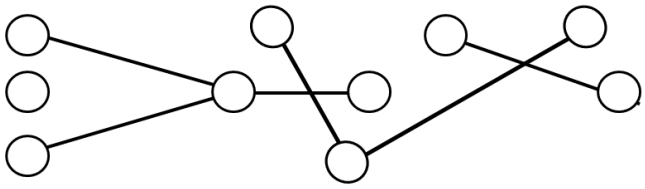
Chapter 3. Section 3

Growing Forests

If a graph is not connected then its vertices may be partitioned into **connected components**: maximally connected subgraphs. A graph whose connected components are all trees is called a **forest**.



A forest consisting of four components (which are trees)

Note: (1) some or all of the components may be single vertices: these are perfectly valid trees.

(2) A single tree is a forest (which contains one tree).

A subgraph of a graph G which contains every vertex of G and is a forest is called a **spanning forest**.

Maximal_Subtree and all its variants grow a spanning tree by *extending a single component*. Using forests gives us a different approach, as in the following MWST algorithm due to Kruskal (and others):

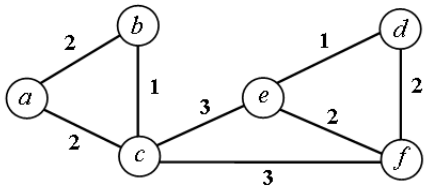
Algorithm **Kruskal**
Input: connected graph $G = (V(G), E(G))$ with edge weights in $\mathbb{R}$
Output: a minimum-weight spanning tree T *of* G

$F := (V(G), \emptyset)$ *# F is a spanning forest all of whose trees are single vertices*
while $X = \{e \in E(G)$ such that $F \cup e$ has no cycles $\} \neq \emptyset$ **do**
choose $e \in X$ with $w(e)$ as small as possible
$F := F \cup e$ *# F has one more edge and one fewer trees*
return(F)
end

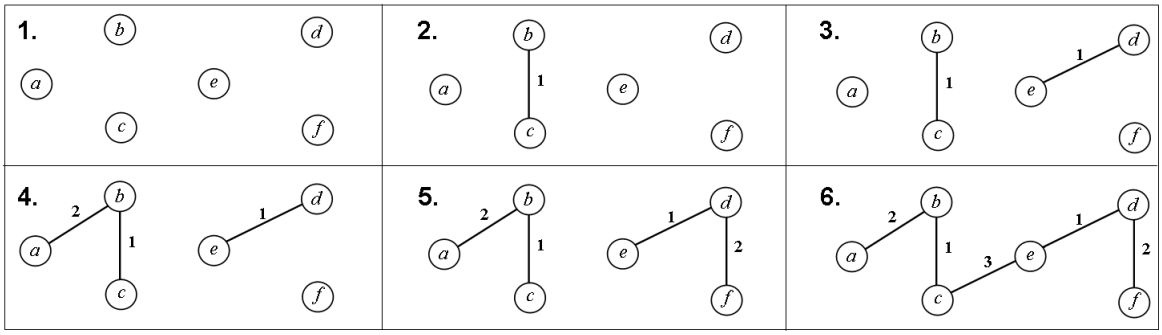
Example:

$V = \{a, b, c, d, e, f\}$,
$E = \{(ab, 2), (ac, 2), (bc, 1), (ce, 3), (cf, 3), (de, 1), (df, 2), (ef, 2)\}$
Note: pairs $= (e, w(e))$

If the graph is specified as lists of vertices and edges then a drawing definitely helps to demonstrate the algorithm!



The algorithm runs as follows. Note that we must show all vertices at each step because they are all trees in our forest F, even if they have no edges.



Correctness: Do we get a spanning tree? We must check that Kruskal's output F is (a) spanning, (b) without cycles, and (c) connected.

Spanning: guaranteed since F is initialised to contain all vertices.

No cycles: guaranteed by definition of the sets X.

Connected: We must check that for any vertices x and y there is walk between them in F. Since G is connected there is certainly a walk W from x to y in G. Now on termination of Kruskal, every edge $e \notin F$ must create a cycle when added to F, otherwise we could create $X \neq \emptyset$ and continue to run Kruskal. Pick an edge uv of W. So $F \cup uv$ contains a cycle, C. Then we can walk from u to v in F by following C. Since we can do this for any edge of W, we see that there must be a walk from x to y, as required.

