Adapted From Virginia Williams' lecture notes.

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

Dynamic Programming III: Floyd-Warshall

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.