CSCE 689 : Special Topics in Sparse Matrix Algorithms
Department of Computer Science and Engineering
Spring 2016 syllabus

Tim Davis

last modified January 21, 2016

1 Catalog Description

CSCE 689. Special Topics in Sparse Matrix Algorithms. (3-0). Credit 3.
Many applications in computational science rely on algorithms for large-scale sparse matrices (circuit simulation, finite-element methods, 'big data', Google StreetView, etc). Course equips students to understand and design methods that exploit sparsity in matrix computations. Focus is direct methods, which rely on combinatorics, graph theory and algorithms, and numerical methods. Prerequisites: undergraduate-level numerical linear algebra and data structures/algorithms

2 Learning outcomes and Course objectives

At the end of the course, students will understand the basic principles of direct methods for sparse linear systems. They will understand how to represent a sparse matrix data structure and how to manipulate it. They will be able to understand and develop algorithms for solving sparse triangular systems, factorization methods (LU, Cholesky, and QR), and the design and use of heuristics for the NP-hard fill-reducing ordering problem. They will develop high-quality software that implements these algorithms.

3 Course Overview

Students in any discipline that uses scientific computing methods will benefit from this course. A wide range of problems in science and engineering require fundamental matrix computations for their solution, and these matrices are often mostly zero. The objective of this course is to equip the student with the background necessary to understand and design algorithms for solving sparse matrix problems. The world's largest sparse matrix arises in Google's web page ranking problem. About once a month, Google finds an eigenvector of sparse matrix that represents the connectivity of the web (of size billions-by-billions). Other sparse matrix problems arise in circuit and device simulation, finite element methods, linear programming and other optimization problems, acoustics, electromagnetics, computational fluid dynamics, 'big data', financial portfolio optimization, structural analysis, and quantum mechanics, to name a few.

Taking effective advantage of the structure of a sparse matrix requires a combination of numerical and combinatorial methods (graph algorithms and related data structures). For example, finding the best ordering to factorize a matrix is an NP-complete graph problem. Topics focus on direct methods, but with some application to iterative methods: sparse matrix-vector multiply, matrix-matrix multiply and transpose, forward/backsolve, LU and Cholesky factorization, singular value decomposition, reordering methods (including the use of graph partitioning methods), and updating/downdating a sparse Cholesky factorization. Projects in the course include programming assignments in C and MATLAB.

4 Class times:

Lecture: 3 50-minute periods per week

5 About the instructor:

- Office: 425E HRBB
- Phone: (979) 845 4094 (office). Cell (352) 359 2812.
- Email: davis@tamu.edu
- Web: http://faculty.cse.tamu.edu/davis/
- Office hours: after class, for 2 hours (or later as needed). I'm often available at other times as well. Feel free to stop by.

6 Book:

- Required: Direct Methods for Sparse Linear Systems, 2006, SIAM Press. Author: Timothy A. Davis. ISBN 0-89871-613-6.
- Required: students will be required to have access to MATLAB.
- Recommended reading: MATLAB Primer, 8th Edition, 2011, CRC Press. Author: Timothy A. Davis. ISBN 978-1-4398-2862-5.

7 Grading:

No exams. All grading is based on the projects. No collaboration will be allowed; all projects are solo. There will be 7 projects, most of which will involve developing algorithms for sparse matrix computations, and implementing them in C and MATLAB. Each project is equally weighted. Projects will be due every two weeks.

Grading is based on five key metrics:

1. Your code must work. It must solve the problem at hand.
2. Your algorithm design must be asymptotically perfect, in both time and memory complexity. Beware of spurious O(n) loops in the middle of your algorithm.
3. Your code must be elegant and well-structured. I must be able to read the code easily, and understand your algorithm.
4. Your code must be accompanied by a robust test suite that exercises the algorithms you have developed.
5. You must write up a short discussion and description of the design of your algorithm.

Grading scale (after curving of individual projects, as needed):

- 90 or higher: A
- 80: B
- 70: C
- 60: D
- less than 60: F

Seven SAMPLE projects below: All projects will be submitted via csnet.cse.tamu.edu. The actual projects will differ.

7.1 SAMPLE PROJECT 1

Four problems: Exercises 2.1, 2.2, 2.3 in the book and the problem listed below. For each exercise, create a mexFunction interface and a test procedure that exercises your code in MATLAB.

- Expand the `Source/cs_gaxpy.c` function so that it can handle both compressed-column and triplet matrices. Then create a mexFunction with the MATLAB prototype of:

  `z = cs_gaxpy_triplet (I, J, X, x, y)` where the vectors I, J, and X present a sparse matrix in triplet form (the kth nonzero in the matrix has row index I(k), column index J(k), and value X(k)). You can create I,J,X via `[I,J,X]=find(A)` where A is a sparse matrix.

- 2.1. Write a `cs_gatxpy` function that computes $y = A'x + y$ without forming A'.

- 2.2. Write a function `cs_find` that converts a cs matrix into a triplet-form matrix, like the find function in MATLAB.

- 2.3. Write a variant of `cs_gaxpy` that computes $y = Ax + y$, where A is a symmetric matrix with only the upper triangular part present. Ignore entries in the lower triangular part.

