CS 3510 Algorithms 1/30/2024

Lecture 5: DFS, Topological Sort, and Strongly Connected Components

Lecturer: Abrahim Ladha *Scribe(s): Joseph Gulian, Saahir Dhanani*

Welcome to the first lecture of unit 2 on graph algorithms! Graphs are useful topological and combinatorial devices to study many problems.

1 Graphs

Graphs are described by a set of vertices V and by a set of edges which are pairs of vertices. This is represented as $G = (V, E)$. Graphs with ordered edges (where there is a direction) are called directed graphs; graphs with unordered edges (where there is no direction) are called undirected graphs. Though not in this lecture, it is sometimes useful to apply a weight or value to edges; the weight of an edge is denoted as w_e where e is the edge.

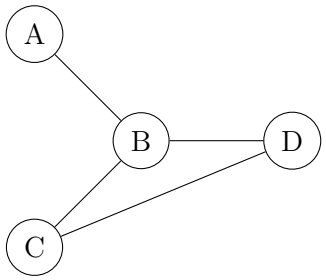

Figure 1: A undirected unweighted graph.

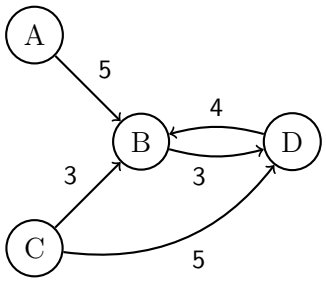

Figure 2: A directed weighted graph.

We consider graphs as these pictures, dots and lines embedded onto the page, but this is not how computers look at graphs. We need to discuss the way that graphs may be encoded. We have two main choices in how to represent graphs like these.

2 Adjacency Matrix

The first is called an adjacency matrix. This is a square matrix of size $|V| \times |V|$. We associate each element of the matrix as an edge in the graph. The above graph14 could be represented as the following matrix

$$\begin{pmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 3 & 0 & 5 \\ 0 & 3 & 0 & 0 \end{pmatrix}$$

Notice here, that there are a lot of zeroes. This format can be wasteful for graphs without a lot of edges. Graphs without many edges are considered sparse, opposed to dense graphs which have many edges. We say a graph is dense if $|E| \approx |V|^2$, and that a graph is sparse if it is not dense. The following is an example of a dense graph

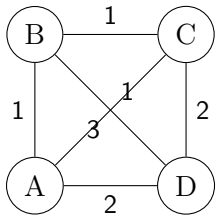

Figure 3: A weighted fully-connected graph.

This would be represented by the following matrix

$$\begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 1 & 3 \\ 1 & 1 & 0 & 2 \\ 2 & 3 & 2 & 0 \end{pmatrix}$$

You may notice, that the matrix above is symmetric. That is that $A = A^T$. This is the case of all undirected graphs. [1]

3 Adjacency List

We come back to this issue though, how do we represent sparse graphs? We could simply store edges instead of all possible edges by using $|V|$ linked lists. These would represent the outgoing edges from each node. The adjacency list representation for the above directed graph is as follows:

[1] You may notice something else as well: the diagonal has all zeroes. This is because there are not edges from nodes back to themselves in this graph. Some graphs may have edges like these; say if you were to model a state machine.

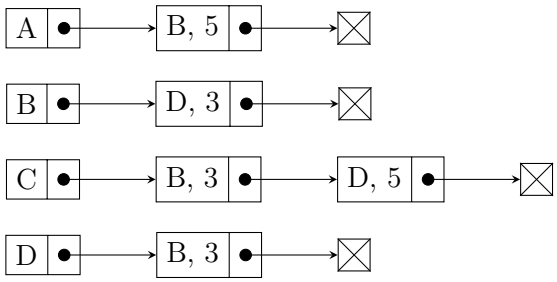

Figure 4: The adjacency list representation of a graph.

This seems like a more efficient option because there's no wasted space, but let's think about the space-time trade-off. If you want a particular edge in this graph, you need to iterate through the adjacency list. If the adjacency list is big, this could take a long time; this happens when graphs are dense. If a graph is sparse, this representation is better. A adjacency matrix takes constant time to read if an edge exists and its weight.
This representation is especially good for many real world situations. Think of the world wide web. We have billions (10^9) of websites with very few links (maybe 10 as an example). If you wanted to represent this as an adjacency matrix, you'd need 10^{18} bytes (assuming each edge has a weight within a byte) or 1 exabyte! Few people have this kind of memory on their computers. If we use the adjacency list representation we need to store roughly 10^{11} (or 10 edges/node $* 10^9$ nodes); this is fairly feasible. Although still large, it's orders of magnitude smaller, and more reasonable. This is just one example of where adjacency lists are better than adjacency matrices. [2] Most of this class will focus on adjacency lists for this reason. The size of an adjaceny list is $O(|V| + |E|)$ because there are $|V|$ linked list entry pointers, and each edge appears in a list once (if directed) or twice (if directed).

