Sparse Matrix Methods
Chapter 4 lecture notes

Tim Davis

2011

Chapter 4: Cholesky factorization

One method: based on Lx=b

$$\begin{bmatrix} L_{11} & \\ l_{12}^T & l_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} \\ & l_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{bmatrix},$$

- L_{11} and A_{11} are $(n-1)$-by-$(n-1)$
- $L_{11}L_{11}^T = A_{11}$,
- $L_{11}l_{12} = a_{12}$,
- $l_{12}^T l_{12} + l_{22}^2 = a_{22}$.

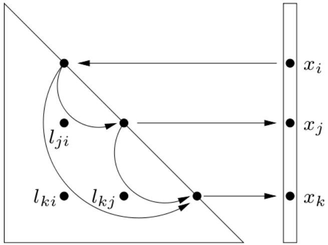
Cholesky factorization

- solve $L_{11}L_{11}^T = A_{11}$ for L_{11}
- solve $L_{11}l_{12} = a_{12}$ for l_{12}
- $l_{22} = \sqrt{a_{22} - l_{12}^T l_{12}}$

MATLAB prototype

```
function L = chol_up (A)
n = size (A) ;
L = zeros (n) ;
for k = 1:n
    L (k,1:k-1) = (L (1:k-1,1:k-1) \ A (1:k-1,k))' ;
    L (k,k) = sqrt (A (k,k) - L (k,1:k-1) * L (k,1:k-1)') ;
end
```

Pruning the directed graph



Thm. 4.2: $a_{ik} \neq 0$ implies $l_{ki} \neq 0$

Thm. 4.3: $l_{ji} \neq 0$ and $l_{ki} \neq 0$ implies $l_{kj} \neq 0$

Thus l_{ki} redundant for $\mathcal{X} = \mathrm{Reach}_L(i)$

Figure 4.1. *Pruning the directed graph G_L yields the elimination tree $\mathcal{T}$*

Elimination tree

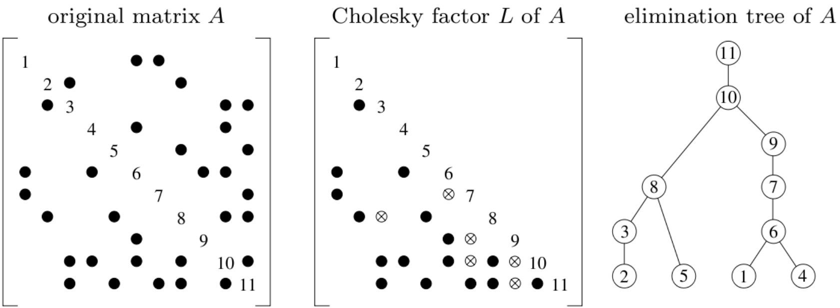


Figure 4.2. *Example matrix A, factor L, and elimination tree*

Elimination tree theorems

Theorem

For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $a_{ij} \neq 0 \Rightarrow l_{ij} \neq 0$. That is, if a_{ij} is nonzero, then l_{ij} will be nonzero as well.

Theorem (Parter)

For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $i < j < k \wedge l_{ji} \neq 0 \wedge l_{ki} \neq 0 \Rightarrow l_{kj} \neq 0$. That is, if both l_{ji} and l_{ki} are nonzero where $i < j < k$, then l_{kj} will be nonzero as well.

Elimination tree theorems

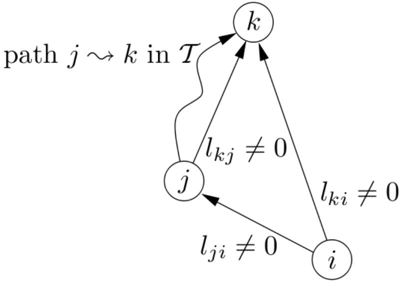


Figure 4.3. *Illustration of Theorem 4.4*

Theorem (Schreiber)

For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $l_{ki} \neq 0$ and $k > i$ imply that i is a descendant of k in the elimination tree $\mathcal{T}$; equivalently, $i \rightsquigarrow k$ is a path in $\mathcal{T}$.

