Algorithms for Data Science: Lecture on Dynamic Programming

Barna Saha

1 Dynamic Programming

Dynamic programming is a powerful algorithmic design technique which most of you have already learned in your undergraduate algorithms class. For a large class of seemingly exponential problems, one can design polynomial time solution often only via dynamic programming. There does not seem to be any other systematic technique with similar universal applicability.

History. Richard E. Bellman (1920-1984, IEEE Medal of Honor 1979)

"*Bellman ... explained that he invented the name 'dynamic programming' to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who had a 'pathological fear and hatred of the term, research.' He settled on the term 'dynamic programming' because it would be difficult to give a 'pejorative meaning' and because 'It was something not even a Congressman could object to'.*" *[John Rust, 2006]*

[Taken from Eric Demaine's lecture note at MIT]

The main bottom-line technique for dynamic programming is to remember (memoize) and reuse solution of subproblems. Let us consider as an example, *computing edit distance of two strings of length n.*

1.1 Edit Distance Computation

We are given two strings s and t of alphabets, each of length n, and we would like to make minimum number of insertion, deletion and substitution on the two strings to make them identical.

Example 1. *The edit distance between "Hello" and "Jello" is 1. The edit distance between "good" and "goodbye" is 3. The edit distance between any string and itself is 0.*

Here is a simple dynamic programming solution for it. We will simply count the number of edits required to convert s to t, but with slight modification, we can also return the edit script. Suppose $D[i,j]$ denotes the number of edits required to convert $s_1s_2...s_i$ to $t_1t_2...t_j$ where s_i denotes the ith alphabet in s and t_j denotes the jth alphabet in t. We can initiate

$$D[i,0] = i, D[0,j] = j, i, j = 1, 2, .., n.$$

Now to compute $D[i,j]$, we have three choices: (1) s_i and t_j match, we match them and then the edit cost is $D[i-1,j-1]$,

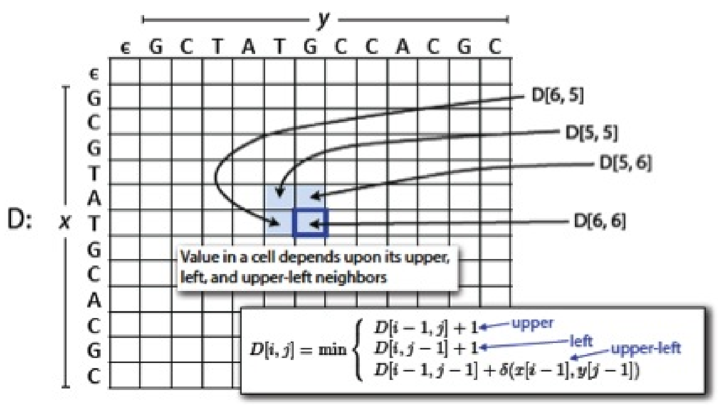
(2) s_i and t_j do not match, we substitute one to match the other and then the edit cost is $D[i-1,j-1]+1$,

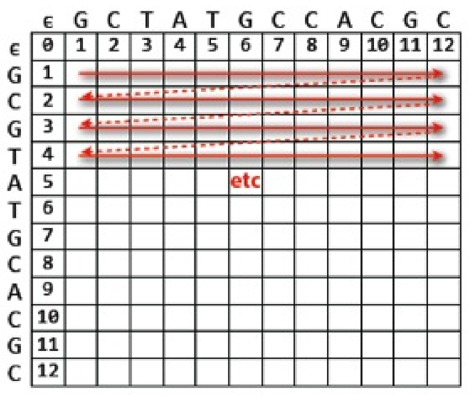
(3) s_i and t_j do not match, we delete s_i and edit cost is $D[i-1,j]+1$,

(4) s_i and t_j do not match, we delete t_j and edit cost $D[i,j-1]+1$.

Then $D[i,j]$ is the minimum among the four costs above, That is,

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 & \text{corresponding to deleting } s_i; \\ D[i,j-1]+1 & \text{corresponding to deleting } t_j;; \\ D[i-1.j-1]+\delta(s_i,t_j) & \text{corresponding to either matching } s_i \text{ and } t_j\text{, or substituting } s_i \text{ for } t_j \end{cases}$$


y
ϵ G C T A T G C C A C G C
ϵ G C G T A T G C A C G C
D:
x
D[6, 5]
D[5, 5]
D[5, 6]
D[6, 6]
Value in a cell depends upon its upper, left, and upper-left neighbors
$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1]+\delta(x[i-1],y[j-1]) \end{cases}$
upper
left
upper-left



where $\delta(a,b) = 0$ if a and b match and 1 otherwise.

The algorithm needs to fill up a dynamic programming table of size n^2 and computing value for each cell requires $O(1)$. Hence, overall the time complexity is $O(n^2)$. Can we do better?

