ICS 46 Spring 2022
Notes and Examples: Graph Connectedness

Background

When we learned previously about graph traversals, we discovered that there were some complications involved in traversing graphs that we didn't encounter when we were traversing trees. One of those complications arose when we tried to traverse a graph that lacked what you might generally call *connectedness*, which is to say that you may not necessarily be able to start at one vertex and reach every other vertex in the graph simply by following the available edges; some of them might be unreachable. This wasn't a problem that was particularly difficult to work around, but we did have to recognize it and allow for it in the design of our traversal algorithms.

It's important to note, also, that a lack of connectedness wasn't at all unrealistic; it's not just a theoretical problem. You could consider the collection of people who use Facebook and their friend relationships to be a directed graph; each person would be one vertex in the graph, while each instance of a person A listing person B as a friend would be a directed edge from vertex A to vertex B. Now suppose that two people create Facebook accounts for the first time, and they each list one another as a friend. For now, these two people are a sort of island in the graph; they're disconnected from anyone else, with no path (via friend relationships) able to reach from either of them to anyone other than them (or vice versa).

So it's certainly the case that we'll need to consider the effect that connectedness has on various graph algorithms. It's not a bad idea, either, to be sure we understand what connectedness really means, and that it's not quite as simple of a concept as it sounds, particularly when we're talking about directed graphs. Let's explore this topic a little more thoroughly.

Connectedness in undirected graphs

What do we mean when we talk about the *connectedness* of an undirected graph? The story actually starts with a related definition, which talks about the *connectedness* of two vertices.

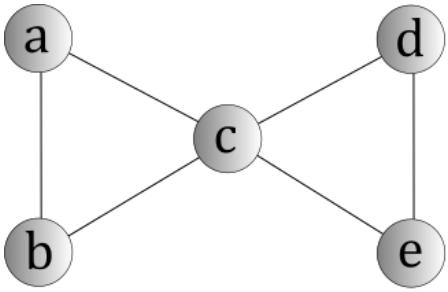
> We say that two vertices *v* and *w* in an undirected graph are *connected* if there is a path containing both *v* and *w*.

Recall that a *path* in a graph is a sequence of vertices where each vertex in the sequence has an edge leading to the vertex that follows it. So, generally, this definition just means that two vertices are connected if you can start with one of them, follow a sequence of edges, and eventually end up at the other one.

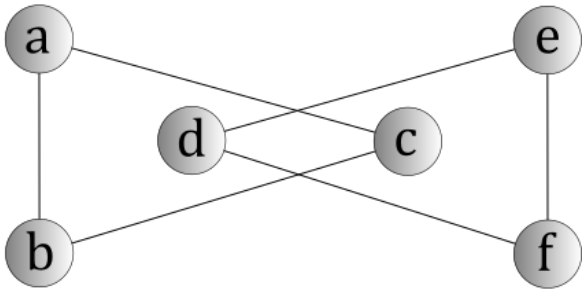
The definition above leads to a broader definition of whether an entire undirected graph is connected.

> We say that an undirected graph is *connected* if every pair of vertices in the graph is connected.

In other words, in an undirected graph that is connected, you can start anywhere and follow edges to get anywhere else. Consider this definition in relation to the two undirected graphs, G_1 and G_2, below.



G_1



G_2

With undirected graphs, it's reasonably straightforward to look at them quickly and determine their connectedness. G_1 is clearly connected; no matter which vertex you choose, you'll be able to reach any of the others. G_2, on the other hand, is clearly not, because it consists of two separate "islands," which are technically called *connected components*.

> We say that a subset of the vertices in an undirected graph is a *connected component* if every pair of vertices in the subset is connected, but none of the vertices is connected to any vertex outside of the subset.

An algorithm for determining the connectedness of an undirected graph

The depth-first graph traversal algorithm we saw previously nicely forms the basis of an algorithm for determining whether an undirected graph is connected. Recall that our depth-first graph traversal algorithm actually had two parts:

- **DFT**, which was the complete depth-first traversal.
- **DFTr**, which was a recursive, depth-first traversal starting at some vertex, which would reach only the vertices that were reachable from where we started.

