Dynamic Programming and Fibonacci numbers

Dynamic programming (DP) is used when there is a need to store intermediate results so that we do not have to recompute them. A very simple example is the computation of Fibonacci numbers, which is done using the following recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \]

An algorithm that will simply use this formula will compute correctly the \( n \)^{th} Fibonacci number, but there will be an exponential number of recursive calls. As we can see in Figure 1, in order to compute \( F(n) \) we will need to compute \( F(n - 1) \) and \( F(n - 2) \), but \( F(n - 1) \) will also need the value of \( F(n - 2) \). As this kind of re-computations will occur, the number of recursive calls will grow exponentially and so, the naive algorithm will be very inefficient.

A simple way to solve this problem using DP is to use an array. Let this array be \( F \). Initially, all the values in the array will be \(-1\); \( F[n] = -1 \) will mean that the \( n \)^{th} Fibonacci number has not been computed yet (note that these numbers are always positive, thus the choice of \(-1\)). Of course, we will also initialize \( F[0] \) and \( F[1] \) with 1. Considering this, the Fibonacci numbers can be computed as in Algorithm 1.

Note that the array \( F \) is globally shared by all recursive calls and so, no Fibonacci number is going to be computed more than once. While this is not a very interesting algorithm, it obeys the fundamental characteristic of DP, namely storing intermediate results so that they can be reused rather than recomputed.
Figure 1: The recursive calls for computing the $n^{th}$ Fibonacci number

**Algorithm 1** Computing the Fibonacci numbers using DP

\[
F(n) = \begin{cases} 
F(i) & \text{if } F[i] > 0 \\
F[i - 1] + F[i - 2] & \text{else}
\end{cases}
\]

**Pairwise shortest paths**

Another example of DP is the computation of all pairwise shortest paths in a graph (all vertices $x$ to all other vertices $y$). This could be resolved by running Dijkstra’s algorithm for each vertex, but this is not the most “elegant” solution. We can see that such an algorithm will need to return a matrix containing all the distances between the vertices; actually, as this matrix is symmetric (the distance from $i$ to $j$ is the same as from $j$ to $i$ in an undirected graph) we need only half of it. Let this matrix be $P$; its will contain $n \times n$ elements, where $n$ is the number of vertices.

In such a graph, the initial upper bound of the shortest path from node $i$ to $j$ will be the length of the edge $(i, j)$, if this edge exists. Note that, this is not necessarily the shortest path, but the one with the fewest edges. If the mentioned edge does not exist, the shortest path needs to be discovered, and so the distance between the two vertices is initially unbounded. Mainly, we are starting with the adjacency matrix and the algorithm will have to iteratively improve this bound until the shortest path is obtained. Let this
matrix be $P$ and so we have:

$$P(i, j) = \begin{cases} 
\text{the length between } i \text{ and } j & \text{if } (i, j) \in E \\
+\infty & \text{otherwise}
\end{cases}$$

Notice that updating the matrix in the usual way (using loops) is somehow difficult in this problem. What we can do instead, is first look at the paths from $i$ to $j$ that contain only one other vertex, and compare these paths with the initial distance between the two vertices. We iteratively increase the number of edges on the paths; note that we will not have to recompute the whole distance at each step, as smaller (in terms of traversed vertices) paths have already been computed at previous step.

But instead of looking at the number of edges on a given path, we will increase the number of vertices that are allowed on the path. Consider that the vertices are ordered and at each step we allow one more of them to be on the path between two nodes. For example, at step 1, between $i$ and $j$ we allow only $v_1$ to be on the path, at step 2 we allow $v_1$ and $v_2$ and so on. By allowing more vertices on a given path we increase its complexity, but the computation is much easier to organize, and this is what we want. Using these ideas we introduce Algorithm 2.

**Algorithm 2** Computing the shortest path between all nodes in $G$

$Floyd'sAlgorithm(G)$$$

1: for $k = 1$ to $n$ do
2: \{$n$ is the number of vertices and $k$ limits the index for the allowed vertices on a path$\}$$$
3: for $i = 1$ to $n$ do
4: \{for $j = 1$ to $n$ do
5: \{if $P(i, k) + P(k, j) < P(i, j)$ then
6: $P(i, j) \leftarrow P(i, k) + P(k, j)$$$
7: waypoint(i, j) \leftarrow k$$$
8: \} end if
9: \} end for
10: \} end for
11: \} end for

The basic idea is that, if at the step in which vertex $v$ is allowed on the path between $i$ and $j$, a new better distance is found, than $v$ must be on that path. All we have to do is check the paths from $i$ to $v$ and from $v$ to
and if the sum of the distances of those paths is better than the current length of the path then we can replace it - this is what the innermost loop in Algorithm 2 does.

Another characteristic of DP can be observed in our algorithm; without the statement in line 7, the shortest path could not be reconstructed. By adding this statement we make sure that the last vertex (intermediate point) added between $i$ and $j$ is stored; let this vertex be $l$. We can now reconstruct the path from $i$ to $j$ in a divide-and-conquer manner: we rebuild the path from $i$ to $l$ and the path from $l$ to $j$. This shows us that DP is divide-and-conquer where we do not know how to divide. If we would have known at the start of the algorithm that $l$ is "between" $i$ and $j$ we could have divided the computation into the two paths.