1.2 Speeding up Edit Distance Computation

When it comes to exact computation of edit distance, despite significant effort over decades, the best running time is $O(\frac{n^2}{\log^2 n})$ due to Masek and Paterson. Such small polylogarithmic improvements can often be obtained over a dynamic programming solution by a method known as the *Four Russian Method*. The idea comes from a paper by Arlazarov, Dinic, Kronrod, and Faradzev, concerning boolean matrix multiplication. Though, only one of the authors is Russian, this method is known in the West as the *Four Russian Method*.

The rough idea of the Four-Russian method is to partition the dynamic programming table into $t \times t$-submatrices which we call t-blocks, and compute the essential values in the table one t-block at a time, rather than one cell at a time. The goal is to spend $O(t)$ time per block (rather than $\Theta(t^2)$ time), achieving a factor of t speed-up over the standard dynamic programming solution.

Consider the standard dynamic programming approach to computing the edit distance of two strings s and t. The value $D(i,j)$ is determined by the values in its three neighboring cells $(i-1,j-1),(i-1,j),(i,j-1)$ and s_i and t_j as long as $i,j > 0$. Similarly, by extension, the values given to the cells in an entire t-block, with upper left-hand corner at position (i,j) say, are determined by the values in the first row and column of the t-block together with the substrings $s_i s_{i+1} ... s_{i+t}$ and $t_j t_{j+1} ... t_{j+t}$. Therefore, if we know the values in the first row and column, we can compute the values for the entire block. We therefore keep a look-up table where there is an entry for every possible values of the first row and column and the substrings indicating the value on the last block and last row (not quite!).

Suppose $n = k(t-1)$. The entire dynamic programming table has size $(n+1) \times (n+1)$ including the index for the 0th position. We can decompose this dynamic programming table in k t-blocks such the adjacent blocks row-wise share one column, and the adjacent blocks icolumn-wise share one row. See Figure 1.

Since, there are $\frac{n^2}{t^2}$ blocks, assuming to output the last row and column requires $O(t)$ time, the total time requirement will be $O(\frac{n^2}{t})$. The total time to construct the look-up table is $\approx O(2^{2t})$. Setting $t = \frac{\log n}{2}$, we get a runtime improvement of $O(\frac{n^2}{\log n})$.

It is not clear from this description why the look-up table construction should take about $O(2^{2t})$ time, since at this points it seems like the input to each t block can have its first row consisting of any numbers from 0 to n, and not just binary. For further details, see here[1].

[1]http://cs.au.dk/$\sim$cstorm/courses/AiBS_e12/papers/Gusfield97_FourRussians.pdf

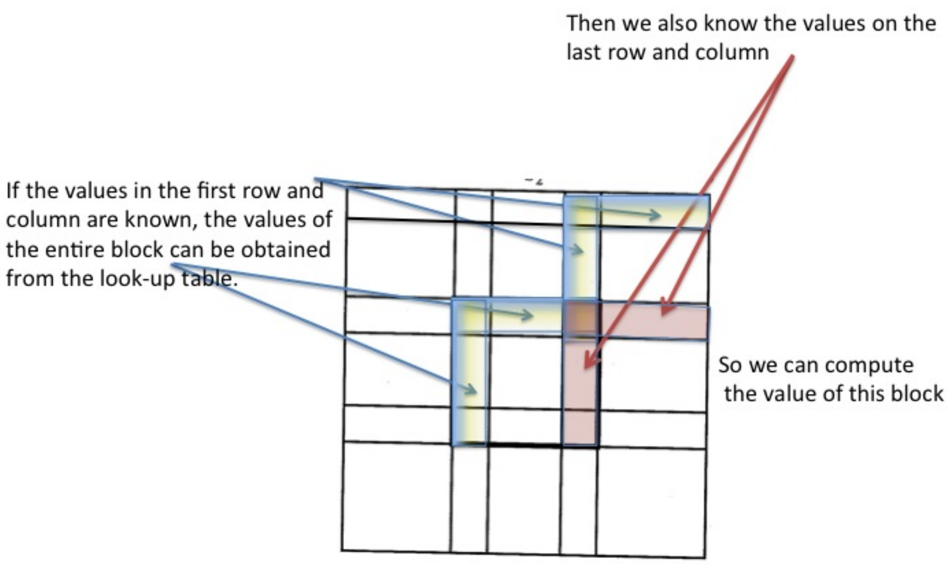

Then we also know the values on the last row and column
If the values in the first row and column are known, the values of the entire block can be obtained from the look-up table.
So we can compute the value of this block

Figure 1: The Four-Russian Method

Also read this blog article `https://rjlipton.wordpress.com/2009/03/22/bellman-dynamic-programming-and-edit-distance/`.

