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

Lecture 14

March 8, 2022

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 Intro to Graph Algorithms

We now start the next chapter of our course: the truly fascinating area of *graph algorithms*.

You can think of a graph simply as a collection of objects (people, houses, webpages, $\cdots$) together with some pairwise relations between these objects (pairs of people who are friend, pairs of houses that are connected by a road, pairs of webpages that cite each other, $\cdots$). Because of this, graphs are extremely helpful in *modeling* various scenarios (think of friendship graphs, maps, or webgraph, in the context of previous examples).

In this part of the course, we are going to both learn how to solve different problems on graphs as an abstraction for various other problems, as well as how to formally model a problem as a graph problem, typically referred to as a (graph) *reduction*, and solve them using these algorithms.

Basic Definitions

An *undirected* graph G is a pair of sets (V, E) where V is the set of *vertices* of the graph, and E is a set of *edges* of the graph, where each edge is simply an *unordered* pair of vertices, i.e., each edge $e = \{u, v\}$ for two different vertices $u, v \in V$. Note that under this definition, there can only be one edge between any pair of vertices, and there are no edges between a vertex and itself (these are often called self-loops)[1].

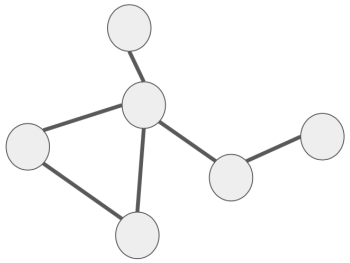

Figure 1: An example of an undirected graph. The circles are vertices and the lines are edges.

A *directed* graph G is also defined very similarly as a pair of sets (V, E); the only difference is that now each edge E is an *ordered* pair of vertices, i.e., each edge $e = (u, v)$ for two different vertices $u, v \in V$. Under this definition, there can only be one *direected* edge between a vertex u and a vertex v (but we may have both the (u, v) edge and the (v, u) edge); as before, there are no-self loops (i.e., edges of the form (u, u)).

Throughout this course, we use n to denote the number of vertices and m to denote the number of edges. We note that for any undirected graph, $m \leq \binom{n}{2} = \frac{n \cdot (n-1)}{2}$ and for a directed graph, $m \leq n \cdot (n-1)$ (why?).

There are various definitions about the graphs including, neighbors, degrees, paths, walks, connected components, trees, etc. Not all of these definitions are covered in the lectures/lecture notes but you can find them

[1] Although we will eventually consider removing these restrictions as for the most part they will *not* change anything in the algorithms we study.

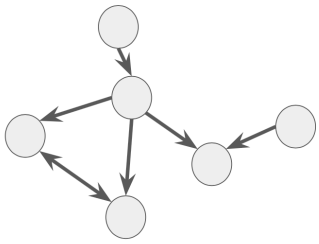

Figure 2: An example of a directed graph. The circles are vertices and the lines are edges.

in the Chapter 5.2 of your textbook. We provide some of these definitions and preliminaries in Appendix A at the end of this lecture; you are strongly encouraged to review them carefully.

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, indexed 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 the adjacency list representation 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. 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 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 3 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

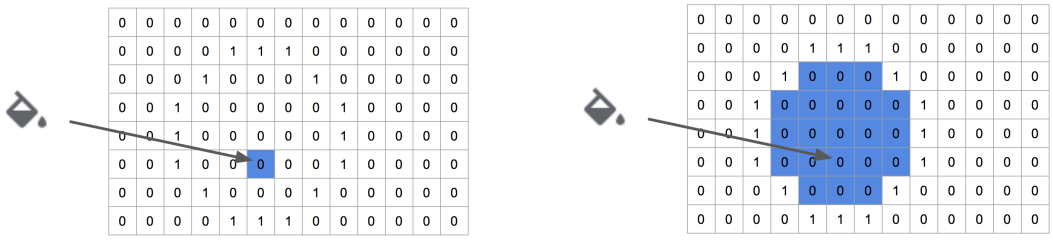

Figure 3: An illustration of the problem of fill coloring pixel maps. The blue cell in the left shows the starting cell s and the blue cells on the right are all the cells that would be colored if we start from cell s.

