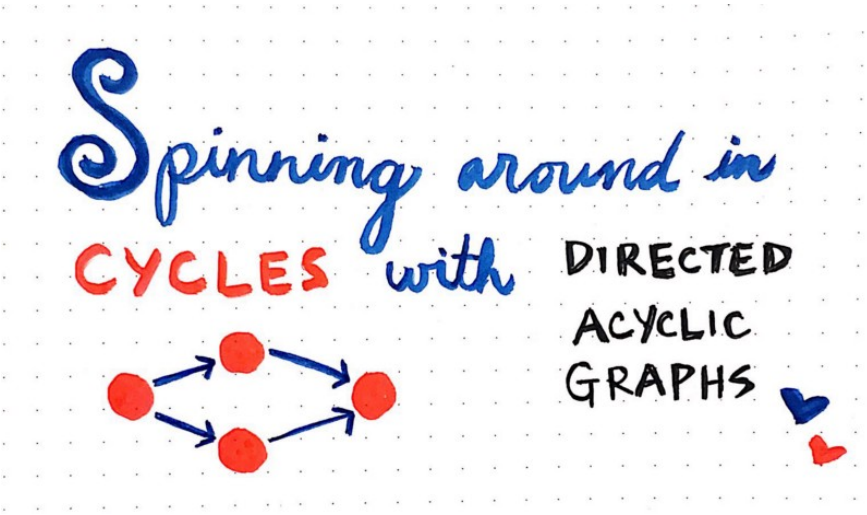
Vaidehi Joshi Follow
Writing words, writing code. Sometimes doing both at once.
Oct 2, 2017 · 15 min read

Spinning Around In Cycles With Directed Acyclic Graphs



Spinning around in cycles with directed acyclic graphs!

Throughout the course of this series, I've written quite a bit about rediscovering things in computer science that I had heard or read about before, but didn't really have enough context to understand.

At first, I thought that this strange cyclic process of stumbling upon the same concepts again and again was unique to computer science. But these days, I am of a different opinion: the process of learning and relearning an idea is fundamental to understanding it at a deeper level. This idea has a name: the hermeneutic circle. At it's core, the hermeneutic circle is grounded in the idea that we must return to ideas again and again, and understand parts of them at a time in order to fully comprehend the whole (and vice versa).

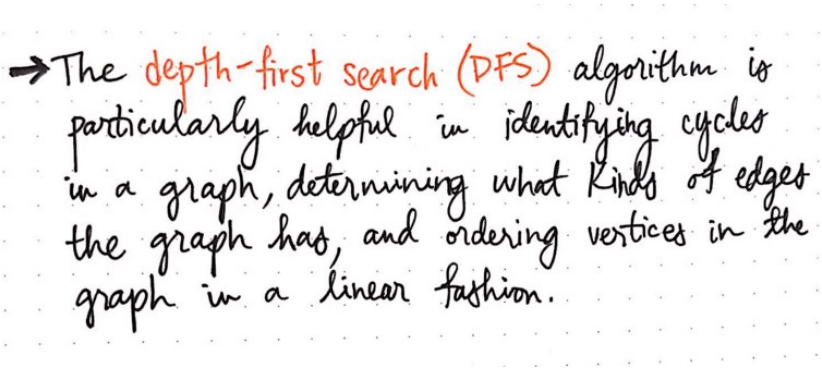
This is cyclical process is exactly what I experienced while learning about different types of graphs, including directed graphs, cyclic graphs, and everything in between. I had heard about these types of graphs before and was slightly familiar with all of them, but didn't fundamentally

understand when they would be used, or why they kept coming up again and again, in everything that I read.

As it turns out, learning is hard! And it takes time, as well as a good amount of repetition. So, let's return to graphs and learn about a specific subset of them that are fundamental to many solutions in computer science.

Edges, edges, everywhere

Throughout our exploration of graphs, we've focused mostly on representing graphs, and how to search through them. Last week, we looked at ***depth-first search (DFS)***, a graph traversal algorithm that recursively determines whether or not a path exists between two given nodes. However, it's worth cycling back to depth-first search again for a few reasons.



The DFS algorithm is helpful in identifying cycles, determining edges, and ordering vertices.

