Chapter 2. Section 3

Analysing Maximal_Subtree

It will be convenient to have our algorithm in front of us:

Algorithm **Maximal_Subtree**
Input: graph $G = (V, E)$, vertex $v \in V(G)$
Output: subgraph $T = (V(T), E(T))$ of G which is a maximal subtree having $v \in V(T)$
1. $V(T) := \{v\}$, $E(T) := \emptyset$ # initialise the subtree to be the single vertex v and no edges
2. **while** some edge $xy \in E(G)$ leaves the tree, i.e. has $x \in V(T)$, $y \notin V(T)$ **do**
3. $V(T) := V(T) \cup \{y\}$ # add the new vertex v to T
4. $E(T) := E(T) \cup \{xy\}$ # add the new edge xy to T

return(T)
end

Correctness

We must convince ourselves of two things: the output T from the algorithm must be

Maximal: the loop at (2) terminates when T cannot be extended; this is the definition of maximal, so maximality is guaranteed.

A subtree: this has two components:

1. T must be a subgraph: this is clear since we always extend T by adding edges and their endpoints;
2. T must be a tree. This itself has two components
   (a) T must be connected;
   (b) T must have no cycles.

So there are two claims remaining to prove:

T **is connected:** Initially T is a single vertex which is trivially connected. It will be enough to show that steps (3) and (4) of the algorithm preserve connectedness. So we assume that T is a tree and show, given any two vertices of $T + xy$, say u and v, that $T + xy$ contains a walk joining u and v, where $T + xy$ denotes the extension of T by the new vertex y and the new edge xy. There are three possibilities:

I: u and v are both vertices of T. Since we are assuming that T is connected this follows by definition;

II: u is a vertex of T but v is not. Then $v = y$ since this is the only vertex of $T + xy$ which is not in T. Now since T is connected there is a walk, say W, from u to x. Now W, xy is the required walk from u to v.

III: neither u nor v are vertices of T. Then we must have $u = v = y$ and the trivial walk joins u to v.

In each case we find extending the tree preserves connectedness, so the output tree is connected.

T **has no cycles:** To show that $T + xy$ does not have a cycle we argue by contradiction. Suppose that $T + xy$ does have a cycle - call it C. Since T is a tree, C is not entirely contained in T, so xy must appear in C. But every vertex in C must have degree 2, and the degree of y in $T + xy$ is 1. So we have contradicted the existence of the cycle C.

Finally we have proved that the output of Maximal_Subtree is a maximal subtree. □

Complexity

How many steps does Maximal_Subtree take to construct a maximal subtree? In this module we will always consider the 'worst case'—that is, we consider the greatest number of steps that the algorithm might take. Sometimes it will take fewer steps: if we execute Maximal_Subtree starting at an isolated vertex then it will terminate after 1 step: no edges leave the initial tree. But what is the *longest* we might have to wait to receive the output? This is called **worst-case analysis**.

Steps (1), (3) and (4) of the algorithm take 1 step each. So the complexity of Maximal_Subtree depends on the number of iterations made by the **while** loop at step (2) and how long its leaving edge test takes.

As with our complexity analyses in Week 1, our analysis rests on a basic result:

Lemma: every tree T on n vertices has $n-1$ edges.

Proof: By induction on n.

BASE CASE: if $n = 1$ then T has zero edges which is indeed $n-1$;

HYPOTHESIS: suppose the result is true for trees on $n = K$ vertices.

INDUCTIVE STEP: suppose T has $K+1$ vertices.

> **Claim:** a finite tree T has a vertex v of degree 1.
> **Proof of Claim:** let P be a maximum length path in T ending at vertex, say, x. Then the degree of x in T, $d_T(x) = 1$. For, if not then there is an edge in $T - P$ incident with x. Either this edge joins x to a vertex of P, creating a cycle and contradicting the fact that T is a tree, or the edge may be added to P, contradicting the fact that P had maximum length. So $v = x$ is the vertex we require.

Now, let v have degree 1 in T. Then $T' = T - v$ is again a tree since deleting v and its incident edge from T cannot create a cycle or create a disconnected graph. Now T' has K vertices, so by hypothesis it has $K-1$ edges. But then $|E(T)| = |E(T')| + 1 = K = |V(T)| - 1$, as required. □

What does this mean for the algorithm? It means we can only iterate the **while** loop $|V(G)| - 1$ times, in the worse case. Each iteration takes 2 steps (for (3) and (4)) + the time for the leaving edge test.

A crude analysis now goes like this: the leaving edge test may have to look through all $|E(G)|$ edges. So the total run time for the algorithm is $(|V| - 1) \times (2 + |E|) = |V||E| + 2|V| - |E| - 2$ steps. We can write this as $O(|V||E|)$ steps by ignoring the lower value terms.

Can we reduce this by a more careful analysis? At the n-th iteration the tree T has n vertices and hence, by the Lemma, $n-1$ edges. We do not have to check edges in T to see if they are leaving edges. So the number of edges to check is

$\|E\|$	at the 1st iteration
$\|E\| - 1$	at the 2nd iteration
	...
$\|E\| - n + 1$	at the n-th iteration
	...
$\|E\| - \|V\| + 2$	at the $(\|V\| - 1)$-th iteration

What is the total of these tests? We have

$$\begin{aligned} |E| \times (|V| - 1) - 1 - 2 - \ldots - (|V| - 2) &= |E|(|V| - 1) - \frac{1}{2}(|V| - 1)(|V| - 2) \\ &= |E||V| - |E| - \frac{1}{2}|V|^2 + \frac{3}{2}|V| - 1 = O(|E||V| - |V|^2) \end{aligned}$$

Have we reduced on $O(|E||V|)$? It depends on $|E|$: if it is much bigger than $|V|$ then $O(|E||V| - |V|^2) = O(|E||V|)$ and we haven't gained anything. If $|E| = O(|V|)$ then we have reduced the run time to $O(1)$, a constant (reasonable enough—G is just a tree (plus a few edges) so it should be quick to find a subtree!