Adapted From Virginia Williams' lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 24

Dynamic Programming II: Bellman-Ford

1 More on the Bellman-Ford Algorithm

We didn't quite make it to the Bellman-Ford algorithm in the last lecture, so we'll re-hash some of that again today. In the notes for the previous lecture, we introduced Bellman-Ford in the context of Dijkstra's algorithm. We'll see it in this lecture in a different way, so as to naturally introduce *dynamic programming*. The Bellman-Ford algorithm is a dynamic programming algorithm, and dynamic programming is a basic paradigm in algorithm design used to solve problems by relying on intermediate solutions to smaller subproblems. The main step for solving a dynamic programming problem is to analyze the problem's **optimal substructure** and **overlapping subproblems**.

The Bellman-Ford algorithm is pretty simple to state:

Algorithm 1: Bellman-Ford Algorithm(G,s)

$d^{(0)}[v] \leftarrow \infty, \forall v \in V$
$d^{(0)}[s] \leftarrow 0$
$d^{(k)}[v] = \texttt{None}, \forall v \in V, \forall k > 0$
for i from $1 \rightarrow n-1$ **do**
  $d^{(k)}[v] \leftarrow d^{(k-1)}[v]$ for all v
  for $(u, v) \in E$ **do**
    $d^{(k)}[v] \leftarrow \min\{d^{(k)}[v], d^{(k-1)}[u] + w(u, v)\}$
  end for
  `// Here we release the memory for` $d^{(k-1)}$ `, we'll never need it again`
end for
return $d^{(n-1)}[v], \forall v \in V$

What's going on here? The value $d^{(k)}[v]$ is the cost of the shortest path from s to v with at most k edges in it. Once we realize this, a proof by induction falls right out, with the inductive hypothesis that "$d^{(k)}[v]$ is the cost of the shortest path from s to v with at most k edges in it."

Runtime and Storage The runtime of the Bellman-Ford algorithm is $O(mn)$; for n iterations, we loop through all the edges. This is slower than Dijkstra's algorithm. However, it is simpler to implement, and further as we saw in Lecture Notes 23, it can handle negative edge weights. For storage, in the pseudocode above, we keep n different arrays $d^{(k)}$ of length n. This isn't

necessary: we only need to store two of them at a time. This is noted in the comment in the pseudocode.

1.1 What's really going on here?

The thing that makes that Bellman-Ford algorithm work is that that the shortest paths of length at most k can be computed by leveraging the shortest paths of length at most $k-1$. More specifically, we relied on the following recurrence relation between the intermediate solutions:

$$d^{(k)}[v] = \min_{u \in V}\{d^{(k-1)}[u] + w(u,v)\}$$

where $d^k[v]$ is the length of the shortest path from source s to node v using at most k edges, and $w(u,v)$ is the weight of edge (u,v). (Above, we are assuming $w(v,v) = 0$).

This idea of using the intermediate solutions is similar to the divide-and-conquer paradigm. However, a divide-and-conquer algorithm recursively computes intermediate solutions once for each subproblem, but a dynamic programming algorithm solves the subproblems exactly once and uses these results multiple times.

2 Dynamic Programming

The idea of dynamic programming is to have a table of solutions of subproblems and fill it out in a particular order (e.g. left to right and top to bottom) so that the contents of any particular table cell only depends on the contents of cells before it. For example, in the Bellman-Ford algorithm, we filled out $d^{(k-1)}$ before we filled out $d^{(k)}$; and in order to fill out $d^{(k)}$, we just had to look back at $d^{(k-1)}$, rather than compute anything new.

In this lecture, we will discuss dynamic programming more.

2.1 Dynamic Programming Algorithm Recipe

Here, we give a general recipe for solving problems (usually optimization problems) by dynamic programming. Dynamic programming is a good candidate paradigm to use for problems with the following properties:

- Optimal substructure gives a recursive formulation; and
- Overlapping subproblems give a small table, that is, we can store the precomputed answers such that it doesn't actually take too long when evaluating a recursive function multiple times.

What exactly do these things mean? We'll discuss them a bit more below, with the Bellman-Ford algorithm in mind as a reference.

2.1.1 Optimal Substructure

By this property, we mean that the optimal solution to the problem is composed of optimal solutions to smaller independent subproblems. For example, the shortest path from s to t consists of a shortest path P from s to k (for node k on P) and a shortest path from k to t. This allows us to write down an expression for the distance between s and t with respect to the lengths of sub-paths:

$$d(s,t) = d(s,k) + d(k,t), \text{for all } k \text{ on a shortest } s-t \text{ path}$$

We used this in the Bellman-Ford algorithm when we wrote

$$d^{(k)}[u] = \min_{v \in V}\{d^{(k-1)}[v] + w(u,v)\}$$

2.1.2 Overlapping subproblems

The goal of dynamic programming is to construct a table of entries, where early entries in the table can be used to compute later entries. Ideally, the optimal solutions of subproblems can be reused multiple times to compute the optimal solutions of larger problems.

For our shortest paths example, $d(s,k)$ can used to compute $d(s,t)$ for any t where the shortest $s-t$ path contains k. To save time, we can compute $d(s,k)$ once and just look it up each time, instead of recomputing it.

More concretely in the Bellman-Ford example, suppose that (v,u) and (v,u') are both in E. When we go to compute $d^{(k)}[u]$, we'll need $d^{(k-1)}[v]$. Then when we go to compute $d^{(k)}[u']$, we'll need $d^{k-1}[v]$ again. If we just set this up as a divide-and-conquer algorithm, this would be extremely wasteful, and we'd be re-doing lots of work. By storing this value in a table and looking it up when we need it, we are taking advantage of the fact that these subproblems overlap.

2.1.3 Implementations

The above two properties lead to two different ways to implement dynamic programming algorithms. In each, we will store a table T with optimal solutions to subproblems; the two variants differ in how we decide to fill up the table:

1. Bottom-up: Here, we will fill in the table starting with the smallest subproblems. Then, assuming that we have computed the optimal solution to small subproblems, we can compute the answers for larger subproblems using our recursive optimal substructure.
2. Top-down: In this approach, we will compute the optimal solution to the entire problem recursively. At each recursive call, we will end up looking up the answer or filling in the table if the entry has not been computed yet.

In fact, these two methods are completely equivalent. Any dynamic programming algorithm can be formulated as an iterative table-filling algorithm or a recursive algorithm with look-ups.

3 Why is it called dynamic programming?

The name doesn't immediately make a lot of sense. "Dynamic programming" sounds like the type of coding that action heroes do in late-90's hacker movies. However, "progamming" here refers to a program, like a plan (for example, the path you are trying to optimize), not to programming a computer. "Dynamic" refers to the fact that we update the table over time: this is a dynamic process. But the fact that it makes you (or at least me) think about action movies isn't an accident. As Richard Bellman, who coined the term, writes in his autobiography:

> An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. Im not using the term lightly; Im using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place, I was interested in planning, in decision-making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying- I thought, let's kill two birds with one stone. Let's take a word which has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in the pejorative sense. Try thinking of some combination which will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.