Advanced Algorithms

Class Notes for Monday, November 19, 2012

Bernard Moret

1 Divide-and-Conquer: Numerical Computing

Divide-and-conquer is especially useful in computational geometry, but also in numerical algebra. One of the most celebrated and influential algorithms in algebra is the Fast Fourier Transform (FFT) algorithm: it is a typical use of divide-and-conquer. Here we look at another famous algorithm: Strassen's algorithm for fast matrix multiplication. The origin of this algorithm goes back to the days when multiplying two values (not matrices, just scalars) was much more expensive than adding them—so that programmers and algorithm designers strove to avoid multiplications. Strassen's algorithm started with an attempt at reducing the number of scalar multiplications needed to compute the product of two 2×2 matrices:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Computed in the standard manner, with each entry of the product computed separately, this product requires 2 scalar multiplications for each entry, or 8 in all. Strassen found a way to compute 7 subexpressions, each using only one multiplication, such that all 8 entries can then be computed from these subexpressions using only additions and multiplications. The subexpressions are not obvious: Strassen used the following:

$$\begin{array}{ll} a_{11}b_{11} & (a_{12}-a_{21}+a_{11}-a_{22})b_{22} \\ a_{12}b_{21} & a_{22}(b_{11}+b_{22}-b_{12}-b_{21}) \\ (a_{11}-a_{21})(b_{22}-b_{12}) & (a_{21}+a_{22}-a_{11})(b_{22}-b_{12}+b_{11}) \\ (a_{21}+a_{22})(b_{12}-b_{11}) & \end{array}$$

Using these, one can compute the product of two 2×2 matrices with 7 multiplications and 15 additions and subtractions, instead of 8 multiplications and 4 additions and subtractions.

Today, of course, multiplication is almost as fast as addition for standard number representation (single or double precision numbers)—although it remains much slower than addition for unbounded-precision arithmetic. Thus Strassen's find would be little more than a footnote in the history of computing if he had not quickly realized that his algorithm could be used recursively in a divide-and-conquer scheme. As stated, his algorithm computes the product of 2×2 matrices by reducing the number of products of 1×1 matrices (scalars); by using this approach recursively, we can take 4×4 matrices, divide each into four 2×2 matrices, and compute the product of 4×4 matrices using 7 products (and 15 additions/subtractions) of 2×2 matrices. When reduced to multiplications and additions of scalars, this yields a total of $7 \cdot 7 = 49$ scalar multiplications and $(7 \cdot 15) + (15 \cdot 4) = 165$ additions and subtractions, whereas the standard method would need $4 \cdot 4 \cdot 4 = 64$ scalar

multiplications and $3 \cdot 4 \cdot 4 = 48$ additions. If we move to 8×8 matrices, we now get $7 \cdot 7 \cdot 7 = 343$ scalar multiplications instead of 512, and $(7 \cdot 165) + (15 \cdot 16) = 1395$ additions and subtractions instead of $7 \cdot 8 \cdot 8 = 448$. Notice that the ratio of excess additions and subtractions decreases steadily: the standard method requires $(n-1) \cdot n^2$, or $\Theta(n^3)$ additions, whereas Strassen's algorithm uses

$$f(n) = 7f(n/2) + 15n^2/4$$

additions and subtractions, which yields $f(n) = \Theta(n^{\log_2 7})$. In other words, the penalty sustained on 2×2 matrices in terms of additions and subtractions is quickly overshadowed by the fact that we have fewer *matrix* multiplications to perform, thereby reducing the number of both multiplicative and additive *scalar* operations. In terms of scalar multiplications, the standard method uses exactly n^3 of them, where Strassen's uses

$$f(n) = 7f(n/2)$$

in all, which is again $\Theta(n^{\log_2 7})$. Hence the standard method computes the product of two $n \times n$ matrices in $\Theta(n^3)$ time, whereas, using divide-and-conquer and a clever way to handle matrices divided into four equal submatrices, Strassen's algorithm takes $O(n^{\log_2 7})$ time, a substantial improvement.

Strassen's algorithm is by no means the fastest matrix multiplication algorithm in asymptotic terms, but it is the fastest practical algorithm. The Copper-Winograd algorithm, inspired from Strassen's algorithm, but far more complex, runs (in some of its variations) in $O(n^{2.38})$, but its coefficients are so large that the algorithm remains impractical today.