Row subtree theorem

Theorem (Liu)

The nonzero pattern $\mathcal{L}_k$ of the kth row of L is given by

$$\mathcal{L}_k = Reach_{G_{k-1}}(\mathcal{A}_k) = Reach_{\mathcal{T}_{k-1}}(\mathcal{A}_k). \tag{1}$$

Row subtrees

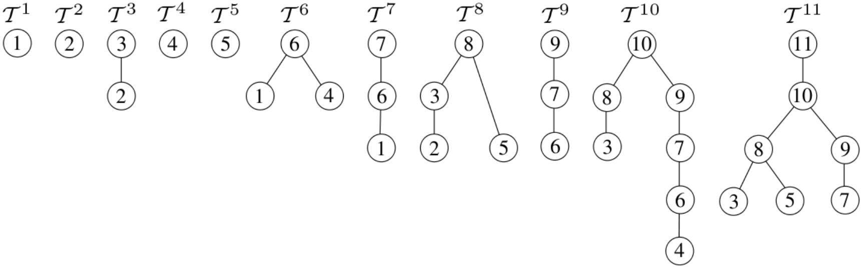


Figure 4.4. *Row subtrees of the example in Figure 4.2*

Row subtree theorems

Theorem (Liu)

Node j is a leaf of $\mathcal{T}^k$ if and only if both $a_{jk} \neq 0$ and $a_{ik} = 0$ for every descendant i of j in the elimination tree $\mathcal{T}$.

Corollary (Liu)

For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $a_{ki} \neq 0$ and $k > i$ imply that i is a descendant of k in the elimination tree $\mathcal{T}$; equivalently, $i \rightsquigarrow k$ is a path in $\mathcal{T}$.

```c
int *cs_etree (const cs *A, int ata)
{
    int i, k, p, m, n, inext, *Ap, *Ai, *w, *parent, *ancestor, *prev ;
    if (!CS_CSC (A)) return (NULL) ;        /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ;
    parent = cs_malloc (n, sizeof (int)) ;              /* allocate result */
    w = cs_malloc (n + (ata ? m : 0), sizeof (int)) ;   /* get workspace */
    if (!w || !parent) return (cs_idone (parent, NULL, w, 0)) ;
    ancestor = w ; prev = w + n ;
    if (ata) for (i = 0 ; i < m ; i++) prev [i] = -1 ;
    for (k = 0 ; k < n ; k++)
    {
        parent [k] = -1 ;                   /* node k has no parent yet */
        ancestor [k] = -1 ;                 /* nor does k have an ancestor */
        for (p = Ap [k] ; p < Ap [k+1] ; p++)
        {
            i = ata ? (prev [Ai [p]]) : (Ai [p]) ;
            for ( ; i != -1 && i < k ; i = inext)   /* traverse from i to k */
            {
                inext = ancestor [i] ;              /* inext = ancestor of i */
                ancestor [i] = k ;                  /* path compression */
                if (inext == -1) parent [i] = k ;   /* no anc., parent is k */
            }
            if (ata) prev [Ai [p]] = k ;
        }
    }
    return (cs_idone (parent, NULL, w, 1)) ;
}
```

```
int cs_ereach (const cs *A, int k, const int *parent, int *s, int *w)
{
    int i, p, n, len, top, *Ap, *Ai ;
    if (!CS_CSC (A) || !parent || !s || !w) return (-1) ;   /* check inputs */
    top = n = A->n ; Ap = A->p ; Ai = A->i ;
    CS_MARK (w, k) ;                   /* mark node k as visited */
    for (p = Ap [k] ; p < Ap [k+1] ; p++)
    {
        i = Ai [p] ;                   /* A(i,k) is nonzero */
        if (i > k) continue ;          /* only use upper triangular part of A */
        for (len = 0 ; !CS_MARKED (w,i) ; i = parent [i]) /* traverse up etree*/
        {
            s [len++] = i ;            /* L(k,i) is nonzero */
            CS_MARK (w, i) ;           /* mark i as visited */
        }
        while (len > 0) s [--top] = s [--len] ; /* push path onto stack */
    }
    for (p = top ; p < n ; p++) CS_MARK (w, s [p]) ;    /* unmark all nodes */
    CS_MARK (w, k) ;                   /* unmark node k */
    return (top) ;                     /* s [top..n-1] contains pattern of L(k,:)*/
}
```