4 Graph Traversal

Say we're at some node on the graph, we want to visit all vertices we can reach from that node. Much like divide and conquer, our trick here will rely on recursion. Instead of implicitly using any advanced data structures, we'll simply use the call stack. We will define the following algorithm:

[2] Of course, there are many other options for representing graphs, all of which can be further fine-tuned for particular applications.

```
def explore(G, v):
    marked[v] = true
    previsit(v)
    for edge (v, u) in E:
        if not marked[u]:
            explore(G, u)
    postvisit(V)
```

Figure 5: Explore routine on a graph and node.

Notice two things about this: `marked` and `pre`/`post`. Marked is a static set available in all calls to the routine. `pre`/`post` are two points in routine where modification allows for more applications. We will change `pre`/`post` to help a lot with future algorithms. DFS isn't an algorithm so much as it is a primitive to build other algorithms. Its just a way to compute and iterate over the graph. We have a default implementation of `pre`/`post` as:

```
counter = 1
def previsit(v):
    pre[v] = counter
    counter++

def postvisit(v):
    post[v] = counter
    counter++
```

Figure 6: Explore routine on a graph and node.

It keeps two numbers per vertice, we denote as the pre and post number. It's also worth noting that this algorithm works on both directed and undirected graphs. The adjacency list representation for directed graphs would only contain outgoing transitions.

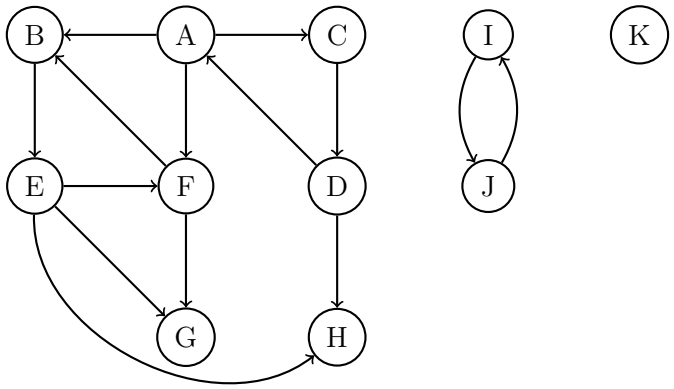

Figure 7: An example graph for explore.

Starting with A, we generate the following recursion tree, with the pre and post numbers for each vertex.

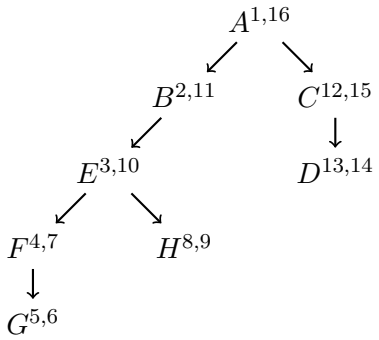

Figure 8: An example exploration of a graph with pre and post labels.

If we were to mark the steps we entered and exited `explore` for a node, we'd get the numbers above.

There's an issue with this graph though. We want to see all nodes, but we're missing I, J, and K. This is because there are disconnected components in the graph. Similarly if we started at K, we would not reach much of the graph. To achieve this, we simply loop over nodes on top of this.

```
def dfs(G):
    for v in V:
        marked[v] = false
    for v in V:
        if not marked[v]:
            explore(G, v)
```

Figure 9: dfs routine on a graph

Let's show that this algorithm is correct. If we start with A as in the previous example, we'll run explore as we did and cover that component. We then return to `dfs` and continue iterating until we hit I; at this point we explore J. Then we leave again, iterate, and reach K. Then, we know it works for our graph above, but how can we show something like it works for all graphs?
Let's setup a proof by contradiction showing `explore` does reach all vertices reachable from V. Assume to the contrary that `explore` does not reach all vertices reachable from V. So, if our dfs makes some mistake, then there is a node u reachable from v which is not touched by dfs. Since it is reachable from v, there exists some path like

$$v \to ... \to u$$
$$v \to ... \to z \to z' \to ... \to u$$

Here, let z is the last marked vertex in the path and w is the first non-marked vertex in a path from v to u. Explore on z was called, meaning it must call explore on all not marked vertices with an edge from z. z' has an edge from z, and it is not marked. Therefore `explore` does reach all vertices reachable from v, contradiction. Therefore, dfs is correct.
There are multiple types of edges in the DFS tree which correspond to where in the DFS tree they can or cannot be found.

