CS 3510 Algorithms 1/26/2023

Lecture 6: Topological Sorting

Lecturer: Abrahim Ladha *Scribe(s): Joseph Gulian*

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.

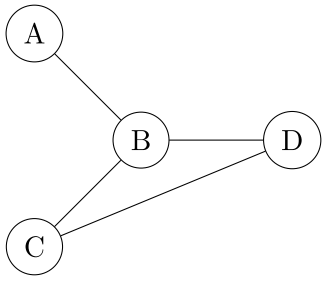

Figure 1: A undirected unweighted graph.

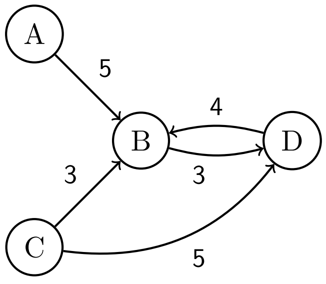

Figure 2: A directed weighted graph.

2 Adjacency Matrix

We have two main choices in how to represent graphs like these. 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 graph12 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 zeros. This format can be wasteful for graphs without a lot of edges. Graphs without many edges are considered sparse and are 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

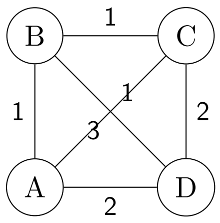

Figure 3: A weighted fully-connected graph.

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; you can convince yourself of this. [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 zeros. 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.

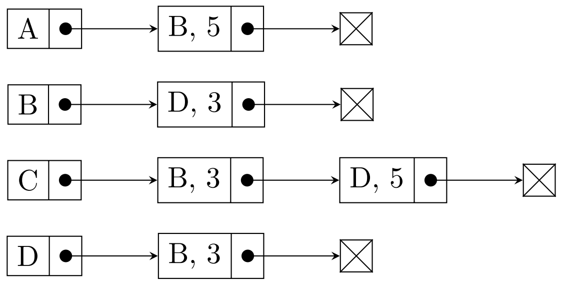

Figure 4: The adjacency list representation of a graph.

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.

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

```
def explore(G, V):
    marked[V] = true
    pre(V)
    for edge (v, u) in G connected to V:
        if not marked[u]:
            explore(G, u)
    post(V)
```

Figure 5: Explore routine on a graph and node.

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

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. It's also worth noting that this algorithm works on both directed and undirected graphs.

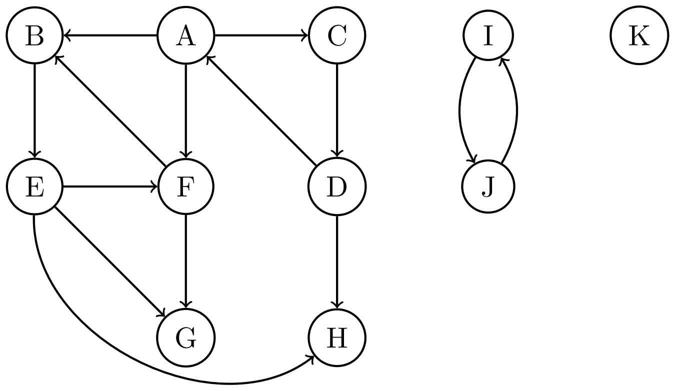

Figure 6: An example graph for explore.

Starting with A, we generate the following recursion tree

$$\begin{tikzcd}
 & & & A^{1,16} \arrow[dl] \arrow[dr] & \\
 & & B^{2,11} \arrow[dl] & & C^{12,15} \arrow[d] \\
 & E^{3,10} \arrow[dl] \arrow[dr] & & & D^{13,14} \\
F^{4,7} \arrow[d] & & H^{8,9} & & \\
G^{5,6} & & & &
\end{tikzcd}$$

Figure 7: 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.

```python
def dfs(G):
    for edge v in G:
        if not marked[v]:
            explore(G, v)
```

Figure 8: 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, there is a path

$$V \to ... \to U$$
$$V \to ... \to Z \to W \to ... \to U$$

Here, Z is the last marked vertex in the path and W is the first non-marked vertex. This is a contradiction. Z was called, meaning it must call explore on all not marked vertices with an edge from Z. W has an edge from Z, and it is not marked. Therefore `explore` does reach all vertices reachable from V.

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.

5 Cycles

This gives 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 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 not marked[u]:
            explore(G, u)
    t-marked[V] = false
```

Figure 9: 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

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

```
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 10: Top-Sort routine for topological sorting.

Consider the DAG below

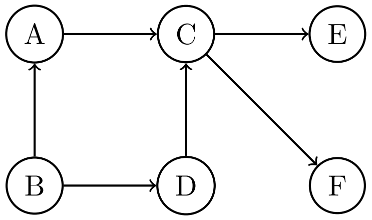

Figure 11: A directed acyclic graph.

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.

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

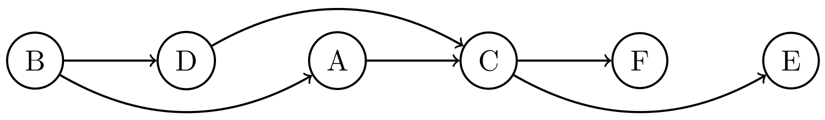

Figure 12: A linearization of a directed acyclic graph.

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

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 $O(|V| + |E|)$.