CITS2200 Data Structures and Algorithms

Topic 20

Dynamic Programming

- The all pairs shortest path problem
- Dynamic programming method
- Matrix product algorithm
- Floyd-Warshall algorithm
- Longest Common Subsequence

Reading: Weiss, Sections 7.5-7.7, CLRS chapter 15

All-pairs shortest paths

Recall the Shortest Path Problem in Topic ??.

Now we turn our attention to constructing a complete table of shortest distances, which must contain the shortest distance between any pair of vertices.

If the graph has no negative edge weights then we could simply make V runs of Dijkstra's algorithm, at a total cost of $O(VE \lg V)$, whereas if there are negative edge weights then we could make V runs of the Bellman-Ford algorithm at a total cost of $O(V^2E)$.

The two algorithms we shall examine both use the adjacency matrix representation of the graph, hence are most suitable for dense graphs. Recall that for a weighted graph the weighted adjacency matrix A has $weight(i, j)$ as its ij-entry, where $weight(i, j) = \infty$ if i and j are not adjacent.

A dynamic programming method

Dynamic programming is a general algorithmic technique for solving problems that can be characterised by two features:

- The problem is broken down into a collection of smaller subproblems
- The solution is built up from the stored values of the solutions to all of the subproblems

For the all-pairs shortest paths problem we define the simpler problem to be

"What is the length of the shortest path from vertex i to j that uses at most m edges?"

We shall solve this for $m = 1$, then use that solution to solve for $m = 2$, and so on . . .

The initial step

We shall let $d_{ij}^{(m)}$ denote the distance from vertex i to vertex j along a path that uses at most m edges, and define $D^{(m)}$ to be the matrix whose ij-entry is the value $d_{ij}^{(m)}$.

As a shortest path between any two vertices can contain at most $V - 1$ edges, the matrix $D^{(V-1)}$ contains the table of all-pairs shortest paths.

Our overall plan therefore is to use $D^{(1)}$ to compute $D^{(2)}$, then use $D^{(2)}$ to compute $D^{(3)}$ and so on.

The case $m = 1$

Now the matrix $D^{(1)}$ is easy to compute — the length of a shortest path using at most one edge from i to j is simply the weight of the edge from i to j. Therefore $D^{(1)}$ is just the adjacency matrix A.

The inductive step

What is the smallest weight of the path from vertex i to vertex j that uses at most m edges? Now a path using at most m edges can either be

(1) A path using less than m edges
(2) A path using exactly m edges, composed of a path using $m-1$ edges from i to an auxiliary vertex k and the edge (k, j).

We shall take the entry $d_{ij}^{(m)}$ to be the lowest weight path from the above choices.

Therefore we get

$$\begin{aligned} d_{ij}^{(m)} &= \min\left(d_{ij}^{(m-1)}, \min_{1\leq k\leq V}\{d_{ik}^{(m-1)} + w(k,j)\}\right) \\ &= \min_{1\leq k\leq V}\{d_{ik}^{(m-1)} + w(k,j)\} \end{aligned}$$

Example

Consider the weighted graph with the following weighted adjacency matrix:

$$A = D^{(1)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

Let us see how to compute an entry in $D^{(2)}$, suppose we are interested in the $(1, 3)$ entry:

$1 \to 1 \to 3$ has cost $0 + 11 = 11$
$1 \to 2 \to 3$ has cost $\infty + 4 = \infty$
$1 \to 3 \to 3$ has cost $11 + 0 = 11$
$1 \to 4 \to 3$ has cost $2 + 6 = 8$
$1 \to 5 \to 3$ has cost $6 + 6 = 12$

The minimum of all of these is 8, hence the $(1, 3)$ entry of $D^{(2)}$ is set to 8.

Computing $D^{(2)}$

$$\begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}\begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 7 \\ 10 & \infty & 0 & 12 & 16 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \infty & 6 & \infty & 0 \end{pmatrix}$$

If we multiply two matrices $AB = C$, then we compute

$$c_{ij} = \sum_{k=1}^{k=V} a_{ik}b_{kj}$$

If we replace the multiplication $a_{ik}b_{kj}$ by addition $a_{ik} + b_{kj}$ and replace summation Σ by the minimum $\min$ then we get

$$c_{ij} = \min_{k=1}^{k=V} a_{ik} + b_{kj}$$

which is precisely the operation we are performing to calculate our matrices.

