Chapter 3
Solving triangular systems

Solving a triangular system, $Lx = b$, where L is square and lower triangular, is a key mathematical kernel. It will be used in Chapter 4 as part of a sparse Cholesky factorization algorithm and in Chapter 6 as part of a sparse LU factorization algorithm. The nonzero pattern of x will be used to construct the nonzero pattern of a column of R for the QR factorization presented in Chapter 5. Solving $Lx = b$ is also essential for solving $Ax = b$ after either a Cholesky or LU factorization of A.

3.1 A dense right-hand side

There are many ways of solving $Lx = b$ (a *forward solve*), but if L is stored as a compressed-column sparse matrix, accessing L by columns is the most natural. Consider the 2-by-2 block decomposition,

first solve x1, then update x2

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \tag{3.1}$$

where L_{22} is the lower right $(n-1)$-by-$(n-1)$ submatrix of L; l_{21}, x_2, and b_2 are column vectors of length $n-1$; and l_{11}, x_1, and b_1 are scalars. This leads to two equations, $l_{11}x_1 = b_1$ and $l_{21}x_1 + L_{22}x_2 = b_2$. To solve $Lx = b$, the first can be solved ($x_1 = b_1/l_{11}$) to obtain the first entry in x. The second equation is a lower triangular system of the form $L_{22}x_2 = b_2 - l_{21}x_1$ that can be solved recursively for x_2. Unwinding the tail recursion leads naturally to an algorithm that iterates over the columns of L. Note that b_1 and b_2 are used just once; this allows x to overwrite b in the implementation:

1

$x = b$
for $j = 0$ to $n-1$ **do**
  $x_j = x_j / l_{jj}$
  for each $i > j$ for which $l_{ij} \neq 0$ **do**
    $x_i = x_i - l_{ij} x_j$

If `x` is a dense vector but `L` is sparse, the algorithm and code are very similar to the matrix-vector multiplication, `cs_gaxpy`. On input, `x` contains the right-hand side

b; on output it contains the solution to $Lx = b$. The `cs_lsolve` function assumes that the diagonal entry of `L` is always present and is the first entry in each column. Otherwise, the row indices in each column of `L` can appear in any order.

```
int cs_lsolve (const cs *L, double *x)
{
    int p, j, n, *Lp, *Li ;
    double *Lx ;
    if (!CS_CSC (L) || !x) return (0) ;                    /* check inputs */
    n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (j = 0 ; j < n ; j++)
    {
        x [j] /= Lx [Lp [j]] ;                                    3
        for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
        {
            x [Li [p]] -= Lx [p] * x [j] ;
        }
    }
    return (1) ;
}
```

Solving $L^T x = b$ (a *backsolve*) is best done by accessing L^T by rows, since L is stored by column. The 2-by-2 block decomposition becomes

$$\begin{bmatrix} l_{11} & l_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \qquad \texttt{x2 is known now!}$$

where each entry in L has the same size as in (3.1).

```
int cs_ltsolve (const cs *L, double *x)
{
    int p, j, n, *Lp, *Li ;
    double *Lx ;
    if (!CS_CSC (L) || !x) return (0) ;                    /* check inputs */
    n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (j = n-1 ; j >= 0 ; j--)
    {
        for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
        {
            x [j] -= Lx [p] * x [Li [p]] ;
        }
        x [j] /= Lx [Lp [j]] ;                        difference here!
    }
    return (1) ;
}
```

