Colorado CSCI 5454: Algorithms **Fall 2020**

Depth-First Search and Topological Sort

Instructor: Bo Waggoner *Lecture 2*

This lecture defines *graphs*, a foundational abstraction. It then discusses depth-first search on graphs and uses it to find a topological sort.

Objectives:

- Understand the definitions of directed and undirected graphs, neighbors, paths, cycles, etc.
- Understand depth-first search (DFS) as a generic procedure, e.g. for solving Reachability.
- Know the definition of a directed acyclic graph (DAG) and topological sort; know how to use DFS to compute a topological sort of a DAG in linear time.

1 Graphs

Recall that graphs represent relationships between pairs of objects. Formally, a **graph** $G = (V, E)$ has a set of vertices V (also called nodes) and edges E. We usually say $|V| = n$ and $|E| = m$. If G is **undirected**, then an edge $e \in E$ is a pair of vertices, e.g. $e = \{u, v\}$, representing a connection between the two vertices u and v. It G is **directed**, then an edge $e \in E$ is an *ordered* pair of vertices, e.g. $e = (u, v)$, representing a connection from u to v.

We usually assume that our graphs do not have "self-loops", i.e. edges connecting u to itself, but such graphs are sometimes used.

Examples. An example of an undirected graph is a friendship network where the vertices are people and there is an edge $\{u, v\} \in E$ if u and v are friends with each other. An example of a directed graph is a hyperlink graph of the World Wide Web where the vertices are webpages and there is an edge $(u, v) \in E$ if page u contains a hyperlink to page v. Notice that edges (u, v) and (v, u) can both be present in E if both pages link to each other.

Paths and cycles. Recall[1] that a **path** in an undirected graph G is a nonempty sequence of vertices $v_1, \ldots, v_k$, where each vertex $v_i \in V$, such that for each pair of consecutive vertices v_i and v_{i+1}, there is an edge $\{v_i, v_{i+1}\} \in E$. The **length** of a path is the number of steps in the path, i.e. length of the sequence minus one. A **cycle** is a path of length at least one whose first and final vertex are the same. In a directed graph, these definitions are the same except that edge $(v_i, v_{i+1}) \in E$ is directed, pointing from v_i to v_{i+1}.

A path is **simple** if it does not contain any vertex more than once, and a cycle is **simple** if it does not contain any vertex more than once except the start/end vertex and does not re-use any edges.

Neighbors and degrees. In an undirected graph, we say v is a *neighbor* of u in a graph if there is an edge $\{u, v\} \in E$. In a directed graph, the term neighbor is ambiguous; one can use *out-neighbor* to refer to a v such that edge $(u, v) \in E$, and similarly for *in-neighbor*.

The *degree* of u is the number of edges incident to it, i.e. edges in which u appears as an endpoint. In a direct graph, we can also define the *out-degree* of u to be the number of edges from u to some other vertex, and the *in-degree* is the number of edges from some other vertex to u.

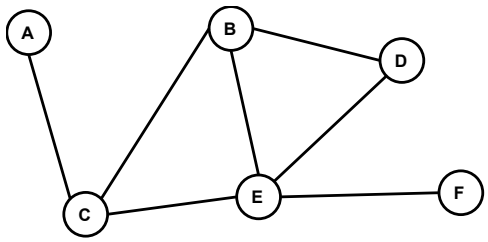
[1] These definitions are not universal; some places use them slightly differently and define the related terms *walk* and *trail*, which we won't use here.

Representation in word-RAM. One way to represent a graph as input to a word-RAM machine is as an *adjacency list*. We assume the vertices are numbered $1, \ldots, n$. Then the input consists of a list of out-neighbors of vertex 1, then 2, ..., n. If the graph is undirected, then the list simply consists of each vertex's neighbors.

With this representation, it takes constant time to look up the start of any vertex u's list (we can suppose the input begins with a list of pointers in order to make this true). To iterate through the list requires a number of iterations equal to the length of the list, of course. But in order to e.g. check whether there is an edge (u, v), we must iterate through the entire list, which can take time $O(n)$ in the worst case as u can have up to n neighbors.

The other popular representation is as an *adjacency matrix*. Here the input consists of an $n \times n$ matrix, flattened into a list, where the (u, v) entry is 1 if $(u, v) \in E$ and is 0 otherwise. In this case, it takes constant time to check whether there is an edge (u, v), since we can calculate the position of the entry in the input and read it immediately. However, to iterate through all of u's neighbors generally takes $O(n)$ time, even if u has much fewer neighbors, since we must check all n entries in the u row of the matrix.

Figure 1: An undirected graph.