The remaining matrices

Proceeding to compute $D^{(3)}$ from $D^{(2)}$ and A, and then $D^{(4)}$ from $D^{(3)}$ and A we get:

$$D^{(3)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & \boxed{6} \\ 10 & \boxed{14} & 0 & 12 & \boxed{15} \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \infty & 6 & \boxed{18} & 0 \end{pmatrix} \qquad D^{(4)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 6 \\ 10 & 14 & 0 & 12 & 15 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \boxed{20} & 6 & 18 & 0 \end{pmatrix}$$

A new matrix "product"

Recall the method for computing $d_{ij}^{(m)}$, the (i, j) entry of the matrix $D^{(m)}$ using the method similar to matrix multiplication.

$d_{ij}^{(m)} \leftarrow \infty$
for $k = 1$ **to** V **do**
    $d_{ij}^{(m)} = \min(d_{ij}^{(m)}, d_{ik}^{(m-1)} + w(k, j))$
end for

Let us use $\star$ to denote this new matrix product.

Then we have

$$D^{(m)} = D^{(m-1)} \star A$$

Hence it is an easy matter to see that we can compute as follows:

$$D^{(2)} = A \star A \qquad D^{(3)} = D^{(2)} \star A \ldots$$

Complexity of this method

The time taken for this method is easily seen to be $O(V^4)$ as it performs V matrix "multiplications" each of which involves a triply nested **for** loop with each variable running from 1 to V.

However we can reduce the complexity of the algorithm by remembering that we do not need to compute *all* the intermediate products $D^{(1)}$, $D^{(2)}$ and so on, but we are only interested in $D^{(V-1)}$. Therefore we can simply compute:

$$D^{(2)} = A \star A$$
$$D^{(4)} = D^{(2)} \star D^{(2)}$$
$$D^{(8)} = D^{(4)} \star D^{(4)}$$

Therefore we only need to do this operation at most $\lg V$ times before we reach the matrix we want. The time required is therefore actually $O(V^3 \lceil \lg V \rceil)$.

Floyd-Warshall

The Floyd-Warshall algorithm uses a different dynamic programming formalism.

For this algorithm we shall define $d_{ij}^{(k)}$ to be the length of the shortest path from i to j whose intermediate vertices all lie in the set $\{1, \ldots, k\}$.

As before, we shall define $D^{(k)}$ to be the matrix whose (i, j) entry is $d_{ij}^{(k)}$.

The initial case

What is the matrix $D^{(0)}$ — the entry $d_{ij}^{(0)}$ is the length of the shortest path from i to j with *no* intermediate vertices. Therefore $D^{(0)}$ is simply the adjacency matrix A.

The inductive step

For the inductive step we assume that we have constructed already the matrix $D^{(k-1)}$ and wish to use it to construct the matrix $D^{(k)}$.

Let us consider all the paths from i to j whose intermediate vertices lie in $\{1, 2, \ldots, k\}$. There are two possibilities for such paths

(1) The path does not use vertex k
(2) The path does use vertex k

The shortest possible length of all the paths in category (1) is given by $d_{ij}^{(k-1)}$ which we already know.

If the path does use vertex k then it must go from vertex i to k and then proceed on to j, and the length of the shortest path in this category is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

The overall algorithm

The overall algorithm is then simply a matter of running V times through a loop, with each entry being assigned as the minimum of two possibilities. Therefore the overall complexity of the algorithm is just $O(V^3)$.

$D^{(0)} \leftarrow A$
for $k = 1$ **to** V **do**
  for $i = 1$ **to** V **do**
    for $j = 1$ **to** V **do**
      $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
    end for j
  end for i
end for k

At the end of the procedure we have the matrix $D^{(V)}$ whose (i, j) entry contains the length of the shortest path from i to j, all of whose vertices lie in $\{1, 2, \ldots, V\}$ — in other words, the shortest path in total.

Example

Consider the weighted directed graph with the following adjacency matrix:

$$D^{(0)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} \qquad D^{(1)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & & \\ 10 & \infty & 0 & & \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

To find the $(2, 4)$ entry of this matrix we have to consider the paths through the vertex 1 — is there a path from 2 – 1 – 4 that has a better value than the current path? If so, then that entry is updated.

