Lecture #9:

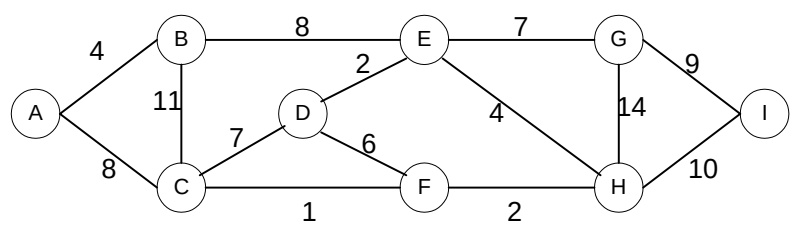
0.0.1 Graph Algorithms: Minimum Spanning Tree (Chapter 24)

Problem 1 *Given a connected undirected simple graph $G = [V, E]$, and a function $w : E \longrightarrow R$, (i.e. a weight $w(u, v)$ for each edge $(u, v) \in E$, find a spanning tree T^* which minimizes $w(T) = \sum w(u, v)$ where the sum is over the edges in T. This tree represents the minimum weight connected subgraph in G.*

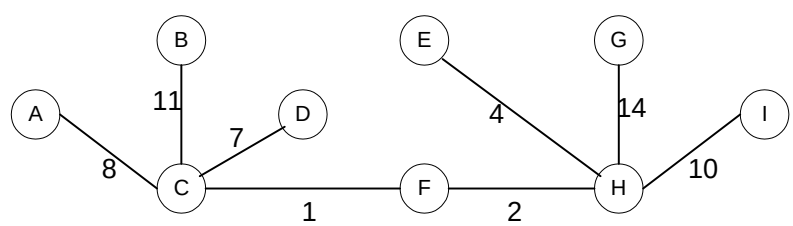
First recall some properties of a spanning tree:

- Any two nodes in a tree are connected by a unique path
- If any edge of a tree is removed, then the resulting graph is a pair of trees that are not connected.
- $|E| = |V| - 1$
- If an edge not in the tree is added to it, we get precisely one cycle.

Consider the graph below:

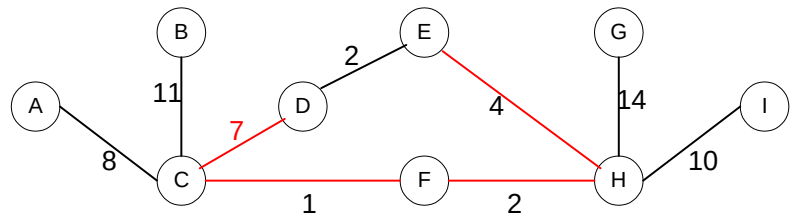


The following is a spanning tree (not one of minimum cost):

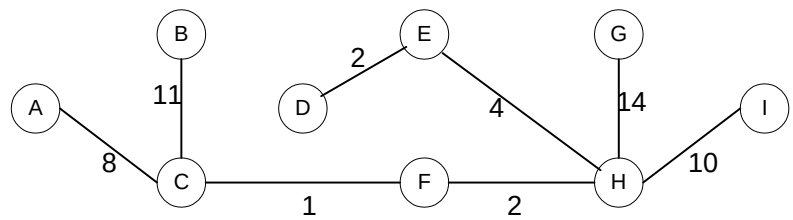


If we add an edge (D, E) not in the above tree, you would get some thing

like:



The red edges indicate the cycle formed by adding the new edge to the tree. If we now delete the edge (C, D) whose cost is the highest among the edges in the cycle, we get an "improved" spanning tree that looks like:



