Dynamic Programming

1 Another look at shortest paths in dags

In the conclusion of our study of shortest paths in graphs we noticed that the problem becomes especially easy when the graph is a dag. What makes the shortest path algorithm run so fast on dags is that each edge is updated only once. Hence, the \textit{dist} of a node \(v\) can be found by just comparing the \textit{dist}(\(u\)) + \(l(u,v)\) of all its predecessors. The algorithm can be recast as follows:

\begin{verbatim}
for each \(v \in V\), in topological order:
  if \(v = s\): \textit{dist}(s) = 0
  else: \textit{dist}(v) = \min_{u,v \in E}\{\textit{dist}(u) + l(u,v)\}
\end{verbatim}

All we do is march from node to node, further and further away from \(s\), and calculate the distance from \(s\) of each node by examining the distances of its predecessors. (If a node \(v\) is not reachable from \(v\), its \textit{dist} is set, quite naturally, to \(\infty\), the minimum over the empty set.)

There are several ways of looking at this simple algorithm. Here is one of the most helpful: We are solving a problem (find the shortest path from \(s\) to \(t\)) by solving larger and larger subproblems. Indeed, finding the shortest path from \(s\) to \(a\) is a subproblem whose solution is needed in order to find the shortest path from \(s\) to \(t\). And these subproblems can be thought of as becoming larger (harder to solve, requiring the solution of more subproblems) as we venture further down in the topological order. For \(s\), the subproblem is so simple that we immediately know its answer: zero.

\textit{Dynamic programming} is a very powerful algorithmic style in which, in order to solve a problem, we identify subproblems and solve them one-by-one, smaller first, using the solutions for the small problems to solve the larger ones until the whole problem is solved. The algorithm for shortest paths in a dag is in some sense the archetypical dynamic programming algorithm.

But there is another way of looking at this algorithm. Fundamentally it entails filling a table, calculating a function, \textit{dist}, of the nodes, where the value of this function at a node depends on the values at the node's predecessors in the dag, and certain local data (the lengths of the edges).

It so happens that the function we are calculating involves taking the minimum of sums, but this is almost irrelevant: We might as well calculate the maximum. Or we could have \(\cdot\) instead of \(+\) inside the brackets, thus calculating the path with the largest \textit{product} of lengths. The same algorithm would accomplish all these tasks.
boxed: The principle of optimality  
boxed: Pascal’s triangle: the oldest dynamic program  
boxed: The risks of recursion  
boxed: The mother of all algorithms?

1.1 Constructing the dag: longest increasing subsequences

Often solving a problem by dynamic programming involves finding shortest paths (of some sort) in a dag — except that we are not given the dag. Constructing the right dag is then the main difficulty in solving such a problem.

For example, consider this problem: We are given an array of \( n \) integers \( a_1, \ldots, a_n \), and we are asked to find the longest increasing subsequence, that is, the maximum \( m \) for which there are indices \( 1 \leq i_1 < \cdots < i_m \leq n \) such that \( a_{i_1} < a_{i_2} < \cdots < a_{i_m} \).

For example, if the array is 5, 1, 3, 2, 8, 6, 3, 9, 7, then the longest increasing subsequence is 1, 3, 6, 9. How do we find it efficiently?

It is natural to construct the following dag \((V, E)\): It has nodes \( V = \{1, 2, \ldots, n\} \), and there is an edge from \( i \) to \( j \) in \( E \) if and only if (a) \( i < j \) and (b) \( a_i < a_j \). That is, we draw an edge from each number to all subsequent larger numbers. It is a dag, as evidenced by the topological ordering 1, 2, \ldots, \( n \) (see Figure ??).

Furthermore, any path \( i_1, \ldots, i_n \) in this dag is an increasing subsequence, since the presence of the edges \((i_1, i_2), (i_2, i_3), \ldots, (i_{m-1}, i_m)\) implies that \( i_1 < i_2 \cdots < i_m \) and \( a_{i_1} < a_{i_2} < \cdots < a_{i_m} \). So, finding the longest increasing subsequence is tantamount to finding the longest path in this dag! Our shortest-paths algorithm can accomplish this, of course, by setting all edge lengths to −1. But here is a more direct version of the same algorithm:

\[
\begin{align*}
\text{for } j = 1, 2, \ldots, n: \\
\text{set } L(j) &= 1 + \max\{L(i) : (i, j) \in E\} \\
\text{return the largest value of } L
\end{align*}
\]
That is, the longest increasing subsequence ending at \( j \) is one plus the maximum \( L(i) \) over all predecessors \( i \) of \( j \) in the dag. (If there are no edges into \( j \), we take the maximum over the empty set to be zero.) Needless to say, in the end we return the largest of all \( L(i) \)’s observed. Notice that the second line of the algorithm requires a loop over all predecessors of \( j \) to be calculated; the adjacency lists of the reverse graph (recall ???) would be handy here. The algorithm is linear in \( E \), which however is about \( n^2 \) in the worst case — the worst case being when the given array is sorted in increasing order. Hence what we have described is an \( O(n^2) \) algorithm for the longest increasing subsequence problem.

For a faster \( O(n \log n) \) algorithm for this problem, see???.

1.2 Constructing the dag: edit distance

Given two strings, how close are they to each other? For example, of the three last names of the authors of this book, which two are the closest?

