Lecture 2: 1/29/2013

Spring 2013

Fibonacci numbers:

$$F_n = F_{n-1} + F_{n-2}$$

Recurrence relation: without memoization, exponential time, with memoization, $O(n)$ time and space.

To save on memory we can forget previous values. We can do this cleanly with linear algebra:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}.$$

Thus we can advance many iterations at a time by raising the matrix $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ to powers:

$$\begin{bmatrix} F_{n+k} \\ F_{n+k-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^k \cdot \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}.$$

One can compute matrix powers with only $O(\log k)$ matrix multiplications and additions via repeated squaring. One can sometimes compute matrix powers even faster if one knows how to *diagonalize* a matrix, which involves *eigenvalues and eigenvectors*, which are a powerful technique that we will see towards the end of the course in the section on graphs.

The eigenvalues of the matrix $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ are ϕ and $-\frac{1}{\phi}$ where ϕ is the golden ratio. This leads to the general expression for Fibonacci numbers: $F_n = a \cdot \phi^n + b \cdot (-\frac{1}{\phi})^n$, where the constants a, b depend on the base case of the recursion for the Fibonacci numbers.

Number of paths in a graph:

(Note: notation has changed slightly from lecture to be more consistent.)

Given a graph G, we can represent it via its *adjacency matrix*, which is a matrix M such that the (i, j) entry in M is 1 if and only if there is a directed edge in the graph from i to j, and 0 otherwise. If V is the set of vertices of G, then M is a $|V| \times |V|$ matrix. Let e_i be the row vector that has a 1 in the ith location and 0's elsewhere. Multiplying $e_i \cdot M$ returns the ith row of M, namely, for each vertex ℓ in G, the ℓth value represents whether there is an edge from i to ℓ or not; viewed differently, this is the number of paths in G from i to ℓ of length 1. Correspondingly, if we multiply by M repeatedly, this corresponds to longer paths: the ℓth entry of the vector $e_i \cdot M^k$ records the number of paths of length k in G going from i to ℓ.

Multiplying a matrix by a vector involves doing something for each nonzero entry of the matrix. If E represents the edges of the graph G, then there are $|E|$ nonzero entries in M, and counting the number of paths of length k thus takes $O(k \cdot |E|)$ time. (Alternatively, we could raise the matrix M to the kth power first, and then multiply it by e_i; matrix multiplication in practice takes $|V|^3$ time, so this approach is typically slower.)

Recursive representation of paths in a graph:

We will now look at the same problem from a rather different perspective: how can we write a recursive procedure (which will be *memoized*) to compute the same thing, namely the number of paths in a graph G from i to ℓ of length k?

PATHS(i, k, ℓ)
  return: $\sum_j$ PATHS$(i, k-1, j) \cdot M(j, \ell)$

Think through the above code slowly: to find the number of paths from i to ℓ of length k, consider all points j which could be the *second-to-last* point on the path, and for each such j, count the number of ways of getting to j from i in $k-1$ steps, and then add it to the sum if $M(j, \ell)$ is 1, namely if there is an edge from j to ℓ in the graph.

How long does this code take to run, provided the recursion is memoized? The time to run the body of the function is $|V|$, the number of values for the vertex j that are summed over. This function needs to be evaluated once for each combination of input parameters that will be encountered: i does not change; calling PATHS(i, n, ℓ) will involve k decrementing from n to 1 for n total possibilities; ℓ will vary among the $|V|$ vertices in the recursive calls leading to $|V|$ possibilities. Thus running PATHS(i, n, ℓ) will take $O(n \cdot |V|^2)$ time. A slightly tighter analysis might note that we only need to do work for nonzero entries of M, so thus the time is $O(n \cdot |E|)$.

Crucially, how does this code relate to the previous approach of repeatedly multiplying a vector by M? It is identical: $e_i \cdot M^k$ is a vector whose ℓth entry is exactly PATHS(i, k, ℓ), and these values for a given k are computed from the values for $k-1$ via exactly the vector-matrix multiplication formula—to find the ℓth component of the product of a vector v_{k-1} by the matrix M, we compute $\sum_j v_{k-1}(j) \cdot M(j, \ell)$, which can be seen to correspond precisely to the expression in our pseudocode.

