Chapter 1

Dynamic programming via DAGs

By Sariel Har-Peled, January 3, 2023[①]

> The events of 8 September prompted Foch to draft the later legendary signal: "My centre is giving way, my right is in retreat, situation excellent. I attack." It was probably never sent.
>
> – – The first world war, John Keegan..

Version: 0.1

1.1. Dynamic programming via DAGs

Dynamic programming gives rise to a natural dependency graph. Sometimes, solving the DP then can be turned into a graph problem on this graph. This provides yet another way of thinking about DP problems.

The configurations

Specifically, the definition of what the subproblem the DP is trying to solve, provides us with an induced configuration space – namely on the different distinct subproblems the DP might consider.

Example 1.1.1. Let $S[1 \ldots n]$ and $T[1 \ldots m]$ be two input strings for the edit-distance problem. The recursive subproblem we are solving here is the minimum edit distance between the prefix $S[1 \ldots i]$ and the prefix $T[1 \ldots j]$. Let $f(i, j)$ denote this value. The configuration space in this case is

$$\mathsf{V} = \{0, \ldots, m\} \times \{0, \ldots, n\} = \{(i, j) \mid i = 0, \ldots, m, j = 0, \ldots, n\}.$$

The value of $f(i, j)$ is associated with the configuration (i, j).

The dependencies

When the computing the value of a subproblem (recursively), associated with a configuration v, one computes recursively the value of other subproblems, say $v_1, \ldots, v_k$. In such a scenario, v ***depends*** on v_i, for $i = 1, \ldots, k$ (we only consider direct dependencies). We introduce the edges (v, v_i), for $i = 1, \ldots, k$, to capture these dependencies. This gives rise to a directed graph, $\mathsf{G} = (\mathsf{V}, \mathsf{E})$. with the edges being

$$\mathsf{E} = \{u \longrightarrow v \mid u, v \in \mathsf{V}, \text{ and } u \text{ depends on } v\}$$

This is a directed graph that can not have cycle – if there was a cycle then it would have circular dependency, and the optimal value can not be computed. Directed graphs without cycles are known as DAG]s.

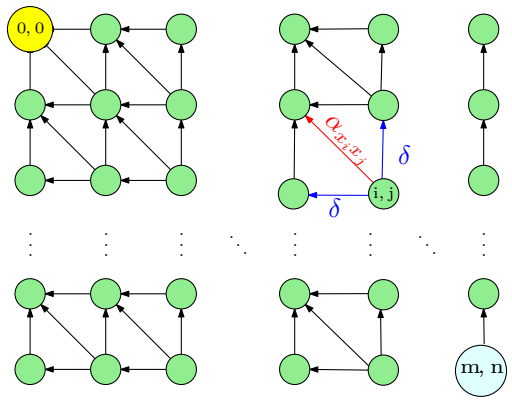

Figure 1.1: The dependency graph for edit-distance.

Example 1.1.2. As a reminder, for simplest version of edit distance, the recursive function was

$$f(i,j) = \begin{cases} 0 & i = 0 \text{ and } j = 0 \\ 1 + f(0, j-1) & i = 0 \\ 1 + f(i-1, 0) & j = 0 \\ \min \begin{cases} \alpha(S[i], T[j]) + f(i-1, j-1) \\ 1 + f(i-1, j) \\ 1 + f(i, j-1) \end{cases} & \text{otherwise.} \end{cases}$$

where 1 is the price of insertion/deletion, and $\alpha(x,y) = \begin{cases} 0 & x = y \\ 1 & x \neq y \end{cases}$ is the price of changing from letter x to letter y.

As such, for any $i > 0$ and $j > 0$ we have the edges

$$(i,j) \to (i-1, j-1), \qquad (i,j) \to (i, j-1), \qquad \text{and} \qquad (i,j) \to (i-1, j)$$

in the dependency graph. In addition, we also have the edges

$$\forall i > 0 \qquad (i,0) \to (i-1, 0), \qquad \text{and} \qquad \forall j > 0 \qquad (0,j) \to (0, j-1).$$

This dependency graph is depicted in Figure 1.1.

Converting the DP into a graph problem

In many DP problems, there is a price associated from moving from one configuration to another configuration that it directly depends on. A natural way to model this is to put a price on the edges in the dependency graph that captures this.

Example 1.1.3. for edit distance, if we have an edge $(i,j) \to (i-1, j-1)$ in the dependency graph its price is $\alpha(i,j)$.

The dynamic problem can then be interpreted as locally moving to the adjacent configuration, such that the price on the dependency edge, plus the price of the target configuration is minimized. Globally, this corresponds in finding the shortest path in the dependency graph from a start vertex to a target vertex.

