CS161 Lecture 13 Dynamic Programming and Greedy Algorithms
Scribe by: Eric Huang **Date:** May 13, 2015

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 from up to down) so that the contents of any particular table cell only depends on the contents of cells before it. Today we will see two examples.

1.1 Dynamic Programming Algorithm Recipe

Here, we give a general recipe for solving problems by dynamic programming. Dynamic programming is a good candidate paradigm to use when solving a problem if it has the following properties.

- Optimal Substructure gives a recursive formulation.
- Overlapping subproblems gives a small table.
  When a recursive function is called, it's called many times. We can store the precomputed answers, so it doesn't actually take too long.

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.

Example: The shortest path from s to t is composed of a shortest path P from s to k (for k on P) + 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.

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

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.

Again, in 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.

These properties lead to two ways to implementat 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 smallest subproblem. Then, assuming that we have computed the optimal solution to small subproblems, we can compute the answers for larger subproblems using our recursive 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 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 computes the All Pairs Shortest Path (APSP) i.e. given graph G, find $d(s,t)$ $\forall s,t \in V$, and also $\pi(s,t)$ which is the node right before t on the s-t shortest path.

Let's speculate about APSP for a moment. Consider the case when the edge weights are nonnegative, We know we can compute APSP by running Dijkstra's Algorithm on each node $v \in V$. The runtime of n Dijkstras is $O(mn + n^2 \log n)$. The runtime of Floyd-Warshall, on the other hand, is $O(n^3)$. We know that in the worst case $m = O(n^2)$, and thus, Floyd-Warshall is at least as bad as running Dijkstra's n times! Why then do we care to explore this algorithm? The reason is that Floyd-Warshall Algorithm is very easy to implement compared to Dijkstra's algorithm. The benefit of using simple algorithms is that they can often be extended and in practice can run relatively faster as compared to algorithms that may have a huge overhead.

An added benefit of Floyd-Warshall is that it also supports negative edge weights, whereas Dijkstra's Algorithm does not [1].

As mentioned, the opimum substructure with overlapping subproblems for shortest paths is that for all k on an s-t shortest path, $d(s,t) = d(s,k) + d(k,t)$. We will refine this: 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 $\leq k-1$ 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 $\leq k$ internally. This motivates the Floyd-Warshall algorithm, Algorithm 1 below (please note that we will refer to the nodes of G by the names $1, 2...., n$).

```
Algorithm 1: Floyd Warshall Algorithm (G)
$\forall u \in V \; and \; \forall k \in \{0,1,2...n\}$ , $d_k(u,u) = 0$
$\forall u,v \in V$ , $u \neq v \; and \; \forall k \in \{1,2...n\} \; we \; set$
$d_k(u,v) \leftarrow \infty \; and,$
$d_0(u,v) \leftarrow \infty \; if \; (u,v) \notin E \; and,$
$d_0(u,v) = w(u,v) \; if \; (u,v) \in E$    // set initial distance estimates
for $k \; from \; 1 \; to \; n$ do
    for $(u,v) \in V$ do
        $d_k(u,v) \leftarrow min\{d_{k-1}(u,v), d_{k-1}(u,k) + d_{k-1}(k,v)\}$    // update estimate of $d(u,v)$
return $d_n(u,v)$ $\forall$ $u,v \in V$
```

Correctness when there are no negative cycles. Consider the k^{th} step of Algorithm 1, $d_k(u,v)$ is the minimum weight of a $u \to v$ path that uses as intermediate nodes only nodes from $\{1,2,...,k\}$. What does the recurrence relation represent? If P is a shortest path from u to v using $1,2,..,k$ as intermediate nodes, there are two cases.

Assume that P is a simple path (shortest paths are simple when there are no negative cycles.).

- *Case 1* – P contains k : In this case, we know that the neither the path from u to k nor the path from k to v contains any nodes that are $> k-1$. 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 already existing edge weight $w(u,v)$ if $(u,v) \in E$, else we store it as ∞ 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

[1]Although, one can still use Dijkstra's algorithm n times, if one preprocesses the edge weights initially via something called Johnson's trick.

results. We continue this process to find the shortest paths that go through nodes 1 and 2, then $1, 2$ and 3 etc. 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 so, there is a negative cycle, and if not, then there isn't. The reason for this is that as above, 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 Floyd-Warshall 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 both the algorithms we covered today (Floyd-Warshall and Bellman-Ford), 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 dependency for the row being populated is always only on the row right below it. The space saving optimization is not a general property of tables formed as a result of Dynamic Programming, 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 talked about shortest path problem in detail, and gave 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 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 Longest Common Subsequence

In lecture 1, we talked about sequence similarity with applications in spell-checking, biology (whether different DNA sequences correspond to the same protein), and more. Today, we'll consider a simplified version of the sequence similarity problem, called the Longest Common Subsequence problem.

