12.1 All pairs shortest paths

Let G be a graph with positive edge weights. We want to calculate the shortest paths between *every* pair of nodes. One way to do this is to run Dijkstra's algorithm several times, once for each node. Here we develop a different dynamic programming solution.

Our subproblems will be shortest paths using *only* nodes $1 \ldots k$ as intermediate nodes. Of course when k equals the number of nodes in the graph, n, we will have solved the original problem.

We let the matrix $D_k[i.j]$ represent the length of the shortest path between i and j using intermediate nodes $1 \ldots k$. Initially, we set a matrix D_0 with the direct distances between nodes, given by d_{ij}. Then D_k is easily computed from the subproblems D_{k-1} as follows:

$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]).$$

The idea is the shortest path using intermediate nodes $1 \ldots k$ either completely avoids node k, in which case it has the same length as $D_{k-1}[i,j]$; or it goes through k, in which case we can glue together the shortest paths found from i to k and k to j using only intermediate nodes $1 \ldots k-1$ to find it.

It might seem that we need at least two matrices to code this, but in fact it can all be done in one loop. (**Exercise:** think about it!)

$D = (d_{ij})$, distance array, with weights from all i to all j
for $k = 1$ to n do
  for $i = 1$ to n do
    for $j = 1$ to n do
      $D[i,j] = \min(D[i,j], D[i.k] + D[k,j])$

Note that again we can keep an auxiliary array to recall the actual paths. We simply keep track of the last intermediate node found on the path from i to j. We reconstruct the path by successively reconstructing intermediate

nodes, until we reach the ends.

12.2 Longest monotone subsequence

Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence of distinct integers. We want to find the longest *subsequence* of A that's *monotone*. A subsequence of A is a sequence $S = (a_{i_1}, a_{i_2}, \ldots, a_{i_m})$ such that $i_1 < i_2 < \cdots < i_m$. (Note that terms of the subsequence don't need to be consecutive within A; for instance, $(3,4,5,9)$ is a subsequence of $(3,1,4,1,5,9)$. A subsequence is monotone iff either

1. $a_{i_1} < a_{i_2} < \ldots < a_{i_m}$) ("S is monotone increasing") or
2. $a_{i_1} > a_{i_2} > \ldots > a_{i_m}$) ("S is monotone decreasing")

Our subproblems will be the length $I(k)$ of the longest increasing subsequence ending at a_k and the length $D(k)$ of the longest decreasing subsequence ending at a_k.

A monotone-increasing subsequence S ending at a_k is of the form $(a_{i_1}, a_{i_2}, \ldots, a_{i_{m-1}}, a_{i_m})$, where $i_m = k$, $a_{i_{m-1}} < a_{i_m}$, and $(a_{i_1}, a_{i_2}, \ldots, a_{i_{m-1}})$ is a monotone-increasing subsequence ending at $a_{i_{m-1}}$. Also, if S is the *longest* monotone-increasing subsequence ending at a_k, then $(a_{i_1}, a_{i_2}, \ldots, a_{i_{m-1}})$ is the longest monotone-increasing subsequence ending at $a_{i_{m-1}}$: if there were a longer monotone-increasing subsequence ending at $a_{i_{m-1}}$, we could append a_{i_m} to get a longer monotone-increasing subsequence ending at $a_{i_m} = a_k$.

Therefore, the length of the longest monotone-increasing subsequence ending at a_k is the maximum of the above over all possible previous elements $a_{i_{m-1}}$ that are less than a_k, so our recurrence is

$$I(k) = \max_{j:a_j<a_k} I(k) + 1.$$

Therefore, the length of the longest monotone-decreasing subsequence ending at a_k is

$$D(k) = \max_{j:a_j>a_k} D(k) + 1.$$

We can implement those recurrences in the same loops:

$A = (A_i)$, integer array
for $k = 1$ to n do
    D[k] = 1

I[k] = 1

for $j = 1$ to n do

  if A[j] ¿ A[k] do

    $D[k] = \max(D[k], D[j] + 1)$

  else

    $I[k] = \max(I[k], I[j] + 1)$

As a side note (and as another reason to consider the problems of longest increasing and longest decreasing subsequences together), the longest monotone subsequence of a sequence of length n has length at least $\sqrt{n}$:

1. The pairs $(I[k], D[k])$ are distinct for all k, because if $(I[k], D[k]) = (I[j], D[j])$, then neither the if nor the else clause in the inner loop could have triggered.

2. If the longest monotone subsequence of some sequence of length n had length less than $\sqrt{n}$, then there would be fewer than $(\sqrt{n})^2 = n$ distinct pairs $(I[k], D[k])$.

12.3 Context-free grammar parsing

Given a sequence of English words like...

1. "The old man the boats"

2. "No rest for the wicked"

3. "The sailor mans the battleship guns"

4. "The the the the the"

...is it complete and grammatical English sentence?

(Note that the first of those examples is complete and grammatical: "the old" refers to "old people" like "the wicked" does in the second example, and those old people are manning boats.)

We'll take the following as our definition of "grammatical" (with apologies to linguists for oversimplification):

1. Each word has some set of *symbols* (e.g. parts of speech) that it can be: e.g. "man" is N or V; "the" is D.

2. Some *production rules* can make two symbols out of one:

   (a) S $\rightarrow$ NP VP

   (b) N $\rightarrow$ N N

   (c) NP $\rightarrow$ D N

   (d) VP $\rightarrow$ V NP

3. A sentence is a complete and grammatical English sentence iff some sequence of production rules can turn the symbol S (for "start" or "sentence") into a sequence of symbols matching that sentence.

For instance, "the old man the boats" is grammatical because we can apply the above production rules to turn S $\rightarrow$ NP VP $\rightarrow$ D N VP $\rightarrow$ D N V NP $\rightarrow$ D N V D N, and those are possible types of "the", "old", "man", "the", and "boats", respectively.

To solve the problem above (that is, given a set of production rules and a target string x, can the production rules generate x), we'll do dynamic programming with the following subproblems:

$$D[i, j, U] = \text{True iff symbol U can generate } x[i : j]$$

(and false if starting from U and applying production rules can't generate the ith through jth symbols of x).

If U can generate $x[i : j]$ and $j - i > 1$, then there must be some production rule applied to U, say $U \rightarrow VW$, such that V can generate part of x and W can generate the rest; that is, there must exist $k \in [i, j)$ such that $D[i,k,V] = \text{True}$ and $D[k, j, W] = \text{True}$. Conversely, if there exists a production rule $U \rightarrow VW$ and there exists $k \in [i, j)$ such that $D[i,k,V] = \text{True}$ and $D[k, j, W] = \text{True}$, then $D[i, j, U] = \text{True}$.

As base cases, if $j - i = 1$, then $D[i, j, U] = 1$ if and only if word i can have symbol U.

As a final answer, a sentence is complete and grammatical if and only if $D[0, n, S] = \text{True}$.

If there are n words in the input and m production rules and symbols, then the runtime for this algorithm is $O(n^3 m^2)$: there are $\Theta(n^2 m)$ subproblems, and for each of them we need to check $O(n)$ possible division points k and $O(m)$ possible production rules.