Postordering a tree

Theorem (Liu)

The filled graphs of A and PAP^T are isomorphic, if P is a postordering of the elimination tree of A. Likewise, the elimination trees of A and PAP^T are isomorphic.

```
function postorder (T)
    k = 0
    for each root node j of T do
        dfstree (j)

function dfstree (j)
    for each child i of j do
        dfstree (i)
    post[k] = j
    k = k + 1
```

```c
int *cs_post (const int *parent, int n)
{
    int j, k = 0, *post, *w, *head, *next, *stack ;
    if (!parent) return (NULL) ;                            /* check inputs */
    post = cs_malloc (n, sizeof (int)) ;                    /* allocate result */
    w = cs_malloc (3*n, sizeof (int)) ;                     /* get workspace */
    if (!w || !post) return (cs_idone (post, NULL, w, 0)) ;
    head = w ; next = w + n ; stack = w + 2*n ;
    for (j = 0 ; j < n ; j++) head [j] = -1 ;               /* empty linked lists */
    for (j = n-1 ; j >= 0 ; j--)                /* traverse nodes in reverse order*/
    {
        if (parent [j] == -1) continue ;        /* j is a root */
        next [j] = head [parent [j]] ;          /* add j to list of its parent */
        head [parent [j]] = j ;
    }
    for (j = 0 ; j < n ; j++)
    {
        if (parent [j] != -1) continue ;        /* skip j if it is not a root */
        k = cs_tdfs (j, k, head, next, post, stack) ;
    }
    return (cs_idone (post, NULL, w, 1)) ;  /* success; free w, return post */
}
```

```
int cs_tdfs (int j, int k, int *head, const int *next, int *post, int *stack)
{
    int i, p, top = 0 ;
    if (!head || !next || !post || !stack) return (-1) ;    /* check inputs */
    stack [0] = j ;                         /* place j on the stack */
    while (top >= 0)                        /* while (stack is not empty) */
    {
        p = stack [top] ;                   /* p = top of stack */
        i = head [p] ;                      /* i = youngest child of p */
        if (i == -1)
        {
            top-- ;                         /* p has no unordered children left */
            post [k++] = p ;                /* node p is the kth postordered node */
        }
        else
        {
            head [p] = next [i] ;           /* remove i from children of p */
            stack [++top] = i ;             /* start dfs on child node i */
        }
    }
    return (k) ;
}
```

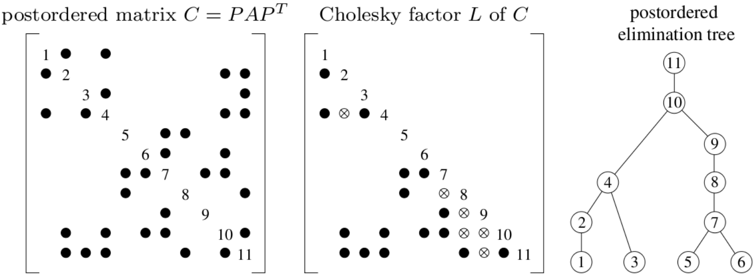


Figure 4.5. *After elimination tree postordering*

Row counts

Requires:

- least common ancestor
- path decomposition
- first descendant
- level
- skeleton matrix

First descendant

```
void firstdesc (int n, int *parent, int *post, int *first, int *level)
{
    int len, i, k, r, s ;
    for (i = 0 ; i < n ; i++) first [i] = -1 ;
    for (k = 0 ; k < n ; k++)
    {
        i = post [k] ;       /* node i of etree is kth postordered node */
        len = 0 ;            /* traverse from i towards the root */
        for (r = i ; r != -1 && first [r] == -1 ; r = parent [r], len++)
            first [r] = k ;
        len += (r == -1) ? (-1) : level [r] ;    /* root node or end of path */
        for (s = i ; s != r ; s = parent [s]) level [s] = len-- ;
    }
}
```