To solve $Ux = b$, where U is stored by column, yet another 2-by-2 decomposition is used:

$$\begin{bmatrix} U_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix},$$

where U_{11} is $(n-1)$-by-$(n-1)$. This results in the two equations $U_{11}x_1 + u_{12}x_2 = b_1$ and $u_{22}x_2 = b_2$. The second equation can be solved for $x_2 = b_2/u_{22}$, and the first becomes $U_{11}x_1 = b_1 - u_{12}x_2$. These equations are encapsulated in the function `cs_usolve`. It assumes the diagonal entry is always present and appears as the last

entry in each column. Row indices in the columns of U can otherwise be in any order.

```
int cs_usolve (const cs *U, double *x)
{
    int p, j, n, *Up, *Ui ;
    double *Ux ;
    if (!CS_CSC (U) || !x) return (0) ;                    /* check inputs */
    n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
    for (j = n-1 ; j >= 0 ; j--)
    {
        x [j] /= Ux [Up [j+1]-1] ;
        for (p = Up [j] ; p < Up [j+1]-1 ; p++)            2
        {
            x [Ui [p]] -= Ux [p] * x [j] ;
        }
    }
    return (1) ;
}
```

The cs_utsolve function solves $U^T x = b$, where U is upper triangular. Its derivation is left as an exercise.

```
int cs_utsolve (const cs *U, double *x)
{
    int p, j, n, *Up, *Ui ;
    double *Ux ;
    if (!CS_CSC (U) || !x) return (0) ;                    /* check inputs */
    n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Up [j] ; p < Up [j+1]-1 ; p++)
        {
            x [j] -= Ux [p] * x [Ui [p]] ;
        }
        x [j] /= Ux [Up [j+1]-1] ;
    }
    return (1) ;
}
```

cs_lsolve(L,x), cs_ltsolve(L,x), cs_usolve(U,x), and cs_utsolve(U,x) correspond to x=L\x, x=L'\x, x=U\x, and x=U'\x in MATLAB, except that the transposed solvers cs_ltsolve and cs_ltsolve do not transpose their inputs.

3.2 A sparse right-hand side

The Cholesky and LU factorization algorithms presented in Chapters 4 and 6 rely on the solution to $Lx = b$ to compute one row or column of the factors, where L, x, and b are all sparse. The QR factorization algorithm presented in Chapter 5 uses the nonzero pattern of x to construct one column of R. This small change, from a dense right-hand side in cs_lsolve to a sparse right-hand side, has a large impact on the algorithm and its underlying theory.

To simplify the discussion, assume that L has a unit diagonal (this will be the case for its use in LU factorization). The algorithm for solving $Lx = b$ becomes

```
x = b
for j = 0 to n − 1 do
    if x_j ≠ 0
        for each i > j for which l_ij ≠ 0 do
            x_i = x_i − l_ij x_j
```

The sparse vector x can be temporarily stored in a dense vector of size n, assumed to be initially zero. Thus, the two statements $x = b$ and $x_i = x_i - l_{ij}x_j$ can be done efficiently. If this algorithm is implemented as in the above pseudocode, the time taken would be $O(n + |b| + f)$, where f is the number of floating-point operations performed and $|b|$ is the number of nonzeros in b. Normally, $|b| < f$, so the time is $O(n + f)$. This looks efficient, but it is not. The floating-point operation count can easily be dominated by n. If b is all zero except for b_n, f is $O(1)$, but the total work is $O(n)$. Basing an LU factorization algorithm on this method for solving $Lx = b$ would lead to an $\Omega(n^2)$-time factorization, which is clearly unacceptable. Factorizing a tridiagonal matrix should take $O(n)$ time, not $O(n^2)$ time.

The problem is the **for** j loop. A better method would assume that the algorithm starts with a list of indices j for which x_j will be nonzero, $\mathcal{X} = \{j \,|\, x_j \neq 0\}$, sorted in ascending order. The algorithm would then be

```
x = b
for each j ∈ X do
    for each i > j for which l_ij ≠ 0 do
        x_i = x_i − l_ij x_j
```

Assuming $\mathcal{X}$ is already given, the run time drops to $O(|b| + f)$, which is essentially $O(f)$, an ideal target.

