Adapted From Virginia Williams' lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 25

Dynamic Programming III: Floyd-Warshal, LCS, Knapsack

1 Overview

Last lecture, we talked about dynamic programming (DP), a useful paradigm and one technique that you should immediately consider when you are designing an algorithm. We covered the Bellman-Ford algorithm for solving the single source shortest path problem, and we talked about the Fibonacci algorithm for solving the n-th Fibonacci number. This lecture, we will cover some more examples of dynamic programming, and start to see a recipe for how to come up with DP solutions. We will talk about three problems today: all-pairs shortest path, longest common subsequence, knapsack, and maximum weight independent set in trees. In general, here are the steps to coming up with a dynamic programming algorithm:

1. **Identify optimal substructure**: how are we going to break up an optimal solution into optimal sub-solutions of sub-problems? We're looking for a way to do this so that there are overlapping sub-problems, so that a dynamic programming approach will be effective.
2. **Recursively define the value of an optimal solution:** Write down a recursive formulation of the optimum, in terms of sub-solutions.
3. **Find the optimal value:** Turn this recursive formulation into a dynamic programming algorithm to compute the value of the optimal solution.
4. **Find the optimal solution:** Once we've figured out how to find the cost of the optimal solution, we can go back and figure out how to keep enough information in our algorithm so that we can find the solution itself.
5. **Tweak the implementation**[1]: Often it's the case that the solutions that we come up with in the previous steps aren't implemented in the best way. Maybe they are storing more than they need to. In this final step (which we won't go into in too much detail in COMP 285), we go back through the DP solution we've designed, and optimize it for space, running time, and so on.

In this class, we'll focus mostly on 1, 2, and 3. We'll see a few examples of 4, and occasionally wave our hands about 5

[1] We won't talk too much about this step in COMP285, even though it is often important in practice

2 Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm solves the All Pairs Shortest Path (APSP) problem: given 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 th $s-t$ shortest path.

Let's speculate about APSP for a moment. Consider the case when the edge weights ar nonnegative. We know we can compute APSP by running Dijkstra's algorithm on each nod $v \in V$ and obtain a total runtime of $O(mn + n^2 \log n)$. The runtime of the Floyd-Warshal algorithm, on the other hand, is $O(n^3)$. We know that in the worst case $m = O(n^2)$, an thus, the Floyd-Warshall algorithm can be at least as bad as running Dijkstra's algorithm n times! Then why do we care to explore this algorithm? The reason is that the Floyd-Warshal 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 fast compared to algorithms that may have a huge overhead.

An added benefit of the Floyd-Warshall algorithm is that it also supports negative edge weights, whereas Dijkstra's algorithm does not.

As mentioned, the optimum substructure with overlapping subproblems for shortest path 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 integer 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 u 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 $1,\cdots,k$ internally. This motivates th Floyd-Warshall algorithm, Algorithm 1 below (please note that we will refer to the nodes of G by the names $1,\cdots,n$).

Algorithm 1: Floyd-Warshall Algorithm(G,s)

$d_k(u,u) = 0, \forall u \in V, k \in \{0,\cdots,n\}$
$d_k(u,v) = \infty, \forall u \in V, u \neq v, k \in \{0,\cdots,n\}$
$d_0(u,v) = c(u,v), \forall (u,v) \in E$
$d_0(u,v) = \infty, \forall (u,v) \notin E$
for $k = 1,\cdots,n$ **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`
  end for
end for
return $d_n(u,v), \forall u,v \in V$

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 both the algorithms we covered today, the Floyd-Warshall and Bellman-Ford algorithms, 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 Longest Common Subsequence

We now consider the longest common *subsequence* problem which has applications in spellchecking, biology (whether different DNA sequences correspond to the same protein), and more.

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 a *subsequence*, because we can obtain `baab` by deleting a, r, cad, and 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 . For instance, the LCS of `abracadabra` and `bxqrabry` is `brabr`.

Using the definition of LCS, we define the LCS problem as follows: Given sequences X and Y , find the length of their LCS, Z (and if we are proceeding to Step 4 of the outline above, output Z). In what follows, suppose that the sequence X is $X = x_1x_2x_3 \cdots x_m$, so that X has length m, and suppose that $Y = y_1y_2 \cdots y_n$ as length n. We'll use the notation $X[1 : k]$ as usual to denote the prefix $X[1 : k] = x_1x_2 \cdots x_k$.

3.1 Steps 1 and 2: Identify optimal substructure, and write a recursive formulation

Our sub-problems will be to solve LCS on prefixes of X and Y. To see how we can do this, we consider the following two cases.

