Basic graph theory definitions for computer science

John MacCormick
Dickinson College

September 2014

Many concepts in computer science are related to graphs, so it's important to know some graph terminology. Figure 1 demonstrates the five most important concepts, which are:

- A *graph* is a collection of vertices with edges between some of the vertices. Sometimes we call this an *undirected graph*, because the edges have no direction.
- A *path* is a sequence of vertices with an edge between each pair of consecutive vertices.
- A *cycle* is a path that starts and ends at the same vertex.
- A *directed graph* is the same as a graph, but with directed edges.
- A *weighted graph* is the same as a graph, but each edge has a numerical weight.

Various obvious combinations of these definitions are possible, such as directed weighted graphs, directed cycles and directed paths. A graph is *connected* if there's a path between every pair of nodes.

We often use graphs, paths, and so on as inputs and outputs to computer programs. One easy way to deal with input and output is to use ASCII strings. Fortunately, it's easy to convert graphs, paths, and other graph-theoretoric concepts into ASCII strings. The exact conventions for doing so are not important, but the right column Figure 1 shows one possible way of doing this, for each of the five main object types.

Trees and rooted trees

Trees and rooted trees are special kinds of graphs that are often encountered in computer science. Their definitions are as follows:

- A *tree* is a connected graph with no cycles. Figure 2(a) gives an example.
- A *rooted tree* is a tree in which one of the nodes has been designated the *root* of the tree. Figure 2(b) gives an example. This is exactly the same tree as in (a), but now node `d` has been designated the root. Note that we can lay out a rooted tree in *levels*: the root is at level 0, its neighbors are at level 1, and so on. This also gives us a notion of parent and child nodes. The parent of a node is its neighbor in the level above; the children of a node are its neighbors in the level below. In Figure 2(b), for example, the parent of `k` is `c`, and the children of `g` are `a`, `b`, and `f`. Sometimes, the exact order of laying out the tree is important. In these cases, we can sort the children of a node according to some agreed ordering, from left to right. For example, in Figure 2(b), children are sorted lexigraphically (i.e. in dictionary order) from left