Again, Algorithm 2 is not the most efficient one (it has cubic worst-case running time while running Dijkstra’s multiple times has the time complexity $O(|V|^2 \cdot \log |V|)$), but is more simple and somehow faster as it contains very basic control structures and operations. Anyway, DP is rarely the most efficient solution as in most cases is a brute-force approach.

Sequence analysis

Another application of DP is analyzing the Hamming distance between two strings. This is very useful if we think about DNA as a sequence of four characters $\{C, G, T, A\}$; we would like to compare such sequences, finding resemblances and differences to gain a better understanding of living organisms.

A more common instance of this problem is spell checking. Given a string, we would like to output the closest string that is legal in the language. When we are interested in the most likely string, we have to take into account characteristics of the language. Thus, different types of typing mistakes are considered: insertions, deletions, replacements (substitutions), and reversals. The goal is to propose a most likely correction, based on a dictionary and the frequency of each type of typing mistake. This is done by using DP.

We analyze this problem by looking at the particular case of biological sequences. Compared to natural language, the alphabet is much smaller: four letters for DNA, and 20 for protein sequences. We would like to know, given two (similar) such sequences, what are the differences between them. For that, we will use events like in the case of spell checking: deletions (a character is missing), insertions (an extra character is introduced) - both
referred to as indels, and substitutions (one character is replaced by another). Reversals are not taken into account since this is more often encountered in typing, and not so much in the biological processes.

For example, given the sequences in Figure 2, we would like to know how one was obtained from the other in terms of the events previously described, with a minimum number of transformations. This analysis is missing the notion of probabilities: any event is considered to be equally likely. For now, we assume that indels and substitutions have the same probability to occur.

The first thing we notice is that they have quite a lot in common. Also, they have different lengths, so some indels occurred for sure. Note that insertions and deletions are equivalent: it is a matter of looking from top to bottom or vice versa.

We are going to do this by considering a (DP) matrix (Figure 3), with a row and a column for each character. We can think of the elements of the matrix as a mapping between positions (indices) in the two strings. The aim is to provide an algorithm that tells us how to “move” in the two strings. Namely, for a given index $i$ in the first string, we would like the corresponding index in the second string is $j$. If we have a substitution, or an exact match (which we can think of as a substitution), both $i$ and $j$ increase at the same time, while an indel will only increase one index (virtually introducing a space in the other sequence).

An assignment of indices will be a path in the matrix. A solution (called alignment) will be a path from the top left corner (corresponding to indices $(1, 1)$) to the bottom right corner (the pair $(10, 11)$ of indices). The path can go through diagonals (increasing $i$ and $j$ at the same time - substitution), or in a vertical/horizontal direction (for an insertion/deletion).

To choose between paths in the matrix, we need costs. For any indel, we consider the same penalty. However, for substitution penalties, recall that a perfect match is also considered a substitution - the cost for a match is in fact a bonus. Moreover, it might be the case that some characters are more likely to be replaced by certain others. This can be expressed by a simple substitution matrix (Figure 4), with values obtained by biological experiments.

All that remains now is to see how the DP matrix is populated with
values. As in the pairwise shortest path, we try to use previously obtained information.

In the DP matrix (Figure 5), the element at line $i$ and column $j$ corresponds to the index mapping for the two sequences. We can reason that this step was either a substitution (or a match), meaning that we arrived from $(i - 1, j - 1)$, a deletion in the first sequence - we went horizontally from $(i, j - 1)$, or a deletion in the second sequence, in which case we arrived from $(i - 1, j)$.

If we denote by $cost(i, j)$ the element at line $i$ and column $j$, we obtain the following recurrence relation for $cost(i, j)$:

$$
\begin{align*}
    cost(i, j) &= \begin{cases}
        cost(i - 1, j - 1) + \text{cost of substituting } S_1[i] \text{ with } S_2[j] \\
        cost(i - 1, j) + \text{cost of indel} \\
        cost(i, j - 1) + \text{cost of indel}
    \end{cases}
\end{align*}
$$

By this recurrence relation, we fill the matrix from left to right, going downwards. The running time of this algorithm is obviously linear, but at the end all we get is a number - the cost of the best alignment. To fix this, like we did in the pairwise shortest path, we simply add waypoints to know how we obtained each element in the matrix.
This algorithm is called *global sequence alignment*, as opposed to more often instances where biologists want to find alignments on subsequences (*local sequence alignment*). In computer science, it is known as *string-to-string editing*.

To complete this algorithm with the notion of probabilities, we next introduce *Hidden Markov Models*.

**Hidden Markov Models**

Consider the following scenario of a dishonest casino. A game of dice is played with just one die, which from time to time is changed; the casino might change the die to a loaded one, which outputs sixes more often than the other five numbers. Given a sequence of produced numbers, can we pinpoint the most likely places where the dice was swapped?

For the example in Figure 6, we would be interested what die was used to produce the rolls $i$, $j$ and $k$, even if we suspect that the first two are most likely produced with a correct die, and the long sequence of repeating sixes was rolled with the loaded one.

To formalize this, we consider a Markov Chain, with two states: *fair* and *loaded*. In the fair state, the die is correct and it’s equally likely that any number is produced, while in the loaded state a six is more probable than
any other number.

Considering we know the state transitions probabilities, we obtain a Hidden Markov Model (Figure 7); it is called hidden because in our experiment we don’t know what the states are, and instead try to deduce what is the most likely sequence of states.

The algorithm we want to develop will assign a most likely state to each roll, and deduce when the dice were switched. This is done using a DP approach, with an algorithm called Viterbi.