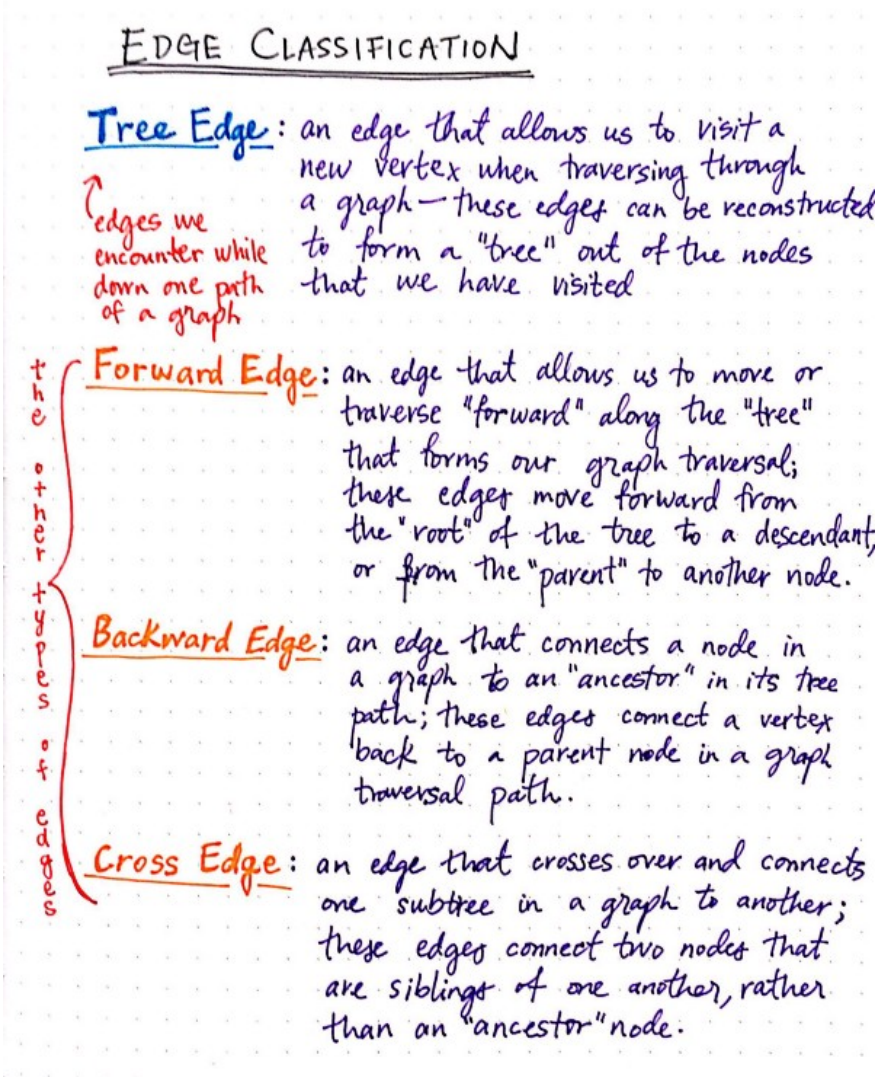
Depth-first search is useful in helping us learn more about a given graph, and can be particularly handy at ordering and sorting nodes in a graph. We can use the DFS algorithm to identify cycles within a graph, to determine what kinds of edges a graph has, and to order the vertices within in a linear fashion.

We'll work our way through each of these three tricks today, and see how using depth-first search can help us better understand what kind of graph we're dealing with in the process of searching *through* it!

Early on in this series, we learned about edges, the elements that connect the nodes in a graph. We also learned that edges can be *directed*, with a flow from one node to another, or *undirected*, with a bidirectional flow

between both vertices. But, keeping with the theme of this post, let's return back to what we *think* we already know.

If we cycle back to edges, we'll see that, in the context of a graph, edges can be more than just "directed" or "undirected". Two directed edges can be very different from one another, depending on where in the graph they happen to occur! Let's take a look at some of the other ways that we can classify edges in a graph.



Edge classification in a graph, part 1

There are two main categories of edges in a graph data structure: a **tree edge** and a **non-tree edge**. We'll recall that in both the BFS algorithm as

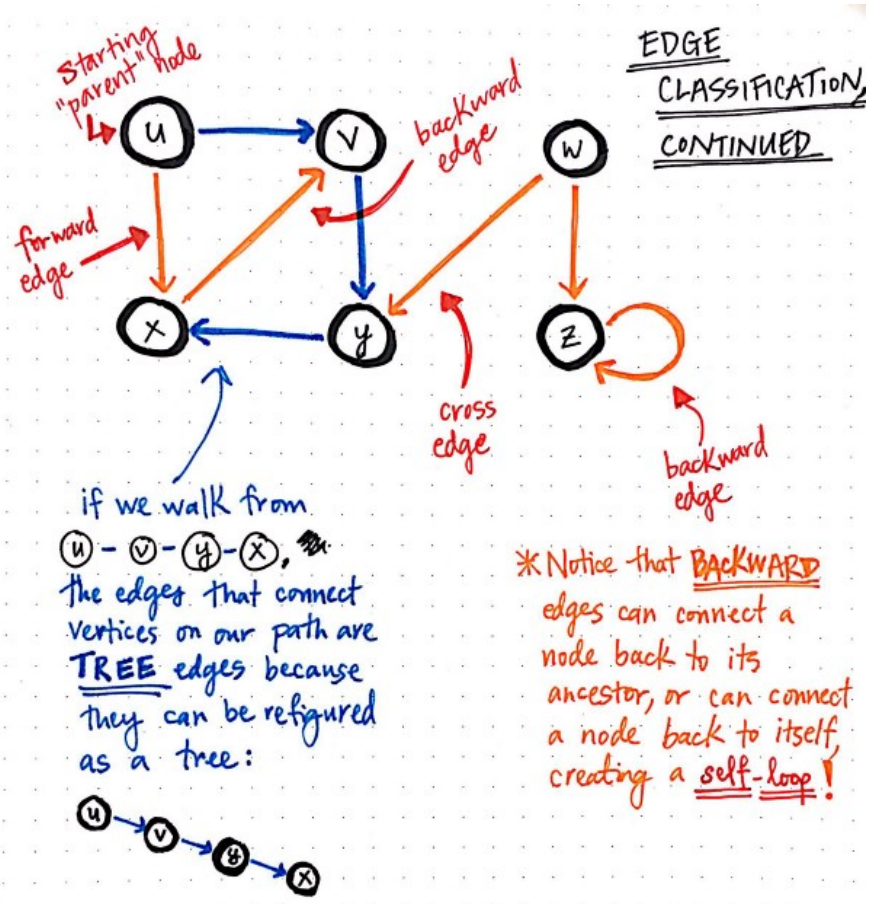
well as in the DFS algorithm, the *parent pointers*, which lead back to the starting node, can be rearranged to form a tree.

