Adapted From Virginia Williams' lecture notes
Additional credits: Eric Huang (2015), Anthony Kim (2016), Mary Wootters (2017)
All typos and mistakes are due to Aviad Rubinstein (2018) **Date:** October 30, 2018

1 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 BFS (breadth-first search) algorithm, we filled out L_{i-1} (list of vertices at distance $i-1$) before we filled out L_i; and in order to fill out L_i, we just had to look back at L_{i-1}, rather than compute anything new.

In this lecture, we will discuss dynamic programming more, and also see another example: the Floyd-Warshall algorithm.

1.1 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, using BFS as a running example. For reference, here is the BFS algorithm:

Algorithm 1: BFS(s)

```
Set vis[v] ← false for all v;
Set all L_i for i = 1 to n − 1 to ∅;
L_0 = s;
vis[s] ← true;
for i = 0 to n − 1 do
    if L_i = ∅ then
        exit
    while L_i ≠ ∅ do
        u ← L_i.pop();
        \\ In the loop below, replace N with N_out for a directed graph.;
        foreach x ∈ N(u) do
            if vis[u] is false then
                vis[u] ← true;
                L_{i+1}.insert(x);
                p(x)← u;
```

1.1.1 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 may consist of a shortest path 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 use this in the BFS algorithm when, for $u \in L_i$, we insert its neighbor v into L_{i+1}. In terms of distances, we're really calculating

$$d(s,v) = d(s,u) + d(u,v) = i+1.$$

(Where $d(s,u) = i$ because $u \in L_i$, and $d(u,v) = 1$ because they are neighbors.)

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

More concretely, in BFS we used the fact that $u \in L_i$ to update all its neighbors. Then we use u's neighbors in L_{i+1} to update their neighbors, etc.; these savings can multiply to an exponential (or even infinite) improvement in running time over a naive recursive computation of shortest paths.

1.1.3 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.

2 Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm solves the All Pairs Shortest Path (APSP) problem: given a graph G, find the shortest path distances $d(s,t)$ for all $s,t \in V$, and, for the purpose of storing the shortest paths, the predecessor $\pi(s,t)$ which is the node right before t on the s-t shortest path.

Let's speculate about APSP for a moment. For the special case of unweighted graphs, running BFS from every vertex $v \in V$ will take time $O(nm + n^2)$. (Later in the quarter we'll also see Dijkstra's algorithm, which achieves almost the same running time for weights that are nonnegative.) In contrast, Floyd-Warshall runs in time $O(n^3)$, but it also works for weighted graphs (even if some weights are negative). Notice that for dense graphs ($m = \Theta(n^2)$) it's running time matches the running time of computing APSP using BFS (which only works for unweighted graphs).

As mentioned, the optimum substructure with overlapping subproblems for shortest paths is that for all node k on an s-t shortest path, $d(s,t) = d(s,k) + d(k,t)$. We refine this observation as follows. Suppose that the nodes of the graph are identified with the integers from 1 to n. Then, if k is the maximum node on an

s-t shortest path, then $d(s,t) = d(s,k) + d(k,t)$ and moreover, the subpaths from s to k and from k to t only use nodes up to $k-1$ internally.

We hence get independent subproblems in which we compute $d_k(s,t)$ for all s,t that are the smallest weight of an s-t path that only uses nodes $1,\ldots,k$ internally. This motivates the Floyd-Warshall algorithm, Algorithm 2 below (please note that we will refer to the nodes of G by the names $1,\ldots,n$).

Algorithm 2: Floyd-Warshall Algorithm (G)

$d_k(u,u) = 0, \forall u \in V, k \in \{0,\ldots,n\}$
$d_k(u,v) = \infty, \forall u,v \in V, u \neq v, k \in \{1,\ldots,n\}$
$d_0(u,v) = c(u,v), \forall (u,v) \in E$
$d_0(u,v) = \infty, \forall (u,v) \notin E$
for $k = 1,\ldots,n$ **do**
  for $(u,v) \in V$ **do**
    $d_k(u,v) = \min\{d_{k-1}(u,v), d_{k-1}(u,k) + d_{k-1}(k,v)\}$ // update the estimate of $d(u,v)$
return $d_n(u,v), \forall\ u,v \in V$

Correctness when there are no negative cycles In the k-th iteration of the Floyd-Warshall algorithm, $d_k(u,v)$ is the minimum weight of a $u \to v$ path that uses as intermediate nodes only nodes from $\{1,\ldots,k\}$. What does the recurrence relation represent? If P is a shortest path from u to v using $1,\ldots,k$ as intermediate nodes, there are two cases. Assume that P is a simple path, since shortest paths are simple when there are no negative cycles:

- *Case 1*: P contains k : In this case, we know that neither the path from u to k nor the path from k to v contains any nodes that are greater than $k-1$. In this case, we can simply use $d_k(u,v) = d_{k-1}(u,k) + d_{k-1}(k,v)$.
- *Case 2*: P does not contain k : We can say that $d_k(u,v) = d_{k-1}(u,v)$

We initialize each $d_0(u,v)$ as the edge weight $c(u,v)$ if $(u,v) \in E$, else we set it to ∞ in the bottom-most row in our dynamic programming table. Now, as we increment k to 1, we effectively find the minimum distance path between $u,v \in V$ that go through node 1, and populate the table with the results. We continue this process to find the shortest paths that go through nodes 1 and 2, then $1,2$, and 3 and so on until we find the shortest path through all n nodes.

Negative cycles. The Floyd-Warshall algorithm can be used to detect negative cycles: examine whether $d_n(u,u) < 0$ for any $u \in V$. If there exists u such that $d_n(u,u) < 0$, there is a negative cycle, and if not, then there isn't. The reason for this is that if there is a simple path P from u to u of negative weight (i.e., a negative cycle containing u), then $d_n(u,u)$ will be at most its weight, and hence, will be negative. Otherwise, no path can cause $d_n(u,u)$ to be negative.

Runtime. The runtime of the Floyd-Warshall algorithm is proportional to the size of the table $\{d_i(u,v)\}_{i,u,v}$ since filling each entry of the table only depends on at most two other entries filled in before it. Thus, the runtime is $O(n^3)$.

Space usage. Note that for the Floyd-Warshall algorithm we can choose to store only two rows of the table instead of the complete table in order to save space. This is because the row being populated always depends only on the row right below it. This space saving optimization is not a general property of tables formed as a result of the dynamic programming method, and the slot dependencies in some dynamic programming problems may lie on arbitrary positions on the table thereby forcing us to store the complete table.

A Note on the Longest Path Problem

We discussed the shortest path problem in detail and provided algorithms for a number of variants of the problem. We might equally be interested in computing the longest simple path in a graph. A first approach is to formulate a dynamic programming algorithm. Indeed, consider any path, even the longest, between two nodes s and t. Its length $\ell(s,t)$ equals the sum $\ell(s,k)+\ell(k,t)$ for any node k on the path. However, this does not yield an optimal substructure: in general, neither subpath $s \to k$, $k \to t$ would be a longest path, and even if one is a longest path, the other one cannot use any nodes that appear on the first since the longest path is required to be simple. Hence the two subproblems $\ell(s,k)$ and $\ell(k,t)$ are not even independent! It turns out that finding the longest path does not seem to have any optimal substructure, which makes it difficult to avoid exhaustive search through dynamic programming. The longest path problem is actually a very difficult problem to solve and is NP-hard. The best known algorithm for it runs in exponential time.

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.