We just add a couple of other examples of divide-and-conquer for numerical programming. We can compute Fibonacci numbers exactly in logarithmic time by considering products of Fibonacci numbers for arguments divided by 2 (obviously, using floating-point numbers, we can compute in constant time by using the irrational function that solves the recurrence, but we get numerical errors for large values). We can multiply two polynomials of degree n in $\Theta(n^{\log_2 3})$ time (instead of $\Theta(n^2)$ by the obvious method) by devising a way to compute $(a_1 \cdot n + a_0) \cdot (b_1 \cdot n + b_0)$ with only three multiplications instead of four—this is a direct analog of Strassen's algorithm. (In this case, we can compute the product even faster using FFTs, since it takes $\Theta(n \log n)$ to compute the forward FFT, linear time to add the two transformed functions in Fourier space, then again $\Theta(n \log n)$ to reverse the FFT. However, FFTs require non-integral operations, whereas the divide-and-conquer uses only integral operations.)

2 Dynamic Programming

Dynamic programming is the last main non-randomized algorithmic design method. It can be viewed simply as a lazy evaluation approach, or as a generalization of divide-and-conquer. The first applies to simple computations, for instance, computing Fibonacci numbers by storing the value returned by a recursive call and making such calls only when the desired value is not already stored. The second is more to the point: divide-and-conquer,

as we have seen, can always divide in any manner without affecting the correctness or optimality of the answer returned—the choice of how to divide affects only the running time, not the answer. Whenever different divisions could return different answers, we must turn to another algorithmic paradigm and in such cases dynamic programming usually works.

2.1 Trivial DPs: lazy evaluation

Consider our old friends the Fibonacci numbers. If we run the code

```
int function Fib(int n)
  int answer
  if (n==0) or (n==1)
      answer = 1
   else
      answer = Fib(n-1) + Fib(n-2)
  return answer
```

then the running time of our procedure is exponential! That is because it will compute the function over and over again for the same argument, within different branches of the recursion tree. (In fact, the running time is $\Theta(Fib(n))$.) If, however, we use a global array initialized to a recognizable marker value, then we can avoid recomputing values as follows:

```
int array storeFib[0..n]
storeFib[0] = 1
storeFib[1] = 1
for i=2 to n storeFib[i] = -1

int function Fib(int n)
  if (storeFib[n] == -1)
storeFib[n] = Fib(n-1) + Fib(n-2)
  return storeFib[n]
```

This new version runs in linear time. Obviously, it is much simpler and more efficient to use a direct loop to fill in the array, and even better to rotate values among three variables and so reduce the storage requirements, but these are minor issues compared to the gulf between exponential and linear time that was bridged by the simple expedient of storing the solutions to subproblems and looking them up whenever possible.

Note, however, that such an approach relies on the fundamental premise that the number of subinstances is a polynomial function of the size of the instance. (For Fibonacci, this is a linear function: n subinstances when solving the instance $F(n)$.) This is an absolute requirement for using dynamic programming, as otherwise both storage and running time will explode. We shall now discover a second requirement: when we must compute an optimal solution according to some given objective function (as opposed to a simple value, such as $F(n)$), the objective function for the full instance must be computable from the values of the objective function for the subinstances.

2.2 Optimal binary search trees

Consider the following optimization problem. We want to build a dictionary from fixed list of words and collect frequencies for all possible queries against the dictionary, where a query is simply "is word XYZ in the dictionary"? The words are ordered lexicographically and so, with n words, say a_1, a_2, ..., a_n, we have $2n+1$ possible outcomes for a query: it can be successful, matching one of the n words in the dictionary, or it can fail, in which case it falls in one of the $n+1$ intervals between and around the n words. Thus assume we have collected frequencies (which we can view as sample probabilities) for each of these $2n+1$ possible types of queries, say p_1, p_2, ..., p_n for the words a_1, a_2, ..., a_n, and q_0, q_1, ..., q_n for the failed queries, where q_0 is the probability of a query for something that is smaller in lexicographic order than a_1, q_n the probability of a query for something that is larger than a_n, and, in general, q_i the probability for a query that is larger than a_i, but smaller than a_{i+1}. We now want to build a binary search tree on the words where the expected number of comparisons (with words of the dictionary) needed to resolve a query is minimized. (The number of required comparisons is the length of the path from the root to the word if the query matches a word; otherwise it is the length of the path from the root to a nil pointer—or a leaf, if we prefer to think of it in this manner.)

The problem seems simple enough and, indeed, it would be very simple if we could decide which a_i word to place at the root of the tree and then, recursively, which to place at the root of each subtree as we continue building the tree downward. However, the choice of root has a huge effect on the expected cost of searches in the tree, so much so that we seem to have little choice other than trying each possible a_i in turn. The choice of how to cut the dictionary into a left subtree and a right subtree is not a question of reducing the time needed to build the tree: it is a question of building the optimal tree or one that is not optimal. So divide-and-conquer cannot be used. Instead we will use dynamic programming.

