Lecture 13

Graph Algorithms I

13.1 Overview

This is the first of several lectures on graph algorithms. We will see how simple algorithms like depth-first-search can be used in clever ways (for a problem known as *topological sorting*) and will see how Dynamic Programming can be used to solve problems of finding shortest paths. Topics in this lecture include:

- Basic notation and terminology for graphs.
- Depth-first-search for Topological Sorting.
- Dynamic-Programming algorithms for shortest path problems: Bellman-Ford (for single-source) and Floyd-Warshall (for all-pairs).

13.2 Introduction

Many algorithmic problems can be modeled as problems on graphs. Today we will talk about a few important ones and we will continue talking about graph algorithms for much of the rest of the course.

As a reminder of basic terminology: a graph is a set of *nodes* or *vertices*, with edges between some of the nodes. We will use V to denote the set of vertices and E to denote the set of edges. If there is an edge between two vertices, we call them *neighbors*. The *degree* of a vertex is the number of neighbors it has. Unless otherwise specified, we will not allow self-loops or multi-edges (multiple edges between the same pair of nodes). As is standard with discussing graphs, we will use $n = |V|$, and $m = |E|$, and we will let $V = \{1, \ldots, n\}$.

The above describes an *undirected* graph. In a *directed* graph, each edge now has a direction. For each node, we can now talk about out-neighbors (and out-degree) and in-neighbors (and in-degree). In a directed graph you may have both an edge from i to j and an edge from j to i.

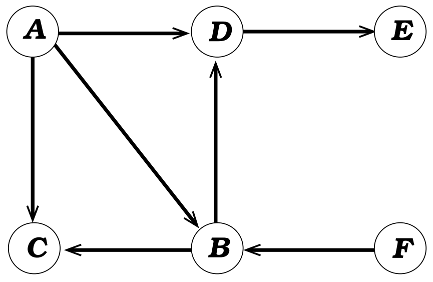
To make sure we are all on the same page, what is the maximum number of total edges in an *undirected* graph? Answer: $\binom{n}{2}$. What about a *directed* graph? Answer: $n(n-1)$.

There are two standard representations for graphs. The first is an *adjacency list*, which is an array of size n where $A[i]$ is the list of out-neighbors of node i. The second is an *adjacency matrix*, which is an n by n matrix where $A[i, j] = 1$ iff there is an edge from i to j. For an undirected graph, the adjacency matrix will be symmetric. Note that if the graph is reasonably sparse, then an adjacency list will be more compact than an adjacency matrix, because we are only implicitly representing the non-edges. In addition, an adjacency list allows us to access all edges out of some node v in time proportional to the out-degree of v. In general, however, the most convenient representation for a graph will depend on what we want to do with it.

We will also talk about weighted graphs where edges may have weights or costs on them. The best notion of an adjacency matrix for such graphs (e.g., should non-edges have weight 0 or weight infinity) will again depend on what problem we are trying to model.

13.3 Topological sorting and Depth-first Search

A **Directed Acyclic Graph (DAG)** is a directed graph without any cycles.[1] E.g.,



Given a DAG, the **topological sorting** problem is to find an ordering of the vertices such that all edges go forward in the ordering. A typical situation where this problem comes up is when you are given a set of tasks to do with precedence constraints (you need to do A and F before you can do B, etc.), and you want to find a legal ordering for performing the jobs. We will assume here that the graph is represented using an adjacency list.

One way to solve the topological sorting problem is to put all the nodes into a priority queue according to in-degree. You then repeatedly pull out the node of minimum in-degree (which should be zero — otherwise you output "graph is not acyclic") and then decrement the keys of each of its out-neighbors. Using a heap to implement the priority queue, this takes time $O(m \log n)$. However, it turns out there is a better algorithm: a simple but clever $O(m + n)$-time approach based on depth-first search.[2]

To be specific, by a *Depth-First Search (DFS) of a graph* we mean the following procedure. First, pick a node and perform a standard depth-first search from there. When that DFS returns, if the whole graph has not yet been visited, pick the next unvisited node and repeat the process.

[1]It would perhaps be more proper to call this an *acyclic directed graph*, but "DAG" is easier to say.