7.2 SAMPLE PROJECT 2

- 2.18. How much time and space does it take to compute $x'y$ for two sparse column vectors x and y, using `cs_transpose` and `cs_multiply`? Write a more efficient routine with prototype `double csdot (cs *x, cs *y)`, which assumes x and y are column vectors. Consider two cases: (1) The row indices of x and y are not sorted. A double workspace w of size x-¿m will need to be allocated. (2) The row indices of x and y are sorted. No workspace is required. Both cases take $O(|x| + |y|)$ time.

- 2.24. The MATLAB statement `C=A(i,j)`, where I and j are integer vectors, creates a submatrix C of A of dimension length(i)-by-length(j). Write a function that performs this operation. Either assume that i and j do not contain duplicate indices or that they may contain duplicates (MATLAB allows for duplicates).

7.3 SAMPLE PROJECT 3

- 3.1. Derive the algorithm used by `cs_utsolve`.
- 3.2. Try to describe an algorithm for solving $Lx = b$, where L is stored in triplet form, and x and b are dense vectors. What goes wrong?
- 3.3. The MATLAB statement `[L,U]=lu(A)` finds a permuted lower triangular matrix L and an upper triangular matrix U so that `L*U=A`. The rows of L have been permuted via partial pivoting, but the permutation itself is not available. Write a permuted triangular solver that computes `x=L\b` without modifying L and with only $O(n)$ extra workspace. Two passes of the matrix L are required, each taking $O(|L|)$ time. The first pass finds the diagonal entry in each column. Start at the last column and work backwards, flagging rows as they are seen. There will be exactly one nonzero entry in an unflagged row in each column. This is the diagonal entry of the unpermuted lower triangular matrix. In any given column, if there are no entries in unflagged rows, or more than one, then the matrix is not a permuted lower triangular matrix. The second pass then performs the permuted forward solve.

7.4 SAMPLE PROJECT 4

- 4.1. Use `cs_ereach` to implement an $O(|L|)$-time algorithm for computing the elimination tree and the number of entries in each row and column of L. It should operate using only the matrix A and $O(n)$ additional workspace. The matrix A should not be modified.
- 4.5. The `cs_ereach` function can be simplified if A is known to be permuted according to a postordering of its elimination tree and if the row indices in each column of A are known to be sorted. Consider two successive row indices i1 and i2 in a column of A. When traversing up the elimination tree from node i1, the least common ancestor of i1 and i2 is the first node a ¿ i2. Let p be the next-to-the-last node along the path i1 to a (where p ¡ i2 ¡ a). Include the path i1 to p in an output queue (not a stack). Continue traversing the tree, starting at node i2. The resulting queue will be in perfectly sorted order. The `while(len>0)` loop in `cs_ereach` can then be removed.

7.5 SAMPLE PROJECT 5

- 4.3. Write a function that solves $Lx = b$ when L, x, and b are sparse and L comes from a Cholesky factorization, using the elimination tree. Assume the elimination tree is already available; it can be passed in a parent array, or it can be found by examining L directly, since L has sorted columns.
- 4.4. Repeat Problem 4.3, but solve $L'x = b$ instead.
- For `cs_43lsolve`, you will need to start at nodes in the etree corresponding to entries in b. Then walk up the tree to find all nodes in x. For `cs_44ltsolve`, you do the opposite, looking *down* the tree to find all descendants, to get the pattern of x. You will need to compute a representation of the etree for problem 4.4 that gives a list of the children of each node. Normally that would be done once (it takes O(n) time) and then the $L'x = b$ solve can take less time than that. However, go ahead and do the O(n) work first, inside your mexFunction.

7.6 SAMPLE PROJECT 6

- Write an efficient implementation of the sparse row-merge QR algorithm on page 80, in C, but with some extra features as discussed below.

  Start with the pattern of R as given by symbfact. You will need to compute it MATLAB and pass it into your mexFunction. It is stored by row, so it's the same as L stored by column. You will also need to pass in the parent array from etree(A). And of course pass in A, but you must pass it in as stored by row, not column (pass in A transpose).

  The result is a numerical valued sparse R.

  You must implement the method as I discussed it in class, not exactly as in the `qr_givens.m` function as printed in the book. In particular, start with an R that is all zero (numerically) but with the pattern as given on input. If you find that the leftmost column index of a row is k, and R(k,k) is zero, then simply place the row as the kth row of R, and terminate the loop (do not keep iterating up the path to the root, but stop early).

- 5.5 Row-merge sparse QR factorization (page 80 in the textbook)

  A Givens-rotation-based QR factorization requires 50% more floating-point operations than a Householder-based QR factorization for a full matrix, but it has some advantages in the sparse case. Rows can be ordered to reduce the work below that of the Householder-based sparse QR. The disadvantage of using Givens rotations is that the resulting QR factorization is less suitable for multifrontal techniques. MATLAB uses Givens rotations for its sparse QR factorization. It operates on the rows of R and A. The matrix R starts out equal to zero but with enough space allocated to hold the final R. Each step of the factorization brings in a new row of A and eliminates its entries with the existing R until it is either all zero or it can be placed as a new row of R. The qr givens full algorithm for full matrices is shown below. It assumes the diagonal of A is nonzero. The innermost loop annihilates the aik entry via a Givens rotation of the incoming ith row of A and the kth row of R.

