Dynamic Programming

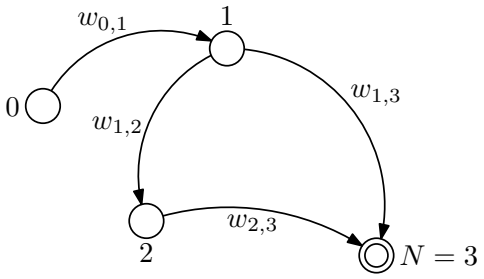
Herman Kamper

Wikipedia describes *dynamic programming* as an optimization or computer programming method where a complicated problem is broken "down into simpler sub-problems in a recursive manner."[1] This is quite vague. This note tries to describe dynamic programming in a more concrete (but still general) way.

Dynamic programming is finding the shortest path on a DAG

This section is roughly based on [Mensch and Blondel, §3.1]. They state that all dynamic programming problems can be (re-)formulated as finding the shortest path between a start and end node in a weighted directed acyclic graph (DAG).

Formally, a DAG consists of a number of nodes (vertices) and weighted edges. Let $w_{i,j}$ denote the weight between parent node i and child node j. Let us also label the nodes from 0 (the start node) to N (the end node). We can do so without loss of generality. An example of such a graph is shown below.



We want to find the path through such a graph that gives the lowest cost when adding up all the weights on that path. Let us define α_i as the cost of the best path up to node i. I.e., α_i is the sum of the weights on the shortest path from node 0 to node i. We will define it more formally in just a little bit. Given this definition, we can find the shortest path by applying the following *forward* recursive steps:

$$\alpha_0 = 0$$
$$\alpha_i = \min_{j \in \mathcal{P}_i} [w_{j,i} + \alpha_j] \qquad \forall\ i = 1, \ldots, N-1$$

with $\mathcal{P}_i$ the set of parent nodes of i.

[1] https://en.wikipedia.org/wiki/Dynamic_programming

To find the best path after calculating all of the forward variables, we now need to *backtrack* through the graph, starting with the final node N:

$$q_N = \arg\min_{j \in \mathcal{P}_N} [w_{j,N} + \alpha_j]$$

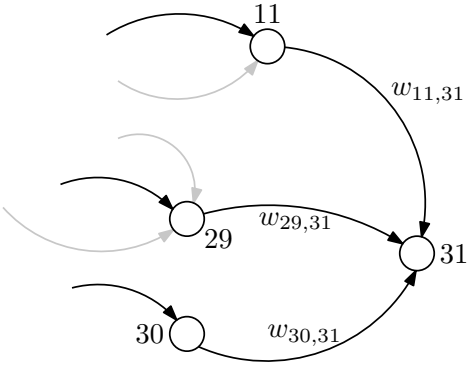
Here q_N denotes the node from which we moved into N on the optimal path. We can then determine from which node we moved into this penultimate node, and so on. In general we will have

$$q_i = \arg\min_{j \in \mathcal{P}_i} [w_{j,i} + \alpha_j]$$

So, after calculating q_N, we set $i = q_N$ and determine the optimal parent node of i using the equation above. We continue to do this, each time setting $i = q_i$ until we backtrack to the start node where $q_i = 0$. This backtracking step can either be done after you've completed the recursive forward steps for calculating the α_i's, or you can keep track of the history of where you came from (stored in the q's) while calculating the α's.

Why does this work?

Let's first answer this question intuitively, using the figure below. Let's say we've run through the algorithm, and we've calculated α for nodes 11, 29 and 30. We therefore know the best way to get to each of these nodes from the first node (0), and we know the cost of getting there. Now, we want to calculate α_{31}, i.e., we want to know the best path of getting to node 31 and we want to know what the cost will be. The optimal path to this node will definitely go through one of its parents (11, 29 or 30). So we simply calculate the total cost for the three paths through the parents. And, since we know how to get to each of the parents (and the cost of getting there), this is easy: you just add the cost of getting to the parent node with the weight of then moving to node 31, i.e. $\alpha_{31} = \min_{j \in \{11,29,30\}} [w_{j,31} + \alpha_j]$. This is exactly as in our algorithm!