Our proof of connectedness is not very formal! What do we mean *exactly* by 'following C', for example. You can see how this kind of argument is made more precise in Question 1 of Workset 3.

The fact that the output of Kruskal is not only a spanning tree but an *minimum-weight* spanning tree follows similarly to our first MWST algorithm (see Chapter 3, Section 1).

Complexity: Kruskal, as specified above, is very slow! Let $m = |E(G)|$ and $n = |V(G)|$. At each iteration of the **while** loop we need to build set X. This involves checking every $e \in E(G)$ for membership of X. Each check may involve looking through all or most of the current forest F for cycles containing e. This is 1 check when F has 1 edge, 2 when F has 2 edges, ..., $n-2$ check in order to add the final edge. Now $1 + 2 + \ldots + n - 2 = O(n^2)$ so the **while** loop test takes $O(mn^2)$ steps. The loop will iterate $O(n)$ times to build the complete spanning tree. So total time is $O(mn^2 \times n) = O(mn^3)$ steps.

A running time of $O(mn^3)$ is bad compared to Prim: $O(n^2 + m)$ (see Week 3, Lecture 2). However we can use various tricks and shortcuts to get a version of Kruskal that is as competitive with Prim. For a start, here is a way of growing minimum-weight spanning forests into spanning trees which is an example of an important speed-up technique in algorithm design. We choose a smallest 'joining' edge for each component tree in our forest F. This will then combine trees in pairs, or even bigger clusters. So we at least halve the number of trees in F at each step. Halving can happen at most $\log_2 |V|$ times, which is a number much smaller than $|V|$. Here is the pseudocode; the most important thing is to appreciate why the resulting complexity has a $\log_2$ in it: replacing an $O(n)$ with an $O(\log n)$ in running time is very often a realistic goal in algorithm design

Algorithm **Turbo Kruskal**
Input: connected graph $G = (V(G), E(G))$ with edge weights in $\mathbb{R}$
Output: a minimum-weight spanning tree T *of* G

```
    F := (V(G), ∅)                  # F is a spanning forest all of whose trees are single vertices
    C = {{v_1}, {v_2}, ..., {v_n}};   # list of components of F
    while |C| > 1 do
        for C_i ∈ C do min(C_i) := ∞; join(C_i) := ∅     # initialise search for min joins
(*)         for xy ∈ E(G) do
                find i, j such that xy joins C_i to C_j;
        if w(xy) < min(C_i) then min(C_i) := w(xy); join(C_i) := xy
        if w(xy) < min(C_j) then min(C_j) := w(xy); join(C_j) := xy
    for each C_i do add join(C_i) to F
    update list C of components
    return(F)
end
```

Efficiency: Denote $|E(G)|$ by m and $|V(G)|$ by n. At every loop iteration we pair up components and this divides $|C|$ by 2. Initially, $|C| = n$ and we can divide n by 2 about $\log_2 n$ times. So there are $O(\log n)$ iterations of the **while** loop (note we ignore the log base since we can change base by multiplying by a constant, and big-Oh notation ignores constant multiples). Now each iteration considers all or most of the edges, at the point marked $*$, so that is $O(m)$. So total running time is $O(m \log n)$.

A time of $O(m \log n)$ is still not as good as $O(n^2 + m)$ Prim. What is more, similar tricks can reduce Prim's time to $O(n \log n + m)$. However, for sparse graphs, meaning not so many edges, say $m = O(n)$ rather than $m = O(n^2)$, Kruskal can be competitive and require less book-keeping (remember those big tables of vertex labels for Prim!) You might think $O(n)$ is not many edges—for example trees have $O(n)$ edges—but remember big-Oh ignores constants. A very important class of graphs is that of **regular graphs** in which all vertices have equal degree. If we fix a value of k then an n-vertex k-regular graph, i.e., regular of degree k, has $\frac{k}{2}n$ edges (by the Handshaking Lemma)) and this is $O(n)$.

Correctness: informally, we cannot create cycles because this can only happen if C_1 joins to C_2, C_2 to C_3 and C_3 to C_1. This is impossible, because once we find a minimum edge joining C_1 to C_2 and another minimum edge joining C_2 to C_3 we have already considered all three components: any edge which joins C_3 to C_1 will be found to have at least the same cost as edges joining C_1 to C_2 and C_2 to C_3, and will therefore be ignored.

Proof of connectedness now follows the argument for Kruskal above.