  For the sparse case, the rotation to zero out aik must be skipped if it is already zero. The entries k that must be annihilated correspond to the nonzero pattern $V(i,:)$ of the ith row of the Householder matrix V , discussed in the previous section.

  Theorem 5.10 (George, Liu, and Ng [95]). Assume A has a zero-free diagonal. The nonzero pattern Vi* of the ith row of the Householder matrix V is given by the path f to min(i,r) in the elimination tree T of A'A, where f = min A_{i*} is the leftmost nonzero entry in the ith row, and r is the root of the tree.

  This method is illustrated in the qr givens M-file as given below.

```
function R = qr_givens (A)
[m n] = size (A) ;
parent = cs_etree (sparse (A), 'col') ;
A = full (A) ;
for i = 2:m
    k = min (find (A (i,:))) ;
    if (isempty (k))
        continue ;
    end
    while (k > 0 && k <= min (i-1,n))
        A ([k i],k:n) = givens2 (A(k,k), A(i,k)) * A ([k i],k:n) ;
        A (i,k) = 0 ;
        k = parent (k) ;
    end
end
R = sparse (A) ;
```

7.7 SAMPLE PROJECT 7

- 6.5. Modify `cs_lu` so that it can factorize both numerically and structurally rank-deficient matrices.
- 6.9. The MATLAB interface for `cs_lu` sorts both L and U with a double transpose. Modify it so that it requires only one transpose for L and another for U. Hint: see Problem 6.1.

8 Topics

- 1 Introduction (1/2 week)
  - 1.1 Linear algebra
  - 1.2 Graph theory, algorithms, and data structures
- 2 Basic algorithms (2 weeks)
  - 2.1 Sparse matrix data structures
  - 2.2 Matrix-vector multiplication
  - 2.3 Utilities
  - 2.4 Triplet form
  - 2.5 Transpose
  - 2.6 Summing up duplicate entries
  - 2.7 Removing entries from a matrix
  - 2.8 Matrix multiplication
  - 2.9 Matrix addition
  - 2.10 Vector permutation
  - 2.11 Matrix permutation
  - 2.12 Matrix norm

  - 2.13 Reading a matrix from a file
  - 2.14 Printing a matrix
  - 2.15 Sparse matrix collections

- 3 Solving triangular systems (1 week)
  - 3.1 A dense right-hand side
  - 3.2 A sparse right-hand side

- 4 Cholesky factorization (3 weeks)
  - 4.1 Elimination tree
  - 4.2 Sparse triangular solve
  - 4.3 Postordering a tree
  - 4.4 Row counts
  - 4.5 Column counts
  - 4.6 Symbolic analysis
  - 4.7 Up-looking Cholesky
  - 4.8 Left-looking and supernodal Cholesky
  - 4.9 Right-looking and multifrontal Cholesky
  - 4.10 Modifying a Cholesky factorization

- 5 Orthogonal methods (2 weeks)
  - 5.1 Householder reflections
  - 5.2 Left- and right-looking QR factorization
  - 5.3 Householder-based sparse QR factorization
  - 5.4 Givens rotations
  - 5.5 Row-merge sparse QR factorization

- 6 LU factorization (2 weeks)
  - 6.1 Upper bound on fill-in
  - 6.2 Left-looking LU
  - 6.3 Right-looking and multifrontal LU

- 7 Fill-reducing orderings (2 weeks)
  - 7.1 Minimum degree ordering
  - 7.2 Maximum matching
  - 7.3 Block triangular form
  - 7.4 Dulmage-Mendelsohn decomposition
  - 7.5 Bandwidth and profile reduction
  - 7.6 Nested dissection

- 8 Solving sparse linear systems (1 week)
  - 8.1 Using a Cholesky factorization
  - 8.2 Using a QR factorization
  - 8.3 Using an LU factorization

  - 8.4 Using a Dulmage-Mendelsohn decomposition
  - 8.5 MATLAB sparse backslash
  - 8.6 Software for solving sparse linear systems

- 9 CSparse (1 week)

- 10 Sparse Matrices in MATLAB (1 week)

9 Policies:

- The use of laptops is prohibited in class, unless I approve otherwise (for example, if you have horrible handwriting and require a laptop to take notes). Laptops are a distraction to your fellow students. Please see me if you need an exception.
- The use of bootleg copies of the textbooks is strictly prohibited.
- The Americans with Disabilities Act (ADA) is a federal anti-discrimination statute that provides comprehensive civil rights protection for persons with disabilities. Among other things, this legislation requires that all students with disabilities be guaranteed a learning environment that provides for reasonable accommodation of their disabilities. If you believe you have a disability requiring an accommodation, please contact Disability Services, currently located in the Disability Services building at the Student Services at White Creek complex on west campus or call 979-845-1637. For additional information, visit http://disability.tamu.edu.
- You must observe the Aggie Code of Honor: http://student-rules.tamu.edu/aggiecode.