Any edge that is part of a path taken in a graph traversal algorithm is known as a ***tree edge***, since it is an edge that allows us to visit a new vertex each time as we traverse through the graph, and can be restructured to form a "tree".

So, what about the non-tree edges? Well, there are three different variations of those, and we've actually already encountered all of them, even if we didn't necessarily know what they were called at the time.

The first of the three is called a ***forward edge***, as it allows us to move "forward" through the graph, and could potentially be part of another path down the tree. By contrast, there is also the ***backward edge***, which connects a node in a graph "back up" to one of its ancestors (its parent or grandparent node, for example, or even back to itself!). Finally, there is the ***cross edge***, which "crosses over" and connects one subtree in a graph to another. In other words, a cross edge connects to sibling nodes that don't necessarily share an ancestor in a tree path, but connects them anyways.

These definitions make a whole lot more sense when we can see them applied to *actual* nodes in a graph, so let's look at an example.

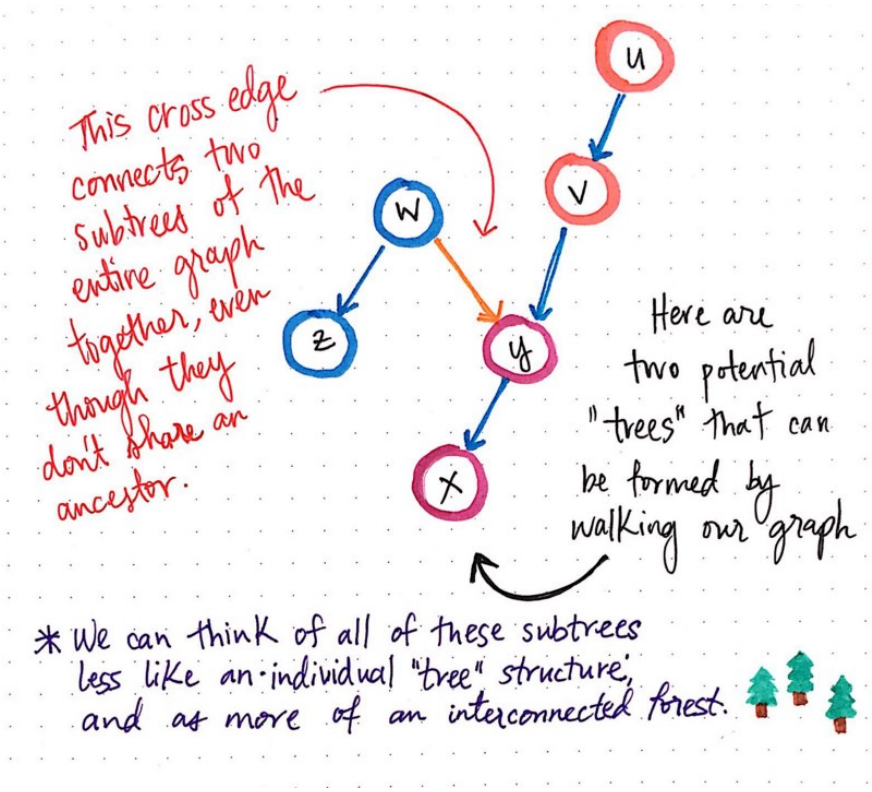


Edge classification in a graph, part 2

In the illustration shown here, we can see that the tree edges, highlighted in blue, are the ones that we walk through as part of our traversal down our path in the graph; in this case, our walk is `u-v-y-x`. The edges connecting these vertices in our path are our *tree edges*. However, we could have also walked "forward" in a different path, and walked from `u` to `x`; thus, the edge `(u, x)` is a *forward edge*.

We'll notice that, when we get to node `x`, the only vertex to navigate to next is `v`. The edge connecting these two nodes in the tree, `(x, v)`, is a *backward edge*, since it connects the node `x` back to one of its ancestors in the path: node `v`. We'll also notice that, in another part of the graph—in a different subtree, rather—node `z` has a backward edge that is actually connecting *back to itself*. This is also known as a ***self-loop***, or a backward edge that connects a node back to itself, or an edge that "references" the node that it came from: `(z, z)`. (We'll come back to self-loops later on, but this is our very first taste of them!)

The edge that connects node `w` and `y`, which we can also call `(w, y)`, is a *cross edge* as it connects the subtree. Cross edges are a little easier to see when we rearrange our tree edges to actually *form* a tree, so let's redraw that same graph to make our cross edge more obvious. In the image below, we'll see two potential "trees" that could be formed by starting at two different nodes in the graph: node `u` or node `w`, respectively.