There is a very natural measure of distance between strings \( x \) and \( y \): The smallest number of edits — insertions, deletions, and overwritings of characters — that would transform \( x \) into \( y \) (or \( y \) into \( x \), since these edit operations are invertible, this distance measure is symmetric). For example, the edit distance between the string to and the string fro is 2: To turn to into fro, we overwrite t with f, and we insert r right after that, a total of two edits. And there is no way to achieve the same effect with one.

boxed: The story of BLAST

But how does one compute this distance? The answer is, by dynamic programming. Because, rather surprisingly, the edit distance of two strings is the shortest path in a particular dag that can be constructed from the two strings.

In designing a dynamic programming algorithm for a problem, the most crucial question is, what is a subproblem? In other words, what are the vertices of the underlying dag? Once the appropriate notion of a subproblem has been defined (one that allows us to use solutions of small problems to solve larger ones), then it is usually an easy matter to write the algorithm: Iterate solving one subproblem after the other, in order of increasing size.

Suppose that our problem is to find the edit distance of two strings, say PAPADIMITRIOU and VAZIRANI. What is a subproblem? One idea comes to mind: A subproblem should be something that goes part of the way in converting one string to the other; say, the smallest number of edits for converting some prefix of the first, say PAPAD, to some prefix of the second, say VAZ.

The next question is, how does one go from one pair of prefixes to another? Let us think.

If we know the edit distance between PAPAD and VAZ, call it \( E(5, 3) \), we can easily calculate the edit distance between certain other prefixes: The distance between PAPADI and VAZ is \( E(5, 3) + 1 \): convert PAPAD to VAZ, and delete the I. The distance between PAPAD and VAZI is also \( E(5, 3) + 1 \): convert PAPAD to VAZ, and then insert an I. Finally, the distance between PAPADI and VAZI is \( E(5, 3) \), simply because the same letter (I) comes next in both words — otherwise it would be \( E(5, 3) + 1 \).

In general, to compute the edit distance between two words, word \( x \) with \( m \) letters and word \( y \) with \( n \), we define \( E(i, j) \) to be the edit distance between \( x[i] \) and \( y[j] \), where by \( x[i] \) we denote the first \( i \) letters of \( x \) and similarly for \( y[j] \). Obviously, our final objective is to compute \( E(m, n) \).

But what is \( E(i, j) \)? To answer, suppose first that the \( i \)th letter of \( x \) is different from the
jth letter of y. But then, the last step in the conversion of \( x[i] \) to \( y[j] \) must be one of these three:

- a deletion, in which case \( E(i, j) = E(i - 1, j) + 1 \); or
- an insertion, implying \( E(i, j) = E(i, j - 1) + 1 \); or
- an overwrite, implying \( E(i, j) = E(i - 1, j - 1) + 1 \).

On the other hand, if the two last letters of the two prefixes are the same, then there is no increase in the edit distance, and the last option is replaced by:

- \( E(i, j) = E(i - 1, j - 1) \).

Therefore, we can calculate \( E(i, j) \) as the minimum of these three possibilities!

Notice that a principle of optimality is at work here: An optimal sequence of edits converting \( x \) to \( y \), always working at the earliest letter at which the two strings still differ, must consist of optimal sequences of edits converting various prefixes of \( x \) to prefixes of \( y \).

Finally, in dynamic programming we need a place to start, usually a place where the indices have zero values. This is easy in the present problem: What is \( E(0, j) \)? It is the fastest way to convert the 0-length prefix of \( x \) — the empty string — to the first \( j \) letters of \( y \). And we know the fastest way to do this: \( j \) insertions. Similarly, \( E(i, 0) = i - i \) deletions.

And this leads directly to a dynamic programming algorithm for computing the edit distance. Let \( \text{diff}(i, j) \) be 1 if the \( i \)th letter of \( x \) and the \( j \)th letter of \( y \) are different, and 0 otherwise:

\[
\text{for all } i \text{ and } j: \quad \text{set } E(i, 0) = i \text{ and } E(0, j) = j
\]
\[
\text{for } i = 1, 2, \ldots, m:\n\]
\[
\text{for } j = 1, 2, \ldots, n:\n\]
\[
\text{set } E(i, j) \text{ equal to the minimum of these three:}
\]
\[
E(i, j) = E(i - 1, j) + 1
\]
\[
E(i, j) = E(i - 1, j) + 1
\]
\[
E(i, j) = E(i - 1, j) + \text{diff}(i, j)
\]
\[
\text{return } E(m, n)
\]

This dynamic programming algorithm entails filling a two-dimensional table, going through the entries in any order, as long as the subproblems become larger and larger —that is to say, indices never decrease: Row after row as in the algorithm above, column after column (the two loops would nest the other way around), or in diagonals, incrementing \( i + j \).

But the same algorithm can also be seen as shortest paths in a dag. Namely, the dag that has as nodes \( V = \{(i, j) : 0 \leq i \leq j, 0 \leq j \leq n\} \) and as edges \( E = \{((i - 1, j), (i, j)), ((i, j - 1), (i, j)) : 1 \leq i \leq j, 1 \leq j \leq n\} \), all of length 1 — except for the edges \( ((i - 1, j - 1), (i, j)) \) when the \( i \)th letter of \( x \) is the same as the \( j \)th letter of \( y \), in which case the length of the edge is zero (see Figure ??). We are seeking the distance between nodes \( s = (0, 0) \) and \( t = (m, n) \).
Figure 1.4 The distance between PAPADIMITRIOU and VAZIRANI is 10.

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th>A</th>
<th>Z</th>
<th>I</th>
<th>R</th>
<th>A</th>
<th>N</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>P</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>P</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>I</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>M</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>R</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>I</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>O</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>U</td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

1.3 The knapsack problem
1.4 All-pairs shortest paths
1.5 The traveling salesman problem
1.6 One-dimensional dynamic programming
1.7 Dynamic programming in trees