Contents

1 Dynamic Programming Overview

Basic Idea: Memoize, memoize, memoize! When you are recursing on a function, it is like to get called with the same input many times. *Only* calculate for that input once, store it in a hash table, and then look it up whenever you need it again.

When does it work? When the solution is a combination of similarly structured subproblems. This is likely to be true when you are using a recursive function since you're using the same function to calculate all of the subproblems, but there are many potential uses.

How does it work? Many ways. One of the easiest is to find a recursive formula for your problem.

What is the running time? It's usually easiest to figure out how many times you call your function overall.

2 All-Pairs Shortest Paths

Let A be the weight matrix so $A_{ij} = w(v_i, v_j)$. There can be negative weight edges, but no negative weight cycles. We let d_{ij}^m be the shortest path from i to j that uses at most m edges. $d_{ij}^0 = 0$ if $i = j$ and ∞ if $i \neq j$.

$$d_{ij}^{(m>0)} = \min_k \left(d_{ik}^{(m-1)} + A_{kj} \right) \tag{1}$$

Ah ha! A recursive formula! Lovely.

2.1 Naive Algorithm

Algorithm:

APSP-RECUR$(V, w, i, j, m, memo)$
1 **if** $memo[i, j, m]$
2 **return** $memo[i, j, m]$
3 **if** $m = 0$
4 **return** 0
5 $memo[i, j, m] \leftarrow \min_k$ (SLOW-APSP-RECUR$(V, w, i, k, m-1, memo) + w(v_k, v_j)$)
6 **return** $memo[i, j, m]$

APSP(V, w)
1 $memo \leftarrow$ Empty memo
2 **for** $v_i \in V$
3 **for** $v_j \in V$
4 $\delta(v_i, v_j) \leftarrow$ APSP-RECUR$(V, W, i, j, |V| - 1, memo)$
5 **return** δ

Running Time: Any call to APSP-RECUR that hits *memo* returns in constant time. There are $O(|V|^2(\text{max path length})) = O(|V|^3)$ entries for *memo* so we make $O(|V|^3)$ calls to APSP-RECUR in the process of filling *memo*. Each of these calls requires $O(|V|)$ (for the min over k) besides the time taken to fill *memo*. Therefore, the total time spent filling *memo* is $O(|V|^4)$. We may make $O(|V|^2)$ extra calls to APSP-RECUR from the for loops in APSP, but the running time is still $O(|V|^4)$.

2.2 Better Algorithm

Motivation: We don't *care* about any values for m except V. When we have $d_{ij}^{(m)}$ we know the shortest path of length m from i to j for *any* i and j... So we can use it to compute $d_{ij}^{(2m)}$:

$$d_{ij}^{(2m)} = \min_k \left(d_{ik}^{(m)} + d_{kj}^{(m)} \right) \tag{2}$$

Note that $d_{ij}^{(1)} = A_{ij}$ so it all makes sense. This is similar in essence to doing powers of matrices by repeated squaring.

Algorithm: Change line 5 of APSP-RECUR to be equation 2 and line 4 of APSP to pass the nearest power of 2 greater than $|V| - 1$.

Running Time: Now we only have $|V|^2 \log |V|$ entries in *memo*, each of which takes $O(|V|)$ to fill for $|V|^3 \log |V|$ running time.

3 Floyd-Warshall

In Section 2, we used a recursion over the number of edges in the path to define an optimal substructure. However, there is another way of finding optimal substructure, which is to consider instead which vertices

are used in the path.

Assume we are trying to find a shortest path from u to v using a set of k vertices $\{w_1, ..., w_k\}$. Then either

- w_k is an intermediate vertex of a shortest path from u to v and we can break the problem down into the shortest path from u to w_k and w_k to v
- OR w_k is *not* an intermediate vertex and we can just consider the shortest path involving $\{w_1, ..., w_{k-1}\}$

Recursive Relation: Let $d_{ij}^{(k)}$ be the shortest path from v_i to v_j using only vertices $\{v_1, ..., v_k\}$. Then:

$$d_{ij}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if k} = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k > 0 \end{cases} \tag{3}$$

Yes that *is* the same k in the sub- and super-script! Each of the choices in the min corresponds to one of the bullet points above. If we know that $\{v_1, ..., v_k\}$ are intermediate vertices then either the shortest path from v_i to v_j involves v_k or it only involves $\{v_1, ..., v_{k-1}\}$ (seems kind of obvious when you put it that way!).