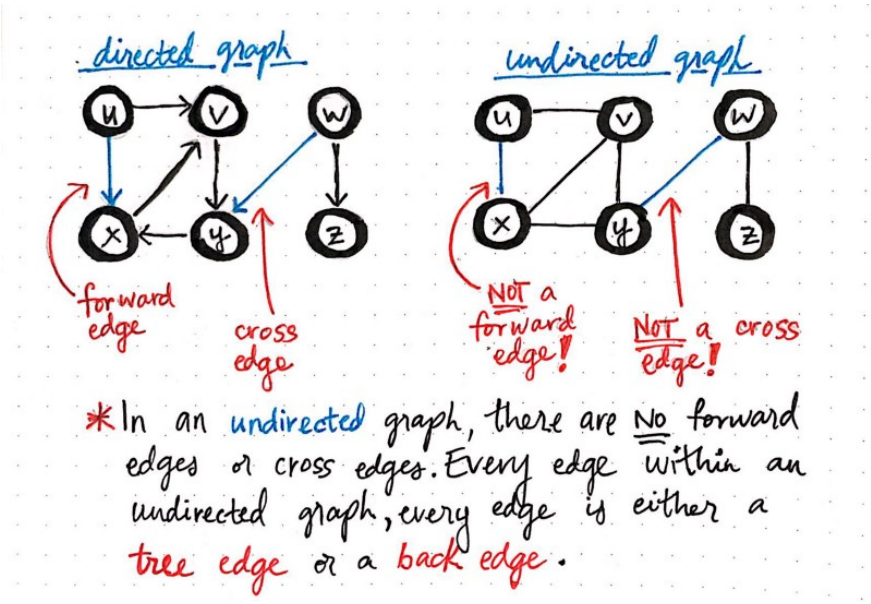
A graph, reimagined as connected (sub)trees

The edge `(w, y)` connects the two subtrees of this entire graph together, even though there is no single "root node" or ancestor that the subtrees actually share. Remember that a graph is a very "loose" version of a tree, and doesn't follow the same rules that a tree would.

> *Since a graph doesn't have one single root node, it doesn't form a single tree. Rather, a graph could contain many small subtrees within it, each of which would become obvious as we walked through the paths of the graph. A graph is less of an individual*

tree data structure, and more of an interconnected forest, with small subtrees within it.

There is one important thing to note about all of these different kinds of edges: they don't always exist! It all depends on what type of graph we're dealing with. In a *directed* graph, tree edges, cross edges, backward edges, and forward edges are all possible. However, for an *undirected* graph, it's a different story altogether.

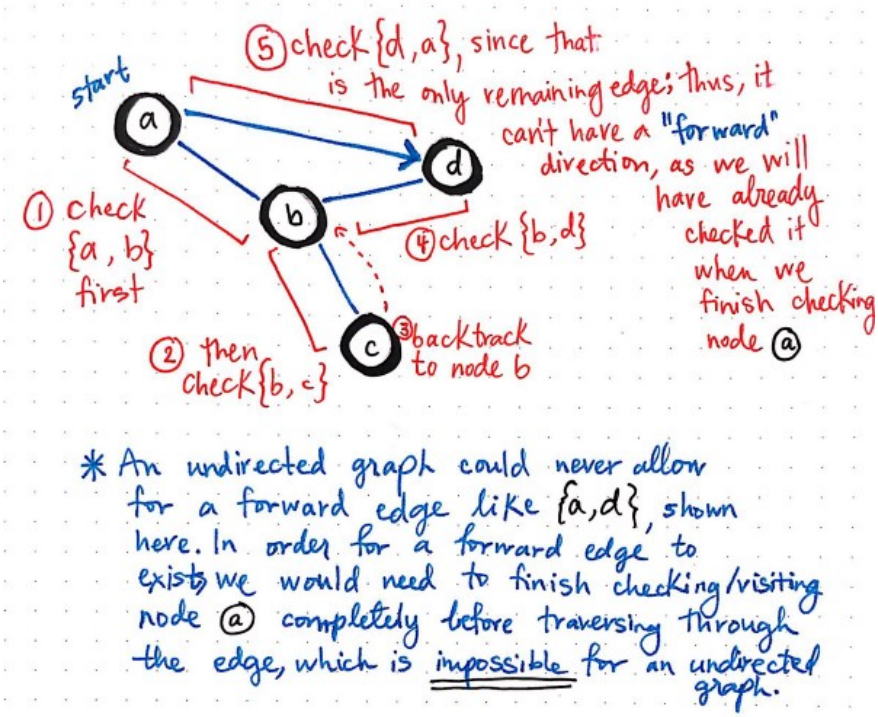


An undirected graph can only ever have tree edges or backward edges, part 1

In an undirected graph, there are no forward edges or cross edges. Every single edge must be either a tree edge or a back edge.

An easy way to remember the rules of edge classification is this: an undirected graph can only have tree edges and back edges, but a directed graph could contain all four edge types.

Okay, but what's the story behind this rule? Well, it's a lot easier to see if we actually try it out on an example, so we'll do exactly that.



An undirected graph can only ever have tree edges or backward edges, part 2

In the graph shown here, we have four vertices, and four edges. Let's say that the edge `(a, d)` is a "forward" edge; in other words, it's an edge that allows us to move from node `a` to node `d`.

