CS 290H: Sparse matrix algorithms // Homework 2

Assigned April 9, 2008

Due by class time Wednesday, April 16

1. [20 points] Consider the 9-vertex grid graph, numbered by rows, with the edges directed to point from lower to higher numbered vertices. Perform `dfs(3,4)`, that is, carry out a depth-first search from vertices 3 and then 4. Show the postorder label assigned to each vertex.

2. [30 points] (See Davis problem 6.14.) In the left-looking LU factorization algorithm presented in class on April 9, one can speed up the structure-prediction step by so-called *symmetric pruning*, which reduces the number of edges in the depth-first-search graph. This speeds up the symbolic step without changing the numerical step. Thus it doesn't change the asymptotic $O(f)$ running time, but in practice it typically makes the whole factorization about four times as fast. Pruning can be done in either the no-pivoting ($A = LU$) or partial-pivoting ($PA = LU$) version of the factorization. For this problem, you can just consider the no-pivoting version.

(2a.) [10 points] Prove that if both l_{jr} and u_{rj} are nonzero for some $r < j$, then when predicting the structure of any column $k > j$, the depth-first search in $G(L^T)$ will still give the correct result if the search ignores nonzeros (edges) of the form l_{sr} with $s > j$. (See the "Symmetric Pruning" slide in the April 9 lecture notes for a diagram.)

(2b.) [10 points] The slide "Left-looking sparse LU without pivoting (pruned)" in the lecture notes outlines a method that uses an auxiliary directed graph S to represent the pruned version of L^T. Describe how to represent and manipulate S, in sufficient detail to show that pruning can be done within $O(f)$ worst-case time. For full credit, show also that you don't need to store an entire separate data structure for S, but rather can represent S by storing L in standard CSC form plus $O(n)$ extra pointers.

(2c.) [10 points] Prove (as the slide says) that if the original matrix A is symmetric, then the pruned graph S is isomorphic to the elimination tree of A.

3. [50 points] (See Davis problem 2.20.) Nobody knows any way to predict the exact number of nonzeros in the product $C = AB$ that is asymptotically faster than actually computing C. (This is very different from Cholesky factorization, where it's much faster to compute the number of nonzeros in the Cholesky factor than to compute the factor itself.) Therefore, any sparse matrix multiplication routine has to include some kind of trial-and-error way of allocating memory for its results. The purpose of this problem is to experiment with different ways of doing this.

You will need a sparse matrix multiplication routine to start with. I suggest that you use `cs_multiply` from the Davis book, and modify it to do your experiments. However, if you prefer, you may write your own routine using the outline from the April 2 class. Either way, you should use a Matlab mex-file interface for testing. The Matlab interface to the Davis code is described in section 10.3.

Experiment with the three (or four) methods below on the following two classes of n-by-n matrices, for values of n ranging from about 100 to as large as you can: first, uniform random matrices created by Matlab's `sprand(n,n,8/n)`; second, power-law matrices created by the Matlab routine `rmat(k)` (see the course web page). Note that the dimension of an `rmat(k)` matrix is actually $n = 2^k$. Use Matlab's `tic` and `toc` to get wall-clock times for the various methods. Make a plot in Matlab that shows f (number of flops) on the x axis, and running time on the y axis, with a point for each matrix. Plot the running time of Matlab's `C = A*B;` on the same figure (say "help hold") to Matlab. Can you beat Matlab's running time?

Turn in all your code, and also the plot with your run times and Matlab's, and also a Matlab transcript of the session that creates the plot and verifies that your output matrices agree with Matlab's.

First method: Guess and expand. This is the method both `cs_multiply` and the Matlab built-in matrix multiplication use. Guess an initial number of nonzeros to allocate for C (`cs_multiply` uses `nnz(A)+nnz(B)`), and then if you run out of space, allocate a larger space and copy the part of C you've computed so far into it. `cs_multiply` approximately doubles its guess each time.

Second method: Compute twice. This is the method suggested in Davis problem 2.20. Do the whole computation of $C = AB$ twice. The first time through, don't allocate any space for C; after computing each column of C in the SPA, discard it, but keep a count of the total number of nonzeros. The second time, allocate exactly the right amount of space for C.

Third method: Cheat. Let the user pass in an upper bound on the number of nonzeros in C, and just fail if you exceed it. This isn't a good idea in practice, but you should time this method (using a big enough bound) just to see how much time the memory allocation is costing.

Fourth method: Probabilistic estimate (extra credit or project possibility). If you're interested in digging deeper into this, you can read Edith Cohen's paper "Structure prediction and computation of sparse matrix products," *J. Combinatorial Optimization* 2:307–332, 1998, also at `http://www.springerlink.com/content/p328542122022748/`, which gives a fast probabilistic way of getting a good estimate of `nnz(A*B)`. The idea is to look at a graph whose vertices represent (separately) the rows and columns of A and B, and estimate the number of column vertices of B that have paths to each row vertex of A. Cohen gives a probabilistic estimate that uses repeated "rounds," each of which looks a lot like multiplying A and B by a dense vector. The more rounds, the better the estimate.