CS 3510 Algorithms Last updated: October 19, 2023

Lecture 17: Dynamic Programming III

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian, Allen Chang*

We've discussed many problems where dynamic programming can turn exponential work into a polynomial amount of work. This works by finding subproblems which we can solve easily and solving those until we get to the main problem. Today, we'll be discussing a hard problem where we can not achieve a polynomial time algorithm, but we do end up being able to solve it in pseudo-polynomial time.

A Note about Pseudo-polynomial Time Complexities: Here, we say that dynamic programming presents a solution with a *pseudo-polynomial* time complexity. Essentially, this means that the time complexity is dependent on the value of the inputs, not the size or length of the input. In this case, we see that the runtime is $\mathcal{O}(nB)$, where B is the budget given. Contrast this to, say, a sorting algorithm such as merge sort, which runtime does not depend on the values in the array to be sorted, making it a polynomial time algorithm.

1 Knapsack

In Knapsack you're given a budget $B \in \mathbb{N}$, and you have n jobs which each have a size s and reward v. Your goal is to find all the jobs you can do within your budget to maximize your reward. [1] It turns out we'll get pseudo-polynomial time of $\Theta(nB)$.

From the runtime you may notice that there are two terms: the number of items and the size of your knapsack. Working backwards you may be able to deduce that we will have a two way table indexed by the item index and an intermediate size of the bag. [2]

First, let's develop some intuition about how we can develop a dynamic programming solution to this problem. To do so, let's consider what information we may want to encode into our solution and how we should build up from a small problem to eventually the full solution. The information we are given is a list of jobs (with corresponding value and time required) and a budget. One strategy we may have is to consider a certain number of jobs, then gradually build up to our final solution by adding more and more jobs. This is helpful because it allows us, in every step, to only consider whether or not we should add exactly one job (rather than choosing a subset from a large set). If we only consider the first i jobs, however, we don't have any information about how much budget we have remaining, and how much budget we have used. Thus, we will also encode another dimension of the table as being a restriction on the budget we have used so far.

So, we'll define our table $M[i, w]$ to be the most valuable items you can get among the first i jobs with a bag of size w.

For our base case, the most trivial base case is where we have one item. We can either take this job or ignore it, and you should always take it if you can afford to. This is

[1] Another formulation of which the problem derives it's name is that you have a knapsack that can fit some items, and you want to steal the most valuable items from a store while being able to fit them in your knapsack.

[2] If you didn't get this there's another solution where you have a one dimensional.

formalized by the following equation.

$$M[0,w] = \begin{cases} 0 & \text{if } s_0 > w \\ v_0 & \text{if } s_0 \leq w \end{cases}$$

We will then need to define a recurrence. For our recurrence, if we can not afford the job meaning that $s_i > w$, then the best thing that we do is to take the previous jobs which will give us a value of $M[i-1,w]$. Now if we can afford the item, we can take the item and have a remaining weight of $w - v_i$, so we will look for the maximum value of the items we can make excluding i with a weight of $w - v_i$ or $M[i-1, w-v_i] + v_i$ where we add the v_i to denote the addition of the ith element. Of course it may be optimal to ignore this item in which case we'd just take $M[i-1,w]$, so we'll take the maximum of when we do take the item and don't take the item. Thus, our general recurrence is:

$$M[i,W] = \begin{cases} M[i-1,W] & \text{if } s_i > W \\ \max\{M[i-1,W], v[i] + M[i-1, W-S_i]\} & \text{if } s_i \leq W \end{cases}$$

To fill in this table, you will notice that each entry will take constant time to fill in and there are $\mathcal{O}(nB)$ entries, meaning the runtime is $\mathcal{O}(nB)$. [3] This is of course pseudo-polynomial which is bad; however this is the best we can do since the problem is NP-Hard [4]

There are many other variations of this problem; as we've described it, it's called 0-1 Knapsack because you can either have an item zero or one times. In another formulation you may be able to have it k times or infinite times which are both of similar difficulty.

2 All-Pairs Shortest Path

The All-Pairs Shortest Path problem is defined as follows. Given a directed graph $G = (V, E)$ with weight $w : E(G) \to \mathbb{R}$, find the shortest path length for all pairs of vertices.

Given that there are $\binom{|V|}{2} = \mathcal{O}(|V|^2)$ elements in our output, we can expect that our algorithm has a time complexity of at least $\mathcal{O}(|V|^2)$. But how close can we get to this bound?

Before providing the correct solution, let's discuss how we can process the information to define the table used in our dynamic programming approach. Like in the Knapsack problem, consider how we can restrict the problem to be defined on a strictly smaller problem, which may allow us to solve a simpler problem by only considering one new element (ex., character in a string, vertex in a graph, entry in a table). In this case, we may want to equate computing one table entry by considering only one new vertex. To do this, we can arbitrarily order the vertices into a list, then restrict our table by defining a dimension that considers only the first k vertices in the list. Thus, we can imagine that if

[3] In the one-dimensional formulation with $\mathcal{O}(B)$ entries each will take $\mathcal{O}(n)$ time to calculate, so you'll still get $\mathcal{O}(nB)$

[4] While solving the problem exactly is NP-Hard, it turns out that there exists polynomial time "good" approximations for the Knapsack problem. The rough intuition is that we divide our jobs into two cases where we have smaller jobs and larger jobs. We can take many small jobs, so making a mistake there will not cause many issues, but if we can guess at the big jobs well, then we can be fairly accurate.

we want to compute the final solution, we only have to consider whether or not adding the new vertex at index k is "worth it".