First descendant in a postordered tree

Case 1: t not a descendant of j Case 2: t a descendant of j

Figure 4.6. *Descendants in a postordered tree*

Skeleton matrix

function *skeleton*

```
maxfirst[0...n-1] = -1
for j = 0 to n-1 do
    for each i > j for which a_ij ≠ 0
        if first[j] > maxfirst[i]
            node j is a leaf in the ith subtree
            maxfirst[i] = first[j]
```

Skeleton matrix

Lemma

Let $f_j \leq j$ denote the first descendant of j in a postordered tree. The descendants of j are all nodes $f_j, f_j + 1, \ldots, j - 1, j$.

Theorem

Consider two nodes $t < j$ in a postordered tree. Then either (1) $f_t \leq t < f_j \leq j$ and t is not a descendant of j, or (2) $f_j \leq f_t \leq t < j$ and t is a descendant of j.

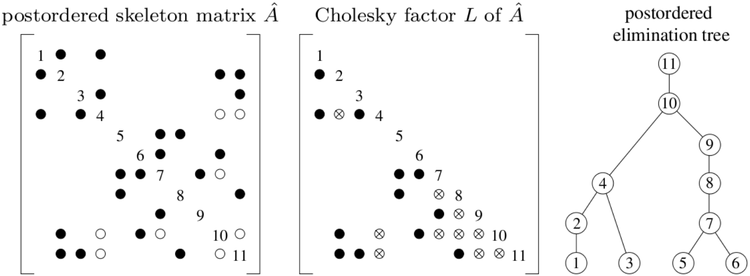
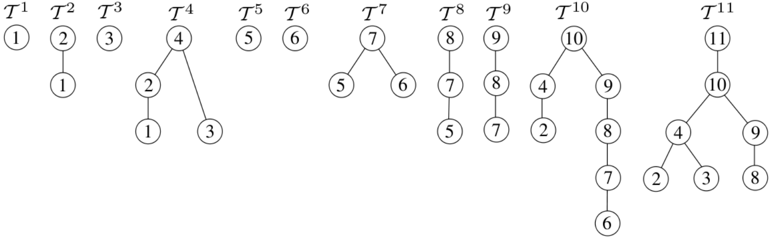


Figure 4.7. *Postordered skeleton matrix, its factor, and its elimination tree*



Corollary

Consider a node j in a postordered tree, and any set of nodes $\mathcal{S}$ where all nodes $s \in \mathcal{S}$ are numbered less than j. Let t be the node in $\mathcal{S}$ with the largest first descendant f_t. Node j has a descendant in $\mathcal{S}$ if and only if $f_t \geq f_j$.

