University of Toronto
CSC148S—Introduction to Computer Science, Spring 1996

Assignment 3: Symbolic Factorization

Due: Wednesday 6 March, 6:00 pm, in the 148 drop boxes
This assignment is worth 10 marks towards your final grade.

Introduction

An important problem in numerical linear algebra is splitting or *factoring* a matrix into the product of two matrices. For example, the 3×3 matrix

$$A = \begin{bmatrix} 7 & 8 & 9 \\ 14 & 46 & 51 \\ 28 & 82 & 163 \end{bmatrix}$$

can be factored into two 3×3 matrices

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 7 & 8 & 9 \\ 0 & 10 & 11 \\ 0 & 0 & 12 \end{bmatrix}$$

such that $A = L \times U$.[1] Note that there are infinitely many ways of doing this. The most "useful" factorizations, however, often make L *lower-triangular* (i.e., containing all zeros above the diagonal) and U *upper-triangular* (i.e., containing all zeros below the diagonal), as in the above example. Also, the ordering of the rows and columns in the matrix often can be changed prior to doing the factorization. For example, if we interchange rows 1 and 3 and columns 1 and 3 in the above matrix we get

$$\bar{A} = \begin{bmatrix} 163 & 82 & 28 \\ 51 & 46 & 14 \\ 9 & 8 & 7 \end{bmatrix}$$

which can be factored into two 3×3 matrices

$$\bar{L} = \begin{bmatrix} 1.0000 & 0 & 0 \\ 0.3129 & 1.0000 & 0 \\ 0.0552 & 0.1707 & 1.0000 \end{bmatrix} \quad \text{and} \quad \bar{U} = \begin{bmatrix} 163.0000 & 82.0000 & 28.0000 \\ 0 & 20.3436 & 5.2393 \\ 0 & 0 & 4.5597 \end{bmatrix}$$

(The elements of $\bar{L}$ and $\bar{U}$ are shown to 4 decimal places only.) The factors of $\bar{A}$ are, of course, different from those of A, but this is fine—the factorization is still useful. If you are interested in why this is so, read the section entitled "An application of matrix factorization" included at the end of this handout. You do *not* need to understand that section in order to do this assignment. It is included for your information only. Also, you do *not* have to worry about the details of how to compute a factorization. We will supply you with an appropriate Turing procedure that inputs a matrix A and computes its factors L and U.

[1] You should check that A does indeed equal $L \times U$.

Symmetric sparse matrices

In this assignment, you will factor a special kind of matrix. As an example, consider

$$S = \begin{bmatrix} 30 & 15 & 12 & 7 \\ 15 & 25 & 0 & 0 \\ 12 & 0 & 36 & 0 \\ 7 & 0 & 0 & 49 \end{bmatrix}.$$

This 4×4 matrix has at least two special properties. It is

- *symmetric*: $S(i, j) = S(j, i)$ for all i, j, and
- *sparse*: a large proportion of the elements are zero.

If we take advantage of these properties, significant storage savings are possible when we represent S on a computer. Since S is symmetric, we need to store only the lower (or upper) triangle. Since it is sparse, we could save even more if we store only the non-zero elements.[2]

Actually, S has one additional property—it is *positive-definite*. For the purposes of this assignment, this simply means that the rows and columns of S may be reordered before beginning the factorization without affecting the accuracy of the algorithm. The only restriction is that *S must remain symmetric*, so if you swap rows i and j, you must also swap columns i and j.