The problem now becomes how to determine $\mathcal{X}$ and how to sort it. Entries in x become nonzero in two places, the first and last lines of the above pseudocode. If numerical cancellation is neglected, these two statements can be written as two logical implications:

1. $b_i \neq 0 \Rightarrow x_i \neq 0$.
2. $x_j \neq 0 \wedge \exists i (l_{ij} \neq 0) \Rightarrow x_i \neq 0$.

These two rules can be expressed as a graph traversal problem. Consider a directed graph $G_L = (V, E)$, where $V = \{1 \ldots n\}$ and $E = \{(j, i) \,|\, l_{ij} \neq 0\}$ (note that this is actually the graph of L^T). The graph is acyclic. If marked nodes in G_L correspond to nonzero entries in x, rule one translates into marking all those nodes $i \in \mathcal{B}$, where $\mathcal{B} = \{i \,|\, b_i \neq 0\}$. Rule two states that if node j is marked, and there is an edge from node j to node i, then node i must be marked. The set $\mathcal{X}$ becomes the set of all nodes in G_L that can be reached via a path from one or more nodes in $\mathcal{B}$. These rules are illustrated in Figure 3.1. In graph terminology, $\mathcal{X} = \text{Reach}_{G_L}(\mathcal{B})$, or more simply $\mathcal{X} = \text{Reach}_L(\mathcal{B})$, to avoid double subscripts. This gives a formal proof of the following theorem.

Theorem 3.1 (Gilbert and Peierls [109]). *Define the directed graph $G_L = (V, E)$ with nodes $V = \{1 \ldots n\}$ and edges $E = \{(j, i) \,|\, l_{ij} \neq 0\}$. Let $\text{Reach}_L(i)$ denote*

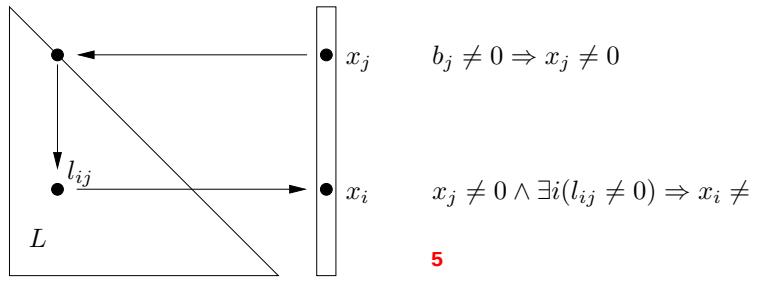

Figure 3.1. *Sparse triangular solve*

the set of nodes reachable from node i via paths in G_L, and let Reach($\mathcal{B}$)*, for a set $\mathcal{B}$, be the set of all nodes reachable from any node in $\mathcal{B}$. The nonzero pattern $\mathcal{X} = \{j \,|\, x_j \neq 0\}$ of the solution x to the sparse linear system $Lx = b$ is given by $\mathcal{X} = \text{Reach}_L(\mathcal{B})$, where $\mathcal{B} = \{i \,|\, b_i \neq 0\}$, assuming no numerical cancellation.*

The set $\mathcal{X}$ can be computed by a depth-first search of the directed graph G_L, starting at nodes in $\mathcal{B}$. The time taken by a depth-first search is proportional to the number of edges traversed, plus the number of initial nodes in $\mathcal{B}$. Each edge reflects exactly two floating-point operations in the numerical solution to $Lx = b$, so the total time is thus $O(|b| + f)$. A depth-first search does not sort the set $\mathcal{X}$, however. Fortunately, the update $x_i = x_i - l_{ij}x_j$ can be computed as soon as x_j is known. This update translates into two nodes j and i in $\mathcal{X}$ with an edge from j to i in the directed graph G_L. An ordering of $\mathcal{X}$ that preserves this precedence is called a *topological order*, and a depth-first search can compute $\mathcal{X}$ in topological order (a breadth-first search cannot).

