CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 14

October 24, 2019

Instructor: Sepehr Assadi

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

1 Graph Reductions

As we said earlier, graphs are perfect tools for modeling other problems. Before we get into any algorithm, let us first see an example of this. Consider the following problem (see Figure 1 for an illustration):

Problem 1 (**Fill Coloring Pixel Maps**)**.** Suppose we are given a pixel map as input, that is, a two dimensional array (or matrix) $A[1:k][1:k]$ with entries equal to either 0 or 1. When *fill coloring* this pixel map, we start from some given cell $s = (i_s, j_s)$ with $A[i_s][j_s] = 0$, color that cell, then go to any neighboring cell (left, right, top, and bottom) and for any of these cells (i, j), if $A[i][j] = 0$, we color those cells, and continue coloring their neighboring cells and so on (if $A[i][j] = 1$ we no longer consider its neighbors although they may get colored from their other neighbors eventually).

Our goal in this problem is to design an algorithm that given the matrix A and a starting cell $s = (i_s, j_s)$ with $A[i_s][j_s] = 0$, outputs the list of all cells that would be colored using this fill coloring operation.

We can certainly design an algorithm from scratch for this problem. However, our goal here is to introduce the notion of *reduction* (in particular in the context of graphs). To do this, let us consider the following problem as well:

Problem 2 (**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[1].

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.

We proposed the following reduction for solving the first problem.

Reduction (Algorithm): Given an instance of the fill coloring problem (i.e., an input 2D-array A and starting cell s), we design the following graph $G(V, E)$:

1. Vertices: for any cell (i, j) in the array A, we create a new vertex v_{ij} in set V;
2. Edges: for any two cells (i, j) and (x, y) that are neighboring to each other, if $A[i][j] = A[x][y] = 0$, we add an edge (v_{ij}, v_{xy}) to edge-set E.

Moreover, we let $s \in V$ be the vertex $v_{i_s j_s}$ for the starting cell (i_s, j_s) defined in the fill coloring problem. We then run the graph search algorithm on this new graph G and vertex s and for any vertex v_{ij} that is in the same connected component as s in G, we output the cell (i, j) in the answer to the fill coloring problem.

[1]A path P in a graph $G(V, E)$ starting from a vertex s to a vertex t is a sequence of vertices $P = v_0, v_1, v_2, \ldots, v_k$ where $v_0 = s$, $v_k = t$, no vertex is repeated in this sequence, and for every $1 \leq i \leq k$, the edge (v_{i-1}, v_i) belongs to the graph.

Proof of Correctness: We prove that a vertex $v_{i,j}$ in the newly created graph is connected to vertex s, if and only if fill coloring the original matrix A starting from cell (i_s, j_s) results in coloring the pixel (i, j). We will be done after proving this since the output of the reduction/algorithm is the connected component of s in G which translates to all pixels (i, j) that will be colored *and* only those pixels.[2]

The first direction: let $s, v_{i_1 j_1}, \ldots, v_{i_t j_t}$ be any path in the new graph G. Then we know that: (1) each cell (i_ℓ, j_ℓ) is neighbor to $(i_{\ell+1}, j_{\ell+1})$ (by the first property of adding an edge to G) and moreover, $A[i_\ell][j_\ell] = A[i_{\ell+1}][j_{\ell+1}] = 0$ (by the second property). Hence, fill coloring A starting from (i_s, j_s) would eventually color the pixel $A[i_t][j_t]$ on this path. This means that any pixel corresponding to any vertex reachable from vertex s will be fill colored correctly (we do not output any "wrong" pixel).

Now for the second direction: let $A[i_t][j_t]$ be any pixel that must be fill colored. By definition, there should exists a sequence of pixels $(i_s, j_s), (i_1, j_1), \ldots, (i_t, j_t)$ that are all marked 0 and (i_ℓ, j_ℓ) and $(i_{\ell+1}, j_{\ell+1})$ are neighboring to each other. Thus, if we consider their corresponding vertices $s, v_{i_1 j_1}, \ldots, v_{i_t j_t}$, this forms a path in the constructed graph G. This means that any pixel that should be fill colored will be endpoint of a path from s in the new graph and hence is output (we output all the "right" pixels).

This concludes the proof of correctness.

Runtime Analysis: The runtime of this algorithm is equal the time needed to construct the graph (which we can do by nested for-loops over $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, k\}$) which is $O(k^2)$ and the time needed to run the *fastest* graph search algorithm that we know. At this point, we have not yet seen an algorithm for the latter problem so we may not be able to answer this question yet (we will study graph search in this lecture so by the end we indeed know the answer).