Exercise 1. Consider the undirected graph of Figure 1. For each of the following, list its neighbors and state its degree.

(a) A

(b) B

(c) C

Exercise 2. Consider the undirected graph of Figure 1. For each of the following, is it a path, simple path, cycle, and/or simple cycle? (Name all that apply.) If it is a path, what is its length?

(a) A

(b) B, D, F

(c) A, C, A

(d) B, C, E, B

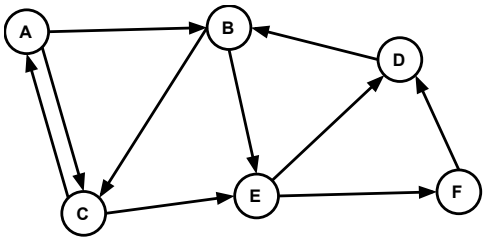
Exercise 3. Consider the directed graph of Figure 2. For each of the following, list its neighbors, state its degree, its in-degree, and its out-degree.

(a) A

(b) B

(c) C

Figure 2: A directed graph.



Exercise 4. Consider the directed graph of Figure 2. For each of the following, is it a path, simple path, cycle, and/or simple cycle? (Name all that apply.) If it is a path, what is its length?

(a) A, B, D

(b) B, C, E, B

(c) B, E, F, D, B

(d) A, C, A

(e) B, C, A, B, E, D, B

Exercise 5. In big-O notation, for a graph with $|V| = n$ and $|E| = m$, how large is the adjacency list representation (i.e. how much space is used)?

Exercise 6. And how large is the adjacency matrix representation?

2 Reachability

The Reachability problem is: given a directed graph $G = (V, E)$ and two vertices s, t, output True if there is a path from s to t, False otherwise. (Note that an algorithm that works on directed graphs will work on undirected graphs as well!)

We will suppose G is represented as an adjacency list. The question: design an algorithm for this problem and prove its correctness and running time.

Algorithm 1 DFS-Reachability

```
1: Input: Graph G = (V, E), vertices s, t
2: Define array is_marked of length n
3: Set is_marked[v] = False for v = 1, ..., n
4: Call DFS-explore(s)
5: Return is_marked[t]
```

Subroutine 2 DFS-explore(u)

```
1: Set is_marked[u] = True
2: for each out-neighbor v of u do
3:    if is_marked[v] == False then
4:       DFS-explore(v)
5:    end if
6: end for
```

Correctness - high-level proof: There are two things to prove: (1) if the algorithm outputs True, then there is a path from s to t; (2) if there is a path from s to t, then the algorithm outputs True. Observe that a vertex v is marked if and only if we call DFS-explore(v), because its first line is the only place we set `is_marked`$[v]$ = True.

First suppose there is a path from s to t, call it $v_1, \ldots, v_k$ with $s = v_1$ and $t = v_k$. We claim every vertex v_i in the path is marked, i.e. has DFS-explore called. Proof sketch by induction: for the base case, $s = v_1$ is marked by the first call to DFS-explore(s). For the inductive step, suppose $v_1, \ldots, v_j$ are marked. Then in the for loop of the call DFS-explore(v_j), we check the neighbor v_{j+1}. It is either already marked, meaning we already called DFS-explore(v_{j+1}), or we call it now. This completes the proof that every v_j in the path is marked, including t.

On the other hand, suppose the algorithm outputs True. Then at some point it called DFS-explore(t). One possible case is that $t = s$, in which case there is a trivial path from s to t. Otherwise, there is some v such that t is a neighbor of v and we called DFS-explore(v). Repeating the above logic for v, we get either $v = s$, or it is a neighbor of some u that was marked. Since the algorithm cannot run infinitely long, this process must terminate with s. Because each vertex is a neighbor of the previous one, this sequence gives a path, e.g. $s, \ldots, u, v, t$.

Running time - high-level proof: The body of DFS-Reachability takes $O(n)$ time, as it only needs to initialize the length-n array, call DFS-explore, and look up t in the array.

For DFS-explore, we first argue it is called at most once per vertex, so at most n times. This follows because we only call it on unmarked vertices, and every time we call it, we immediately mark v.

We analyze DFS-explore carefully be looking at each line and asking how many times it executes total over the entire course of the algorithm (this is the idea behind "amortized analysis"). Line 1 executes once per call, and we argued it is called at most n times, so this contributes $O(n)$ to the running time.

The body of each `for` loop, lines 3-5, each contribute a constant amount of operations. And the `for` loop executes once per edge out of v, in other words, the total amount of work in the form loop is $O(\text{out-degree}(v))$. Over the course of the entire algorithm, this totals at most

$$O\left(\sum_{v \in V} \text{out-degree}(v)\right) = O(m)$$

where m is the number of edges.