A depth-first search is most easily written as a recursive algorithm, stated in pseudocode below. The *reach* function computes $\mathcal{X} = \text{Reach}_L(\mathcal{B})$ by starting a depth-first search at each node $i \in \mathcal{B}$.

```
function X = reach (L, B)
    assume all nodes are unmarked
    for each i for which b_i ≠ 0 do
        if node i is unmarked
            dfs (i)

function dfs (j)
    mark node j
    for each i for which l_ij ≠ 0 do
        if node i is unmarked
            dfs (i)
    push j onto stack for X      enter the stack!
```

7

These two pseudocodes can be implemented with `reachr` and `dfsr` below.

```
int reachr (const cs *L, const cs *B, int *xi, int *w)
{
    int p, n = L->n ;
    int top = n ;                                   /* stack is empty */
    for (p = B->p [0] ; p < B->p [1] ; p++)         /* for each i in pattern of b */
    {
        if (w [B->i [p]] != 1)                      /* if i is unmarked */
        {                                       w:mark whether it has been reached, len=n
            dfsr (B->i [p], L, &top, xi, w) ;   /* start a dfs at i */
        }    rhs row ind
                                                top:if marked top--
    }
    return (top) ;                                  /* return top of stack */
}                                                xi:len=n, while tail top stores the locations

void dfsr (int j, const cs *L, int *top, int *xi, int *w)
{
    int p ;    always update w first, nested functions
    w [j] = 1 ;                                     /* mark node j */
    for (p = L->p [j] ; p < L->p [j+1] ; p++)       /* for each i in L(:,j) */
    {
        if (w [L->i [p]] != 1)                      /* if i is unmarked */
        {
            dfsr (L->i [p], L, top, xi, w) ;        /* start a dfs at i */
        }
    }
    xi [--(*top)] = j ;                             /* push j onto the stack */
}
```

The `reachr` function computes $\mathcal{X} = \mathrm{Reach}_L(\mathcal{B})$, where $\mathcal{B}$ is the nonzero pattern of the n-by-1 sparse column vector b. The function returns $\mathcal{X}$ in the `xi` array, in locations `top` to `n-1`, in topological order. The array `w` is of size `n` and must be all zero on input. In `reachr`, a depth-first search is started at each node in $\mathcal{B}$, unless that node is already marked. The `dfsr` function starts a depth-first search at node `j`. It simply marks node `j` and then starts a depth-first search at any unmarked neighbors. When it finishes, it pushes `j` onto a stack containing $\mathcal{X}$ on output, topologically sorted.

Recursive algorithms are easy to understand, but they can cause stack overflow if the recursion goes too deep. The stack depth is up to n, which limits the algorithm to solving matrices of modest size. Thus, CSparse does not rely on `dfrs` and `reachr` (they are included but only for reference). The `cs_dfs` function below uses its own recursion stack `xi[0]` to `xi[head]` that does not overlap with the output stack in `xi[top]` to `xi[n-1]`, since no node can be in both stacks at the same time. The input matrix is called `G`, because it need not be lower triangular. Two parameters are added in anticipation of Chapter 6 (`pinv` and `k`). The parameter `k` specifies which column of `B` contains the right-hand side b. For now, assume `pinv` is `NULL`.