Imagine that we walk through this graph, from node `a` to `b`, and then from `b` to `c`. When we get to node `c`, there is nowhere else for us to go, so we must backtrack up to node `b`.

Now, in order for us to properly run the depth-first search algorithm, we need to check if there are any other vertices that we can "visit" from node `b`. As it turns out, there is one! We'll visit node `d`. We're once again in a similar situation, and we have to check if there is another node that we can visit from `d`. As it turns out, since this is an undirected graph, there is one place we can visit: node `a`. At this moment, it becomes impossible for the edge `(a, d)` to be a "forward" edge. Since the rules of DFS dictate that we *must* visit node `a` from `d` since we must explore the graph on the *deepest* level, we must visit it; we cannot backtrack back *up* to node `a` and then visit `d` from there!

An undirected graph can never allow for a forward edge like `(a, d)` since, in order for a forward edge to exist, we would need to completely

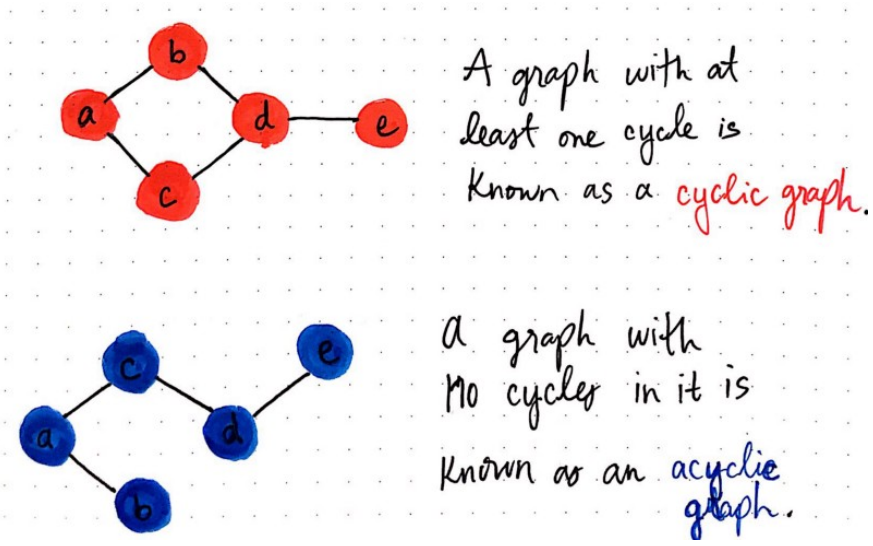
finish visiting all the nodes reachable from our starting node `a` before traversing through the edge. In an undirected graph, this is literally impossible to do! So, the edge between these two nodes really ought to be written like this: `{a, d}`, since we can traverse in either direction on this edge, depending on which node we start with.

Even though we won't look at an example of it here, we could disprove cross edges similar to forward edges, since they also cannot exist in an undirected graph.

Round and round

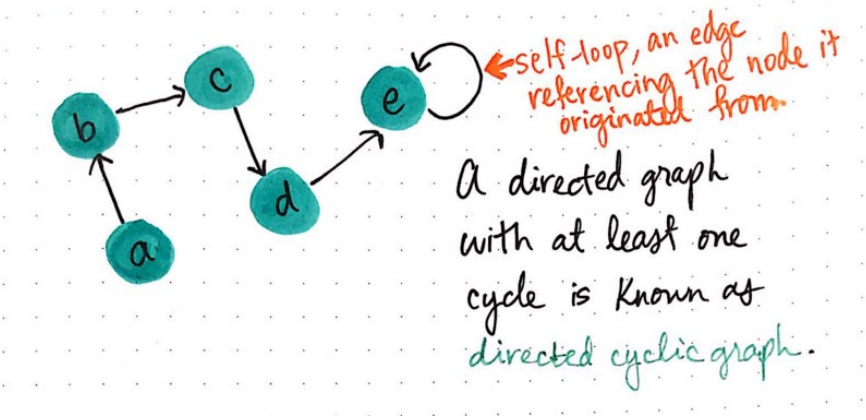
Another great strength of the depth-first search algorithm is its ability to identify cycles in a graph. Before we get too far into how to do that, let's familiarize ourselves with some important terms that we'll end up using along the way.

A ***cycle***, in the context of a graph, occurs when some number of vertices are connected to one another in a closed chain of edges. A graph that contains at least one cycle is known as a ***cyclic graph***. Conversely, a graph that contains zero cycles is known as an ***acyclic graph***.



Cyclic vs. acyclic graphs

In the previous section, we saw our first instance of a ***self-loop***, a type of backward edge. Self-loops can only ever occur in a *directed* graph, since a self-loop is a type of *directed* edge.