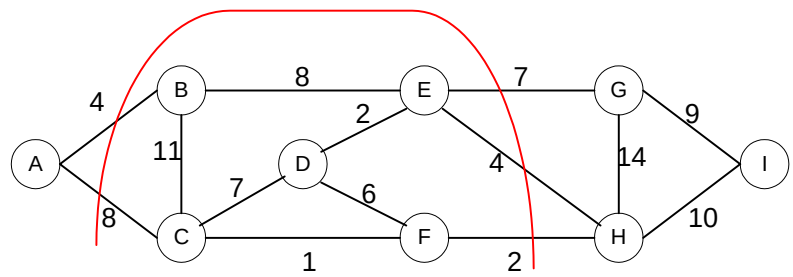
By doing this repeatedly until no such exchange is possible, yields an algorithm for finding the minimum spanning tree. There are two things with this statement that need proving/checking: (i)that the final result (the one where no such exchange improves the solution) is optimal; (ii)the number of such exchanges required. Algorithms of this sort which "move" from one feasible solution to an improved feasible solution are called *improvement type algorithms.* In this particular case, we can show that an edge that is "removed" is never again part of the subsequent spanning trees. This means that the number of such exchanges is no more than $|E| - |V| + 1$. But there is work to be done in each time in identifying the cycle and checking if the exchange improves the solution. Faster algorithms are available for this problem all of which are greedy methods. There are several of these and we present some of them.

0.0.2 General Greedy Method:

This process consists of coloring edges all of which are initially uncolored. We use two colors – blue (to indicate that this edge is included in the final tree) and *red* (to indicate that this edge is not included in the final tree). To color the edges we maintain the following *color invariant:*

There is a minimum spanning tree that includes all blue edges and none of the *red* edges. At the end, the blue edges form the required spanning tree. To formulate the coloring rules, we need some definition:

Definition 2 *A **cut** in a graph $G = [V, E]$ is a partition of V into two parts, X and $\overline{X} = V - X$. An edge **crosses** the above cut if one end is in X and the other is in $\overline{X}$. The remaining edges "respect" the cut.*



is the cut corresponding to $X = \{A, G, H, I\}$ and $\overline{X} = \{B, C, D, E, F\}$. Edges $(A, B), (A, C), (E, G), (E, H), (F, H)$ cross the cut and others "respect" the cut.

All Greedy methods use two coloring rules:

Blue Rule: Select a cut that blue edges *respect* (do not cross). Among uncolored edges crossing the cut, select one of minimum cost and color it blue.

Red Rule: Select a simple cycle containing no red edges. Among uncolored edges on the cycle, select one of maximum cost and color it red.

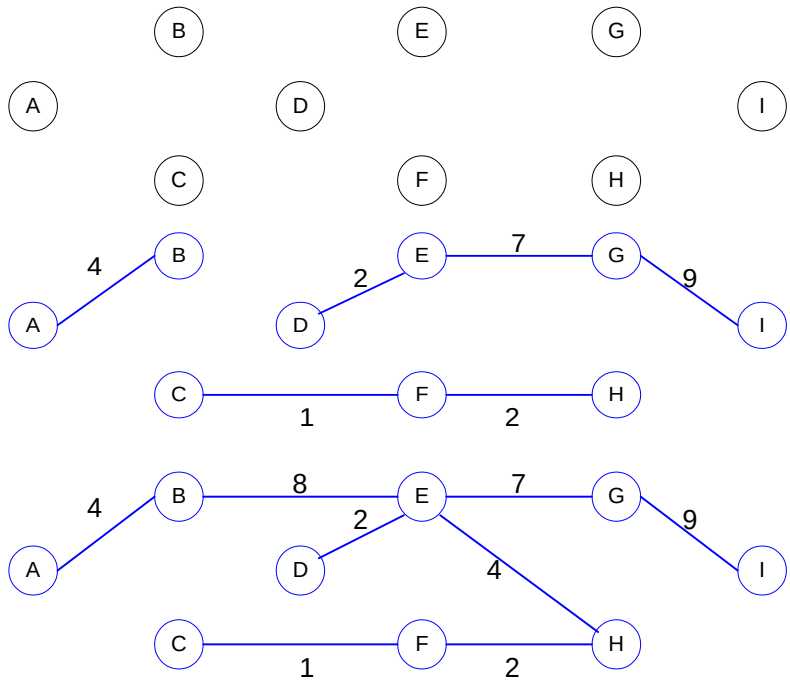
We are free to apply either rule at any time, in arbitrary order, until all edges are colored.