The entire sequence of matrices

$$D^{(2)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \boxed{3} & \boxed{7} \\ 10 & \infty & 0 & \boxed{12} & \boxed{16} \\ \boxed{3} & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} \qquad D^{(3)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & 3 & 7 \\ 10 & \infty & 0 & 12 & 16 \\ 3 & 2 & 6 & 0 & 3 \\ \boxed{16} & \infty & 6 & \boxed{18} & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & \boxed{4} & \boxed{8} & 2 & \boxed{5} \\ 1 & 0 & 4 & 3 & \boxed{6} \\ 10 & \boxed{14} & 0 & 12 & \boxed{15} \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \boxed{20} & 6 & 18 & 0 \end{pmatrix} \qquad D^{(5)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 6 \\ 10 & 14 & 0 & 12 & 15 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & 20 & 6 & 18 & 0 \end{pmatrix}$$

Finding the actual shortest paths

In both of these algorithms we have not addressed the question of actually finding the paths themselves.

For the Floyd-Warshall algorithm this is achieved by constructing a further sequence of arrays $P^{(k)}$ whose (i, j) entry contains a predecessor of j on the path from i to j. As the entries are updated the predecessors will change — if the matrix entry is not changed then the predecessor does not change, but if the entry does change, because the path originally from i to j becomes re-routed through the vertex k, then the predecessor of j becomes the predecessor of j on the path from k to j.

Longest Common Subsequence

Consider the following problem

LONGEST COMMON SUBSEQUENCE
INSTANCE: Two sequences X and Y
QUESTION: What is a longest common subsequence of X and Y

Example

If

$$X = \langle A, B, C, B, D, A, B \rangle$$

and

$$Y = \langle B, D, C, A, B, A \rangle$$

then a longest common subsequence is either

$$\langle B, C, B, A \rangle$$

or

$$\langle B, D, A, B \rangle$$

A recursive relationship

As is usual for dynamic programming problems we start by finding an appropriate recursion, whereby the problem can be solved by solving smaller subproblems.

Suppose that

$$X = \langle x_1, x_2, \ldots, x_m \rangle$$
$$Y = \langle y_1, y_2, \ldots, y_n \rangle$$

and that they have a longest common subsequence

$$Z = \langle z_1, z_2, \ldots, z_k \rangle$$

If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1}.

Otherwise Z is either a LCS of X_{m-1} and Y or a LCS of X and Y_{n-1}.

(This depends on whether $z_k \neq x_m$ or $z_k \neq y_n$ respectively — at least one of these two possibilities must arise.)

A recursive solution

This can easily be turned into a recursive algorithm as follows.

Given the two sequences X and Y we find the LCS Z as follows:

If $x_m = y_n$ then find the LCS Z' of X_{m-1} and Y_{n-1} and set $Z = Z'x_m$.

If $x_m \neq y_n$ then find the LCS Z_1 of X_{m-1} and Y, and the LCS Z_2 of X and Y_{n-1}, and set Z to be the longer of these two.

It is easy to see that this algorithm requires the computation of the LCS of X_i and Y_j for all values of i and j. We will let $l(i, j)$ denote the length of the longest common subsequence of X_i and Y_j.

Then we have the following relationship on the lengths

$$l(i,j) = \begin{cases} 0 & \text{if } ij = 0 \\ l(i-1, j-1) + 1 & \text{if } x_i = y_j \\ \max(l(i-1, j), l(i, j-1)) & \text{if } x_i \neq y_j \end{cases}$$

Memoization

The simplest way to turn a top-down recursive algorithm into a sort of dynamic programming routine is *memoization*. The idea behind this is that the return values of the function calls are simply stored in an array as they are computed.

The function is changed so that its first step is to look up the table and see whether $l(i, j)$ is already known. If so, then it just returns the value immediately, otherwise it computes the value in the normal way.

Alternatively, we can simply accept that we must at some stage compute all the $O(n^2)$ values $l(i, j)$ and try to schedule these computations as efficiently as possible, using a *dynamic programming table*.

The dynamic programming table

We have the choice of memoizing the above algorithm or constructing a bottom-up dynamic programming table.

In this case our table will be an $(m+1) \times (n+1)$ table where the (i, j) entry is the length of the LCS of X_i and Y_j.

Therefore we already know the border entries of this table, and we want to know the value of $l(m, n)$ being the length of the LCS of the original two sequences.

In addition to this however we will retain some additional information in the table - namely each entry will contain either a left-pointing arrow $\leftarrow$, a upward-pointing arrow $\uparrow$ or a diagonal arrow $\nwarrow$.

These arrows will tell us which of the subcases was responsible for the entry getting that value.

Our example

For our worked example we will use the sequences

$$X = \langle 0, 1, 1, 0, 1, 0, 0, 1 \rangle \qquad \text{and} \qquad Y = \langle 1, 1, 0, 1, 1, 0 \rangle$$

Then our initial empty table is:

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i							
1	0							
2	1							
3	1							
4	0							
5	1							
5	0							
7	0							
8	1							

The first table

First we fill in the border of the table with the zeros.