Algorithm:

FW-RECUR$(V, i, j, k, w, memo)$
1 **if** $memo[i, j, k]$
2 **return** $memo[i, j, k]$
3 **if** $k = 0$
4 **return** $w(v_i, v_j)$
5 $memo[i, j, k] \leftarrow$
$\min[\text{FW-RECUR}(V, i, j, k-1, w), \text{FW-RECUR}(V, i, k, k-1, w) + \text{FW-RECUR}(V, k, j, k-1, w)]$
6 **return** $memo[i, j, k]$

FLOYD-WARSHALL(V, w)
1 $memo \leftarrow$ Empty memo
2 **for** $v_i \in V$
3 **for** $v_j \in V$
4 $\delta(v_i, v_j) \leftarrow \text{FW-RECUR}(V, i, j, |V| - 1, w, memo)$
5 **return** δ

Running Time: At some point during all calls to FW-RECUR, we must fill up the *memo* data structure, which is size $|V|^3$. Each entry in the memo takes only $O(1)$ to fill (assuming the necessary other entries are filled) so the total time it takes to fill the *memo* is $O(|V|^3)$. Calls to FW-RECUR that do not fill in *memo* take $O(1)$. Therefore, the for loops on lines 2 and 3 of FLOYD-WARSHALL may incur an extra $O(|V|^2)$ calls to FW-RECUR each of $O(1)$. So filling *memo* dominates for a total running time of $O(|V|^3)$.

4 When Dynamic Programming Fails: Unweighted Longest Path

Simple Paths: A path is *simple* if it does not contain a cycle.

Longest Path Subproblems: Assume we have an unweighted graph p is the longest simple path (i.e. no cycles) from u to v and w lies on this path. Then the subpath from u to w is *not* necessarily the longest subpath from u to w. In fact, the problem of longest path is NP-Complete so there is no known way of solving it in polynomial time.

Dependent Subproblems: What is the problem? The problem is that the subproblems for longest simple path are not *independent*. Two subproblems are independent if the solution to one does not affect the solution to another. For longest subpaths, using a vertex in the solution to one sub-problem means we cannot use it in the solution to another. Both sub-problems depend on the same resource!

For shortest subpaths, however, if w is on the shortest path from u to v then it is impossible that a shortest path from u to w shares any vertices (besides w) with a shortest path from w to v. (Prove that yourself).

5 Dynamic Programming Examples

5.1 Markov Decision Processes

A Markov decision process (MDP) is a way of representing uncertainty in the world. An MDP is a tuple $M = \langle S, A, T, R, G \rangle$ where S is a set of states in the world, A is a set of actions, $T : S \times A \times S \to [0, 1]$ is a transition function giving the probability that an action transitions one state to another, $R : S \to \Re$ is a reward function giving the reward for each state, and $G \subset S$ is a set of goal states. The reward for a goal state is 0 and the reward for any non-goal state is less than 0. Jenny's research is on solving very large MDPs.

For example, you could set up a robot navigation problem by considering each state to be a different square in a grid and the actions to be `move-up`, `move-down`, `move-right`, `move-left`. Because robot actuators are not very good it might have only an 80% chance of getting to the square where it is actually aiming, a 15% chance of of winding up in one of the other 3 squares, and a 5% chance of staying where it is. If we want the robot to take the shortest path, we give the goal square 0 reward and all other squares -1 reward. However, we could also give certain squares even lower rewards if they represent obstacles that the robot should avoid.

