CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 15

March 10, 2022

Instructor: Sepehr Assadi

1 Algorithms for Graph Search: Part One

Recall the graph search problem from the last lecture:

Problem 1 (Graph Search (Undirected)). Suppose we are given an undirected graph $G = (V, E)$ and a starting vertex $s \in V$. We define the *connected component* of s in the graph G as the set of vertices in G that are *reachable* from s, namely, the vertices that can be reached from s by following a *path* in G.

Our goal in this problem is to design an algorithm that given the graph $G = (V, E)$ and a starting vertex s, outputs all the vertices in the connected component of s in G.

There are twi canonical algorithms for graph search: the *depth-first-search (DFS)* algorithm, and the *breadth-first-search (BFS)* algorithm. We are going to describe these algorithms for undirected graphs. However, they work *exactly the same way* for directed graphs as well as you will see shortly (in a directed graph, the graph search problem asks for finding all vertices reachable from s, i.e., the ones that s has a path to; note that in directed graphs, the set of vertices that are reachable from s is not necessarily the same as vertices that can reach s and thus there is no notion of a connected component in directed graphs). We cover DFS algorithm in this lecture and BFS in the next one.

1.1 Depth-First-Search Algorithm

Our first algorithm for graph search is called *depth-first-search* or *DFS* for short. The algorithm is simply as follows: we start from the vertex s and go to its first neighbor, say v_1, then go to vertex v_1 and recursively do the same thing, i.e., go to the first neighbor of v_1 and so on and so forth. A problem with this approach? We may end up in a loop eventually by just going to the *same set of vertices* again and again, so how can we fix this? Well of course by dynamic programming! We simply store which vertices we have visited and next time when we visit them we simply ignore them.

More formally, the algorithm is the following:

Algorithm: Initialize an array $mark[1:n]$ with 'FALSE'. Run the following recursive algorithm on s, i.e., return `DFS`(s):

```
DFS($v$) :
  1. If $mark[v]$ = 'TRUE' terminate.
  2. Otherwise, set $mark[v]$ ='TRUE' and for every neighbor $u \in N(v)$ (this is the list of neighbors of $v$
     that we have direct access to in the adjacency list representation):
       • Run DFS($u$).
```

At the end, we return all vertices v with $mark[v]$ = 'TRUE' as the answer, i.e., as vertices that are in the same connected component of s.

Proof of Correctness: We need to prove that every vertex connected to s will be output and no other vertex is also output by this algorithm. The second part is straightforward because we are only visiting s, neighbors of s, their neighbors and so on and so forth, so for any vertex marked, there is a path from s to that vertex in the graph.

The other part is also easy: consider any unmarked vertex u. We prove that u is not in the connected component of s. Since u is unmarked, none of the neighbors v of u can be marked: otherwise, when we visited v for the first time, we would have marked u also at some point later (because we run `DFS`(u) for that vertex when visiting v, i.e., in `DFS`(v)). We can then continue like this to all neighbors of v (since they are not marked either) and say that all neighbors of v needs to be unmarked also. We expand this until we find all of the vertices that are connected to u and we know that they are all unmarked. But this means that s was not any of those vertices since s is actually marked; hence u cannot be part of the connected component of s since there is no path from s to u (again, this proof is simply a proof by induction although we were indeed rather informal about it here; do you see how?).

Runtime Analysis: We visit each vertex v *at most* once (after that it is marked and we do not spend more time in visiting the vertex (think of memoization)) and it takes $O(|N(v)|)$ time to process this vertex. Hence, the total runtime is at most $c \cdot \sum_{v \in V} |N(v)|$ for some constant c: since the total degree of vertices is proportional to the number of edges, the runtime of this algorithm is $O(n + m)$.

Further Extensions

There are various extra things one can do in the DFS algorithm. For instance, we can use it to return a *spanning tree* of the connected component of s, namely, a subgraph of G (a graph on some subset of vertices and edges of G) that connects all vertices in the connected component of s to each other and furthermore has no cycle[1] (a tree is simply a graph with no cycle).