Observe that the number of subproblems is quadratic: a subproblem is simply a consecutive range of dictionary words. So, if we have n dictionary words for our problem, a subproblem is any range of words $[a_k, \ldots, a_l]$, with $1 \leq k < l \leq n$, and thus we have $n(n+1)/2$ subproblems, of sizes varying from 1 to n. Subproblems of size 1 are trivial, of course: as they have just one dictionary word, that word, say a_i, is at the root, and the two pointers correspond to failed queries (of probabilities q_{i-1} and q_i). Thus we can initially fill in the solution to every subproblem of size 1. After that, we proceed to compute the solution to every subproblem of size 2, then size 3, and continue until we reach and solve size n, which gives us the desired solution. We can set up an array of subproblems, indexed by k and l (in which case we will move diagonal by diagonal) or by $l-k$ and k (in which case we will move row by row, which is perhaps a bit more intuitive). Suppose we do the second. Our array, call it SOLUTION, has n rows, corresponding to sizes of subproblems, and n columns, corresponding to the index in the dictionary of the first word of the subproblem. Entry (i, j) in that matrix stores the total frequency of the words and failed queries in the subproblems (the sum of the p_is and q_js in that subproblem), call it $F(i, j)$, the index of the word chosen as the root of the subtree for that subproblem, call it $r(i, j)$,

and the expected query cost for that subtree, call it $c(i,j)$. We initialize the bottom row, $i = 1$, for all j, $1 \leq j \leq n$, by setting $F(i,1) = p_i + q_{i-1} + q_i$, $r(i,1) = i$, and $c(i,1) = 1$. (This last setting assumes that the cost of testing a word against another is one unit, but the cost of testing a pointer against `nil` is zero.)

Now our algorithm proceeds to fill in each row (not to the end: row i has entries $(i,1)$ through $(i,n-i+1)$ only). To fill in row i, where each subproblem has size i, we try each possible choice of root: that is, for entry (i,j), we try $r(i.j) = i, i+1, \ldots, n-i+1$; for each choice, we compute the expected cost by looking up the total frequency and expected cost of the two induced subtrees (which have already been computed) and combining them with the query frequency $p_{r(i,j)}$. The indices get a bit ugly, but the computation takes constant time for each choice of subtree root. We then retain the choice of lowest cost and store that choice, its cost, and the total frequency of the queries in the subtree in the array location (i,j) and move to $(i,j+1)$, if any—otherwise we move to to $(i+1,1)$ to fill in the next row. Once we have reached and filled in $(1,1)$, we have the solution: position $(1,1)$ gives us the root word, $a_{r(1,1)}$; knowing the root choice, we compute the size of the left subtree, $s_l = r(1,1) - 1$, and of the right subtree, $s_r = n - r(1,1)$, so, unless one or the other is equal to zero, we can move to the array entries $(s_l,1)$ and $(s_r, r(1,1)+1)$, and continue in this vein to retrace the tree downwards.

Since filling in each entry in the array takes $O(n)$ time (trying out each word in the subproblem as a possible choice of root, taking $O(1)$ time for each trial) and we have $O(n^2)$ entries to fill in, the algorithm runs in $O(n^3)$ time.

Obviously, a dynamic program is much more computationally demanding than a divide-and-conquer algorithm: the divide-and-conquer algorithm does not test all possible ways of dividing a subproblem, but chooses one a priori (such as choosing the middle one, for instance), whereas the dynamic program must test every choice, in what amounts to a recursive setup (to compute the value of a choice, we must find out the optimal way to choose roots for its subtrees, and so forth all the way down). That the dynamic program is even able to run in polynomial time is due to a combination of two crucial factors. We have already remarked on the first: the number of subproblems–the size of the array to be filled in—is polynomial. The second is the computation of the value of an optimal solution to a subproblem: it requires only knowledge of the optimal solutions to smaller subproblems and can be computed in polynomial time using that knowledge. Another way to put it is that we can set up a recurrence relation to describe the optimal cost of a subproblem, a recurrence that depends only on entries for smaller subproblems; it is sometimes known in the literature as the *principle of optimality*. The principle of optimality by itself enables us to devise a dynamic program for an optimization problem, but in order for that algorithm to be useful, we also need to know that the number of subproblems is polynomially bounded. For instance, it is quite easy to devise a dynamic program for the Travelling Salesperson problem, but that DP will require exponential storage and take exponential time, because it will have an exponential number of subproblems.