[2]You can also directly improve the first approach to $O(m + n)$ time by using the fact that the minimum always occurs at zero (think about how you might use that fact to speed up the algorithm). But we will instead examine the DFS-based algorithm because it is particularly elegant.

Continue until all vertices have been visited. Specifically, as pseudocode, we have:

```
DFSmain(G):
 For v=1 to n: if v is not yet visited, do DFS(v).

DFS(v):
  mark v as visited. // entering node v
  for each unmarked out-neighbor w of v: do DFS(w).
  return.            // exiting node v.
```

DFS takes time $O(1 + \text{out-degree}(v))$ per vertex v, for a total time of $O(m + n)$. Here is now how we can use this to perform a topological sorting:

1. Do depth-first search of G, outputting the nodes as you *exit* them.
2. Reverse the order of the list output in Step 1.

Claim 13.1 *If there is an edge from u to v, then v is exited first. (This implies that when we reverse the order, all edges point forward and we have a topological sorting.)*

Proof: [In this proof, think of $u = B$, and $v = D$ in the previous picture.] The claim is easy to see if our DFS entered node u before ever entering node v, because it will eventually enter v and then exit v before popping out of the recursion for DFS(u). But, what if we entered v first? In this case, we would exit v before even entering u since there cannot be a path from v to u (else the graph wouldn't be acyclic). So, that's it. ■

13.4 Shortest Paths

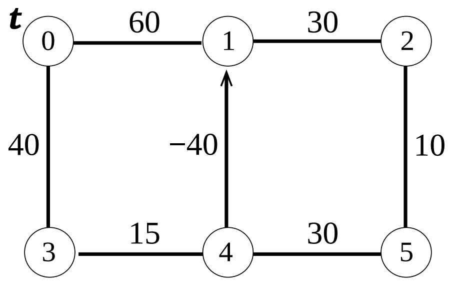
We are now going to turn to another basic graph problem: finding shortest paths in a weighted graph, and we will look at several algorithms based on Dynamic Programming. For an edge (i, j) in our graph, let's use $len(i, j)$ to denote its length. The basic shortest-path problem is as follows:

Definition 13.1 *Given a weighted, directed graph G, a start node s and a destination node t, the* **s-t shortest path** *problem is to output the shortest path from s to t. The* **single-source** *shortest path problem is to find shortest paths from s to every node in G. The (algorithmically equivalent)* **single-sink** *shortest path problem is to find shortest paths from every node in G to t.*

We will allow for negative-weight edges (we'll later see some problems where this comes up when using shortest-path algorithms as a subroutine) but will assume no negative-weight cycles (else the shortest path can wrap around such a cycle infinitely often and has length negative infinity). As a shorthand, if there is an edge of length ℓ from i to j and also an edge of length ℓ from j to i, we will often just draw them together as a single undirected edge. So, all such edges must have positive weight.

13.4.1 The Bellman-Ford Algorithm

We will now look at a Dynamic Programming algorithm called the Bellman-Ford Algorithm for the single-sink (or single-source) shortest path problem.[3] Let us develop the algorithm using the following example:



How can we use Dyanamic Programming to find the shortest path from all nodes to t? First of all, as usual for Dynamic Programming, let's just compute the *lengths* of the shortest paths first, and afterwards we can easily reconstruct the paths themselves. The idea for the algorithm is as follows:

1. For each node v, find the length of the shortest path to t that uses at most 1 edge, or write down ∞ if there is no such path.

   This is easy: if $v = t$ we get 0; if $(v, t) \in E$ then we get $len(v, t)$; else just put down ∞.

2. Now, suppose for all v we have solved for length of the shortest path to t that uses $i - 1$ or fewer edges. How can we use this to solve for the shortest path that uses i or fewer edges?

   Answer: the shortest path from v to t that uses i or fewer edges will first go to some neighbor x of v, and then take the shortest path from x to t that uses $i-1$ or fewer edges, which we've already solved for! So, we just need to take the min over all neighbors x of v.