Note that reversing the direction of paths is equivalent to taking the transpose of the matrix M; having numbers other than 0 or 1 in the matrix M is quite natural, and, for example, a "3" in the (i, j)th location of M might be interpreted as "there are 3 ways to get from i to j in one step."

Minimum cost paths in a graph:

Instead of computing the *number* of paths in a graph, we often want to compute the *shortest* path between two nodes in a graph. Here the meaning of M is slightly different: the (i, j)th location now represents the cost/weight/distance of going from i to j in one step; in the simplest case, M is either 1 or ∞, denoting respectively that yes you can get from i to j in one step, or no it is impossible to get from i to j in one step. (Note that, since $\log 1 = 0$ and $\log 0 = -\infty$, the new matrix M is essentially negative the log of the old matrix M.) In general, beyond just 1 or ∞ values, the matrix M may be filled with arbitrary numbers, positive or negative, expressing general cost/weight/distance values.

Consider the following code to recursively compute the minimum cost path between nodes i and ℓ in a graph:

PATHS(i, k, ℓ)
  return: $\min_j$ PATHS$(i, k-1, j) + M(j, \ell)$

Think through the above code slowly: to find the minimum cost path from i to ℓ involving k nodes, consider all points j which could be the *second-to-last* point on the path, and take the minimum over such j, of the minimum cost way of getting from i to j involving $k-1$ nodes plus the cost of going from j to ℓ in one hop. Compare this code to the previous code, and see that the structure is identical, and the mathematical operations are related (sum and product are replaced by min and sum respectively).

One issue remains: what is the meaning of k? It is straightforward to prove (check the Wikipedia article for "Bellman-Ford") that for any nodes i, j that have a minimum cost path between them, that path is of length at most $|V|$ (the alternative is that there is a cycle with negative total cost, and going around the cycle repeatedly can drive the cost to $-\infty$). Thus we should run the algorithm with $k = |V|$ and record the lowest-cost path of any intermediate length up to $|V|$ that we encounter.

This is the Bellman-Ford algorithm, which is presented here as an unexpected adaptation of matrix multiplication. (The standard presentation of the Bellman-Ford algorithm is a bit different, and makes it less clear what the meaning of "k" is. See Wikipedia for details.) The minimum cost path from i to j in a graph will be computed in time $O(|V| \cdot |E|)$, even if there are negative weights (costs) in the graph.

Edit distance: We turn to a completely different application of dynamic programming: edit distance. Given two words like "cat" and "crab", we want to efficiently compute the "cost" of the best way to edit the first word so that it becomes the second word. Applications include all forms of spell check/correction, and many bioinformatics techniques for analyzing DNA to hypothesize the evolutionary "edit" path between two species. Exactly how to define "cost" appropriately in each case is a delicate matter, but we can get the flavor of the problem—and its dynamic programming solution—by taking a very simple model: inserting a letter costs 1; deleting a letter costs 1; modifying a letter costs 1.

Designing the right recursion for this is slightly tricky: we cannot store edits to the strings in the recursive calls because this might lead to an exponential space of possible recursive calls. Instead, consider comparing a word w_1 of length i to a word w_2 of length j. There are 3 possible things which can happen: the ith letter of w_1 can be *edited* into the jth letter of w_2, at a cost of 1, or 0 if the letters were already equal, leaving the first $i-1$ letters of word w_1 and the first $j-1$ letters of word w_2 that have yet to be matched up and edited; the ith letter of w_1 can be deleted, leaving the first $i-1$ letters of w_1 and the first j letters of w_2; or the jth letter of w_2 can be inserted, leaving the first i letters of w_1 and the first $j-1$ letters of w_2. We have thus reduced the problem into three recursive subcases, and the crucial thing is that we do not need to know what edits are done in these recursive subcases, only the total cost. This leads to the following pseudocode:

editDistance(w_1, w_2, i, j)
  return: min{
    editDistance$(w_1, w_2, i-1, j-1) + [w_1(i) = w_2(j)]$,
    editDistance$(w_1, w_2, i-1, j) + 1$,
    editDistance$(w_1, w_2, i, j-1) + 1$
    },

where the notation $[w_1(i) = w_2(j)]$ denotes 1 if the equality is true and 0 if it is false.