BORUVKA's Algorithm (1926):

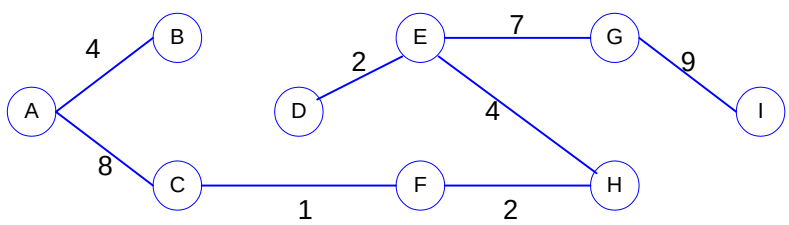
Begin with n blue trees each consisting of a node in G.

Coloring Step: For every blue tree T, select a minimum cost edge incident from some node of T to some node not in T. Color all selected edges blue. The

evolution of this algorithm on the above graph is shown below:



An alternate solution for the last step is given by:

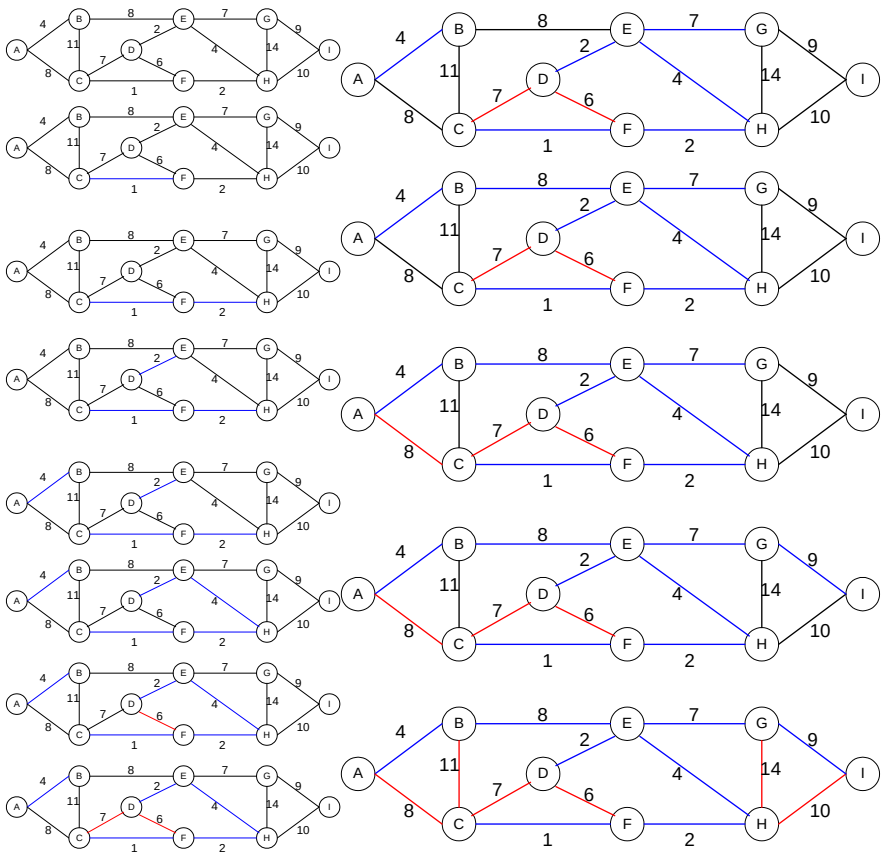


Kruskal's Algorithms: (1956)

Algorithm A: Apply the coloring step in nondecreasing order of costs.