To solve this MDP, we want to know what action to take in every state. An assignment of actions to states is a *policy* $\pi : S \to A$. If we have a policy, the quantity of interest for an MDP is a *value*, which is the expected reward of a state over all time. The value of a policy π is given by

$$V^{\pi}(s) = R(s) + \sum_{s' \in S} T(s, \pi(s), s') V^{\pi}(s'). \tag{4}$$

We would like to find the optimal value (corresponding to some optimal policy, but let's focus on the value for now) $V^*(s)$ for a state s:

$$V^*(s) = \begin{cases} 0 & \text{if } s \in G \\ \max_{a \in A} \left[R(s) + \sum_{s' \in S} T(s, a, s') V^*(s') \right] & \text{else} \end{cases} \tag{5}$$

This is a recursion relation! Or is it? No, in fact it is a system of equations because $V^*(s)$ appears on both sides of the equation. And it is a nasty system of equations because it is non-linear. So, dynamic programming to the rescue!

We think of this as a dynamic programming problem by thinking of how *many* actions we can take total. (Eventually, obviously, we need to deal with the case of being able to take an infinite number of actions, but we'll get there). Clearly, if we have k total actions we can take, we want to figure out the best action to take on the first time step, assuming we then act optimally thereafter. Specifically, the optimal value $V_k^*(s)$ given only k actions is given by

$$V_k^*(s) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{a \in A} \left[R(s) + \sum_{s' \in S} T(s, a, s') V_{k-1}^*(s') \right] & \text{else} \end{cases} \tag{6}$$

and now *this* we know how to solve by dynamic programming! Or do we? No one limits the number of actions we can take! So we could just assume that after 1000 or 100000 actions we get to the goal, but there is a better way...

At some time t the probability that, if we start at any state and act optimally for t actions we will reach the goal after those t actions approaches 1. Once we get there, there is no point in continuing beyond t because V^* won't change. How do we know where t is? We just said it! V^* won't change! So, we do the update given by equation 6 until V^* stops changing:

```
VALUE-ITERATION(S, A, T, R, G)
 1  V_0 ← 0, d ← ε + 1, k ← 1
 2  while d > ε
 3      for s ∈ S
 4          if s ∈ G
 5              V_k(s) ← 0
 6          else
 7              V_k ← max_{a∈A} (R(s) + Σ_{s'∈S} T(s, a, s') V_{k-1}(s'))
 8      d ← max_{s∈S} |V_k(s) − V_{k-1}(s)|
 9      k ← k + 1
10  return V*
```

Running Time: Each iteration takes $O(|S|^2|A|)$ time. It can be shown that only $O(|S|)$ iterations are needed for convergence so the overall algorithm is $O(|S|^3 A)$.

5.2 Sums of Integers

Problem: Suppose you are given an array of n integers $\{a_1, ..., a_n\}$ between 0 and M. You wish to divide these integers into two sets x and y such that the sum of the integers in each set is the same. For example, given the set $\{2, 3, 2, 7, 8\}$, you can divide it into $\{2, 2, 7\}$ and $\{3, 8\}$, both of which sum to 11. The set $\{1, 7, 4\}$, however, cannot be divided. Your task is, given the array of integers, create an algorithm to tell if the integers can be divided.

Solution: First check if $\sum_j a_j$ is even. If not, return false. Otherwise, we know we need a subset of the a_j that sums to $\left(\sum_j a_j\right)/2$

Consider just the set of the numbers $\{a_1, ..., a_j\}$. What sums can we make with that set or subsets of it? We can make

- Any sums we could make with a subset of $\{a_1, .., a_{j-1}\}$

- Any sums we could make with a subset of $\{a_1, ..., a_{j-1}\} + a_j$

So: Let c_{ij} be 1 if a subset of $\{a_1, ..., a_i\}$ adds to j and 0 otherwise. The recursion relation for c_{ij} is

$$c_{ij} = \begin{cases} 1 & \text{if } i = 0 \text{ and } j = 0 \\ \max\left[c_{i-1,j}, c_{i-1,j-a_j}\right] & \end{cases} \tag{7}$$

Running Time: We need only let j go to nM since the integers are bounded. Therefore, the size of c is n^2M and filling it in takes $O(1)$ per entry for a total running time of $O(n^2M)$.