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	3	3	3	2	2	2

D	R	E	A	D
D	E	E		D

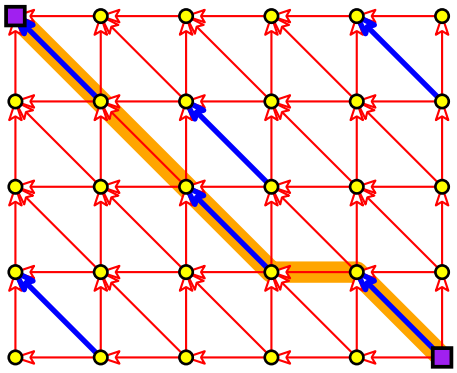

Figure 1.2: Edit distance between `DREAD` and `DEED`. The blue edges have price 0, all other edges have price 1.

Example 1.1.4. For edit-distance, we price all the horizontal and vertical edges as having price one. Diagonal edges have price 0 or 1 depending if their correspond to aligning identical or different characters, respectively. The problem is then computing the shortest path from (m, n) to $(0, 0)$ in this dependency graph, under this pricing function. This is illustrated in Figure 1.2.

A benefit of computing the shortest path in the dependency graph is that not only we computed the price of the optimal solution, but one can then easily extract the optimal solution from the computed path. We state the following without providing any details (for now):

Lemma 1.1.5. *The shortest path in a DAG with weights on the edges (positive or negative) can be computed in linear time in the number of vertices and edges of the graph. The same holds for the longest path in the DAG.*

This readily implies an efficient algorithm that can be modeled as a shortest path problem on the DAG– compute the dependency graph of the problem at hand (for the given input), compute the shortest path, and output the shortest path, and the solution it encodes. In particular, the above examples implies the following.

Lemma 1.1.6. *Given two strings $S[1 \ldots m]$ and $T[1 \ldots n]$, computing the minimum edit distance between S and T can be done in $O(mn)$ time (and the sequence of edit operations realizing the edit distance), via computing shortest path in DAG.*

Proof: We compute the dependency graph G for the edit distance as described above, and for every edge we assign weight 0 or 1 as appropriate. We then compute the shortest path between (m, n) and $(0, 0)$ in G. Since G is a DAG, this takes $O(mn)$ time using the algorithm of Lemma 1.1.5. ■

1.1.1. Example: Longest increasing subsequence

The input is a sequence $\alpha_1, \ldots, \alpha_n$ of n real numbers, and the task is to compute the longest increasing subsequence. To this end, we create a graph, with vertices being

$$\mathsf{V} = \{s, t, 1, 2, \ldots, n\},$$

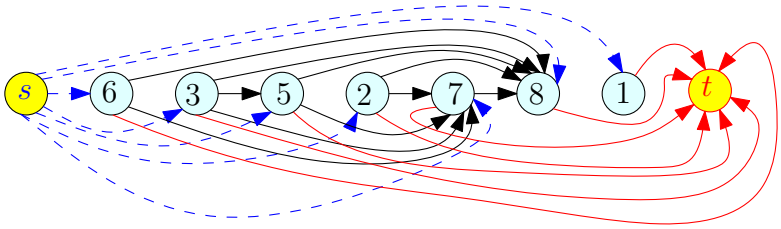

Figure 1.3: Longest increasing subsequence via longest path in a DAG. Here the input sequence is $6, 3, 5, 2, 7, 8, 1$.

and the set of edges being

$$\mathsf{E} = \left\{ i \to j \;\middle|\; 1 \le i < j \le n \text{ and } \alpha_i < \alpha_j \right\} \cup \left\{ s \to s, i \to t \;\middle|\; i = 1, \ldots, n \right\}.$$

Let $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ be the resulting graph. Clearly, as the edges go from lower values to higher values, this graph does not have cycles. Namely, it is a directed acyclic graph (i.e., DAG). See Figure 1.3 for an example of this graph.

Clearly, an increasing subsequence $\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_k}$ corresponds a path $s \to i_1 \to i_2 \to \cdots \to i_k \to t$ in the resulting graph. This correspondence also works in the other direction – any path between s and g in G encodes an increasing subsequence. Thus, computing the longest increasing subsequence is equivalent to computing the longest path in this DAG. Since computing the longest path in a DAG can be done in linear time in the size of the DAG, and this DAG can be computed in $O(n^2)$ time, we get the following result.

Lemma 1.1.7. *Let $\alpha_,, \ldots, \alpha_n$ be a sequence of n numbers. One can compute the longest increasing subsequence, in $O(n^2)$ time, by computing the longest path in a DAG.*