Initializing the workspace `w` of size `n` takes $O(n)$ time. This can be avoided by marking nodes using the matrix `G` itself. To denote a marked node `j`, `Gp[j]` is set to `CS_FLIP(Gp[j])`, which exploits the fact that `Gp[j]` ≥ 0 in an unmodified matrix `G`. A marked node `j` will have `Gp[j]` < 0. To unmark a node or to obtain the original value of `Gp[j]`, `CS_FLIP` can be applied again, since the function is its own inverse. The name "flip" is used because the function "flips" its input about the integer `-1`. `CS_UNFLIP(i)` "flips" `i` if it is negative or returns `i` otherwise.

```
#define CS_FLIP(i) (-(i)-2)
#define CS_UNFLIP(i) (((i) < 0) ? CS_FLIP(i) : (i))
#define CS_MARKED(w,j) (w [j] < 0)
#define CS_MARK(w,j) { w [j] = CS_FLIP (w [j]) ; }

int cs_reach (cs *G, const cs *B, int k, int *xi, const int *pinv)
{
    int p, n, top, *Bp, *Bi, *Gp ;
    if (!CS_CSC (G) || !CS_CSC (B) || !xi) return (-1) ;    /* check inputs */
    n = G->n ; Bp = B->p ; Bi = B->i ; Gp = G->p ;
    top = n ;
    for (p = Bp [k] ; p < Bp [k+1] ; p++)
    {
        if (!CS_MARKED (Gp, Bi [p]))    /* start a dfs at unmarked node i */
        {
            top = cs_dfs (Bi [p], G, top, xi, xi+n, pinv) ;
        }          no nested again!
    }
    for (p = top ; p < n ; p++) CS_MARK (Gp, xi [p]) ;  /* restore G */
    return (top) ;
}

int cs_dfs (int j, cs *G, int top, int *xi, int *pstack, const int *pinv)
{
    int i, p, p2, done, jnew, head = 0, *Gp, *Gi ;
    if (!CS_CSC (G) || !xi || !pstack) return (-1) ;    /* check inputs */
    Gp = G->p ; Gi = G->i ;
    xi [0] = j ;                /* initialize the recursion stack */
    while (head >= 0)
    {
        j = xi [head] ;         /* get j from the top of the recursion stack */
        jnew = pinv ? (pinv [j]) : j ;
        if (!CS_MARKED (Gp, j))
        {
            CS_MARK (Gp, j) ;       /* mark node j as visited */
            pstack [head] = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew]) ;
        }
        done = 1 ;                  /* node j done if no unvisited neighbors */
        p2 = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew+1]) ;
        for (p = pstack [head] ; p < p2 ; p++)  /* examine all neighbors of j */
        {
            i = Gi [p] ;            /* consider neighbor node i */
            if (CS_MARKED (Gp, i)) continue ;   /* skip visited node i */
            pstack [head] = p ;     /* pause depth-first search of node j */
            xi [++head] = i ;       /* start dfs at node i */
            done = 0 ;              /* node j is not done */
            break ;                 /* break, to start dfs (i) */
        }
        if (done)               /* depth-first search at node j is done */
        {
            head-- ;            /* remove j from the recursion stack */
            xi [--top] = j ;    /* and place in the output stack */
        }
    }                       Just a way to break the nested structure!
    return (top) ;
}
```

The `cs_dfs` function starts by placing `j` in the recursion stack at `xi[0]`. Each

iteration of the `while` loop starts, or continues, the `j`th instance of `cs_dfs`. If `j` is on the recursion stack and it is not marked, then this is the first time it has been visited. In this case, the node is marked, and `pstack[head]` is set to point to the first outgoing edge of node `j`. If an unmarked node `i` is found, it is placed on the recursion stack, and the iteration for node `j` is paused. The next `while` loop iteration will then start the depth-first search for node `i`. When the depth-first search for node `j` eventually finishes, it is removed from the recursion stack and placed in the output stack.

The `cs_reach` function is nearly identical to `reachr`. It computes $\mathcal{X} = \mathrm{Reach}_G(\mathcal{B}_k)$, where $\mathcal{B}_k$ is the nonzero pattern of column `k` of B.