Improving the running time to $O(nd)$. Now, suppose we know an upper bound d on the actual edit distance. Then, we know that an optimum algorithm does not need to compute $D[i,j]$ where $|i-j|>d$. Hence, in the dynamic programming table, we only need to compute $2nd$ entries where computing each entry takes only $O(1)$ time. This leads to an $O(nd)$ algorithm. Often, in practice d is small, hence the algorithm runs much faster. In fact, the algorithm can be made to run in $O(n+d^2)$ time.

Impossibility of subquadratic algorithm for edit distance computation. It has been recently shown [1] that string edit distance does not have an $O(n^{2-\delta})$ algorithm for some positive constant δ unless k-SAT on n variables and m clauses has an algorithm running in $O(m^{O(1)}2^{n(1-\epsilon)})$ for a constant $\epsilon > 0$–the later will violate the *Strong Exponential Time Hypothesis*. Even improving over the Masek and Peterson's result is unlikely [2].

2 All-Pairs Shortest Paths

Suppose we are given a graph $G=(V,E)$ with edge weights $w: V \to \mathbb{R}$–both positive and negative (but no negative weight cycle). Our goal is to compute the shortest path between every pair of nodes. Using Floyd-Warshall's algorithm, one can do that in $O(|V|^3)$ time using again a dynamic programming.

Let $|V|=n$, and consider the vertices to be numbered 1, 2, ..., n. Let $\textit{shortest}(i,j,k)$ denote the weight of the shortest path between i j using only vertices 1 through k. For each of these pairs of vertices, the true shortest path could be either (i) a path that only uses vertices in the set $\{1,2,..,k\}$, or (ii) a path that goes from i to $k+1$ (using only vertices from 1 to k), and then $k+1$ to j (using only vertices from 1 to k). Hence,

$$\textit{shortest}(i,j,k+1) = \min\left(\textit{shortest}(i,j,k), \textit{shortest}(i,k+1,k) + \textit{shortest}(k+1,j,k)\right)$$

The base case is

$$\textit{shortest}(i,j,0) = w(i,j)$$

Running time of the algorithm is $O(n^3)$–it completes an n^3 sized dynamic programming table, and computing each entry takes constant amount of time.

Even after repeated attempts, this $O(n^3)$ running time has not been improved substantially. The best known running time when the edge weights are integers is $O(\frac{n^3}{2^{\sqrt{\log n}}})$ [3].

2.1 Shortest paths with small integer weights

When the edges weights are small, say integers bounded in between $[-M, +M]$, then the all pairs shortest path can be computed in $O(Mn^\omega)$ time where $\omega = 2.373$ is the exponent of the fast matrix multiplication. To do so, we look at the all-pairs shortest path computation through a different angle.

Given the weighted graph, define a matrix A such that $A[i, i] = 0$ for all i, $A[i, j] = w(i, j)$ if there is an edge between i and j, and ∞ otherwise. That is, $A[i, j]$ is the length of the shortest path from i to j using 1 or fewer edges. Now, following the basic dynamic programming idea, we can use this to produce a new matrix B where $B[i, j]$ is the length of the shortest path from i to j using 2 or fewer edges.

$$B[i, j] = \min_k (A[i, k] + B[k, j])$$

I.e., what we want to do is compute a matrix product $B = A \times A$ except we change "*" to "+", and we change "+" to "min" in the definition. In other words, instead of computing the sum of products, we compute the min of sums.

What if we now want to get the shortest paths that use 4 or fewer edges? To do this, we just need to compute $C = B \times B$ (using our new definition of matrix product). I.e., to get from i to j using 4 or fewer edges, we need to go from i to some intermediate node k using 2 or fewer edges, and then from k to j using 2 or fewer edges. So, to solve for all-pairs shortest paths we just need to keep squaring $O(\log n)$ times. Each matrix multiplication takes time $O(n^3)$. So the overall running time is $O(n^3 \log n)$.

So what did we gain?

In fact, this new matrix product can be computed in $O(Mn^\omega)$ time when the edge weights are small [**?**], using which, it is possible to obtain an $OMn^\omega)$ time algorithm for all-pairs shortest path when the edge weights are integers bounded by M.

References

[1] Backurs, Arturs, and Piotr Indyk. "Edit distance cannot be computed in strongly subquadratic time (unless seth is false)." In Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, pp. 51-58. ACM, 2015.

[2] Abboud, Amir, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. "Simulating branching programs with edit distance and friends or: A polylog shaved is a lower bound made." n 47th ACM Symposium on Theory of Computing (STOC 2015).

[3] Williams. R. "Faster All-Pairs Shortest Paths via Circuit Complexity. " In 46th ACM Symposium on Theory of Computing (STOC 2014).

[4] Alon, N., Galil, Z., Margalit, O., "On the exponent of all pairs shortest path problem", Proceedings, 32th Annual Symp. on Foundation of Comput. Sci, 1991.