3. How far do we need to go? Answer: at most $i = n - 1$ edges.

Specifically, here is pseudocode for the algorithm. We will use `d[v][i]` to denote the length of the shortest path from v to t that uses i or fewer edges (if it exists) and infinity otherwise ("d" for "distance"). Also, for convenience we will use a base case of $i = 0$ rather than $i = 1$.

Bellman-Ford pseudocode:

```
initialize d[v][0] = infinity for v != t.  d[t][i]=0 for all i.
for i=1 to n-1:
   for each v != t:
     d[v][i] = $\min_{(v,x)\in E}$ (len(v,x) + d[x][i-1])
For each v, output d[v][n-1].
```

Try it on the above graph!

We already argued for correctness of the algorithm. What about running time? The min operation takes time proportional to the out-degree of v. So, the inner for-loop takes time proportional to the sum of the out-degrees of all the nodes, which is $O(m)$. Therefore, the total time is $O(mn)$.

[3] Bellman is credited for inventing Dynamic Programming, and even if the technique can be said to exist inside some algorithms before him, he was the first to distill it as an important technique.

So far we have only calculated the *lengths* of the shortest paths; how can we reconstruct the paths themselves? One easy way is (as usual for DP) to work backwards: if you're at vertex v at distance $d[v]$ from t, move to the neighbor x such that $d[v] = d[x] + len(v, x)$. This allows us to reconstruct the path in time $O(m + n)$ which is just a low-order term in the overall running time.

13.5 All-pairs Shortest Paths

Say we want to compute the length of the shortest path between *every* pair of vertices. This is called the **all-pairs** shortest path problem. If we use Bellman-Ford for all n possible destinations t, this would take time $O(mn^2)$. We will now see two alternative Dynamic-Programming algorithms for this problem: the first uses the matrix representation of graphs and runs in time $O(n^3 \log n)$; the second, called the *Floyd-Warshall* algorithm uses a different way of breaking into subproblems and runs in time $O(n^3)$.

13.5.1 All-pairs Shortest Paths via Matrix Products

Given a weighted graph G, define the matrix $A = A(G)$ as follows:

- $A[i, i] = 0$ for all i.
- If there is an edge from i to j, then $A[i, j] = len(i, j)$.
- Otherwise, $A[i, j] = \infty$.

I.e., $A[i, j]$ is the length of the shortest path from i to j using 1 or fewer edges. Now, following the basic Dynamic Programming idea, can we use this to produce a new matrix B where $B[i, j]$ is the length of the shortest path from i to j using 2 or fewer edges?

Answer: yes. $B[i, j] = \min_k(A[i, k] + A[k, j])$. Think about why this is true!

I.e., what we want to do is compute a matrix product $B = A \times A$ except we change "*" to "+" and we change "+" to "min" in the definition. In other words, instead of computing the sum of products, we compute the min of sums.

What if we now want to get the shortest paths that use 4 or fewer edges? To do this, we just need to compute $C = B \times B$ (using our new definition of matrix product). I.e., to get from i to j using 4 or fewer edges, we need to go from i to some intermediate node k using 2 or fewer edges, and then from k to j using 2 or fewer edges.

So, to solve for all-pairs shortest paths we just need to keep squaring $O(\log n)$ times. Each matrix multiplication takes time $O(n^3)$ so the overall running time is $O(n^3 \log n)$.

13.5.2 All-pairs shortest paths via Floyd-Warshall

Here is an algorithm that shaves off the $O(\log n)$ and runs in time $O(n^3)$. The idea is that instead of increasing the number of edges in the path, we'll increase the set of vertices we allow as intermediate nodes in the path. In other words, starting from the same base case (the shortest path that uses no intermediate nodes), we'll then go on to considering the shortest path that's allowed to use node 1 as an intermediate node, the shortest path that's allowed to use $\{1, 2\}$ as intermediate nodes, and so on.

```
// After each iteration of the outside loop, A[i][j] = length of the
// shortest i->j path that's allowed to use vertices in the set 1..k
for k = 1 to n do:
  for each i,j do:
    A[i][j] = min( A[i][j], (A[i][k] + A[k][j]);
```

I.e., you either go through node k or you don't. The total time for this algorithm is $O(n^3)$. What's amazing here is how compact and simple the code is!