We say that a sequence Z is a *subsequence* of a sequence X if Z can be obtained from X by deleting symbols. For example, `abracadabra` has `baab` as subsequence, because we can obtain `baab` by deleting `a`,`r`,`cad`,`ra`. We say that a sequence Z is a *longest common subsequence* (LCS) of X and Y if Z is a subsequence of both X and Y, and any sequence longer than Z is not a subsequence of at least one of X or Y. Continuing our example, you can verify that the LCS of `abracadabra` and `bxqrabry` is `brabr`.

With this in mind, we define the LCS problem as follows: Given sequences X, Y, find length of their LCS, Z (and possibly output Z). In our algorithm and analysis, we'll let $X = x_1x_2x_3...x_m$, so $|X| = m$ and let $Y = y_1y_2y_3...y_n$, so $|Y| = n$, which will allow us to provide the runtime of our algorithm in terms of m and n.

3.1 Optimal substructure

To begin, we note that we can identify a way to break down the optimal solution to the LCS problem into optimal subsolutions to LCS subproblems. In particular, consider the following two cases.

- Case 1: If $x_m = y_n = \ell$, then any LCS Z has ℓ as its last symbol. It's easy to see that in this case any common subsequence Z' that does not end in ℓ can be lengthened by appending ℓ to Z' to obtain another (longer) legal common subsequence.

  Thus, if $|Z| = k$ and $x_m = y_n = \ell$, then we can express the first $k-1$ symbols of Z as

  $$Z[1:k-1] = LCS(X[1:m-1], Y[1:n-1])$$

  where $Z = Z[1:k-1]\ell$.

- Case 2: If $x_m \neq y_n$, then the last letter of Z, which we call z_k is not equal to x_m or not equal to y_n (or both). In this case, we can express the LCS of X and Y as

  $$Z = \max\{LCS(X[1:m-1], Y), LCS(X, Y[1:n-1])\}.$$

Suppose we keep a table C, where $C[i,j]$ maintains the length of of $LCS(X[1:i], Y[1:j])$, the longest common subsequence of $X[1:i]$ and $Y[1:j]$. Then, we can fill in the values of C using the following recurrence.

$$C[i,j] = \begin{cases} 1 + C[i-1, j-1] & \text{if } X[i] = Y[j] \\ \max(C[i-1,j], C[i,j-1]) & \text{otherwise} \end{cases}$$

3.2 Overlapping Subproblems

Note that there are only $n \times m$ entries in our table C. We only need to compute each one once. Morover, we can also see that $C[i,j]$ only depends on three possible prior values: $C[i-1,j]$, $C[i,j-1]$, and $C[i-1,j-1]$.

Thus, we can start to see how to obtain an algorithm for filling in the table and obtaining the LCS. First, we know that any string of length 0 will have an LCS of length 0. Thus, we can start by filling out $C[0,j] = 0$ for all j and similarly, $C[i,0] = 0$ for all i. Then, we can fill out the rest of the table, filling the rows from bottom up (i from 1 to m) and filling each row from left to right (j from 1 to n).

In order to fill each entry, we only need to perform a constant number of lookups / additions. Thus, we need to do a constant amount of work for each of the $m \times n$ entries, giving a running time of $O(mn)$.

3.3 Recovering the actual LCS

The algorithm that we've described computes the length of LCS. What if we want to recover the actual longest common subsequence? In Algorithm 2, we show how we can construct the actual LCS, given the dynamic programming table that we've filled out.

In each step, we decrement i or j or both, eliminating one character from X or Y from consideration (and possibly discovering a letter in the common subsequence). i and j start at m and n and are decremented until one of them is equal to 0. One can think of this as follows: in each step, the sum $i + j$ is decremented by at least 1, and the algorithm stops before $i + j$ goes below 0. Thus, the runtime for reconstructing the LCS is $O(m+n)$, which can be subsumed by the table-filling runtime of $O(mn)$.

Interestingly, this simple dynamic programming algorithm is the best known algorithm for solving the LCS problem. It is conjectured that this algorithm may be essentially optimal. It turns out that giving an algorithm that (polynomially) improves the dependence on m and n over the $O(mn)$ strategy outlined above would imply a major breakthrough in algorithms for the boolean satisfiability problem – a problem widely believed to be computationally hard to solve.

Algorithm 2: LCS(X, Y, C)

```
// C is filled out already
$L \leftarrow \varnothing$
$i \leftarrow m$
$j \leftarrow n$
while $i > 0$ and $j > 0$ do
    if $X[i] = Y[j]$ then
        append $X[i]$ to the front of $L$
        $i \leftarrow i - 1$
        $j \leftarrow j - 1$
    else if $C[i, j] = C[i, j - 1]$ then
        $j \leftarrow j - 1$
    else
        $i \leftarrow i - 1$
```