- Tree edge (A, B) if it appears in the dfs tree.
- Forward edge (E, G) if it exists in the graph but not in the tree because it goes from a parent to a child that was already explored.
- Back Edge (D, A) if it exists in the graph but not in the tree because it goes from a child to a parent that was already explored.
- Cross Edge (D, H) if it exists in the graph but not in the tree because the other vertex was marked, even though it doesn't go to a parent or child.

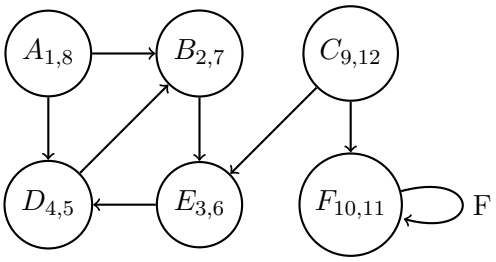

Figure 10: An example of pre and post values generated after running dfs

5 Cycles

Pre and post numbers give us a lot of information, like it allows us to find a cycle (a path where the first and last vertex are the same) in a graph. Checking for cycles has very real world applications like ensuring packages in a package manager don't have dependency cycles.

Now do we do this? Suprisingly we already have all the tools we need. We will now show that a graph has a cycle in it if and only if there is a backedge. We'll do this proof in two parts.

If a graph has a cycle, it will have a back edge. Define a cycle $v_0 \to ... \to v_k \to v_0$. Let v_i be the first traversed vertex in this cycle. The $v_i \to v_{i+1}$ will be traversed next and $v_{i-1} \to v_i$ will be the backedge because it will point to it's parent.

Now let's go the other way. Suppose there exists a backedge (B, A). Then the dfs tree path $A, ..., B + (B, A)$ would produce a cycle. Then we have shown that there is a cycle if there is a backedge.

This is very powerful, we can check for cycles in $O(n)$ with a modified `explore`: `explore-cd`. Now as well as marked, we will maintain a second set, `t-marked` which will store whether or not a vertex is a parent. `explore-cd` is as follows

```
def explore(G, v):
    marked[v] = true
    t-marked[v] = true
    for edge (v, u) in G connected to v:
        if t-marked[u]:
            print('cycle found!!!')
        if not marked[u]:
            explore(G, u)
    t-marked[v] = false
```

Figure 11: Explore-cd routine for cycle detection.

You may be asking about disconnected components for this, say we run explore on one component but there is a cycle in another disconnected component. It turns out we can

basically just do the same strategy we did above for `dfs`, but have a global `t-marked`.

6 Topological Sort

What is topological sort[3]? IF we have a directed acyclic graph, and we'd like to sort the graph by precedence we could use a topological sort. This could be very useful in a situation like evaluating a series of variables in an unspecified order.
Essentially what we're going to do is use `dfs` and the dfs tree again. Here is that the dfs tree is a subset of the graph which is a directed acyclic graph (DAG[4]). Our strategy for this will be to add them to a list as we return. We'll call this modification `top-sort` as shown here

```
def top-sort(G, v):
    marked[v] = true
    for edge (v, u) in G connected to V:
        if not marked[u]:
            explore(G, u)
    L = [v] + L
```

Figure 12: Top-Sort routine for topological sorting.

Consider the DAG below

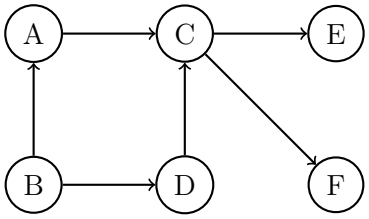

Figure 13: A directed acyclic graph.

We're not going to run top sort on the source because then we'd have to scan the graph, so instead we'll start on A like we normally do and iterate. This is still correct for disconnected components.

[3]The DPV book uses the term linearization.
[4]Know this acronym.