If all we want to know is whether an undirected graph is connected, there's an easy way to do it. Below is an algorithm that uses **DFTr**, with the additional presumption that the **visit** step simply adds 1 to the **visitedCount**.

```
IsConnected(UndirectedGraph g):
    startVertex = any vertex in g
    visitedCount = 0

    DFTr(g, startVertex)

    return visitedCount == number of vertices in g
```

So, generally, we run a single **DFTr** from the start vertex, adding 1 to **visitedCount** every time we visit a vertex. When we're done, the counter will tell us how many vertices were reached. If we reached all of the vertices (i.e., if the counter equals the number of vertices in the graph), the undirected graph is connected; otherwise, it's not. (If it seems strange to you that it's possible for **visit** to add 1 to a local variable in **IsConnected**, note that there are lots of ways to achieve this kind of goal in C++: a member variable in a class, a lambda expression that captures the local variable by reference, or even a global variable.)

We can also modify our algorithm slightly to find the connected components instead. For this, we'll instead say that the **visit** step will assign a **componentNumber** to the vertex being visited.

```
FindConnectedComponents(UndirectedGraph g):
    for each vertex v in g:
        v.visited = false

    componentNumber = 0
```

```
for each vertex v in g:
    if not v.visted:
        componentNumber++
        DFTr(g, v)
```

The **FindConnectedComponents** algorithm is a lightly-modified version of **DFT**, which increments a **componentNumber** variable each time we start a new call to **DFTr**. If the **visit** step in **DFTr** assigns that **componentNumber** to each vertex it visits, then when we're done, all of the vertices in each connected component will be marked with the same **componentNumber**, and any two vertices in different connected components will be marked with a different **componentNumber**.

Try these algorithms on the two undirected graphs above to make sure you understand how they work.

Asymptotic analysis of the algorithms

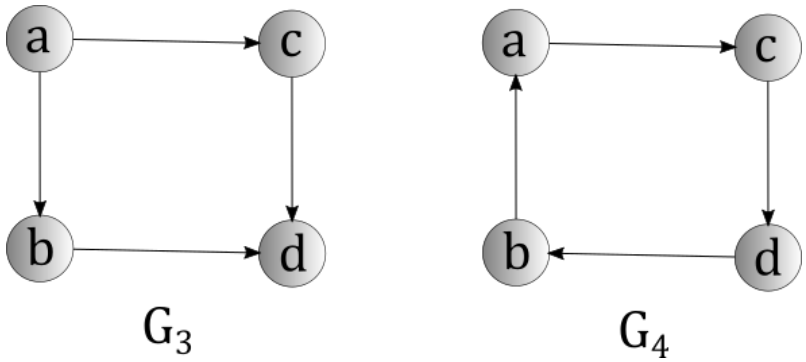
Since these algorithms are ultimately nothing more than depth-first traversals with slightly different **visit** implementations, each of which takes $\Theta(1)$ time to run, the analysis here is straightforward.

- Recall that a complete depth-first traversal runs in $\Theta(v^2)$ or $\Theta(v + e)$ time, depending on how the graph is implemented.
- **IsConnected** runs in $O(v^2)$ or $O(v + e)$ time, then, since it may not run a complete traversal (i.e., if the graph is disconnected and you choose a vertex that has relatively few other vertices in its connected component, you'll only reach those).
- **FindConnectedComponents** runs in $\Theta(v^2)$ or $\Theta(v + e)$ time, since it does a complete depth-first traversal of the entire graph.

Connectedness in directed graphs

Strong connectedness and weak connectedness

Connectedness in directed graphs is a slightly more intricate concept, because the directionality of the edges introduces the possibility of two vertices being connected in one direction but not the other (i.e., just because there is a path leading from *v* to *w* does not imply that there is a path leading from *w* to *v*). This basic observation leads to two separate definitions of connectedness, embodied by the two directed graphs below.



The graph on the left, G_3 is one that appears to be connected when you look at it quickly, but the directionality of its edges reveals an important shortcoming: From *a*, it's possible to reach every vertex; but from any other vertex, there is at least one other vertex that you can't reach. On the other hand, the graph on the right, G_4 doesn't have this problem; no matter where you start, you'll be able to reach every vertex in the graph.

To describe the distinction between these kinds of connectedness in a directed graph, we use the terms *weakly connected* and *strongly connected*, with G_3 being weakly connected and G_4 being strongly connected.

To be more formal, we define strong connectedness this way:

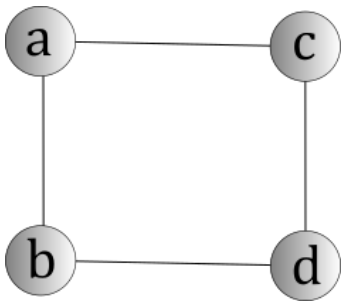
> We say that a directed graph is *strongly connected* if, for every pair of vertices *v* and *w*, there is a path from *v* to *w* and a path from *w* to *v*.