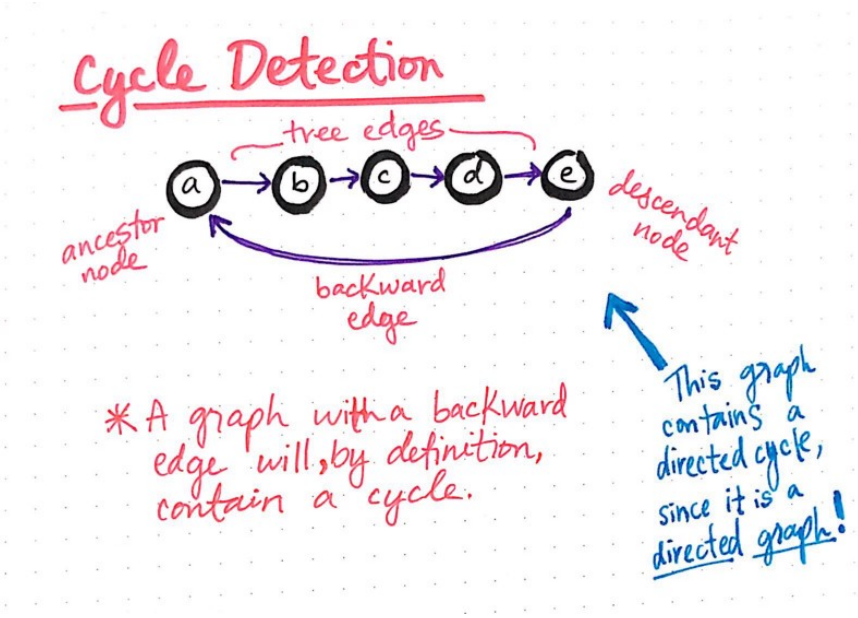


A directed cyclic graph, with a self-loop

Both directed and undirected graphs can have cycles in them, but it's worth noting that a self-loop can only ever occur in a ***directed cyclic graph***, which is a directed graph that contains at least one cycle in it.

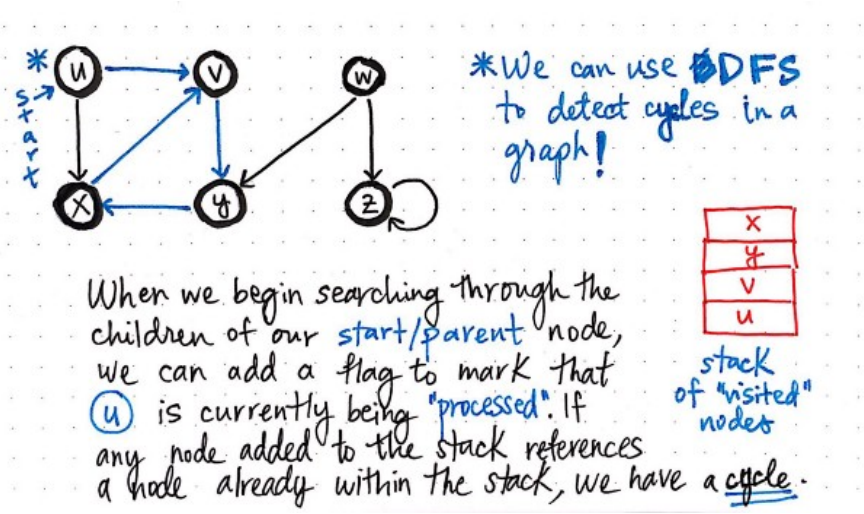
> *As it turns out, the reason that the depth-first search algorithm is particularly good at detecting cycles is because of the fact that it is efficient at finding backward edges.*

By definition, any graph that has a backward edge will contain a cycle. In the graph drawn below, we can see that node `e` connects back to an ancestor in the path, node `a`. Because of a backward edge exists in this directed graph, we know that this graph also contains a cycle—it is a *directed cyclic graph*.



Cycle detection and backward edges

Okay, but how does DFS actually detect cycles, exactly? And why is this algorithm in particular so efficient at determining whether or not a cycle exists within a graph? Well, it all goes back to the recursive nature of the depth-first search algorithm. We'll recall that DFS uses a stack data structure under the hood. This makes is slightly easier to know whether a backward edge exists in the path as we walk through the graph.



We can use DFS to detect cycles in a graph.

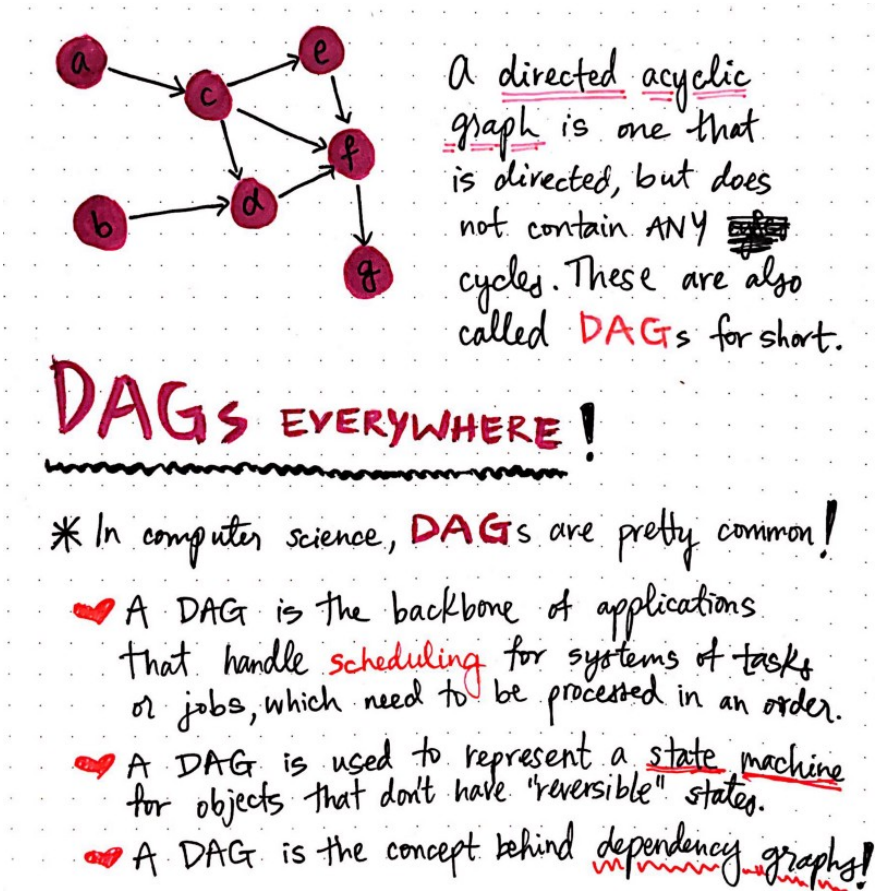
For example, in the example graph from earlier, when we traverse from `u` to `v` to `y` to `x`, we add each of these elements to the stack of

