Chapter 2. Section 2

The Maximal_Subtree algorithm

Maximal vs Maximum

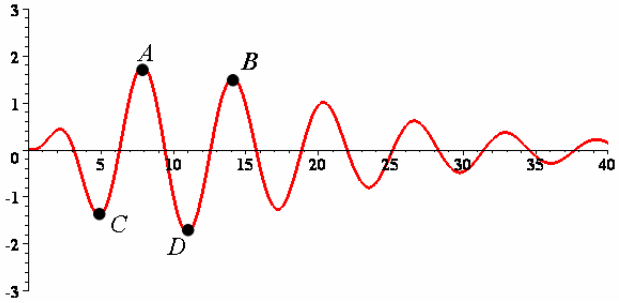
If G is a graph then a **subtree** of G is a subgraph of G which is a tree. Our aim is to find subtrees which are as large as possible. However, the easiest thing to achieve with an algorithm turns out to be: add edges to a subtree until this is no longer possible. This is *not the same thing!*

Key Distinction: a construction is **maximal** if it cannot be extended to make it bigger.

A construction is **maximum** if no bigger construct exists.

Note that maximum ⇒ maximal; hence the contrapositive: not maximal ⇒ not maximum.

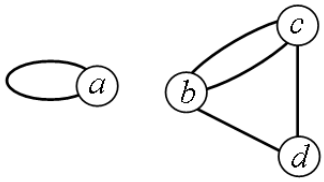
We can visualise this important distinction in terms of classical optimisation:



If we imagine trying to maximise the function whose curve is sketched here by 'hill climbing' then point A is maximal: we cannot extend our uphill walk at this point. It is also maximum — no higher point is reached by this function[1]. The point B is also maximal but not maximum. In classical terminology this is the difference between a local maximum and a global maximum.

The terms **minimal** and **minimum** are defined exactly analogously. The points C and D in the above sketch are both minimal but only D is minimum for this curve.

E.g. To return to graph theory, in the graph given in Chapter 2, Section 1:



$(\{b, c\}, \{bc\})$ is a subgraph which is a subtree. It is not maximal since it can be extended by adding vertex d and edge bd. The graph $(\{b, c, d\}, \{bc, bd\})$ is a subtree which is both maximal and maximum. The subgraph $(\{a\}, \emptyset)$ is a maximal subtree: the only edge at a is the self-loop, which cannot be added since this would create a cycle. This subtree is not maximum since we have seen that a maximum subtree of 3 vertices exists.

Growing Maximal Subtrees

Here is a very 'high-level' version of an algorithm which constructs a maximal subtree:

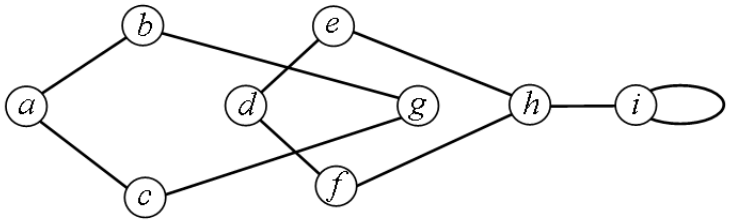
Algorithm **Maximal_Subtree**
    Start with a single vertex tree T
    while there an edge leaving T **do**
        add it to T

[1] the function is $y = \dfrac{x^3 \sin x}{e^{2\sqrt{x}}}$.

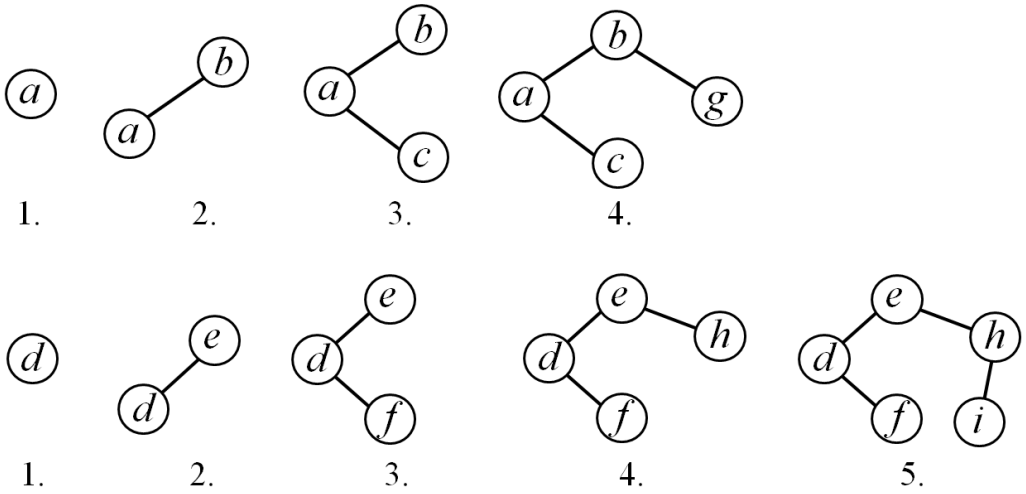
We need a more detailed ('lower level') specification to be sure we are clear what the algorithm does and to support a correctness proof and complexity analysis:

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

E.g. we show the execution of Maximal_Subtree on the following graph G:



Here is the execution of Maximal_Subtree(G, a) and Maximal_Subtree(G, d) i.e. starting with $v = a$ and then with $v = d$:



The subtree at $v = a$ is maximal but not maximum; the subtree at $v = d$ is maximal *and* maximum.

Note that we do not have to reproduce the whole graph over and over again: it is clear how the growing subtree 'emerges' inside the graph G. We do not have to respect the original drawing either, we are free to redraw our picture to keep things tidy and easy to look at.

We end with an example: provided with Maximal_Subtree we can test whether a graph is connected.

Algorithm **Is_Connected**
Input: graph $G = (V, E)$
Output: True if G is connected; False otherwise
1. v := some vertex of V # we need not specify how this choice is made
2. T := Maximal_Subtree(G,v)
3. **if** $|V(T)| = |V(G)|$ **then return**(True) **else return**(False)
end

For the above graph Is_Connected(G, v) will return False regardless of which vertex v we choose—we will never construct a tree containing $|V(G)|$ vertices.