Memoization and All-Pairs Shortest Path

Instructor: Richard Peng

Oct 7, 2016

• DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

• Main topics:
  – Memoization view of Dynamic Programming.
  – All-pairs shortest path.

• Textbook:
  – Chapter 6.6.
  – Relevant exercises: shortest reliable paths in Chapter 6.6, Ex 6.26 - 6.29.

We start by giving a different view of dynamic programming. Here instead of figuring out the states first, our starting point is the brute force search algorithm.

Recall the longest path problem in a directed acyclic graph, which is an ordering on the vertices so that any edge \( i \to j \) has \( i < j \).

For this problem we can simply write out a brute force search:

\[
\text{LONGEST}(v)
\]

1. Initialize \( answer \leftarrow 0 \).

2. For each \( u \to v \)
   
   (a) \( answer = \max\{answer, \text{LONGEST}(u) + l_{u\to v}\} \).

3. Return \( answer \).

This can take time at least \( 2^{n/2} \): consider a ‘ladder’ graph where each layer goes to the next. However, note that it only gets called with \( n \) different inputs: one for each \( v \).

So in some functional programming languages, these call values are remembered, and the answer will be instantly returned on a ‘redundant’ call.

Dynamic programming can be viewed this way:

1. The states are the call parameters.

2. The transitions are the recursive calls being made.
3. The base case are the terminations of the recursion.

One difference though is that this view no longer needs an ordering of the states: they are implicitly given by the order of the recursive calls, and the fact that no cycles are found.

Something similar can also be done for edit distance: in search, we simply consider what happened at the last step:

\[
\text{EditDistance}(x[1\ldots i], y[1 \ldots j])
\]

1. Initialize \(\text{answer} \leftarrow 0\).
2. If \(i > 0\) and \(j > 0\) \(\text{answer} \leftarrow \min\{\text{answer}, x[1\ldots i - 1], y[1\ldots j - 1] + 1\}\).
3. If \(i > 0\) and \(j > 0\) and \(x[i] = y[j]\) \(\text{answer} \leftarrow \min\{\text{answer}, x[1\ldots i - 1], y[1\ldots j - 1]\}\).
4. If \(i > 0\) \(\text{answer} \leftarrow \min\{\text{answer}, x[1\ldots i - 1], y[1\ldots j] + 1\}\).
5. If \(j > 0\) \(\text{answer} \leftarrow \min\{\text{answer}, x[1\ldots i], y[1\ldots j - 1] + 1\}\).
6. Return \(\text{answer}\).

Note that this is exactly how we arrived at the state/transition in the iterative case too. The number of states is \(O(n^2)\) because all states have \(0 \leq i, j \leq n\), and the number of transitions is \(O(1)\) per state.

Finally, we shift gears to the shortest path problem. We start with a distance matrix \(d\), and want to compute lengths of shortest paths between all pairs of vertices. The dense case of Dijkstra’s algorithm runs in \(O(n^2)\) time, which gives a total of \(O(n^3)\).

Here we will show a much simpler algorithm based on the fact that each vertex appears on the shortest \(jk\) path at most once.

1. State: \(d[i][j][k]\): shortest \(j \rightarrow k\) path containing intermediate vertices \(1\ldots i\).
2. Base case: \(d[0][j][k] = \text{dist}[i][j]\).
3. Transition:

\[
d[i][j][k] = \min\{d[i - 1][j][k], d[i - 1][j][i] + d[i - 1][i][k]\}.
\]
4. Ordering: order by \(i\), any order of \(j, k\).

The motivation for the transition is that the path either

- does not use the vertex \(i\), in which case the maximum id on it is \(i - 1\);
- or it uses \(i\), in which case the portions between \(j\) and \(i\), as well as \(i\) and \(k\), don’t use \(i\), and hence use only ids between 1 and \(i - 1\).

This has \(O(n^3)\) states, but only \(O(n^3)\) transitions as well, giving a simple \(O(n^3)\) time algorithm.