1. **Case 1:** $x_m = y_n$. If $x_m = y_n = \ell$, then any LCS Z has ℓ as its last symbol. Indeed, suppose that Z' is any common subsequence that does not end in ℓ: then we can always extend it by appending ℓ to Z' to obtain another (longer) legal common subsequence. Thus, if $|Z| = k$ and $x_m = y_n = \ell$, we can write

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

   and

   $$Z = Z[1 : k-1] \circ \ell$$

   , where $\circ$ denotes the concatenation operation on strings.

2. **Case 2:** $x_m \neq y_n$. As above, let Z be the LCS of X and Y . In this case, the last letter of Z (call it z_k) is either not equal to x_m or it is not equal to y_n. (Notice that this or is not an exclusive or; maybe z_k isn't equal to either x_m or y_n). In this case, at least one of x_m or y_n cannot appear in the LCS of X and Y ; this means that either

   $$\text{LCS}(X, Y) = \text{LCS}(X[1 : m-1], Y)$$

   or

   $$\text{LCS}(X, Y) = \text{LCS}(X, Y[1 : n-1])$$

   , whichever is longer. That is, we can shave one letter off the end of either X or Y . In particular, the length of $\text{LCS}(X, Y)$ is given by

   $$\text{lenLCS}(X, Y) = \max\{\text{lenLCS}(X[1 : m-1], Y), \text{lenLCS}(X, Y[1 : n-1])\}$$

   .

This immediately gives us our recursive formulation. Let's keep a table C, so that

$$C[i,j] = \text{length of LCS}(X[1:i], Y[1:j])$$

Then we have the relationship:

$$C[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & X[i] = Y[j], i,j > 0 \\ \max\{C[i-1,j], C[i,j-1]\} & X[i] \neq Y[j], i,j > 0 \end{cases}$$

Suppose we keep a table C, where $C[i,j]$ maintains the length 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:

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

Technically, we should do a proof here to show that this recurrence is correct. See CLRS for the details, but it is true that if we define $C[i,j]$ recursively as above, then indeed, $C[i,j]$ is equal to the length of LCS$(X[1:i], Y[1:j])$. (Good exercise: prove this for yourself using induction).

3.2 Step 3: Define an algorithm using our recursive relationship.

The recursive relationship above naturally gives rise to a DP algorithm for filling out the table C:

Note that there are only $n \times m$ entries in our table C. This is where the overlapping subproblems come in: we only need to compute each entry once, even though we may access it many times when filling out subsequent entries.

We can also see that $C[i,j]$ only depends on three possible prior values: $C[i-1,j]$, $C[i,j-1]$, and $C[i-1,j-1]$. This means that each time we compute a new value $C[i,j]$ from previous entries, it takes time $O(1)$.

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). The pseudocode is given in Algorithm 2.

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

Algorithm 2: lenLCS(X,Y)

Initialize an $n+1 \times m+1$ zero-indexed array C
Set $C[0,j] = C[i,0] = 0$ for all $i,j \in \{1,\cdots,m\} \times \{1,\cdots,n\}$
for $i = 1,\cdots,m$ **do**
  for $j = 1,\cdots,n$ **do**
    if $X[i] == Y[j]$ **then**
      $C[i,j] = 1 + C[i-1,j-1]$
    else
      $C[i,j] = \max\{C[i-1,j], C[i,j-1]\}$
    end if
  end for
end for
return C

3.3 Step 4: Recovering the actual LCS

Algorithm 2 only computes the length of the LCS of X and Y . What if we want to recover the actual longest common subsequence? In Algorithm 3, we show how we can construct the actual LCS, given the dynamic programming table C that we've filled out in Algorithm 1.

In this algorithm, we start from the end of X and Y and work backward, using our table C as a guide. We start with $i = m$ and $j = m$. If at some point (i,j), we see that $X[i] = Y[j]$, then decrement both i and j. On the other hand, if $X[i] \neq Y[j]$, then we know that we need to drop a symbol from either X or Y . The table C will tell us which: if $C[i,j] = C[i,j-1]$, then we can drop a symbol from Y and decrement j. If $C[i,j] = C[i-1,j]$, then we can drop a symbol from X and decrement i. Of course, it might be the case that both of these hold; in this case it doesn't matter which we decrement, and our pseudocode will be default decrement j.

How long does this take? Notice that in each step, the sum $i+j$ is decremented by at least one (maybe two) and stops as soon as one of i,j is equal to zero; this is at least before $i+j = 0$. Thus, the number of times we decrement $i+j$ is at most $m+n$, which was their total value to start.

Because at each step of Algorithm 3, the work is $O(1)$, the total running time is thus $O(n+m)$, which is subsumed by the runtime of $O(mn)$ necessary to fill in the table.

The conclusion is that we can find LCS(X,Y) of a sequence X of length m and a sequence Y of length n in time $O(mn)$.