The graph G_4, being one that contains a cycle that includes every vertex, is one that certainly has this property. On the other hand, G_3 does not meet this property, because, for example, there is no path leading from the vertex *b* to the vertex *a*.

Meanwhile, we define weak connectedness a little bit differently, first defining something else called the *underlying undirected graph*.

> The *underlying undirected graph* of a directed graph is one with the same set of vertices, but a different set of edges. In particular, between any pair of vertices *v* and *w*, if the directed graph has an edge $v \rightarrow w$ or an edge $w \rightarrow v$, the underlying undirected graph includes an edge $\{v, w\}$.

The underlying undirected graph for the graph G_3 above looks like this:



Additionally, note that since undirected graphs can't have "self-edges" (i.e., an edge $v \rightarrow v$, for some vertex *v*), we simply remove them; they're not included in the underlying undirected graph.

Given this definition, we define weak connectedness as follows:

> We say that a directed graph is *weakly connected* if its underlying undirected graph is connected.

This definition codifies the general notion that weakly connected directed graphs are the ones that "look connected" when you don't consider the directionality of their edges, while also suggesting an algorithm for determining weak connectedness.

There's one more thing to note: All graphs that are strongly connected are also weakly connected. For example, the graph G_4 has the same underlying undirected graph as G_3 does, so it, too, is weakly connected.

An algorithm for determining weak connectedness of a directed graph

If we want to test that a directed graph is weakly connected, we have a couple of choices:

- Build the underlying undirected graph
- Traverse the directed graph as though it was the underlying undirected graph, by treating the edges as though they have no direction

How you make that decision has mostly to do with the way that the graph is implemented.

- If the directed graph is implemented as an adjacency matrix, it can be traversed as though it was the underlying undirected graph by simply looking at an entire row and an entire column each time you

want to find the outgoing edges from some vertex. This will easily find all edges $v \rightarrow$ anything and anything $\rightarrow v$
- If the directed graph is implemented as adjacency lists, it will be better to simply build the underlying undirected graph, since finding all incoming edges for some vertex will require searching all of the lists.

From there, the algorithm is the same as the one for finding out whether an undirected graph is connected. If the underlying undirected graph is connected, the directed graph is weakly connected; if not, it isn't.

The asymptotic analysis is straightforward:

- Building the underlying undirected graph would require iterating through the entire structure and making a (slightly different) copy of it, which would take $\Theta(v^2)$ or $\Theta(v + e)$, depending on how the graph is implemented (i.e., matrix or lists).
- Testing for connectedness of the underlying undirected graph would require $O(v^2)$ or $O(v + e)$ time, again depending on how the graph is implemented.
- The total time would be $\Theta(v^2)$ or $\Theta(v + e)$ if the underlying undirected graph is built, or $O(v^2)$ or $O(v + e)$ if not.

An algorithm for determining strong connectedness of a directed graph

As we've seen, depth-first traversals can form the basis of many graph algorithms, so we might expect a depth-first traversal to be a good choice here. And, indeed, you can use a depth-first traversal to find out whether a directed graph is strongly connected, but you have to be a bit careful about how you apply it.

To understand what we need to do, consider the graph G_3 above, which is not strongly connected. Suppose that you ran a **DFTr** starting from the vertex *a*. If you did, you'd reach every vertex, which is a positive result, but isn't enough; it establishes that there is a path from *a* to every vertex, but not that there is a return path from those vertices back to *a*. This tells us that just performing a **DFTr** that reaches every vertex isn't enough.

One approach that would work would be to simply run a separate **DFTr** starting at every vertex. If they *all* reach every vertex, then you would be sure that the graph was strongly connected.

The asymptotic analysis, again, would be straightforward. We know that a single call to **DFTr** takes $O(v^2)$ or $O(v + e)$ time (depending on the graph's implementation). Running as many as v of these calls — one for each vertex — would then take $O(v^3)$ or $O(v^2 + ve)$ time.

It should be noted that there are better algorithms than this one, such as Tarjan's Algorithm and Kosaraju's algorithm, though we didn't discuss either of these in class and are outside the scope of this course, but might make for interesting additional study if you're curious about them.