Coloring Step: Begin with n trees each consisting of a node. If the current edge e has both ends in the same tree, color it *red*; otherwise color it blue.

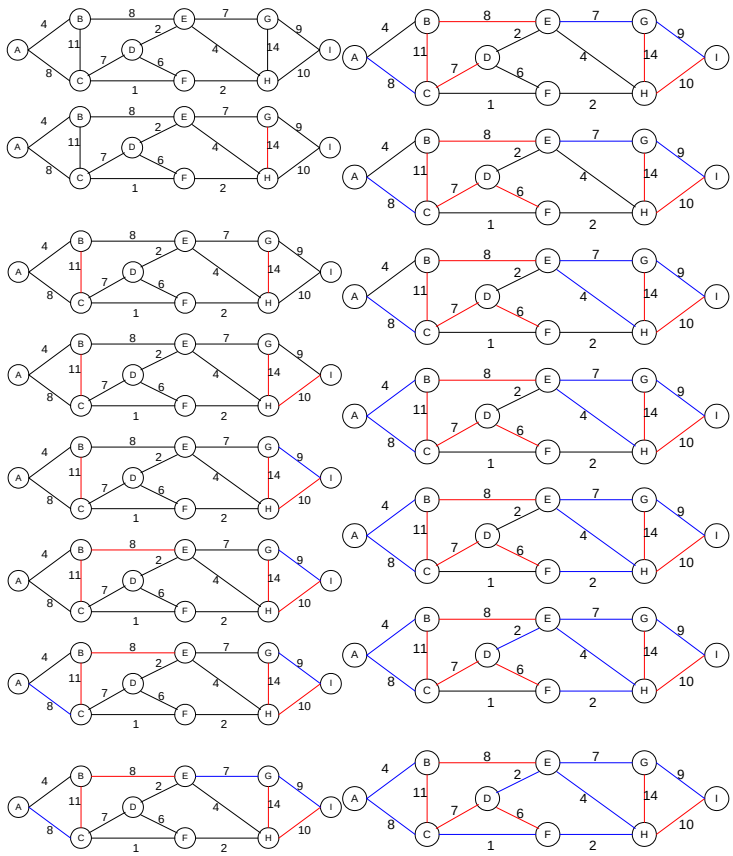
The evolution of this algorithm on the above graph is shown below:



Algorithm B: Apply the coloring step in nonincreasing order of costs.

Coloring Step: Begin with the original graph with all edges present. If the removal of the current edge would disconnect the graph, color it blue; else color it *red*.

The evolution of this algorithm on the above graph is shown below:



Please note that this is one of two possible cases since there was a tie at one point in the algorithm.

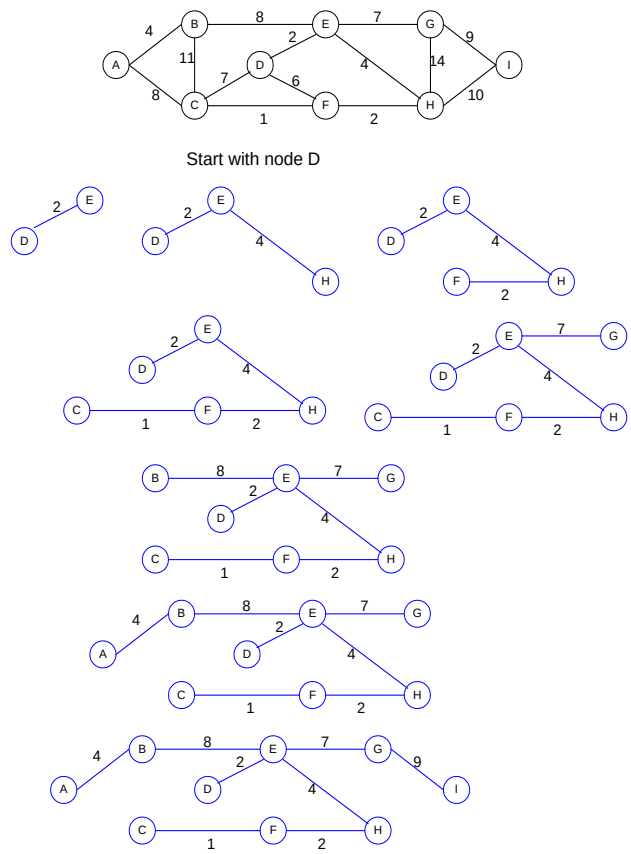
Jarnik (1930)-Prim (1957)-Dijkstra (1959) Algorithm:

This is commonly known as Prim's algorithm.

The algorithm starts with an arbitrarily selected node of the graph say s. It repeatedly uses the following coloring step:

Coloring Step: Let T be the blue tree containing s. In the beginning, T has only the node s. Select a minimum cost edge one end of which is in T and the other not in T and color it blue.

The evolution of this algorithm on the above graph is shown below:



0.0.3 Proofs of validity:

We have already defined cuts, "*crossing a cut*" or "*respecting*" a cut. An edge is a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut. There may be ties in selecting this minimum cost edge. Given a set A of edges which is contained in some minimum cost spanning tree, an edge (u, v) is a "*safe edge*" for A if $A \cup (u, v)$ is also in some minimum cost spanning tree. All these algorithms (with the exception of the improvement algorithm) add safe edges each time starting with the empty set. This is what needs proving in each algorithm.

Theorem 3 *Let A be a subset of some minimum cost spanning tree edges in the graph $G = [V, E]$. Let $[S, V - S]$ be any cut that respects A and let (u, v) be a light edge crossing the above cut. Then, edge (u, v) is safe for A.*

Proof. See page 501-502 in the book. ■

Corollary 4 *Let A be a subset of some minimum cost spanning tree edges in the graph $G = [V, E]$ and let C be a connected component (tree) in the forest $G_A = [V, A]$. If (u, v) is a light edge connecting C with some other component of G_A, then it is safe for A.*

0.0.4 Implementation:

Kruskal Algorithm A:

Here, the main question is: Does the current edge e have both ends in the same tree? For this purpose, we use the **UNION-FIND** data structure (found in Chapter 22 of your book)

This data structure also called **disjoint-set data structure** maintains a collection of disjoint dynamic sets. Each set in the collection is identified by a representative, which is usually a member of the set. The main operations are:

MAKE-SET(x): creates a new set whose only member (and thus its representative) is pointed to by x. For this x must not be already in any existing set.

UNION(x, y): unites sets containing x and y say S_x and S_y, into a new set that is the union of these two sets. It is assumed that these were disjoint before. The representative of this new set is usually chosen from among the representatives of the sets S_x and S_y. After doing this we destroy the sets S_x and S_y and now we have only the set $S_x \cup S_y$ in the collection.

FIND-SET(x): returns a pointer to the representative of the (unique) set containing x.

Analysis of this data structure is based on two parameters: the number, n, of MAKE-SET operations and the total number, m, of MAKE-SET, UNION, and FIND-SET operations. There can be at most $n-1$ UNION operations since there are only n elements and these sets are disjoint. Also, $m \geq n$.

