MTH6105 Algorithmic Graph Theory

Week 3. Lecture 1
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: a single vertex is a perfectly valid tree.
(2) A single tree is a forest (which contains one tree).

A subtree 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 $G$

$$ F := (V(G), \emptyset) \quad \# \text{ $F$ is a forest all of whose trees are single vertices} $$

while $X = \{ e \in E(G) \text{ such that } F \cup e \text{ has no cycles } \} \neq \emptyset$

choose $e \in X$ with $w(e)$ as small as possible

$F := F \cup e \quad \# \text{ $F$ has one more edge and one fewer trees}$

return $(F)$

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 a walk \( W \) from \( x \) to \( y \) 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 Coursework 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 Week 3, Lecture 1).

Complexity: Kruskal, as specified above, is very slow! Let \( m = \left| E(G) \right| \) and \( n = \left| V(G) \right| \). 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, \ldots, \( n - 2 \) check in order to add the final edge. Now \( 1 + 2 + \ldots + n - 2 = O(n^2) \) so the while loop test takes \( O(nm^2) \) 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. This is bad compared to Prim: \( O(n^2 + m) \) (see Week 3, Lecture 2).

However: (1) we can use various tricks and shortcuts to get a version of Kruskal that is as efficient as Prim (we see an example below), and

(2) suppose we call sets of edges which form forests ‘independent’. Such independent sets in graphs share some key properties with linearly independent sets of vectors in a vector space. This leads to a generalised theory of independence called Matroid Theory (which you don’t need to know about for this module but is a big deal, from a theory point of view. You can read about matroid theory in the recommended book Papadimitriou and Steiglitz).

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 \( \log_2 \left| V \right| \) times, which is a number much smaller than \( \left| V \right| \). Here is detailed pseudocode—you do not need to know these details but you do need to know why the resulting complexity has a \( \log_2 \) in it!

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 \( G \)

\[
F := (V(G), \emptyset) \quad \text{# } F \text{ is a forest all of whose trees are single vertices}
\]

\[
C = \left\{ \{v_1\}, \{v_2\}, \ldots, \{v_n\} \right\}; \quad \text{# list of components of } F
\]

\[
\text{while } |C| > 1 \text{ do}
\]

\[
\text{for } C_i \in C \text{ do } \min(C_i) := \infty; \text{join}(C_i) := \emptyset \quad \text{# initialise search for min joins}
\]

\[
(*) \quad \text{for } xy \in E(G) \text{ do}
\]

\[
\text{find } i, j \text{ such that } xy \text{ joins } C_i \text{ to } C_j;
\]

\[
\text{if } w(xy) < \min(C_i) \text{ then } \min(C_i) := w(xy); \text{join}(C_i) := xy
\]

\[
\text{if } w(xy) < \min(C_j) \text{ then } \min(C_j) := w(xy); \text{join}(C_j) := xy
\]

\[
\text{for } \text{ each } C_i \text{ do } \text{add } \text{join}(C_i) \text{ to } F
\]

\[
\text{update list } C \text{ of components}
\]

\[
\text{return}(F)
\]

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 $\ast$, so that is $O(m)$. So total running time is $O(m \log n)$.

**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 $C_3$ 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 follows the argument for Kruskal above.