Theorem

Assume that the elimination tree $\mathcal{T}$ is postordered. The least common ancestor of two nodes a and b where $a < b$ can be found by traversing the path from a towards the root. The first node $q \geq b$ found along this path is the least common ancestor of a and b.

```
int *rowcnt (cs *A, int *parent, int *post) /* return rowcount [0..n-1] */
{
    int i, j, k, len, s, p, jprev, q, n, sparent, jleaf, *Ap, *Ai, *maxfirst,
        *ancestor, *prevleaf, *w, *first, *level, *rowcount ;
    n = A->n ; Ap = A->p ; Ai = A->i ;                    /* get A */
    w = cs_malloc (5*n, sizeof (int)) ;                   /* get workspace */
    ancestor = w ; maxfirst = w+n ; prevleaf = w+2*n ; first = w+3*n ;
    level = w+4*n ;
    rowcount = cs_malloc (n, sizeof (int)) ;    /* allocate result */
    firstdesc (n, parent, post, first, level) ; /* find first and level */
    for (i = 0 ; i < n ; i++)
    {
        rowcount [i] = 1 ;      /* count the diagonal of L */
        prevleaf [i] = -1 ;     /* no previous leaf of the ith row subtree */
        maxfirst [i] = -1 ;     /* max first[j] for node j in ith subtree */
        ancestor [i] = i ;      /* every node is in its own set, by itself */
    }
    for (k = 0 ; k < n ; k++)
    {
        j = post [k] ;          /* j is the kth node in the postordered etree */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            q = cs_leaf (i, j, first, maxfirst, prevleaf, ancestor, &jleaf) ;
            if (jleaf) rowcount [i] += (level [j] - level [q]) ;
        }
        if (parent [j] != -1) ancestor [j] = parent [j] ;
    }
    cs_free (w) ;
    return (rowcount) ;
```

```c
int cs_leaf (int i, int j, const int *first, int *maxfirst, int *prevleaf,
    int *ancestor, int *jleaf)
{
    int q, s, sparent, jprev ;
    if (!first || !maxfirst || !prevleaf || !ancestor || !jleaf) return (-1) ;
    *jleaf = 0 ;
    if (i <= j || first [j] <= maxfirst [i]) return (-1) ;  /* j not a leaf */
    maxfirst [i] = first [j] ;      /* update max first[j] seen so far */
    jprev = prevleaf [i] ;          /* jprev = previous leaf of ith subtree */
    prevleaf [i] = j ;
    *jleaf = (jprev == -1) ? 1: 2 ; /* j is first or subsequent leaf */
    if (*jleaf == 1) return (i) ;   /* if 1st leaf, q = root of ith subtree */
    for (q = jprev ; q != ancestor [q] ; q = ancestor [q]) ;
    for (s = jprev ; s != q ; s = sparent)
    {
        sparent = ancestor [s] ;    /* path compression */
        ancestor [s] = q ;
    }
    return (q) ;                    /* q = least common ancester (jprev,j) */
}
```

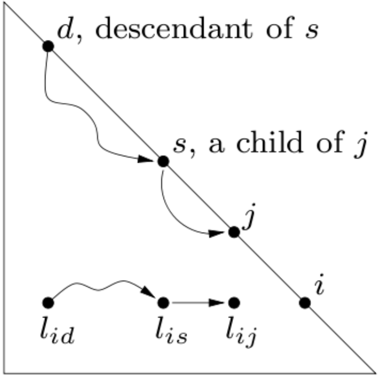
Column counts

Theorem (George and Liu)

If $\mathcal{L}_j$ denotes the nonzero pattern of the jth column of L, and $\mathcal{A}_j$ denotes the nonzero pattern of the strictly lower triangular part of the jth column of A, then

$$\mathcal{L}_j = \mathcal{A}_j \cup \{j\} \cup \left(\bigcup_{j=parent(s)} \mathcal{L}_s \setminus \{s\} \right). \tag{2}$$

Nonzero pattern of column j is union of its children



Column counts

$$c_j = |\widehat{\mathcal{A}}_j| + \left| \bigcup_{j=\text{parent}(s)} \mathcal{L}_s \setminus \{s\} \right| = |\widehat{\mathcal{A}}_j| - e_j + \left| \bigcup_{j=\text{parent}(s)} \mathcal{L}_s \right|$$

$$c_j = |\widehat{\mathcal{A}}_j| - e_j - o_j + \sum_{j=\text{parent}(s)} c_s .$$

1. If $j \notin \mathcal{T}^i$, then $i \notin \mathcal{L}_j$ and row i does not contribute to the overlap o_j.
2. If j is a leaf of $\mathcal{T}^i$, then by definition a_{ij} is in the skeleton matrix. Row i does not contribute to the overlap o_j, because it appears in none of the children of j. Row i contributes exactly one to c_j, since $i \in \widehat{\mathcal{A}}_j$.
3. If j is not a leaf of $\mathcal{T}^i$, let d_{ij} denote the number of children of j that are in $\mathcal{T}^i$. These children are a subset of the children of j in the elimination tree $\mathcal{T}$. Row i is present in the nonzero patterns of each of these d_{ij} children. Thus, row i contributes $d_{ij} - 1$ to the overlap o_j. If j has just one child, row i appears only in that one child and there is no overlap.