Applications:

CONNECTED-COMPONENTS(G):
for each vertex $v \in V[G]$
    do MAKE-SET(v)
for each edge $(u, v) \in E[G]$
    do if FIND-SET(u) $\neq$ FIND-SET(v)
        then UNION(u, v)

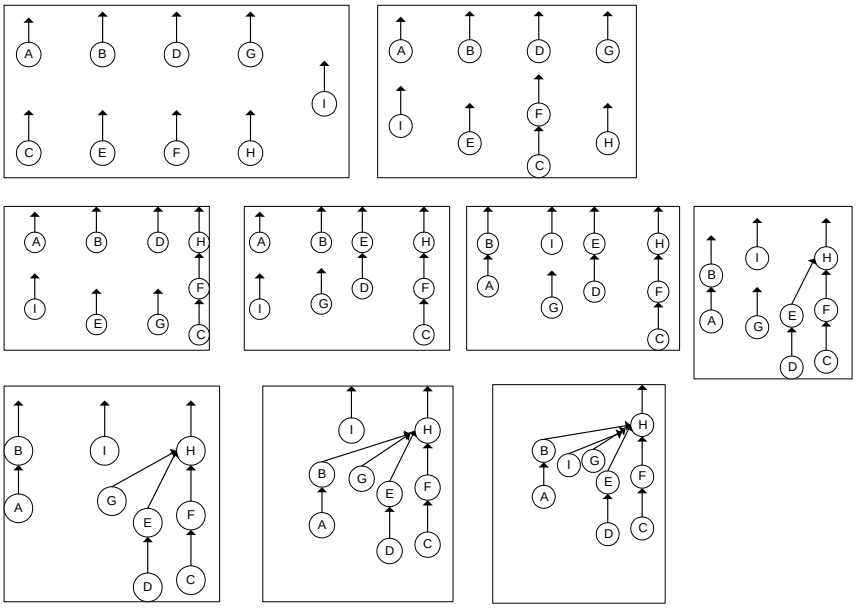
SAME-COMPONENT(u, v)
if FIND-SET(u) = FIND-SET(v)
    then return TRUE
    else return FALSE

Kruskal Algorithm A:

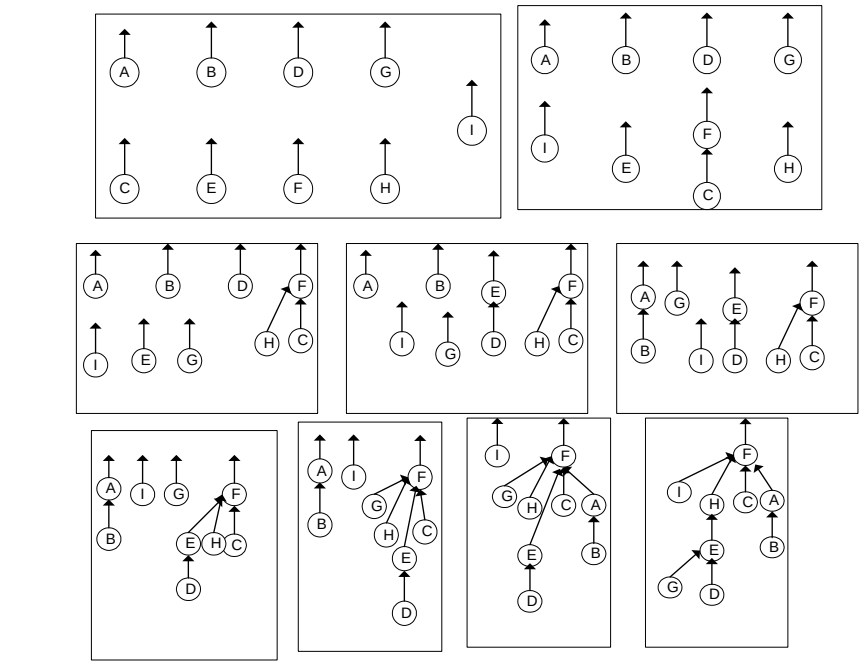
MST-KRUSKAL(G, w)
$A \longleftarrow \phi$

for each vertex $v \in V[G]$
  do MAKE-SET(v)
sort the edges of E by nondecreasing weight w
for each edge $(u, v) \in E$, in order of nondecreasing weight
  do if FIND-SET(u) $\neq$ FIND-SET(v)
    then $A \longleftarrow A \cup \{(u, v)\}$
      UNION(u, v)
return A