With `cs_reach` defined, solving $Lx = b$, where L, x, and b are all sparse, becomes a straightforward translation of the pseudocode. The `cs_spsolve` function computes the solution to $Lx = b_k$ (if `lo` is nonzero), where b_k is the kth column of B. When `lo` is nonzero, the function assumes `G` $= L$ is lower triangular with the diagonal entry as the first entry in each column. It takes an optimal $O(|b| + f)$ time. Solving an upper triangular system $Ux = b$ is almost identical to solving $Lx = b$. Its derivation is left as an exercise. With `lo` equal to zero, the `cs_spsolve` function assumes `G` $= U$ is upper triangular with the diagonal entry as the last entry in each column.

```
int cs_spsolve (cs *G, const cs *B, int k, int *xi, double *x, const int *pinv,
    int lo)
{
    int j, J, p, q, px, top, n, *Gp, *Gi, *Bp, *Bi ;
    double *Gx, *Bx ;
    if (!CS_CSC (G) || !CS_CSC (B) || !xi || !x) return (-1) ;
    Gp = G->p ; Gi = G->i ; Gx = G->x ; n = G->n ;
    Bp = B->p ; Bi = B->i ; Bx = B->x ;
    top = cs_reach (G, B, k, xi, pinv) ;        /* xi[top..n-1]=Reach(B(:,k)) */
    for (p = top ; p < n ; p++) x [xi [p]] = 0 ;    /* clear x */
    for (p = Bp [k] ; p < Bp [k+1] ; p++) x [Bi [p]] = Bx [p] ; /* scatter B */
    for (px = top ; px < n ; px++)
    {
        j = xi [px] ;  current col                  /* x(j) is nonzero */
        J = pinv ? (pinv [j]) : j ;                 /* j maps to col J of G */
        if (J < 0) continue ;                       /* column J is empty */
        x [j] /= Gx [lo ? (Gp [J]) : (Gp [J+1]-1)] ;/* x(j) /= G(j,j) */
        p = lo ? (Gp [J]+1) : (Gp [J]) ;            /* lo: L(j,j) 1st entry */
        q = lo ? (Gp [J+1]) : (Gp [J+1]-1) ;        /* up: U(j,j) last entry */
        for ( ; p < q ; p++)
        {
            x [Gi [p]] -= Gx [p] * x [j] ;          /* x(i) -= G(i,j) * x(j) */
        }
    }
    return (top) ;                                  /* return top of stack */
}
```

The function returns the nonzero pattern $\mathcal{X}$ in `xi[top]` through `xi[n-1]`, an array of size `2*n`. The first `n` entries of `xi` holds the output stack and the recursion stack for `j`. The second `n` entries holds the stack for `p` in `cs_dfs`. The numerical values are in the dense vector `x`, which need not be initialized on input. To solve $Lx = b$, a `NULL` pointer must be passed for `pinv`, and `lo` must be nonzero.