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[2].

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.

Both problems seem quite similar to each other except that the latter one might be considered much more general than the former. Indeed, it looks like that if we know how to solve the second problem, we should be able to solve the first one as well. But how do we formalize this? *Reductions!*

We will show that given any instance of the fill coloring problem, we can design an instance of the graph search problem so that the solution to the graph search problem uniquely identifies the solution to our original fill coloring problem. This approach has a significant benefit: even if we have absolutely no idea how to solve the graph search problem ourselves, as long as someone gives us an algorithm for this problem, we can simply use it as a black-box and solve the fill coloring problem also!

Reduction: Our reduction in this case looks as follows. 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 the 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.

We also need to specify the vertex s in the graph search problem: for that, 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. This is now a valid input to the graph search 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. This finalizes our reduction (algorithm).

Remark: A reduction is simply another algorithm for the problem we want to solve (here, fill coloring problem) that uses *any* algorithm for the problem that we want to reduce to (here, graph search problem).

This concludes the description of our reduction. We are still not done though; we also need to prove the correctness of the reduction (the same way we prove correctness of any other algorithm) and analyze its runtime.

[2]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.[3]

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 the next lecture so by the end of it 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[4].

[3] 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!).

[4] As we will see in the next lecture, 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.

Appendix

A Graphs: Definitions and Preliminaries

We review some basic definitions and preliminaries on graphs here. For more information, see Chapter 5.2 of your textbook.

Undirected Graphs

An undirected graph G is a set of *vertices* V together with a set of *edges* E; an edge is an (unordered) pair of distinct vertices. We abbreviate the edge $\{u, v\}$ as uv. Graphs are of two kinds: undirected graphs and directed graphs. In undirected graphs, the edge uv is the same as the edge vu, which should be clear from its definition.

We will always use n to denote the number of vertices of a graph and m to denote the number of edges. Note that an algorithm which runs in time $O(n + m)$ is *linear* in the size of the input (which may not be linear in n).

Basic Terminology

- We say that vertices u and v are the *endpoints* of the edge uv and that uv *joins* u and v.
- A vertex u is *adjacent to* a vertex v if u and v are joined by some edge.
- We say an edge e is *incident* to a vertex v if v is one of the endpoints of e.
- The *degree* of a vertex u, written $\deg(u)$, is the number of edges incident to it.
- The *neighborhood* of a vertex v, denoted $N(v)$, is the set of adjacent vertices to v – any vertex $u \in N(v)$ is also called a *neighbor of* v.
- A *walk* is a sequence P of vertices $v_1, \ldots, v_{k+1}$ such that $v_i v_{i+1}$ is an edge for all $i \in \{1, \ldots, k\}$. A *path* is a walk in which all vertices are distinct. We say that P is a v_1-v_k *path*. The *length* of a such a walk (path) is k, the number of edges traversed.
- A *closed walk* is a walk that starts and ends at the same vertex. A *cycle* is a closed walk of length at least 3 in which all vertices are distinct, except of course at the first and last vertex.
- The *distance* between two vertices u and v is the length of the shortest path between them.
- Consider the equivalence relation in which two vertices are related if there is a path between them (you should verify that this is an equivalence relation). The equivalence classes of this relation are called *components* of G. That is, the components of G are the maximally connected subgraphs.
- A graph is *connected* if there is a path between any two vertices $u \in V$ and $v \in V$. Equivalently, a graph is connected if it contains exactly one component.
- A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- An (undirected) graph without any cycle is a *forest*.
- A forest with one component is a *tree*. Alternatively, a tree is an undirected acyclic connected graph.
- A *leaf* of a tree is a vertex of degree 1.
- A *spanning tree* of a graph $G = (V, E)$ is a subgraph $T = (V, F)$ which is a tree. Note that spanning trees are on the *same vertex set*.
- A graph is *bipartite* if its vertex set can be partitioned into two classes A and B such that all of the edges of the graph go between A and B.