Interestingly, this simple dynamic programming algorithm is basically the best known algorithm or 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 andn over the $O(mn)$ strategy outlined above would imply a major breakthrough in algorithmsfor the boolean satisfiability problem – a problem widely believed to be computationally hard to

Algorithm 3: LCS(X,Y)

```
// C is filled out already
L ←
i ← m
j ← n
while i > 0 and j > 0 do
  if X[i] = Y[j] then
    Append X[i] to the beginning of L
    i ← i − 1
    j ← j − 1
  else if C[i, j] = C[i, j − 1] then
    j ← j − 1
  else
    i ← i − 1
  end if
end while
```

Item	Weight	Value
A	6	25
B	3	13
C	4	15
D	2	8

solve.

4 The Knapsack Problem

This is a classic problem, defined as the following:

We have n items, each with a value and a positive weight. The i-th item has weight w_i and value v_i. We have a knapsack that holds a maximum weight of W. Which items do we put in our knapsack to maximize the value of the items in our knapsack? For example, let's say that $W = 10$; that is, the knapsack holds a weight of at most 10. Also suppose that we have four items, with weight and value:

We will talk about two variations of this problem, one where you have infinite copies of each item (commonly known as Unbounded Knapsack), and one where you have only one of each item (commonly known as 0-1 Knapsack).

What are some useful subproblems? Perhaps it's having knapsacks of smaller capacities, or maybe it's having fewer items to choose from. In fact, both of these ideas for subproblems are useful. As we will see, the first idea is useful for the Unbounded Knapsack problem, and a combination of the two ideas is useful for the 0-1 Knapsack problem.

4.1 The Unbounded Knapsack Problem

In the example above, we can pick two of item B and two of item D. Then, the total weight is 10, and the total value 42.

We define $K(x)$ to be the optimal solution for a knapsack of capacity x. Suppose $K(x)$ happens to contain the i-th item. Then, the remaining items in the knapsack must have a total weight of at most $x - w_i$. The remaining items in the knapsack must be an optimum solution. (If not, then we could have replaced those items with a more highly valued set of items.) This gives us a nice subproblem structure, yielding the recurrence

$$K(x) = \max_{i:w_i \leq x} (K(x - w_i) + v_i)$$

.

Developing a dynamic programming algorithm around this recurrence is straightforward. We first initialize $K(0) = 0$, and then we compute $K(x)$ values from $x = 1, \cdots, W$. The overall runtime is $O(nW)$.

Algorithm 4: UnboundedKnapsack(W, n, w, v)

```
K[0] ← 0
for x = 1, ··· , W do
  K[x] ← 0
  for i = 1, ··· , n do
    if w_i <= x then
      K[x] ← max{K[x − w_i] + v_i}
    end if
  end for
end for
return K[W]
```

Remark 1. *This solution is not actually polynomial in the input size because it takes* $\log W$*) bits to represent* W*. We call these algorithms "pseudo-polynomial." If we had a polynomial time algorithm for Knapsack, then a lot of other famous problems would have polynomial time algorithms. This problem is NP-hard.*

4.2 The 0-1 Knapsack Problem

Now we consider what happens when we can take at most one of each item. Going back to the initial example, we would pick item A and item C, having a total weight of 10 and a total value of 40.

The subproblems that we need must keep track of the knapsack size as well as which items are allowed to be used in the knapsack. Because we need to keep track of more information in our state, we add another parameter to the recurrence (and therefore, another dimension

to the DP table). Let $K(x, j)$ be the maximum value that we can get with a knapsack of capacity x considering only items at indices from $1, \cdots, j$. Consider the optimal solution for $K(x, j)$. There are two cases:

1. Item j is used in $K(x, j)$. Then, the remaining items that we choose to put in the knapsack must be the optimum solution for $K(x - w_j, j - 1)$. In this case, $K(x, j) = K(x - w_j, j - 1) + v_j$.
2. Item j is not used in $K(x, j)$. Then, $K(x, j)$ is the optimum solution for K(x, j - 1). In this case, $K(x, j) = K(x, j - 1)$.

So, our recurrence relation is: $K(x, j) = \max\{K(x - w_j, j - 1) + v_j, K(x, j - 1)\}$. Now, we're done: we simply calculate each entry up to $K(W, n)$, which gives us our final answer. Note that this also runs in $O(nW)$ time despite the additional dimension in the DP table. This is because at each entry of the DP table, we do $O(1)$ work.

Algorithm 5: ZeroOneKnapsack(W, n, w, v)

```
for j = 1, ··· , n do
    K[0, j] ← 0
end for
for x = 0, ··· , W do
    K[x, 0] ← 0
end for
for x = 1, ··· , W do
    for j = 1, ··· , n do
        K[x, j] ← K[x, j − 1]
        if w_j <= x then
            K[x, j] = max{K[x − w_j, j − 1] + v_j, K[x, j]}
        end if
    end for
end for
return K[W, n]
```