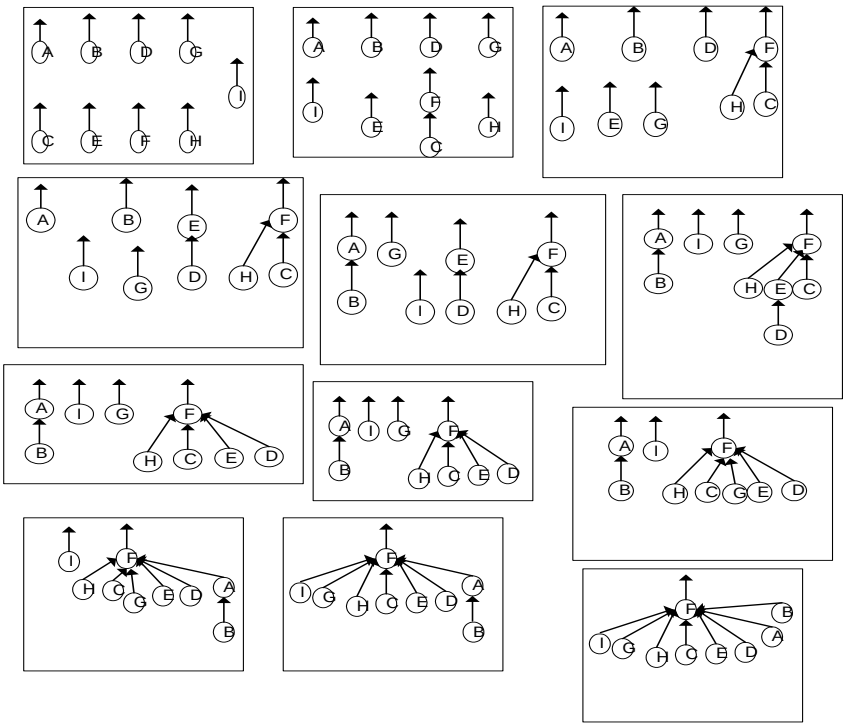
The simplest implementation of UNION-FIND is shown in the diagrams below corresponding to UNION steps.



In this simple implementation, the tree may have a large depth. This is not good for FIND-SET operations. Two strategies: always join the smaller tree to the larger one. For this we need to keep either the information on depth of the trees or the number of nodes. Usually, the latter is kept since it is easier to do. The second strategy is known as PATH-COMPRESSION. Each time we do a FIND-SET operation, we traverse the tree twice once to find the root and the second time to make pointers of all nodes on the way to point directly to the root. This keeps the tree shallow. These are shown below for our case:



==========



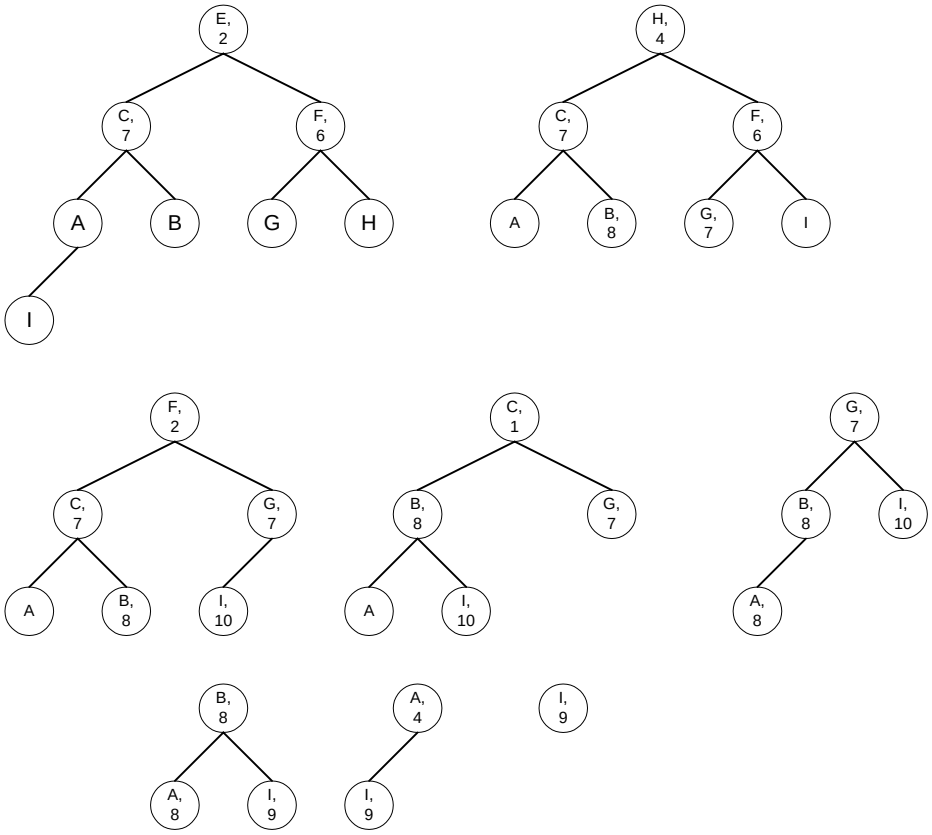
The second one is with path compression.

Prim Algorithm:

The key to implementing this algorithm is to find the minimum cost edge connecting the current tree to a node outside the tree efficiently. All nodes not in the tree reside in a priority queue Q based on a *key* field. For each node v, $key[v]$ is the minimum weight of any edge connecting v to a node in the tree; $key[v] = \infty$, if no such edge exists. The field $\pi(v)$ names the parent of v in the tree. The set A of edges in the tree is kept implicitly by the collection of $\{(v, \pi(v)\}; v \in V - \{r\} - Q$ where r is the starting node and hence the root of the tree. At termination, Q is empty and A gives the minimum spanning tree.

```
MST-PRIM(G, w, r)
Q ⟵ V[G]
for each u ∈ Q
    do key[u] ⟵ ∞
key[r] ⟵ 0
π[r] ⟵NIL
while Q ≠ φ
    do u ⟵EXTRACT-MIN(Q)
        for each v ∈ Adj[u]
            do if v ∈ Q and w(u, v) < key[v]
                then π[v] ⟵ u
                    key[v] ⟵ w(u, v)
```

The priority queue as it occurs in our example is shown below:



0.0.5 Analysis:

Kruskal Algorithm A:

Time to sort edges by weight takes $O(|E| \lg |E|)$. Initialization takes $O(|V|)$. The time required for UNION and FIND-SET operations involved with both strategies – joining the smaller tree to the larger one and path compression is given by $O(|E| \, \alpha(|E|, |V|)$ where the function $\alpha(m, n)$ is very slowly growing function and can for all practical purposes be taken to be a constant. Thus, the overall complexity is $O(|E| \lg |E|) = O(|E| \lg |V|)$.

Prim Algorithm:

The performance of this algorithm depends on how the priority queue is implemented. We assume that binary heap is used. Initialization is to use BUILD-HEAP and takes $O\,|V|)$ time. The loop is performed $|V|$ times and each EXTRACT-MIN takes $O(\lg(|V|))$ time so that the total effort so far is $O(|V| \lg(|V|))$. Test for membership in Q is implemented in constant time. DECREASE-KEY operation takes $O(\lg |V|)$ time each time (once for each edge) it is called. Thus, the overall time for Prim's algorithm with this implementation is $O(|V| \lg(|V|) + |E| \lg(|V|)) = O(|E| \lg(|V|))$. This is the same as in Kruskal's algorithm. If we use other types of heaps, Prim's algorithm's bound can be

improved to $O(|E| + |V| \lg(|V|))$.

This completes the discussion on this problem.