We will supply you with a factorization procedure tailored for these kinds of matrices. This procedure, when given S, produces the following factors:

$$L = \begin{bmatrix} 5.4772 & 0 & 0 & 0 \\ 2.7386 & 4.1833 & 0 & 0 \\ 2.1909 & -1.4343 & 5.3984 & 0 \\ 1.2780 & -0.8367 & -0.7410 & 6.7910 \end{bmatrix} \text{ and } U = \begin{bmatrix} 5.4772 & 2.7386 & 2.1909 & 1.2780 \\ 0 & 4.1833 & -1.4343 & -0.8367 \\ 0 & 0 & 5.3984 & -0.7410 \\ 0 & 0 & 0 & 6.7910 \end{bmatrix}.$$

Two things are worth noting here. First, L and U are very similar. In fact, the rows of L are the columns of U. We say that L is the *transpose* of U and vice-versa (denoted $L = U^T$ or $U = L^T$). *This is always the case when symmetric matrices are factored using the procedure we supply.* For this reason, the procedure computes L only. U can easily be obtained afterwards by transposing L. Secondly, in this example, the computed factors of S are *dense*—the lower triangle of L and the upper triangle of U contain no zeros. This is surprising, given that the original matrix S was sparse. When this happens, we say that *fill-in* has occurred during the factorization, or that the factors suffer from *fill-in*.

Fill-in during factorization is something to be avoided. It increases storage costs and, even worse, can significantly slow down the algorithm. Fortunately, it sometimes can be reduced or even prevented. Consider what happens if we first reorder the rows and columns of S before factorization. One possible reordering is matrix $\bar{S}$ below; row/column $(1, 2, 3, 4)$ of $\bar{S}$ is row/column $(4, 3, 2, 1)$ of S. With this reordering, our factorization procedure computes the lower-triangular factor $\bar{L}$ shown below ($\bar{U}$ is just $\bar{L}^T$):

$$\bar{S} = \begin{bmatrix} 49 & 0 & 0 & 7 \\ 0 & 36 & 0 & 12 \\ 0 & 0 & 25 & 15 \\ 7 & 12 & 15 & 30 \end{bmatrix}, \quad \bar{L} = \begin{bmatrix} 7 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix}.$$

[2] The storage savings achieved by exploiting sparsity are far more dramatic in larger problems. Think about how much you could save when storing a 1000×1000 matrix containing only 50,000 non-zeros.

This is *much* better—$\bar{L}$ is as sparse as the original matrix S! One goal of this assignment is to investigate how row/column order affects fill-in during sparse matrix factorization.

Cholesky factorization

Figure 3 on page 9 shows a Turing implementation of *Cholesky factorization*, the algorithm you will use to factor matrices in this assignment. This procedure computes the factorization of a real $n \times n$ symmetric positive-definite matrix S. The lower triangular factor is returned in L; the upper triangular factor is simply L^T. Since elements above the main diagonal of S are not accessed by the procedure, they need not be initialized in the calling program.

Note that each of the following statements in procedure Cholesky could cause a run-time error:

```
L(j,j) := sqrt(S(j,j) - sum)
L(i,j) := (S(i,j) - sum)/L(j,j)
```

The first will cause an error if `(S(j,j) - sum)` is negative, and the second if `L(j,j)` is zero. These errors will *not* occur if the input matrix is positive-definite.

To save typing, you can pick up procedure Cholesky in file `148chol.t` using the "Pickup standard file" icon on the PCs.

Symbolic factorization

If we knew in advance exactly which row/column ordering results in factors with the least fill-in, we could save storage and reduce the running time of procedure Cholesky. As a first approach to this problem, one might simply try a few different orderings and select the best one. This process can be made more efficient by considering only the *position* of each non-zero element in the matrix rather than its numerical value. For example, matrices S and L above could be represented as

$$S = \begin{bmatrix} * & * & * & * \\ * & * & & \\ * & & * & \\ * & & & * \end{bmatrix} \text{ and } L = \begin{bmatrix} * & & & \\ * & * & & \\ * & * & * & \\ * & * & * & * \end{bmatrix}$$

where each $*$ represents a non-zero element. Unmarked positions are assumed to contain zero. Similarly, $\bar{S}$ and $\bar{L}$ above could be represented as

$$\bar{S} = \begin{bmatrix} * & & & * \\ & * & & * \\ & & * & * \\ * & * & * & * \end{bmatrix} \text{ and } \bar{L} = \begin{bmatrix} * & & & \\ & * & & \\ & & * & \\ * & * & * & * \end{bmatrix}.$$

These representations are often called *sparsity patterns*. One of your tasks in this assignment is to modify procedure Cholesky so that it processes sparsity patterns; i.e., the matrix to be factored will be input as a sparsity pattern, and the computed factor will be output as a sparsity pattern. This is called a *symbolic factorization*.

