CS 3510 Algorithms 2/29/2024

Dynamic Programming IV

Lecturer: Jan van den Brand *Scribe(s): Ankith Thalanki*

This is the last lecture on Dynamic Programming, and today we covered three different dynamic programming algorithms: Longest Increasing Sub-sequence, Bellman-Fords, and Floyd-Warshall. While we have already seen Bellman-Ford, in this lecture we explore it from a dynamic programming approach.

1 Longest Increasing Subsequence

The longest increasing subsequence of an array $A[1...n]$ is the largest sub-sequence of A such that each element is greater than all preceding elements.

The longest-increasing sub-sequence problem is an example of how we can use dynamic programming to solve similar adjacent problems instead of solving our problem directly. Sometimes we might have trouble splitting up a problem into sub-cases. If we know of a similar problem we can easily split into sub-problems, whose solution we can use to easily solve ours, we can use dynamic programming to indirectly to solve our problem. In this problem, of finding the longest increasing subsequence of an array $A[1...n]$, we first solve the adjacent problem of finding the longest increasing subsequence of an array $A[1...n]$ that must contain $A[n]$.

1.1 Table Definition

Let us define $T[i]$ as the longest increasing subsequence in the array $A[i]$ that must include i as an element of the subsequence.

1.2 Base Cases

$T[1]$ is the base case, which is simply $A[i]$, as that is the only sub-sequence available in an array with only one element. We don't have to worry about the "longest increasing" part, as there is only one element.

1.3 Recursive Formula

How can we calculate $T[i]$ given $T[1]$ up to $T[i-1]$? Well we know the longest increasing sub-sequence in $T[i]$ must include the element $A[i]$. One question we can ask is whether or not we will have another element in the longest increasing sub-sequence before $A[i]$, and what that element will be. If we don't have any elements before $A[i]$, then $T[i]$ will just be the single element $A[i]$. If we do choose to have an element before $A[i]$, at say x, then it will have to be $A[x] < A[i]$. Note however that $T[x]$ contains $A[x]$, and all elements in longest sub sequence associated with $T[x]$ will be less than or equal to $A[x]$ and therefore less than $A[i]$. This means that we can add all the elements in $T[x]$ to the longest sub sequence associated with $T[i]$. Therefore if we choose to have element x be the next element in the

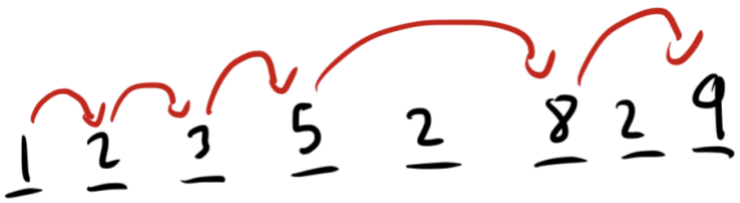

Figure 1: Longest increasing sub sequence of an array, where elements included in the sub sequence are pointed out by the red arrows.

longest increasing sub-sequence, then $T[i]$ is equal to $T[x]+1$. Our recursive relation would have to go through all the possible outcomes above and pick the one with the maximum value for $T[[i]$, giving us the equation below:

$$T[i] = max_{x=\{0,1,2,..,i-1\}}(1, \begin{cases} T[x]+1 & A[x] < A[i] \\ 0 & A[x] \geq A[i] \end{cases})$$

As you can see in the recursive relation above we loop through all possible sub-cases. Because $0 < 1$, we will never choose a previous element that doesn't give us an increasing subsequence.

1.4 Solution

The longest increasing subsequence shows that the solution to a dynamic programming problem doesn't necessarily have to be an element of the defined table. By calculating all the entries in the table, we have found all the longest increasing sub-sequences that end at each element of the array. The longest increasing subsequence of A doesn't necessarily need to include the last element, so we can't simply look at the $T[i]$. However the longest increasing subsequence has to end at some element in the array, meaning it must be one of the entries in T. We can therefore loop through every entry in T to find the max value, which will be the total of the longest increasing sub-sequence.

1.5 Runtime

Our table is of size n, and since each entry $A[n]$ considers all previous elements, the runtime to calculate each entry is $O(n)$. Therefore the runtime to determine the whole table is $O(n^2)$. Since the runtime to find the solution from the table is $O(n)$ (since we loop through all n elements of the table), it has a neglibible impact on the total runtime, and thus the total runtime of the algorithm is still $O(n^2)$.