Combining the correction terms

- If j is a leaf of the elimination tree, $c_j = \Delta_j = |\widehat{\mathcal{A}}_j| + 1$.
- Otherwise, $\Delta_j = |\widehat{\mathcal{A}}_j| - e_j - o_j$
- then $c_j = \Delta_j + \sum_{j=\text{parent}(s)} c_s$,
- example for column 4, $\Delta_4 = 0 - 2 - 2$ and $c_4 = -4 + c_2 + c_3 = -4 + 4 + 3 = 3$.

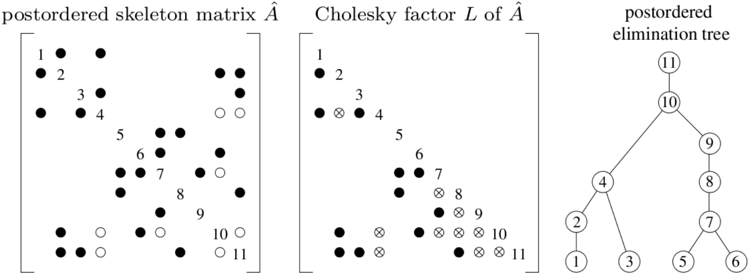
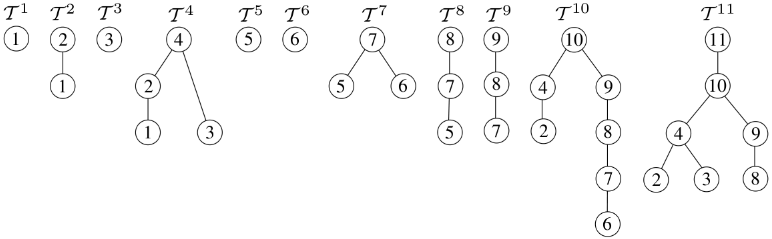


Figure 4.7. *Postordered skeleton matrix, its factor, and its elimination tree*



Column count algorithm, part 1 of 3

```
 #define HEAD(k,j) (ata ? head [k] : j)
 #define NEXT(J)   (ata ? next [J] : -1)
static void init_ata (cs *AT, const int *post, int *w, int **head, int **next)
{
    int i, k, p, m = AT->n, n = AT->m, *ATp = AT->p, *ATi = AT->i ;
    *head = w+4*n, *next = w+5*n+1 ;
    for (k = 0 ; k < n ; k++) w [post [k]] = k ;     /* invert post */
    for (i = 0 ; i < m ; i++)
    {
        for (k = n, p = ATp[i] ; p < ATp[i+1] ; p++) k = CS_MIN (k, w [ATi[p]]);
        (*next) [i] = (*head) [k] ;     /* place row i in linked list k */
        (*head) [k] = i ;
    }
}
```

Column count algorithm, part 2 of 3

```
int *cs_counts (const cs *A, const int *parent, const int *post, int ata)
{
    int i, j, k, n, m, J, s, p, q, jleaf, *ATp, *ATi, *maxfirst, *prevleaf,
        *ancestor, *head = NULL, *next = NULL, *colcount, *w, *first, *delta ;
    cs *AT ;
    if (!CS_CSC (A) || !parent || !post) return (NULL) ;    /* check inputs */
    m = A->m ; n = A->n ;
    s = 4*n + (ata ? (n+m+1) : 0) ;
    delta = colcount = cs_malloc (n, sizeof (int)) ;    /* allocate result */
    w = cs_malloc (s, sizeof (int)) ;                   /* get workspace */
    AT = cs_transpose (A, 0) ;                          /* AT = A' */
    if (!AT || !colcount || !w) return (cs_idone (colcount, AT, w, 0)) ;
    ancestor = w ; maxfirst = w+n ; prevleaf = w+2*n ; first = w+3*n ;
    for (k = 0 ; k < s ; k++) w [k] = -1 ;      /* clear workspace w [0..s-1] */
    for (k = 0 ; k < n ; k++)                   /* find first [j] */
    {
        j = post [k] ;
        delta [j] = (first [j] == -1) ? 1 : 0 ;  /* delta[j]=1 if j is a leaf */
        for ( ; j != -1 && first [j] == -1 ; j = parent [j]) first [j] = k ;
    }
    ATp = AT->p ; ATi = AT->i ;
    if (ata) init_ata (AT, post, w, &head, &next) ;
    for (i = 0 ; i < n ; i++) ancestor [i] = i ; /* each node in its own set */
```