So we have shown that the algorithm's total running time is at most $O(n) + O(m) \in O(n + m)$.

Exercise 7. If the input graph is represented as an adjacency matrix rather than adjacency list, explain how the running time changes.

Exercise 8. If the input graph is undirected, explain what changes in the algorithm, proof of correctness, and proof of running time.

3 Topological sort

Definitions. A *directed, acyclic graph (DAG)* is a directed graph that has no cycles. Such graphs arise in many applications, especially where edges represent a dependency: vertex u must be visited, e.g. its task must be completed, before we move on to vertex v.

A permutation (or ordering) of the vertices is a bijective function $\pi : \{1, \ldots, n\} \to V$ where $\pi(1)$ is the first vertex in the ordering, $\pi(2)$ is the second, etc. Here $\pi^{-1}(u)$ is inverse function of π, which gives the location of u in the ordering. When we plug an index into π, it gives us a vertex. When we plug the vertex into π^{-1}, it tells us the index.

A *topological sort* of a DAG $G = (V, E)$ is a permutation π such that every edge in the graph points forward, i.e., for all edges $(u, v) \in E$, $\pi^{-1}(u) < \pi^{-1}(v)$. In other words, u must be located prior to v in the ordering.

As usual, the question is to give an algorithm for this problem and argue correctness and efficiency (which again is time and space usage).

Algorithm description. We maintain a list A, initially empty. We also start with each vertex unmarked. Then, we do a depth-first search from vertex 1. After visiting all of its neighbors, we add 1 to the beginning of A. We then do the same DFS from vertex 2 if it is not yet marked, then $3, \ldots, n$.

Meta note. If this were a homework or exam problem, then given that we have covered DFS in class and in the textbook chapter, the above description would be perfectly acceptable. To be clear, and because it's early in the class, we will also write out the algorithm formally below.

Algorithm 3 DFS-Topo

```
1: Input: Graph G = (V, E), vertices s, t
2: Define array is_marked of length n
3: Set is_marked[v] = False for v = 1, ..., n
4: Create list A, initially empty
5: for each vertex v do
6:    if is_marked[v] == False then
7:       DFS-explore-2(v)
8:    end if
9: end for
10: Return A
```

Subroutine 4 DFS-explore-2(v)

```
1: Set is_marked[v] = True
2: for each neighbor w of v do
3:    if is_marked[w] == False then
4:       DFS-explore-2(w)
5:    end if
6: end for
7: add v to beginning of list A
```

Correctness. We argue that every vertex is added to A exactly once, so it is a permutation. Then we argue that it is a topological sort, i.e. all edges point forward.

First, we add a vertex v to A only when we call DFS-explore-2(v). We only call DFS-explore-2(v) at most once per vertex, because we only call it if v is unmarked and then we immediately mark v. Finally, we call it for every vertex because of the `for` loop in DFS-Topo (line 5). So A is a permutation.

Now, consider any edge (u, v). We must argue that u is added to the list A *after* v is added to the list. Then, u will be earlier in the list than v.

At some point, we call DFS-explore-2(u). During the for loop, we reach neighbor v. There are two cases. If v is already marked, then we have already called DFS-explore-2(v) and it has completed. So v has already been added to list A. If v is not already marked, then we call DFS-explore-2(v) now and wait for it to complete. Then, v will have been added to list A. Only after this loop do we add u to the beginning of A, so in either case u is before v in the list. So it is a topological sort.

Running time. We claim that DFS-Topo runs in time $O(n+m)$, assuming the input graph is a DAG in adjacency list format.

Proof sketch: we just need to consider how the previous DFS analysis changes. We have added a `for` loop to DFS-Topo (line 5). However, the total work done in DFS-Topo is still $O(n)$, since we execute the loop n times. Furthermore, it is still true that DFS-explore-2 is called at most n times, and the analysis of its running time in lines 1-5 is the same, so the total work done is still at most $O(n+m)$. We now need to consider the amount of work required to build the list A. One implementation is to build it backwards: make A an array, and add each new vertex to the *end* of the array. This takes $O(1)$ time

per addition, so $O(n)$ work total. Then, at the end of the algorithm, we write A onto the output array in reverse, which takes $O(n)$ time. So the total running time is still $O(n+m)$.

Exercise 9. Suppose the algorithm only called DFS-Explore-2 on vertex 1 rather than looping through all vertices. How could it go wrong? Give an example input where the algorithm is incorrect.

Exercise 10. If a graph G is a DAG, then we know it has some topological ordering, because we proved our algorithm finds one. Prove the converse: if a graph is not a DAG, then it has no topological order. (Recall the definition of topological order).