When we run the algorithm we get the following list when we finish each vertex.

$$\begin{aligned} A \to C \to E &: [E] \\ A \to C \to F &: [F, E] \\ A \to C &: [C, F, E] \\ A &: [A, C, F, E] \\ B \to D &: [D, A, C, F, E] \\ B &: [B, D, A, C, F, E] \end{aligned}$$

This is a topological sort.[5] Importantly, we see the source of the graph first and the sinks last. If you wanted to draw this on paper, you could do

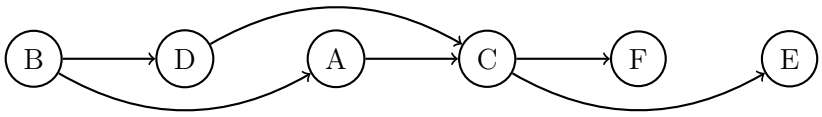

Figure 14: A linearization of a directed acyclic graph.

Note that the topological sort of a DAG can be given by the post numbers from highest to lowest, that is what is returned here essentially.

7 Runtime

The last thing we'll discuss is the runtime of these algorithms. Note that we visit every node once in the algorithm and cross every edge, so the runtime of these algorithms are $O(|V| + |E|)$.

8 Connected Components

Most graphs we cover in this class are connected because if we had a graph that was not connected, we could decompose it into it's connected parts and study those. It's trivial to find connected components in an undirected graph: run DFS. [6] The nodes found in each explore call will be the connected components; if you wanted to count the number of components, you could count the number of top level explore calls made from DFS.
In a directed graph, we say two nodes are strongly connected if there exists a path to and from the nodes with respect to all other nodes. Nodes v, u are connected if there is a path from v to u and also from u to v. In other words, if there is a cycle. This means that your graph won't always be disconnected, sometimes it may simply be that a node can reach another node, but the second node can not go back.

[5] Note that there could be many depending on which way you traverse

[6] There's an important distinction explore and dfs in the context of disconnected graphs. The explore routine that we've outlined will only explore within one connected component; dfs will explore all components. Confusing these is a common mistake, don't make it.

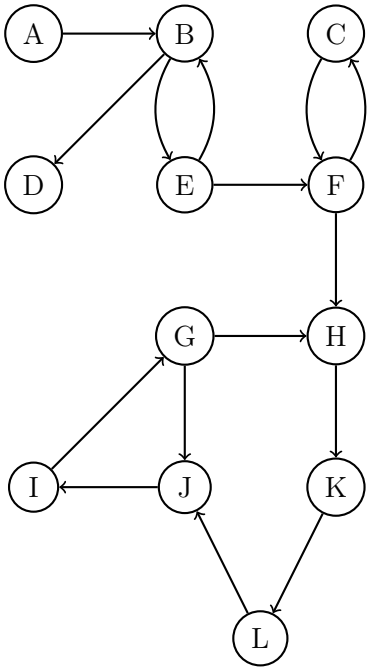

Figure 15: A directed graph.

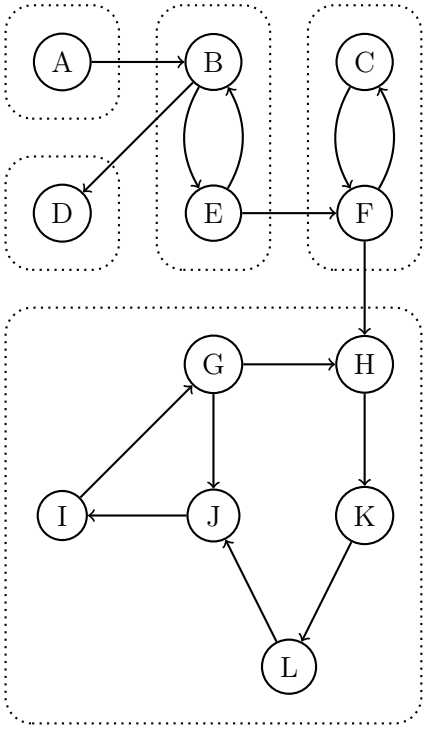

Figure 16: The components in a directed graph.

In this graph, the strongly connected components can be found by looking at it. Note that a vertex can not be contained in two components. This is because if a vertex is part of one cycle and part of another cycle, then there is a path between all nodes on both cycles meaning its one large component. Now, let's construct the metagraph, a graph with these components as vertices, and edges between components where they appear in the original graph.

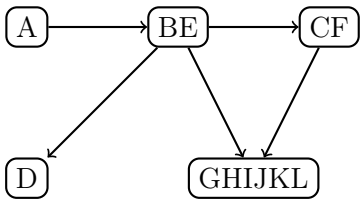

Figure 17: Components in a directed graph.