But there is an important lesson here: by designing a reduction like this, the running time of our algorithm for the fill coloring problem is (asymptotically) the same as the runtime of the fastest graph search algorithm (the other parts of reduction take linear time in input which is always necessary to just read the input). This means that *any* improvement on the running time of graph search algorithms would automatically carry over to our fill coloring problem also with no further effort on our side[3].

2 Graph Representations

In order to study graph algorithms in detail, we need to specify how the input graph is given to the algorithms. There are two general way of specifying a graph, called *adjacency list* and *adjacency matrix* (these are not the only ways in general, but we will not consider other ways in this course).

Adjacency list: For every vertex $v \in V$, we store a list $N(v)$ of all the neighbors of v. We store these lists themselves in an array, say A, index by each vertex $v \in V$, i.e., $A[v]$ is a list itself. This way, we can access the list of any given vertex $v \in V$ in constant time and then iterating over this list takes time proportional to the number of neighbors of v (for the purpose of this course, you can assume vertices are $V = \{1, \ldots, n\}$, namely, we simply write their name as 1, 2, etc. so that we can access them easily in an array). This means reading (or writing) the entire graph in adjacency list takes $O(n + m)$ time. (For directed graphs, we store *out-neighbors* of v in $N(v)$, namely vertices to which v has a directed edge).

Adjacency matrix: We store a matrix $M[1:n][1:n]$ where $M[i][j] = 1$ if there is an edge between the i-th vertex and j-th vertex (i.e., $\{v_i, v_j\} \in E$ for undirected graphs and $(v_i, v_j) \in E$ in directed graphs) and is otherwise 0. This way we can find whether there is an edge between two vertices v_i and v_j in constant time by checking $M[i][j]$. However, iterating over all neighbors of a vertex in this format takes $O(n)$ time.

[2]It is quite common that when proving the correctness of a reduction, one forgets to prove that we are actually "not outputting things that we should not"; please make sure this does not happen for you (at least too often!).

[3]As we will see soon, this is a rather moot point for this particular problem as we already know the *fastest possible* algorithm for graph search (asymptotically). However, for many other problems (some of which we will later visit), we (the entire scientific community) have not been able to design the fastest possible algorithm yet (or if we did, we do not know it (!) since we cannot prove their runtime is indeed fastest): by doing these type of reductions, we will ensure that if at any point in the future someone is able to improve the runtime of algorithms for those problems, the runtime of our problem will also improve automatically.

This means reading (or writing) the entire graph in adjacency matrix takes $O(n^2)$ time.

Throughout the course, unless explicitly stated otherwise, when working with graphs, we always assume the input is given to the algorithm **in the adjacency list representation**.

3 Graph Search

We are going to describe our graph search algorithms for undirected graphs. However, they work *exactly the same way* for directed graphs as well as you will see shortly.

3.1 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 $\texttt{DFS}(s)$:

$\texttt{DFS}(v)$:

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 $\texttt{DFS}(u)$.

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 $\texttt{DFS}(u)$ for that vertex when visiting v, i.e., in $\texttt{DFS}(v)$). 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[4] (a tree is simply a graph with no cycle).

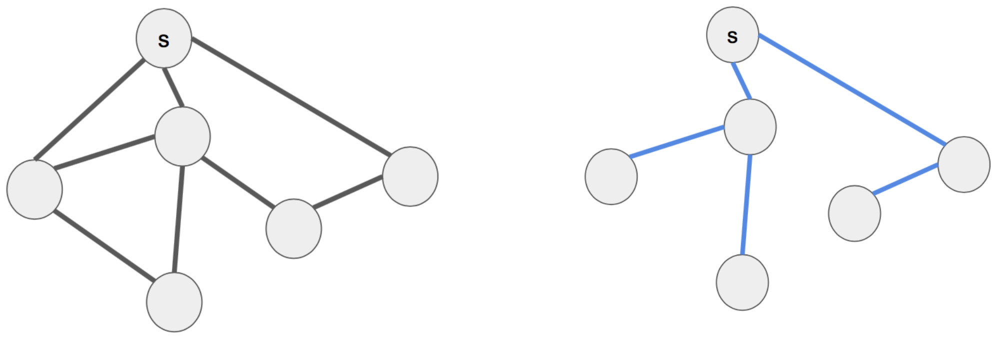

Figure 1: An illustration of a graph G on the left and one of its spanning trees on the right for the connected component of s (here s is connected to all vertices).

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):

```
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[5].

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

[4]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).

[5]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.

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).

3.2 Breadth-First-Search Algorithm

The second main algorithm for doing graph search is *breadth-first-search* or *BFS* for short. We will go over this algorithm in the next lecture.