Colorado CSCI 5454: Algorithms August 29, 2019

Lecture 2

Lecturer: Bo Waggoner *Scribe: Bo Waggoner*

Depth First Search

These notes give more formal proofs of topics covered in class. They are also examples of how homework solutions could be written.

1 Definitions

Recall that a directed graph $G = (V, E)$ has a set of vertices V and edges E. We usually say $|V| = n$ and $|E| = m$. We represent an edge $e \in E$ as an ordered pair $e = (u, v)$. Recall that if G is *undirected*, then whenever u and v have an edge between them, it's true both that $(u, v) \in E$ and $(v, u) \in E$.

Recall that a *path* in G is a sequence of vertices $v_1, \dots, v_k$, 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 edges in the path, i.e. number of vertices minus one.

A *cycle* is a path whose first and final vertex are the same. 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. (Note in an undirected graph, (u, v) and (v, u) are considered the same edge.)

In an undirected graph, we say v is a *neighbor* of u in a graph if there is an edge (u, v); and the *degree* of u is the number of neighbors it has.

In a directed graph, we usually say v is a *neighbor* of u if either edge (u, v) or edge (v, u) or both are present. In this case, the *out-degree* of u is 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.

2 Reachability

The Reachability problem is: given a graph $G = (V, E)$ and two vertices s, t, output True if there is a path from s to t, False otherwise.

We will suppose G is represented as an adjacency list.

Question: design an algorithm for this problem and prove its correctness and running time.

Algorithm 1 DFS-Reachability

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

Subroutine 2 DFS-explore(v)

```
Set is_marked[v] = True
for each neighbor w of v do
    if is_marked[w] == False then
        DFS-explore(w)
    end if
end for
```

Correctness - proof sketch: We show that the algorithm outputs True if and only if there is a path from s to t. The algorithm outputs True if and only if t is marked at some point, which occurs if and only if we call DFS-explore(t) at some point. This is true if and only if DFS-explore(v) was called on some v where there is an edge (v, t). This is true if and only if either (1) $v = s$ or (2) DFS-explore(u) was called on some vertex u where there is an edge (u, v). By repeating this argument for u and so on, we see that DFS-explore(t) is called if and only if there is a sequence of vertices starting at s and ending at t, for example s, u, v, t, such that there is an edge between each pair of consecutive vertices: a path from s to t.

Note. The above is not a fully formal proof, but would be a good proof sketch to give on a homework. A fully formal proof would use e.g. induction.

Running time - proof sketch: 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 $O(n) + O(m) \in O(n + m)$.

3 Topological sort

A *directed, acyclic graph (DAG)* is a directed graph that has no cycles.

A permutation (or ordering) π of the vertices is a function where $\pi(1)$ is the first vertex in the ordering, $\pi(2)$ is the second, etc.

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.

Here $\pi^{-1}(u)$ is inverse function of π, which gives the location of u in the ordering.[1] When we plug a location into π, it gives us a vertex. When we plug the vertex into π^{-1}, it tells us the location.

Algorithm 3 DFS-Topo

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

[1] Thanks to a student for mentioning that, in lecture, I forgot to write the inverse signs. -Bo.

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
```

Algorithm description - note for homework. If this were a homework problem, then given that we have covered DFS in class and in the textbook chapter, the following would be a good description of the algorithm. "We do a depth-first search from all vertices in order, skipping them if they are already marked. We maintain a list A, initially empty. When the depth-first-search visits a vertex, after iterating through all of its neighbors, we add it to the beginning of the list A."

Correctness - proof sketch. 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 (v, w). We must argue that v is added to the list A *after* w is added to the list. Then, v will be earlier in the list than w.

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

Running time - proof sketch. We use the previous DFS analysis. Here, 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 $O(n + m)$.