Why does this work? Let's now answer this question formally. We start by mathematically defining α_i as follows:

$$\alpha_i \triangleq \min_{\mathcal{Q}_i} \sum_{(u,v) \in \mathcal{Q}_i} w_{u,v}$$

$\mathcal{Q}_i$ denotes potential paths from node 0 to node i. Formally, it can be seen as a set of tuples (u, v) of start-end nodes which are connected along this path. For instance, in the first figure

you could have $\mathcal{Q}_3 = \{(0,1),(1,3)\}$. The definition therefore defines α_i as the cost along the optimal path from node 0 to i (as explained earlier, here it is just formalized). To show why the recursive dynamic programming equations solve the problem of finding the most optimal path, we need to show that the steps in the forward equations actually matches this definition.

We show this by writing out α_i as follows:

$$\begin{aligned}
\alpha_i &= \min_{\mathcal{Q}_i} \sum_{(u,v)\in\mathcal{Q}_i} w_{u,v} \\
&= \min_{\mathcal{Q}_j} \min_{j\in\mathcal{P}_i} \left[w_{j,i} + \sum_{(u,v)\in\mathcal{Q}_j} w_{u,v} \right] \\
&= \min_{j\in\mathcal{P}_i} \left[w_{j,i} + \min_{\mathcal{Q}_j} \sum_{(u,v)\in\mathcal{Q}_j} w_{u,v} \right] \\
&= \min_{j\in\mathcal{P}_i} \left[w_{j,i} + \alpha_j \right]
\end{aligned}$$

This is exactly the equation given above in the forward steps! This completes the proof that the calculation of α_i in the forward equations indeed corresponds to the definition.

A bit more general

Erickson describes the shortest path algorithm a little more generally. He states that any shortest path algorithm can be formulated according to the pseudo code:

```
# Initialise
alpha[0] = 0            # cumulative cost
q[0] = None             # predecessor (history) of starting node undefined
for i = 1 to N:
    alpha[i] = inf      # start with undefined path
    q[i] = None

# Relax edges
while there is at least one tense edge:
    # Relax any tense edge
    alpha[i] = w[j, i] + alpha[j]
    q[i] = j
```

A "tense" edge is described as an edge between nodes (j,i) where $w_{j,i} + \alpha_j < \alpha_i$. I.e., the current (tentative) α_i is incorrect since moving through node j actually provides a shorter way to get from node 0 to i. We should therefore reassign it to the shorter path, and we continue to do this until there are no more tense edges.

We see that the DAG formulation in the first section matches the definition here. What I like about this slightly more general formulation is that it highlights that there are actually still some design decisions required to solve the problem! In what order should we consider

the nodes? For the DAG formulation, if we decide to always go from node 0 to node N, how do we number our nodes? And, for a complete algorithm, we will of course also need to know the edge weights and whether there are any constraints on the connectivity.

In practice

In some cases it is convenient to have a multi-indexed cumulative cost α. You see this, for instance, in the typical implementations of dynamic time warping, hidden Markov models and the edit distance algorithm.

As a brief example of why this could be useful, consider the case where we want to find the shortest path through the graph, but we want the path to consist of exactly C connections. For instance, if $C = 5$, then we are requiring the algorithm to move along exactly five edges: it can't move directly from the start to the end node (one step), but it can also not take six steps to get from the start to the end node.

In this case we can define

$$\alpha_{i,c} \triangleq \min_{\mathcal{Q}_i} \sum_{(u,v)\in\mathcal{Q}_i} w_{u,v} \qquad \text{such that } |\mathcal{Q}_i| = c$$

I.e., $\alpha_{i,c}$ is the score of the optimal path to node i consisting of exactly c connections. We can calculate $\alpha_{i,c}$ recursively as

$$\alpha_{i,c} = \min_{j\in\mathcal{P}_i} [w_{j,i} + \alpha_{j,c-1}] \qquad \forall\ i = 1, \ldots, N \text{ and } c = 1, \ldots, C$$

We will then backtrack, starting with the final node N with C connections, i.e. we will consider the $\alpha_{N,C}$. Let's say it's parent is node j. From the parent j we will then backtrack using $\alpha_{j,C-1}$, i.e. the quickest way to get to node j along a path of $C - 1$ connections. And so forth.

Importantly, this solution could be reformulated to match the DAG formulation exactly: we could simply think of a graph where each node is indexed by a tuple (i, c), i.e. the graph would consist of $(N + 1) \times C$ nodes.

If this multi-indexed idea doesn't make entire sense, that's okay. If you follow the steps for deriving dynamic time warping, you should be able to see this connection.

Acknowledgements

Thanks to Leanne Nortje and Christiaan Jacobs for catching mistakes and for helpful feedback.