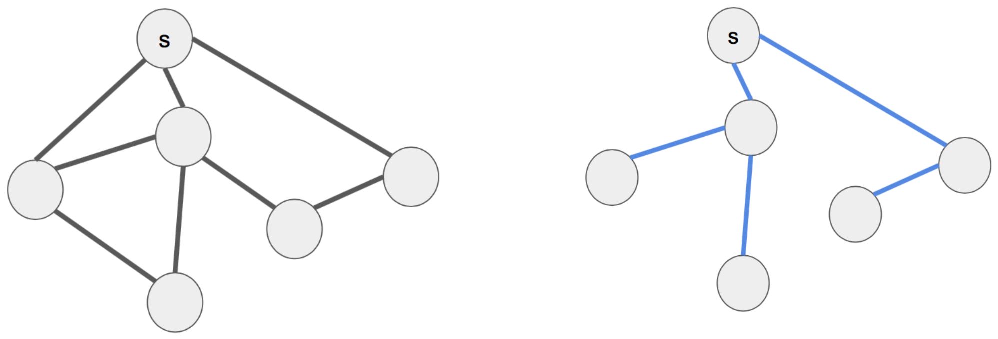

Figure 1: An illustration of a graph G on the left and one of its spanning trees on the right for the connected component of s (here s is connected to all vertices).

How to find a spanning tree using a DFS? We slightly tweak the algorithm so that when we are calling `DFS`(u) from the vertex v (inside `DFS`(v)), we directly check there whether u is marked or not; if it is marked we ignore it right away (instead of calling it recursively and then check in the recursive call if it is marked or not). If it is not marked however, we pick the edge (v, u) inside our tree and recursively call `DFS`(u). The algorithm can be described formally as follows:

Algorithm: Initialize an array $mark[1:n]$ with 'FALSE'. Let $T[1:n]$ be an array of *lists* so that $T[v]$ contains the list of neighbors of v inside the *tree* we want to return (originally $T[v]$ is an empty list for all v). Run the following recursive algorithm on s, i.e., return `DFS-TREE`(s):

[1]A cycles is a sequence of vertices $v_0, v_1, \ldots, v_k$ where $v_0, \ldots, v_{k-1}$ is a path, (v_{k-1}, v_k) is an edge, and $v_0 = v_k$. In other words, it is a "path" that starts and ends in the same vertex (the reason it is not correct to call it a path is that in a path we are not allowed to repeat a vertex).

```
DFS-TREE($v$) :

  1. Set $mark[v]$ =‘TRUE’,
  2. For every neighbor $u \in N(v)$:
     (a) If $mark[u]$ = ‘TRUE’ go to the next neighbor.
     (b) Otherwise, add $u$ to the list $T[v]$ and run DFS-TREE($u$).
```

How to Find a s-t Path (even in Directed Graphs) using DFS? As we said earlier, the DFS algorithm works as it is for directed graphs as well. The only difference is that instead of finding a connected component of s (which is undefined for directed graphs), it finds the set of all vertices *reachable* from s, i.e., vertices such that there is path *from* s to them[2].

In undirected graphs, once we have a spanning tree of a connected component, we can find a path between any two vertices in the same connected component since in a tree, there is a unique path that connects these two vertices (simply start from the two vertices, go to their parent, their parent's parent and so on until these two reaches the same vertex, then return the vertices as the path). For directed graphs, a spanning tree is undefined again (the same way that a connected component was undefined). However, still we can find a path from the vertex s to any other vertex t reachable from s by doing DFS. The strategy is exactly the same as `DFS-TREE` by storing the edges that are going from a vertex v to its *unmarked* neighbor u that we are going to do a DFS over and calling v the *parent* of u. Then by following the parent of each vertex u repeatedly until we reach s, we obtain an s-u path in G (after reversing the order of this sequence).

[2]but not necessarily a path from those vertices to *to s* – for undirected graphs these two concepts are the same since we can traverse a path both ways, for directed graphs however they are different.