- A graph $G = (V, E)$ is *k-colorable* if there is a color assignment $c : V \to [k]$ such that $uv \in E \implies c(u) \neq c(v)$. It is easy to verify that a graph is bipartite if and only if it is 2-colorable.
- A graph is *d-regular* when all of its vertices have degree d.
- A *clique* is a set of vertices in a graph which are all connected to each other.

Directed Graphs

A directed graph (sometimes called a digraph) is a set of *vertices* V together with a set of *arcs* A; an arc is an *ordered* pair of distinct vertices. The difference between directed graphs and undirected graphs is precisely that arcs are ordered, whereas edges are not. We abbreviate the arc (u, v) as $\overrightarrow{uv}$. We may abuse the notation and refer to arcs as edges as well whenever it is clear from the context.

Observe that undirected graphs can be viewed as a special case of directed graphs, since for every edge uv in we can create two arcs $\overrightarrow{uv}$ and $\overrightarrow{vu}$.

Basic Terminology

- We say that the arc $a = \overrightarrow{uv}$ *enters* v and *leaves* u. We also call u the *tail* of a and v the *head* of a.
- The *in-degree* of a vertex is the number of arcs entering it. The *out-degree* of a vertex is the number of arcs leaving it.
- A *source* is a vertex of in-degree 0, and a *sink* is a vertex of out-degree 0.
- The notions of walks, paths, cycles, distance, etc., are the same as in undirected graphs, except that we must follow the orientation of the now directed edges.
- We say that a vertex u is *reachable* from a vertex v if there is a directed path from v to u; in that case, we may also say that v can *reach* u.
- A directed graph is *strongly connected* if for any two vertices u and v, there is a directed path between them.
- A *directed acyclic graph* or a *DAG* is a directed graph which contains no directed cycle.
- The *transitive closure* of a directed graph $G = (V, A)$ is a directed graph on V where there is an arc from u to v whenever G contains a path from u to v.

Properties

- The Handshaking Lemma: For any undirected graph G, the sum of the degrees of its vertices is twice its number of edges, i.e., $\sum_{v \in V} \deg(v) = 2m$. This lemma gets its name from the following observation: if you sum up the number of hands shaken by each person during the course of some event, the result will be an even number.
- For any undirected graph G, the number of vertices of odd degree is even. (This is an immediate corollary of the handshaking lemma.)
- Every directed acyclic graph (DAG) G has at least one source and one sink. Moreover, if there is *exactly* one source in G, then all vertices of G are reachable from this sink; similarly, if there is *exactly* one sink, then all vertices of G can reach this sink.
- A forest with n vertices and k components has exactly $n - k$ edges. An immediate corollary is that a tree on n vertices has $n - 1$ edges.
- An undirected graph is bipartite if and only if it does not contain an odd cycle (as a subgraph).
- An undirected graph has between 0 and $\binom{n}{2}$ edges. A directed graph has between 0 and $n \cdot (n - 1)$ edges.

Graph Representations

There are two common ways to represent graphs: the adjacency list and the adjacency matrix.

In the adjacency list representation, we maintain n lists, one for each vertex. The list corresponding to vertex u contains all vertices that are adjacent to u in the graph. For a directed graph, the list corresponding to vertex u contains all vertices v such that $\overrightarrow{uv} \in A$ (that is, the arcs leaving u in G).

As the name suggest, the adjacency matrix representation of G is a matrix that encodes the adjacency relationships between all pairs of vertices in the graph G. Concretely, the adjacency matrix A is the $n \times n$ matrix such that $A(i, j) = 1$ if $ij \in E$ and $A(i, j) = 0$ otherwise.

Weighted Graphs

A *weighted graph* is a graph $G = (V, E)$ in which every edge has an associated weight. Formally, have a function $w : E \to \mathbb{R}$. We may also speak of weighted directed graphs, which are exactly the same but for directed graphs; in the directed setting, the weight of $\overrightarrow{uv}$ need not be the same as the weight of $\overrightarrow{vu}$.