2 Bellman-Ford

Bellman-Ford is an $O(|V||E|)$ dynamic programming algorithm that computes the shortest path from a single vertex s to any other vertex in any weighted directed graph G. It still works with negative edges, as long as there are no negative cycles.

2.1 Rational

In Dynamic Programming in general we solve a problem by splitting it up into several smaller problems. We can consider a DP problem to be a series of choices (Do I select element x or not?), and each call/step of the recursive formula answers the last choice based on our solution is trying to maximize/minimize (cost, value, etc.). In Bellman-Ford, we are trying to determine which path of length i from s to some node u we can use as a "sub-path" to determine the shortest path of length $i+1$ from to v. Once we've answered that question (In section 2.4), we've split our problem into a choice between several sub-problems, allowing us to use Dynamic Programming to solve our problem in polynomial time.

2.2 Table Definition

Let us define $T[i, v]$ as the length of the shortest path from s to v using at most i steps. If such a path is not possible, $T[i, v] = \infty$.

2.3 Base Cases

Our base cases occur when $i = 0$. This indicates that we are looking for a path to v with a length of zero, which does not exist, unless v is s itself. This means that we have two different base cases, $T[0, s] = 0$, since the shortest path from s to s is just s, and $T[0, v] = \infty \ \forall v \in V \wedge v \neq s$, as there is no path of length 0 that can reach a different node from s.

2.4 Recursive Formula

As asked in section 2.1, how can we use the answers to previous sub-problems to solve our current problem? We wish to find $T[i, v]$, or the shortest path from s to v with a length of at most i. Well, any shortest path from s to v would have to go through one of the neighbours of v, which we define as the set $N(v)$. We could therefore consider all the paths of length $i - 1$ to elements of $N(v)$, and all the edge costs from elements of $N(v)$ to v. What if the shortest path from s to v has already been found, and it is smaller or equal to $i - 1$? Then such a path would be contained in $T[i - 1, v]$, so we can also consider that as a potential contender for the value of $T[i, v]$. Putting all this information together we get the following recursive relation:

$$T[i, v] = min_{(u,v)\in E}(T[i - 1, u] + c_{u,v}, T[i - 1, v])$$

. In this recursive relation we consider $|N(v)| + 1$ contenders for the shortest path of length less than or equal i; either the previous path of length less than or equal to $i - 1$ or a path built from the shortest paths of the neighbours.

2.5 Solution

How can we use our table $T[i, v]$ to find the shortest path from s to v of any length? It may seem that we would have to calculate all the way up to $T[i, \infty]$, because we don't have any knowledge of the max length of the shortest path. However, if we assume that there are

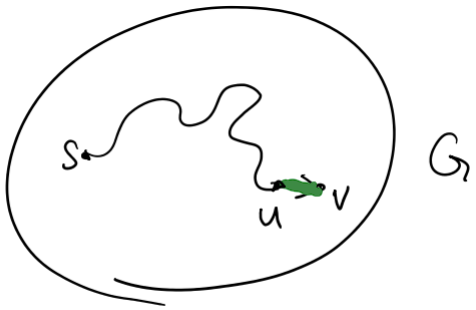

Figure 2: Image of how we can use the path from s to u of length $i-1$ to construct a path from s to v of length i

no negative cycles, we can find an upper bound on the length of the shortest path. If we assume that the shortest path of length n is greater than or equal to the number of nodes, $n \geq |V|$, then such a path will have covered at minimum $|V|+1$ nodes. This is because a path of length 0 will have started of reaching one node (s) and every additional edge would make the path reach another node. Since there are only $|V|$ nodes, that means one node has been covered twice. This means that a cycle exists. Since such a cycle must have a non-negative weight, it only increases the length/cost of the path, meaning that this path is not the shortest path. We can therefore see by this contradiction that $n < |V|$, meaning that the greatest possible path length is $|V|-1$. This means that we only need to find $T[|V|-1, v]$ to find the length of the shortest path from s to v.

2.6 Runtime

In order to calculate the shortest path length for every node, we will have to calculate up to $T[|V|-1, v]$ for every node v. Notice that each table entry $T[i, v]$ has a runtime of $O(|N(v)|)$. That means for calculating all the shortest paths of length smaller or equal to i we are considering the degrees of every node, $O(|N(v_1)|+|N(v_2)|+...)$. In a directed graph the sum of all the degrees is equal to $|E|$, meaning that calculating all the shortest paths of length smaller than or equal to i for all nodes takes $O(|E|)$. Since we do $|V|-1$ iterations of this, the total runtime of the entire Bellman Ford algorithm is $O(|E||V|)$ (Constant factors ignored).