nodes that we've visited. We can take one additional step to make it easy to classify edges: we can mark the node `u` as currently `beingProcessed` with a simple boolean flag when we start searching down its "deepest" path. Once we flag node `u` as being the node that we're searching through, if any node that we add to the stack that references *back up* to `u`, we know that we have found a cycle. Indeed, this same logic applies to other internal cycles in the graph, and not just cycles connecting back up to the starting node.

> *If any node added to the stack has a reference to a node that is already within the stack, we can be sure that we have a cycle in this graph.*

But wait—there's more! Well, okay—there is just *one* more. The final type of graph that we need to define is a directed graph *without* cycles.

Most implementations of depth-first search will check to see if any cycles exist, and a large part of that is based on the DFS algorithm checking to see whether or not a graph is a ***directed acyclic graph***, also known as a ***DAG***. A directed acyclic graph is, as its name would suggested, directed, but without any cycles. The golden rule of DAGs is that, if we start at any node in the graph, no sequence of edges will allow us to loop back to the starting node. Thus, by definition, a directed acyclic graph can never contain a self-loop.



Directed acyclic graphs (DAGs)

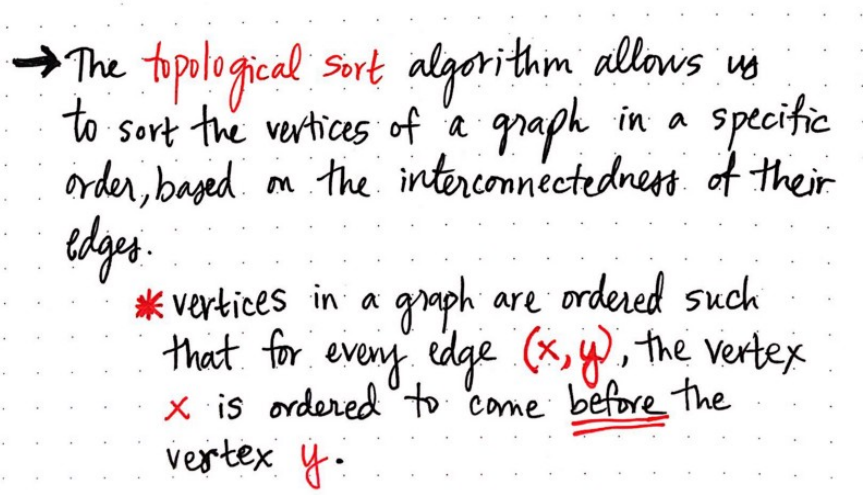
DAGs are somewhat infamous in computer science because they're pretty much *everywhere* in sofware. For example, a directed acyclic graph is the backbone of applications that handle scheduling for systems of tasks or handling jobs—especially those that need to be processed in a particular order. At a higher level, DAGs are used to represent state machines, which are often used to keep track of objects that have a mutable state in web and mobile applications.

My personal favorite example, however, is the use of DAGs to represent dependency graphs. If you've ever had to resolve dependencies in a Gemfile, or handle updating packages or navigate dependency issues using package managers like npm or yarn, then you were interacting with a DAG, on a very abstracted level! The dependency graph is also a great example of how DAGs can be complicated and massive and size, and why it can be useful to sort through and order the nodes within such a graph.

So, let's learn how to do that!

