. Work the first step of Johnson's algorithm for the all-pairs shortest path problem on G, *i.e.* adjust the weights so that there are no negative weight edges. Show the adjusted weights in part (b) of the figure below.



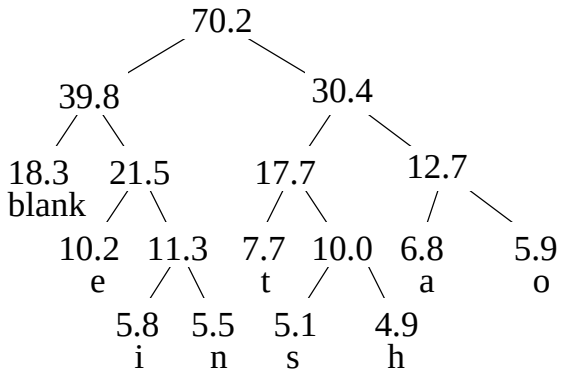
(a) (b)

The values of h are as follows:

S	A	B	C	D	E	F	G	H	I
0	0	0	−2	0	0	−2	−3	−4	−4

4. Work Problem 5.18 on page 150 of your textbook, but where we only use the first column of the table. The resulting Huffman code will only encode blank space and the letters e,t,a,o,i,n,s,h.

Here is a Huffman tree and the corresponding binary code.



blank	00
e	010
t	100
a	110
o	111
i	0110
n	0111
s	1010
h	1011

5. It is well known that the Knapsack problem, explained in Section 6.4 of the textbook, is $\mathcal{NP}$ hard. The Knapsack problem, given in Section 6.4 of your textbook is this: given a set of n items, each of which has a weight and a value, and given a number W (the maximum weight you can carry in your knapsack) find the most valuable set of items whose weight does not exceed W.

It is well known that the Knapsack problem is $\mathcal{NP}$ hard, which implies that there is no known polynomial time algorithm for that problem. However, there are several variations of the Knapsack problem that can be solved in polynomial time. Here are two of these.

(a) **Pseudo-Polynomial.** All weights are integers in the range $0..N$. There is a dynamic program given in the textbook which solves that variation of the knapsack problem in $O(nN)$ time. This is not always polynomial in n, since N might not be polynomial in n, but it is polynomial in n if N is polynomial in n.

(b) **Powers of 2.** All weights are of the form 2^i where i is integer. There is a polynomial time algorithm for this case. To simplify the problem, we can insist that W also be a power of 2, although that restriction simplifies the problem only slightly.

Suppose $W = 20$ and the weights and values of the items are as given in the table below. Find the optimal set of items by walking through the pseudo-polymial algorithm. Show your work.

weight	value
3	2
5	6
10	7
13	10

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	2	0			2																	
5	6	0			2		6			8												
10	7	0			2		6			8		7			9		13			15		
13	10	0			2		6			8		7			10		13	12		16		

16 is the largest value of any set of items whose weight does not exceed 20: the weights of the items are 5 and 13.