With symbolic factorization, we have a relatively inexpensive tool for investigating how specific row/column orderings in a matrix affect the sparsity of its factors. It is inexpensive because we do not have to compute the actual values of the nonzeros, only their positions. Note, however, that it is not practical to try to find the *optimal* ordering using this approach. Think about why not. (*Hint*: How many different orderings are there for a 1000×1000 matrix?)

Your task

Step 1: The first thing you should do is write a short driver program and try out procedure Cholesky to make sure you understand how to use it. Remember that the procedure factors symmetric positive-definite matrices only. Since you probably do not know how to construct such a matrix, just use S above as input for now. Check that the returned factors are correct; i.e., check that $S = L \times L^T$.

Step 2: Once you are familiar with how procedure Cholesky works on matrices with numerical values, modify it so that it performs a symbolic factorization on a sparsity pattern. Modifying the procedure is straightforward. Essentially, you replace all arithmetic operations with comparisons. *Note that your modified procedure must not use any arithmetic operations.* They are not necessary in a symbolic factorization and will make the procedure inefficient. You may assume that cancellation does not occur during the factorization, so you do not have to worry about a potential divide-by-zero. For now, store the sparsity pattern as a two-dimensional array of boolean. ($A(i,j)$ is `true` if it contains a non-zero, `false` otherwise.) Check that the symbolic factorization is computed correctly for the sparsity pattern of S above.

Step 3: Although you will not be graded for these first two steps, we recommend that you do them anyway in preparation for the next step: changing over to a linked-list data structure. This is the difficult part of your task. When a sparsity pattern is stored as a linked list, accessing existing non-zeros and creating new non-zeros during factorization is not quite as straightforward as with a two-dimensional array. See the section below entitled "Data structures" for some hints on how to organize the linked lists.

In order to test your modified symbolic factorization procedure, include it as part of a program that does the following:

> **loop**
>   **exit** when **eof**
>   input n, followed by an $n \times n$ sparsity pattern S
>   input p, followed by a series of p row/column orderings
>   check that S can be factored
>   **if** S cannot be factored **then**
>   print out an error message
>   **else**
>   **for** each ordering
>   factor $\bar{S}$
>   output $\bar{S}$, $\bar{L}$ and $\bar{L}^T$
>   **end for**
>   **end if**
> **end loop**

An example of input and output is shown in figures 1 and 2 and discussed in the next section.

Figure 1: A sparsity pattern and three orderings.

```
4
****
**
* *
*  *
3
1 2 3 4
2 1 4 3
4 3 2 1
```

To check that a sparsity pattern S can be factored, test if the following conditions hold:

- $S(i,j) \neq 0$ if and only if $S(j,i) \neq 0$ for all i,j (this is a symmetry test), and
- $S(i,i) \neq 0$ for all i.

You may assume that cancellation will not occur if these conditions hold. The notion of "positive-definite" does not apply to sparsity patterns.

Note that you are welcome to recycle any part of *your* code from previous assignments for use in this assignment. In particular, your maze input procedure from assignment #2 might be useful.

Example input and output

Figure 1 shows example input containing one sparsity pattern and three orderings. The first line contains a single integer n specifying the number of rows and columns in the pattern. In this example, $n = 4$. This is followed by the sparsity pattern, made up of $*$'s and blanks. You may assume there are no errors here—n will be followed by *exactly* n lines, each of length n, and each containing $*$'s and blanks. Note, though, you *may not* assume that the sparsity pattern is symmetric.

The sparsity pattern is followed by a line containing a single integer p specifying the number of row/column orderings or *permutations* to try. In this example, $p = 3$. A permutation is specified by a set of n integers, each of which is between 1 and n. For example, consider the second permutation: 2 1 4 3. This tells us to form a new sparsity pattern $\bar{S}$ from the original pattern S as follows. First, we permute rows 2 1 4 3 so that they become rows 1 2 3 4, respectively. Second, we permute columns 2 1 4 3 so that they become columns 1 2 3 4, respectively. (Row interchanges are followed by equivalent column interchanges to ensure that the pattern remains symmetric.) You may assume that the permutations are input correctly.