3 Floyd-Warshall

Floyd-Warshall is another application of dynamic programming, which finds the shortest paths between every nodes in a directed weighted graph in $O(V^3)$ time. It works on all directed weighted graphs that contain no negative cycles. It uses the same idea as Bellman-Ford, of iteratively determining the best shortest path that uses a limited number of vertices. Unlike Bellman-Ford however, it doesn't just find all shortest paths from a specific node, it finds all the shortest paths between any two nodes.

3.1 Rational

In previous dynamic programming algorithms, like Bellman-Ford, we are essentially "cutting off the last piece of the solution" to determine what sub-problems we would need to solve the problem. However, in Floyd-Warshall, we are essentially "cutting of the last piece of the input" to determine what sub-problems to solve. The sub-problems are correct for specific sub-graphs, unlike in Bellman-Ford, where the sub-problems are specific to other restrictions.

3.2 Table Definition

We define $T[s, t, k]$ as the length of the shortest path from s to t , where all nodes on the path except for s and t must be part of the first k vertices. The first k vertices define the sub-graph upon which $T[s, t, k]$ correctly represents the shortest path.

3.3 Base Cases

If we set $k = 0$, then we are not allowed to use any intermediate nodes between s and t. Therefore there only exists a path between s and t if they are the same node. These two different base cases can be represented by the equations below:

$$T[s, t, 0] = \infty, \ s \neq t$$

$$T[s, t, 0] = 0, \ s = t$$

3.4 Recursive Formula

Let's say we want to calculate $T[s, t, k]$, the shortest path from s to t with the first k nodes. As we are "cutting of the last piece of the input" in this problem, let us consider the shortest paths with only $k-1$ nodes. This would be $T[s, t, k-1]$. If $T[s, t, k]$ does not include the kth node, that means it is equivalent to $T[s, t, k-1]$. If $T[s, t, k]$ does however include the kth node, then we can split up our path to two parts, s to k and t to k. This would make $T[s, t, k]$ be equal to the total length of both smaller paths combined, of $T[s, k, k-1]T[k, t, k-1]$. Since in both of these smaller paths k is either a start or an end node, it doesn't technically need to be part of the sub-graph (see the table definition at 3.2), so we only need to consider the sub-graph containing $k-1$ nodes. Whether $T[s, t, k]$ includes or doesn't include k depends on which path is the shortest, giving us the recursion formula below:

$$T[s, t, k] = min(T[s, t, k-1], T[s, k, k-1] + T[k, t, k-1])$$

3.5 Solution

The solution to find the distance between any two vertices s and t in a graph G of size $|V|$ is simply $T[s, t, |V|]$. Since we are considering the first $|V|$ vertices, and there are only $|V|$ vertices, this is the shortest path in the entire graph.

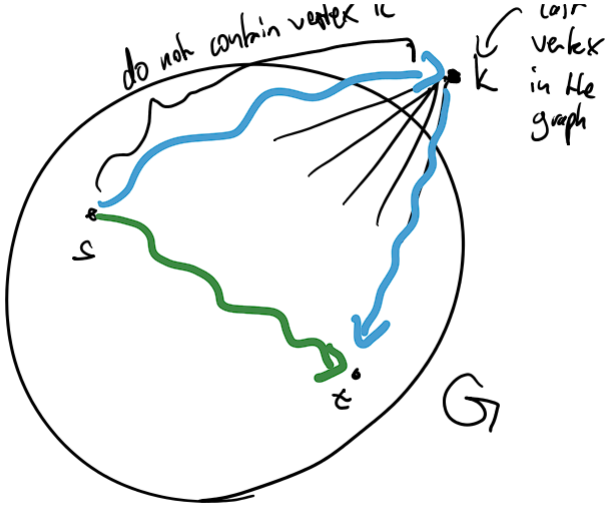

Figure 3: Comparing whether or not the shortest path includes the kth node or doesn't.

3.6 Runtime

The time it takes to fill out each element of the array is simply $O(1)$, we are only considering a linear number of base cases. Since there are $|V|^3$ entries in the table, the total runtime of Floyd-Warshall is $O(|V|^3)$.