What about the other dimensions in the dynamic programming table? For the algorithm to work, we need $|V|^2$ entries in the table that we can output. Naturally, we can consider the other two dimensions to be the start and end vertices of a path that we would like to compute; by encoding this information, we can return the entry in the table that considers every vertex (i.e., the "last" element where $k = |V|$) to recover the true shortest path length.

It turns out that the algorithm for this is called the Floyd-Warshall algorithm; this algorithm has interesting properties among them being its ability to handle negative edges. For the algorithm we'll have vertices and we'll label them $v_1, v_2, \ldots v_n$; the ordering doesn't really matter but it helps us create sub-problems. We define a three dimensional table M; we define $M[i, j, k]$ to be the size of the shortest path from i to j only through vertices $\{1, 2, \ldots k\} \cup \{i, j\}$[5].

Let's consider the base cases; where would a base case for shortest paths be? When there is no path (path length of zero), meaning the path goes from a vertex to itself. Another base case we could use is where there is a path of one edge, the length of this path would just be the weight of the edge (or infinity if the edge doesn't exist). Then, the base cases are as follows

$$M[i, i, k] = 0 \text{ and } M[i, j, 0] = w_{i,j}$$

Now we'll define our recurrence. Previously we've been trying to maximize items, but here we're trying to find the shortest path, so we'll be taking the minimum. The first case is where we don't use the kth vertex which would mean the shortest path is just $M[i, j, k-1]$. The second case is where we do use the vertex k in which case we'll find the path from i to k which will have a cost of $M[i, k, k-1]$, and from k to j which will have a cost of $M[k, j, k-1]$.

$$M[i, j, k] = \min \begin{cases} M[i, j, k-1], \\ M[i, k, k-1] + M[k, j, k-1] \end{cases}$$

We remark that the time complexity is $\mathcal{O}(|V|^3)$, as there are $|V|^3$ tries in M, and each entry takes $\mathcal{O}(1)$ time to compute.

3 Longest Common Subsequence

In this problem you're given two arrays of n symbols (letters, numbers, etc.), and the goal is to find two subsequences that are the same. Formally this is finding two subsets X, Y where $|X| = |Y|$ and $a[x_i] = b[y_i]$. As an example you might get the two strings "ABCDEFG" and "AKBZCRD". The common substring is "ABCDE".

Like before, we will also simplify the problem so that computing subproblems require only considering one new character in the string. This problem, however, requires us to consider two strings (thus, we need to encode information about both strings in our matrix). Naturally, we can do this by using this strategy twice, once for each string. Using this,

[5]The intuition here is that when constructing the path, for the vertices in the inside of the path (i.e., without considering the ends of the path) from i to j, we only use vertices $\{1, 2, \ldots, k\}$.

consider a 2D array which restricts the problem on the first i characters in the first string and the first j characters in the second string. Then, to compute each entry, we would theoretically only need to consider exactly one new character and determine whether or not it increases our longest common subsequence.

We'll define a simple two-way table indexed by i, j where $i \in [|A|]$ and $j \in [|B|]$[6] which represents the length of the longest common subsequence using only characters up to i in A and j in B.

In the base case with one element, they can either be the same element in which, the longest subsequence will include them both. Alternatively they could ignore the element to have a length of zero. For the base case, we have

$$M[0,0] = \begin{cases} 0 & \text{if } A[0] \neq B[0] \\ 1 & \text{if } A[0] = B[0] \end{cases}$$

For the general case, we define

$$M[i,j] = \begin{cases} 1 + M[i-1, j-1] & \text{if } A[i] = B[j] \\ \max\{M[i, j-1], M[i-1, j]\} & \text{if } A[i] \neq B[j] \end{cases}$$

To motivate some intuition, we will describe each case. The first case is when the last characters checked are the same, and we should always include the last character in this case. The second case, we where the two characters are not equal, we will ignore one of the characters; if we ignore the character of B we'll use $M[i, j-1]$, and if we ignore the character of A, we'll use $M[i-1, j]$. We'll choose the character that maximizes the length, and we are done.

As we've discussed before, the runtime of this algorithm is the number of entries multiplied by the time it takes to fill in each element. In this case there are $\mathcal{O}(nm)$ elements which each take constant time, meaning the overall runtime is $\mathcal{O}(nm)$. [7]

[6] The notation $[x]$ means $\{1, 2, \ldots, x\}$

[7] We believe this algorithm is optimal under the exponential time hypothesis.