Since the sparsity pattern in our example is symmetric and has all non-zeros on the main diagonal, it can be factored. Figure 2 shows the output that should result. Clearly, the last of the three permutations results in a factorization with the least fill-in. Note that the first permutation is the "identity"—it does not change the original pattern. This allows us to see the original sparsity pattern and its factors.

Figure 2: Sample output for figure 1.

```
----------------------------------
permutation   S       L       L^T
----------------------------------
  1 2 3 4
             ****     *       ****
             **       **       ***
             * *      ***       **
             *  *     ****       *
  2 1 4 3
             **       *       **
             ****     **       ***
              **       **       **
              * *      ***       *
  4 3 2 1
             *  *     *       *  *
              * *      *       * *
               **       *       **
             ****     ****       *
```

Data structures

A two-dimensional array is not the best data structure for storing a symmetric sparse matrix. As mentioned above, considerable savings are possible if we store only the non-zeros in the lower triangle. There are a few ways we could do this. One possibility is to pack the non-zeros consecutively in a one-dimensional array and then use a clever indexing scheme to determine their true position in the original matrix. Another possibility is to use a linked list. Since this assignment is intended to give you some practice in using pointers, we require that you use the latter.

There is more than one way of using a linked list for this application. For example, should you use single or double links? Probably the most appropriate choice is to use an array of singly linked lists, where each list holds the non-zeros in one row of the lower triangle. Remember that our goal is to save storage. Store only the non-zeros in the lower triangle.

Also, since you are storing a sparsity pattern rather than a matrix of numerical values, the mere *presence* of a node in a linked list is enough to represent an element. Of course you must also include a column index in the node in order to determine the element's true position in the original matrix. When you output a sparsity pattern, however, it should appear as in the above examples (i.e., use $*$'s).

Testing

Test your symbolic factorization procedure on a variety of sparsity patterns and orderings.

You can easily create such patterns—just make sure that they are symmetric, sparse, and non-zero on the diagonal.

A couple of days before the assignment is due, we will supply test cases to submit for marking. These are the only test runs that you are to hand in.

What to hand in

Hand in your Turing program, the output resulting from the test cases we supply, and your report. Your report should follow the general guidelines given in the CSC148 Course Information Booklet, but it will also include two things particular to this assignment:

- A complexity analysis of procedure Cholesky (Figure 3). Be sure to show all steps in your analysis; don't just give the final answer.
- A description of an alternative data structure that could be used to store a sparsity pattern, and a brief discussion of the advantages and disadvantages it has over the one you chose.

Since the only test runs you hand in will be for our test cases, do not include a Testing section in your report. However, correctness marks for your code will be allocated based on these test runs, so don't forget to include printouts of input and output for these tests!

Put your program, testing, and report in an 8 1/2 by 11 inch envelope. Fill out and sign the cover sheet provided with this handout and tape it to the outside of the envelope. Please do not glue, staple, or tape the envelope shut—just fold in the flap. Hand in your assignment to the CSC148 drop boxes by 6:00 p.m. on Wednesday 6 March. The location of the drop boxes is described on page 9 of the Course Handbook.

An application of matrix factorization

(This section is included for your information only. You do not need to understand its content in order to do the assignment.)

