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

Lecture 13

October 21, 2019

Instructor: Sepehr Assadi

1 Maximum Disjoint Intervals

Let us recall the maximum disjoint intervals problem from the last lecture.

Problem 1 (**Maximum Disjoint Intervals**)**.** We are given a collection of n intervals specified by their two endpoints: $[a_1, b_1], [a_2, b_2], \cdots, [a_n, b_n]$. We say that two intervals $[a_i, b_i]$ and $[a_j, b_j]$ are *disjoint* if there is no point that belongs to both of them. Our goal in this problem is to find the *maximum* number of intervals in the input that are all disjoint from each other.

Algorithm:

1. Sort the intervals based on their *last point* (i.e., b_i for interval $[a_i, b_i]$) in the non-decreasing order.
2. Pick the first interval in the solution. Go over this ordering until you find the next interval that does not intersect the previously chosen interval and pick it in the solution. Continue like this until you iterate over all the intervals.

Proof of Correctness: We have to prove both that the output of the algorithm is a valid solution (the intervals it outputs are all disjoint) and that it is also optimal (it contains the largest number of disjoint intervals).

Let $G = \{g_1, \ldots, g_\ell\}$ be the set of intervals chosen by the algorithm. By construction of the algorithm, we know that each g_i is disjoint from g_{i-1}. Thus, we only need to show that g_i is also disjoint from g_j for $j < i$ (we do not need to worry about $j > i$; do you see why?). This is true because g_i's starting point is larger than g_{i-1}'s endpoint, while g_{i-1}'s endpoint is larger than the endpoint of all intervals g_j for $j < i$; thus g_i is also disjoint from all g_j.

We now prove the optimality of the algorithm by an exchange argument. Let $O = \{o_1, \ldots, o_k\}$ be any optimal solution (we do not know a priori whether $\ell = k$ or not – in fact, this is what we want to prove).

If $G = O$ we are already done since it means the greedy algorithm is also optimal. So let us assume that $G \neq O$. We are going to also sort the intervals in O based on their endpoint so that $b_{o_1} \leq b_{o_2}, \ldots, b_{o_k}$; this clearly does not change the solution O and its size. Let j be the index where $g_1 = o_1, \ldots, g_{j-1} = o_{j-1}$ but $g_j \neq o_j$: note that such an index always exist since $G \neq O$ and also $j \leq \ell$ because if G and O are equal in their first ℓ choices, it cannot be the case that size of O is larger than ℓ because if there was an interval disjoint from $g_1, \ldots, g_j$ (in particular o_{j+1}), we would have picked it in G also.

We now define a new "solution" $O' = \{o'_1, \ldots, o'_k\}$ obtained from O by exchanging o_j with g_j, i.e., for all $i \neq j$, $o'_i = o_i$ but $o'_j = g_j$ instead. We call O' "solution" at this point because it is not clear yet whether it is indeed a valid solution to the problem or not: we need to ensure that all intervals in O' are disjoint. To prove this, note that since we only changed o_j to g_j, we need to prove $o'_j = g_j$ does not intersect with any other interval in O'. To prove this we have:

- g_j does not intersect with $g_1, \ldots, g_{j-1}$ by definition of the greedy algorithm.

- g_j has the smallest endpoint among all intervals that are *not* disjoint from $g_1, \ldots, g_{j-1}$; this is again by definition of the greedy algorithm after we sort the intervals based on their endpoint.
- o_j is disjoint from $g_1, \ldots, g_{j-1}$ because O is a solution to the problem and $\{o_1, \ldots, o_{j-1}, o_j\} = \{g_1, \ldots, g_{j-1}, o_j\}$, so o_j cannot intersect with the previous intervals chosen in O.
- The two points above imply that endpoint of g_j can only be smaller than the endpoint of o_j. On the other hand, starting point of any interval $o_{j+1}, \ldots, o_k$ can only be larger than the starting point of o_j (otherwise they will intersect). This implies that g_j is also disjoint from $o_{j+1}, \ldots, o_k$.

The last two points above imply that O' is also a valid solution to the problem indeed. Now, since size of O and O' are the same, we have that O' is another optimal solution. But now O' has its first j indices equal to G (hence O' is "more similar" to G that O was). We are done now as we can repeat this argument again and again (exactly as before) until we end up with G itself (by changing all the ℓ indices of the solution to become equal to G): this proves the correctness of the algorithm.

Runtime Analysis: The algorithm involves sorting n intervals by their endpoints which can be done in $O(n \log n)$ time and then iterating over intervals in $O(n)$ time. Hence, the total runtime is $O(n \log n)$.

2 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].

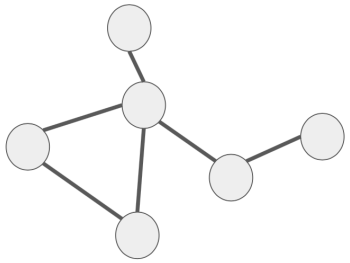

Figure 1: An example of an undirected graph. The circles are vertices and the lines are edges.

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

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

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)).

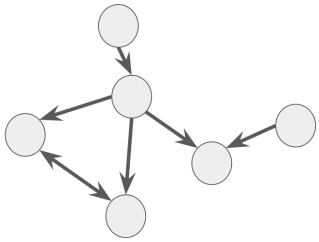

Figure 2: An example of a directed graph. The circles are vertices and the lines are edges.

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 in the textbook as well as Chapter 5.2 of the suggested reading book.

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 2 (**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.

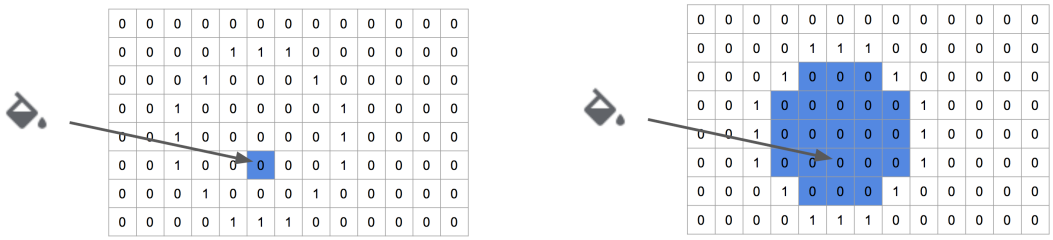

Figure 3: An illustration of the problem of fill coloring pixel maps. The blue cell in the left shows the starting cell s and the blue cells on the right are all the cells that would be colored if we start from cell s.

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 3 (**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 ourself, 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 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. 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.

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. This is done in the next lecture.

[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.