Topological sorting

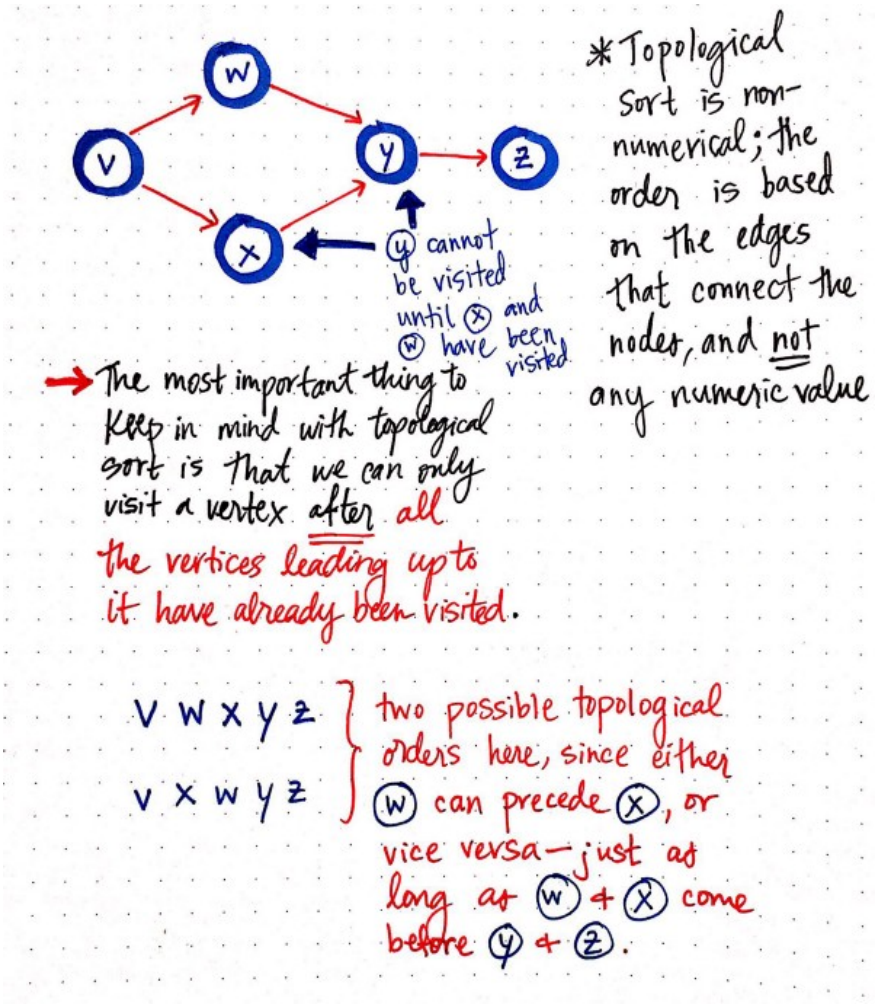
Since we now know how vast and complicated a directed acyclic graph can actually be, being able to sort through and order vertices and make sense of the data *within* a DAG can be super helpful. Thankfully, there is an algorithm that does exactly that!



Topological sort: a defintion

The ***topological sort algorithm*** allows us to sort through the vertices of graph in a specific order, based on the interconnectedness of the edges that connect the vertices. If two vertices, `x` and `y` exist in a graph, and a directed edge `(x, y)` exists between them, then topological sort will order the vertices in the way that we would encounter them in a path: `x`, followed by `y`.

If we consider the ordering of topological sort, we'll notice that the sorting itself is ***non-numerical***. In other words, the order of the vertices is based on the *edges* that connect them, and not any numeric value.



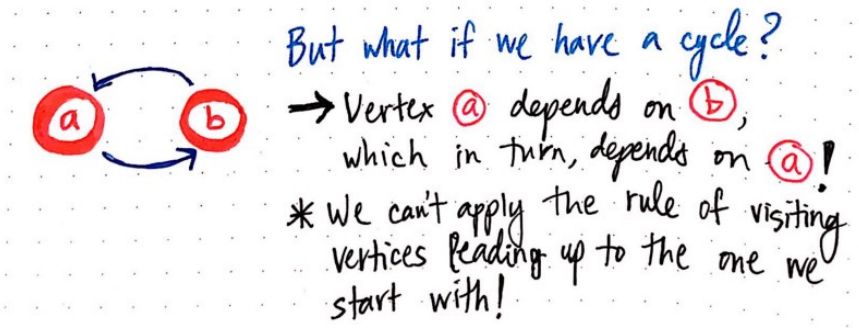
Topological sort and its resulting topological order

The most important thing to keep in mind when it comes to topological sort is the fact that we can only ever visit a vertex after all the vertices leading up to it have *already* been visited.

For example, in the graph shown here, we have five nodes: `v`, `w`, `x`, `y`, and `z`. In order to run topological sort on this directed graph, we need to be sure that, as we order and visit our vertices, the node `y` cannot be visited until both `x` and `w` have been visited.

Just from looking at this graph, we can see that there are two possible routes that we could take: `v-w-x-y-z` or `v-x-w-y-z`. In both of these two paths, we would still be visiting node `y` after both `x` and `w`. Thus, both of them are valid, and each of them return valid ***topological orders***, which is the ordered set of vertices that is the returned result of running topological sort.

In this graph, we don't have a cycle. But what if we did? The simplest version of such a graph would have two nodes, each of which reference one another, forming a cycle. Here's an example of what that graph might look like:



Topological sort can only ever be applied to DAGs, or acyclic graphs.

In this example, vertex `a` depends on `b`, which in turn, depends back on `a`. The issue here is that we can't apply the rule of visiting the vertices that "lead up" to the one we're starting with, because they both lead up to one another.

> *A topological sort can only ever be applied to directed acyclic graphs, or DAGs. It is impossible to run a topological sort on a directed graph with a cycle, since it is unclear where the sort itself should start.*

There are a few different ways to actually implement topological sort. However, for today, we don't even need to learn any new ones, because we already *know* one intuitively! I'm talking, of course, about *depth-first search*.

So, let's take a look at topological sort in action—using DFS to help us!