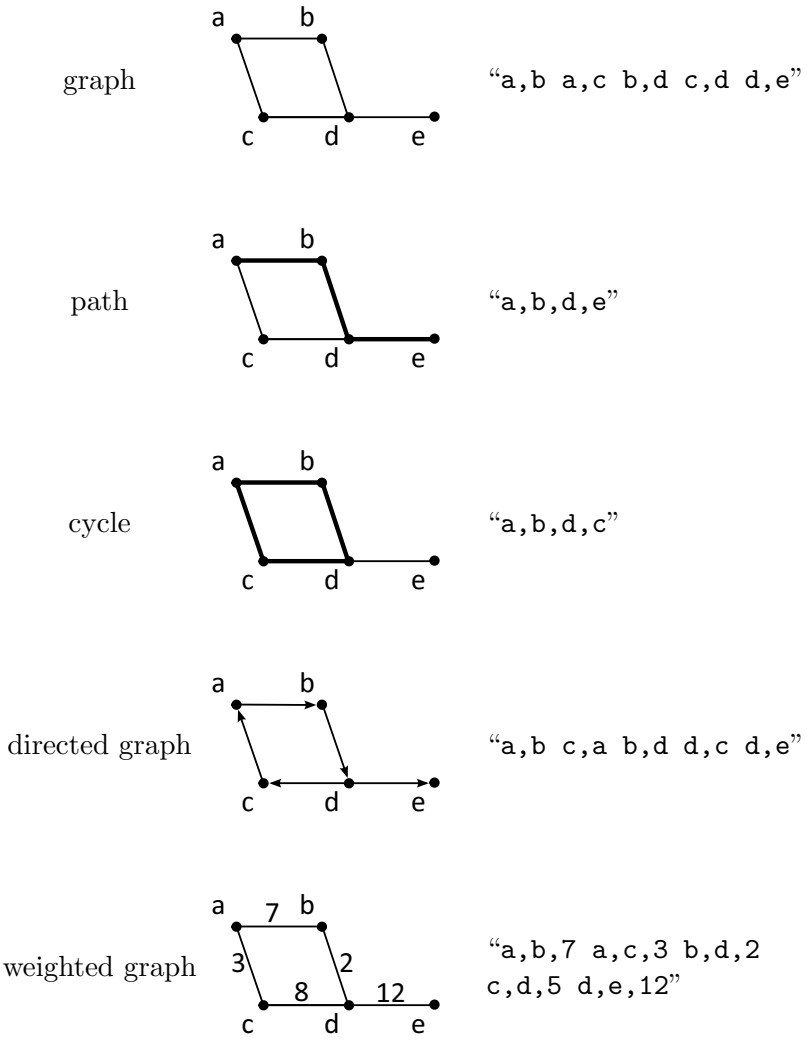

Figure 1: **Graph concepts.** The right column shows possible ASCII string representations of the corresponding examples.

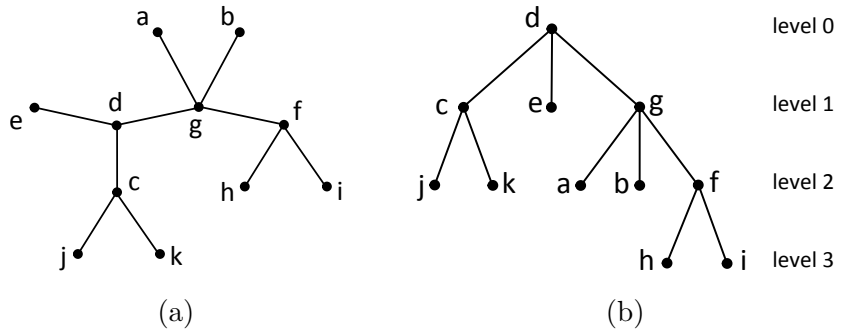

Figure 2: (a) A tree with 11 nodes. (b) The same tree, this time represented as a rooted tree with root node `d`.

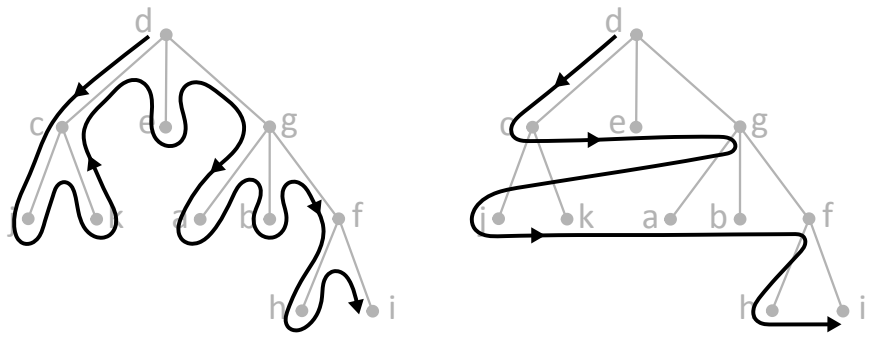

Figure 3: Depth-first (left) and breadth-first (right) order for visiting the nodes in a rooted tree.

to right. That's why the children of `g` are listed in the order `a`, `b`, `f`, for example. A node with no children is called a *leaf*. In Figure 2(b), the leaves are: `j`,`k`,`e`,`a`,`b`,`h`,`i`. You may have noticed that, in computer science, rooted trees are upside down. Whereas a real-world tree has its roots at the bottom and leaves at the top, computer science trees have the root at the top and leaves at the bottom. Get used to it!

In practice, rooted trees are so common in computer science that they are often simply called "trees". It's usually clear from the context whether or not a tree has a designated root. And we can easily convert a tree into a rooted tree by selecting any node as the root.

Often, we want to search a (rooted) tree to find a particular node. There are two common methods of searching trees: *depth-first* and *breadth-first*. These are both demonstrated in Figure 3. In a depth-first search, the children of a node N are always visited before moving on to the next sibling of node N. Breadth-first works the opposite way: the siblings of N are visited before its children. A simple way to think of breadth-first search is that we scan each level in turn, from left to right. In contrast, depth-first search always goes as deep as possible into the tree, keeping to the left. The search only moves up or to the right when everything below the current node has already been explored.