Now each entry (i, j) depends on x_i, y_j and the values to the left $(i, j-1)$, above $(i-1, j)$, and above-left $(i-1, j-1)$.

In particular, we proceed as follows:

If $x_i = y_j$ then put the symbol $\nwarrow$ in the square, together with the value $l(i-1, j-1)+1$.

Otherwise put the greater of the values $l(i-1, j)$ and $l(i, j-1)$ into the square with the appropriate arrow.

The first row

It is easy to compute the first row, starting in the $(1, 1)$ position:

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0
1	0	0	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2	1	0						
3	1	0						
4	0	0						
5	1	0						
6	0	0						
7	0	0						
8	1	0						

Computation proceeds as described above.

The final array

After filling it in row by row we eventually reach the final array:

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0
1	0	0	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2	1	0	$\nwarrow 1$	$\nwarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\nwarrow 2$	$\leftarrow 2$
3	1	0	$\nwarrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\nwarrow 2$	$\nwarrow 3$	$\leftarrow 3$
4	0	0	$\uparrow 1$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$	$\uparrow 3$	$\nwarrow 4$
5	1	0	$\nwarrow 1$	$\nwarrow 2$	$\uparrow 3$	$\nwarrow 4$	$\nwarrow 4$	$\uparrow 4$
6	0	0	$\uparrow 1$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 4$	$\uparrow 4$	$\nwarrow 5$
7	0	0	$\uparrow 1$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 4$	$\uparrow 4$	$\nwarrow 5$
8	1	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 3$	$\nwarrow 4$	$\nwarrow 5$	$\uparrow 5$

This time we have kept enough information, via the arrows, for us to compute what the LCS of X and Y is.

Finding the LCS

The LCS can be found (in reverse) by tracing the path of the arrows from $l(m, n)$. Each *diagonal* arrow encountered gives us another element of the LCS.

As $l(8, 6)$ points to $l(7, 6)$ so we know that the LCS is the LCS of X_7 and Y_6.

Now $l(7, 6)$ has a diagonal arrow, pointing to $l(6, 5)$ so in this case we have found the last entry of the LCS — namely it is $x_7 = y_6 = 0$.

Then $l(6, 5)$ points (upwards) to $l(5, 5)$, which points diagonally to $l(4, 4)$ and hence 1 is the second-last entry of the LCS.

Proceeding in this way, we find that the LCS is

$$11010$$

Notice that if at the very final stage of the algorithm (where we had a free choice) we had chosen to make $l(8, 6)$ point to $l(8, 5)$ we would have found a different LCS

$$11011$$

Finding the LCS

We can trace back the arrows in our final array, in the manner just described, to determine that the LCS is 11010 and see which elements within the two sequences match.

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0
1	0	0	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2	1	0	$\boxed{\nwarrow 1}$	$\nwarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\nwarrow 2$	$\leftarrow 2$
3	1	0	$\nwarrow 1$	$\boxed{\nwarrow 2}$	$\leftarrow 2$	$\nwarrow 2$	$\nwarrow 3$	$\leftarrow 3$
4	0	0	$\uparrow 1$	$\uparrow 2$	$\boxed{\nwarrow 3}$	$\leftarrow 3$	$\uparrow 3$	$\nwarrow 4$
5	1	0	$\nwarrow 1$	$\nwarrow 2$	$\uparrow 3$	$\nwarrow 4$	$\boxed{\nwarrow 4}$	$\uparrow 4$
6	0	0	$\uparrow 1$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 4$	$\uparrow 4$	$\nwarrow 5$
7	0	0	$\uparrow 1$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 4$	$\uparrow 4$	$\boxed{\nwarrow 5}$
8	1	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 3$	$\nwarrow 4$	$\nwarrow 5$	$\uparrow 5$

A match occurs whenever we encounter a diagonal arrow along the reverse path.

Analysis

The analysis for longest common subsequence is particularly easy.

After initialization we simply fill in mn entries in the table — with each entry costing only a constant number of comparisons. Therefore the cost to produce the table is $O(mn)$

Following the trail back to actually find the LCS takes time at most $O(m+n)$ and therefore the total time taken is $O(mn)$.

Summary

1. Dynamic Programming is a general approach for solving problems which can be decomposed into sub-problems and where solutions to sub-problems can be combined to solve the main problem.
2. Dynamic Programming can be used to solve the shortest path problem directly or via the Floyd-Warshall formulation.
3. The Longest Common Subsequence problem is a typical dynamic programming problem.