9 Metagraphs and Graph properties

Here's our metagraph; it happens to be a DAG. It's interesting to consider whether all metagraphs are DAGs, so let's try to prove it. Assume to the contrary that there exists a metagraph with a cycle. This would mean, the two components would themselves be strongly connected as we've deduced earlier. This means those nodes should have been in the same metanode, a contradiction to our correctness.

We want to create an algorithm to find strongly connected components in directed graphs. This isn't that simple. Say you call DFS on the graph like we did for connected components with undirected graphs. [7] Instead we use an algorithm which is a little more advanced than dfs called the SCC algorithm.

As an aside, many people have this intuition that there's an entrance and an exit to a graph. This could be formalized by the concept of sources and sinks. A source is a vertex with in-degree 0 (no edges entering), and a sink is a vertex with out-degree 0 (no edges exiting). Think about why this could be useful along with some of the algorithms we've previously devised by running explore from the sink and source (although we already ran it from the sink). Doing this you might notice that running explore from the sink of the metagraph reveals one strongly connected component of the graph.

Of course now, we have another problem, how do we get only the sinks of a graph. Recall what we did for top-sort: noting when a vertex was entered and exited. Performing this on the above graph, we get the following tree.

[7] Think about this. Calling on the first graph from the letter A would produce A, B, C, D, E, F, G, H, I, J, K, and L. Clearly this is not what we want as we can discover by looking at the answer.

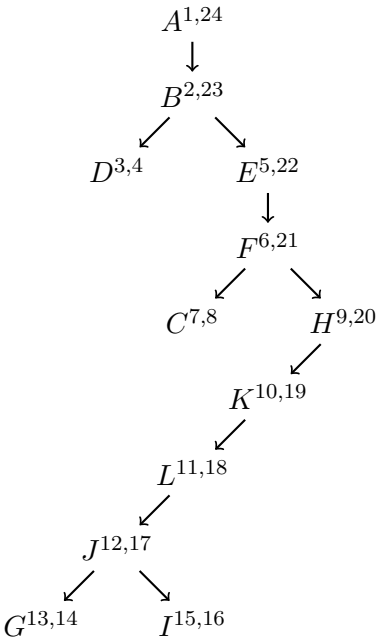

Figure 18: The dfs traversal of the graph.

If you thought about how you would top-sort this (ignoring cycles), it would put the source in the back. We're not guaranteed the sink would be the last element. Consider ordering by post label the following graph with a dfs which explores in alphabetical order. You'd get the ordering A, B, and C which is not correct because C is actually in the source of the graph.

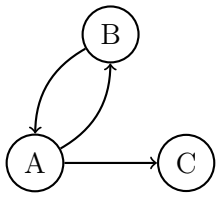

Figure 19: A graph whose loose topological sort does not yield a sink as the last vertex.

10 Proof and Algorithm

We will need to prove one thing before we can actually get a correct algorithm: If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest post number in C is bigger than the highest post number in C'. We'll prove this by cases. Say DFS finds C before C', it will explore $C \to C'$, and C' will have a smaller post label. Say DFS finds C' before C, C' will be added and then C will be added later through a separate explore call, so it will have a higher post label.
Considering that the first element in a "top-sort" will be a source, but we're looking for a sink. Consider what reversing all the edges will do; it will make the sources sinks and the

sinks sources. Now, we can sort by post labels, and our last element will be a source of the reverse graph G^R, but a sink in our real graph G.
An interesting proof arises with G and G^R wherein you might think that G and G^R have the same metagraph, simply with the reversed edges. This is correct and consider why. Consider the definition of a strongly connected component: there are cycles between all the vertices in a strongly connected component. Now consider if you reverse all the edges. All the edges in the cycle will be reversed but it will still form a cycle (simply in the reverse direction). All other edges will be reversed.
Considering all of this we come to the following algorithm for computing strongly connected comonents.

```
def scc(G):
    dfs(G) to get post labels in decreasing order
    compute reversal GR from G
    while posts not empty:
        v = max(posts)
        component = explore(GR,v)
        print or mark the component
        remove component from GR and posts
```

Figure 20: Strongly connected components algorithm

This algorithm works how you might think given the ideas we've devised. We find a source of `G`, so its guaranteed to be in the sink of `GR`. We then explore this entire sink component in `GR`. We remove this explored component from `GR`, and its nodes from our topsort, and repeat until we have all components. A common confusion is that even though top sort doesn't work on cyclic graphs, as we've devised top-sort, it will sort by post labels which is all we care about. **Note carefully that we call our "top-sort" on `G`, but we explore on `GR`.** It's also worth noting that this is essentially doing the same amount of work as two DFS calls, meaning strongly connected components run in linear time.