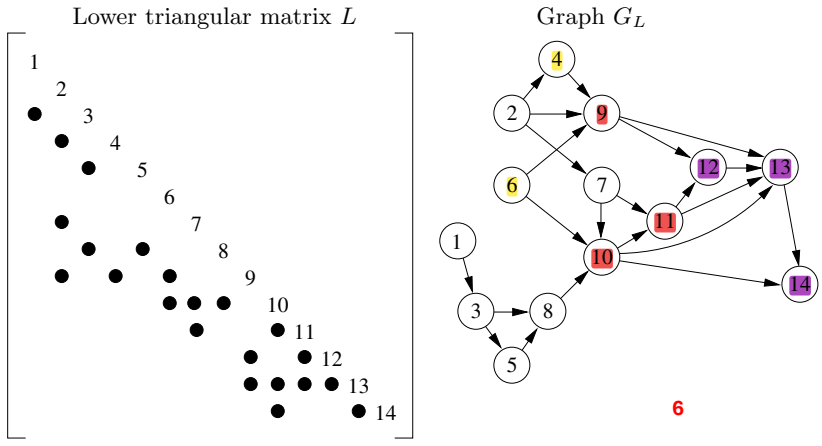

Figure 3.2. *Solving $Lx = b$ where L, x, and b are sparse*

An example is shown in Figure 3.2. Suppose $\mathcal{B} = \{4, 6\}$. The depth-first search can start at either node 4 or node 6, and the neighbors of any node can be searched in any order. If node 4 is searched first and the row indices in each column of L are sorted, Reach$(4) = \{4, 9, 12, 13, 14\}$ in order. This list of 5 nodes is placed on the stack `xi`, and node 6 is searched next; Reach$(6) = \{6, 9, 10, 11, 12, 13, 14\}$, but some of these nodes are already marked. The stack `xi` will contain the list $\{6, 10, 11, 4, 9, 12, 13, 14\}$ in topological order. The forward solve will access the columns of L in this order. The work done at columns 6, 10, and 11 is not affected by the work done at columns 4 and 9. **Potential for parallel!** **It is not unique**

MATLAB does not have an exact equivalent to `cs_reach` or `cs_spsolve`, except as used internally in `[L,U,P]=lu(A)`. The MATLAB expression `x=L\b` takes $O(n + |L|)$ time to compute a sparse `x` if `L` is sparse and `b` is a sparse vector.

3.3 Further reading

The sparse triangular solve forms the basis of the GPLU algorithm in MATLAB [105, 109]. All sparse matrix packages with direct methods include forward solvers and backsolvers. Not all provide transposed solvers such as `cs_ltsolve` or sparse solvers such as `cs_spsolve`. Cormen, Leiserson, and Rivest [23] present algorithms for the depth-first search and topological sort of a graph.

Exercises

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.

3.4. Repeat Problem 3.3 for a matrix `U` that is an upper triangular matrix whose rows have been permuted. The algorithm is almost identical to Problem 3.3.

3.5. Repeat Problem 3.3 for a matrix `L` that is a lower triangular matrix whose columns have been permuted. Assume the first entry in each column has the smallest row index. This problem is simpler than Problem 3.3. Two passes of `L` are still required, but the first takes only $O(n)$ time.

3.6. Repeat Problem 3.5 for a matrix `U` that is an upper triangular matrix whose columns have been permuted. The solution will be similar to Problem 3.5.

3.7. This problem is a generalization of Problems 3.3 through 3.6 and is the method used in MATLAB (due to Gilbert). Consider a matrix A that may be a permuted upper or lower triangular matrix with both rows and columns permuted by unknown permutations P and Q. Write an algorithm that determines if the matrix is in this form and, if so, solves $Ax = b$. A single integer `s` can represent a set of integers of arbitrary size that supports the following operations: `s=0` clears the set; `s=s^j` (the exclusive-or) removes `j` from the set if `j` is not in the set or adds it to the set otherwise; `j=s` gets the member of the set if the set has size one. Let `r[i]` be the count of nonzeros in row `i` of `A`. Let `z[i]` represent the `i`th set; it starts as the exclusive-or of all column indices of nonzeros in row `i`. Create a linked list of all row singletons (rows with only one entry). For `n` iterations where `A` has dimension `n`, select a row `i` from the list. If the list is empty, the matrix is not a permuted triangular matrix. Otherwise, the corresponding column is `j=z[i]`. Append `i` and `j` to the permutations `p` and `q`. Remove `j` from the sets `z[t]` for all `t` $\in \mathcal{A}_{*j}$ and decrement their row counts by one. Add any new row singletons to the linked list. If successful, follow this by a permuted triangular solve using the newly discovered permutations `p` and `q`. The permuted matrix `A(p,q)` need not be formed explicitly.

3.8. The `cs_lsolve` and `cs_usolve` functions assume that `b` contains no zero entries. The time can be reduced if it does. Note that the inner `for` loop in the two functions can be skipped if `x[j]` is zero. Add this test and compare the run times of the modified and original functions. The modified functions take $O(n + f)$ time; the original ones take $O(|L|)$ and $O(|U|)$ time.

3.9. Prove that the `cs_spsolve` function solves $Ux = b$ when `lo` is zero.