Consider the following system of simultaneous linear equations:

$$\begin{aligned} 7x_1 + 8x_2 + 9x_3 &= 20 && (1) \\ 14x_1 + 46x_2 + 51x_3 &= -11 && (2) \\ 28x_1 + 82x_2 + 163x_3 &= -509 && (3) \end{aligned}$$

In high school you learned how to solve such a system by the method of *elimination*. This involves adding multiples of one equation to another in order to eliminate selected variables. For example, by first eliminating x_1 from equations 2 and 3, and then x_2 from equation 3, the above system becomes

$$\begin{aligned} 7x_1 + 8x_2 + 9x_3 &= 20 && (4) \\ 10x_2 + 11x_3 &= -17 && (5) \\ 12x_3 &= -84 && (6) \end{aligned}$$

(See if you can do this. Note that some equations have been scaled.) Because we are consistent in the way we eliminate variables, equations 4, 5 and 6 and equations 1, 2, and 3 have the *same* solution. The system with equations 4, 5 and 6, however, is "easier" to solve. In particular, equation 6 immediately gives $x_3 = -7$. Substituting this back into equation 5 and rearranging gives $x_2 = 6$. Finally, substituting $x_3 = -7$, $x_2 = 6$ back into equation 4 and rearranging gives $x_1 = 5$.

There is an alternative, more precise way of describing this solution process if we first represent the equations in *matrix-vector* form. For example, equations 1, 2, and 3 could be represented as $Ax = b$ where

$$A = \begin{bmatrix} 7 & 8 & 9 \\ 14 & 46 & 51 \\ 28 & 82 & 163 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 20 \\ -11 \\ -509 \end{bmatrix}.$$

The solution process can now be described as follows. First, we find a *factorization* of A; i.e., we find a lower-triangular factor L and an upper-triangular factor U such that $A = LU$. Substituting this factorization into $Ax = b$ gives

$$Ax = b \iff LUx = b,$$

and letting $y = Ux$ gives

$$Ax = b \iff LUx = b \iff Ly = b.$$

This last system, $Ly = b$, is "easy" to solve because L is lower-triangular. (We can solve for y in much the same way as we solved equations 4, 5, and 6.) Once we have y, we can compute the desired solution x by solving $Ux = y$. This last system is exactly equations 4, 5, and 6 in the example above.

This application illustrates why we still get a "useful" factorization even if we first reorder the rows and columns of A. Reordering the rows of A corresponds to reordering the *equations* of $Ax = b$. The solution remains the same. Reordering the columns of A corresponds to reordering the *variables* of $Ax = b$ (i.e., x_1, x_2, x_3). Again, as long as we note the new order of the variables, the solution remains the same.

Figure 3: Cholesky factorization.

```
procedure Cholesky (n : int, S : array 1..*,1..* of real,
                         var L : array 1..*,1..* of real)

    %  This procedure computes the lower triangular Cholesky factor L
    %  of a real n x n matrix S.  S must be symmetric positive-definite.
    %  If it is not, the fatal errors of divide-by-0 or square root of
    %  a negative number could occur.
    %
    %  Upon return, S = L x transpose(L).
    %
    %  Elements above the main diagonal of S are not accessed.

    var sum : real

    % L is constructed column-wise
    for j : 1 .. n
        % elements above main diagonal are zero
        for i : 1 .. j-1
            L(i,j) := 0.0
        end for
        % compute element on main diagonal
        sum := 0
        for k : 1 .. j-1
            sum := sum + L(j,k)**2
        end for
        L(j,j) := sqrt(S(j,j) - sum)
        % compute elements below main diagonal
        for i : j+1 .. n
            sum := 0
            for k : 1 .. j-1
                sum := sum + L(i,k)*L(j,k)
            end for
            L(i,j) := (S(i,j) - sum)/L(j,j)
        end for
    end for

end Cholesky
```

University of Toronto
CSC148S—Introduction to Computer Science, Spring 1996

Cover sheet for Assignment 3

Complete this page and tape it to the front of the envelope containing your assignment.

Name: ______________________ Tutor: ______________________
(Underline your surname)

Student #: ______________________

I have discussed this assignment with the following people:

I have read section 7.2 of the CSC148 Course Handbook on plagiarism, and I declare that this assignment is my own work.

Signature: ______________________

Tentative marking scheme:

Program		
Correctness	________	/ 45
Style	________	/ 10
Comments	________	/ 10
Report		
Use section	________	/ 5
Method section	________	/ 10
Complexity analysis	________	/ 15
Overall quality of writing	________	/ 5
Total	________	/ 100