Column count algorithm, part 3 of 3

```
    for (k = 0 ; k < n ; k++)
    {
        j = post [k] ;          /* j is the kth node in postordered etree */
        if (parent [j] != -1) delta [parent [j]]-- ;    /* j is not a root */
        for (J = HEAD (k,j) ; J != -1 ; J = NEXT (J))   /* J=j for LL'=A case */
        {
            for (p = ATp [J] ; p < ATp [J+1] ; p++)
            {
                i = ATi [p] ;
                q = cs_leaf (i, j, first, maxfirst, prevleaf, ancestor, &jleaf);
                if (jleaf >= 1) delta [j]++ ;   /* A(i,j) is in skeleton */
                if (jleaf == 2) delta [q]-- ;   /* account for overlap in q */
            }
        }
        if (parent [j] != -1) ancestor [j] = parent [j] ;
    }
    for (j = 0 ; j < n ; j++)           /* sum up delta's of each child */
    {
        if (parent [j] != -1) colcount [parent [j]] += colcount [j] ;
    }
    return (cs_idone (colcount, AT, w, 1)) ;    /* success: free workspace */
}
```

Putting it all together: the symbolic analysis

1. fill-reducing ordering, P
2. $C = PAP^T$
3. find etree of C
4. postorder the etree
5. find column counts of L
6. find column pointers of L
7. (nonzero pattern of L not required)

Symbolic analysis

```
typedef struct cs_symbolic  /* symbolic Cholesky, LU, or QR analysis */
{
    int *pinv ;     /* inverse row perm. for QR, fill red. perm for Chol */
    int *q ;        /* fill-reducing column permutation for LU and QR */
    int *parent ;   /* elimination tree for Cholesky and QR */
    int *cp ;       /* column pointers for Cholesky, row counts for QR */
    int *leftmost ; /* leftmost[i] = min(find(A(i,:))), for QR */
    int m2 ;        /* # of rows for QR, after adding fictitious rows */
    double lnz ;    /* # entries in L for LU or Cholesky; in V for QR */
    double unz ;    /* # entries in U for LU; in R for QR */
} css ;
```

Symbolic analysis

```
css *cs_schol (int order, const cs *A)
{
    int n, *c, *post, *P ;
    cs *C ;
    css *S ;
    if (!CS_CSC (A)) return (NULL) ;         /* check inputs */
    n = A->n ;
    S = cs_calloc (1, sizeof (css)) ;        /* allocate result S */
    if (!S) return (NULL) ;                  /* out of memory */
    P = cs_amd (order, A) ;                  /* P = amd(A+A'), or natural */
    S->pinv = cs_pinv (P, n) ;               /* find inverse permutation */
    cs_free (P) ;
    if (order && !S->pinv) return (cs_sfree (S)) ;
    C = cs_symperm (A, S->pinv, 0) ;         /* C = spones(triu(A(P,P))) */
    S->parent = cs_etree (C, 0) ;            /* find etree of C */
    post = cs_post (S->parent, n) ;          /* postorder the etree */
    c = cs_counts (C, S->parent, post, 0) ; /* find column counts of chol(C) */
    cs_free (post) ;
    cs_spfree (C) ;
    S->cp = cs_malloc (n+1, sizeof (int)) ; /* allocate result S->cp */
    S->unz = S->lnz = cs_cumsum (S->cp, c, n) ; /* find column pointers for L */
    cs_free (c) ;
    return ((S->lnz >= 0) ? S : cs_sfree (S)) ;
}
```

Numerical factorization: Up-looking Cholesky

$$\left[\begin{array}{cc} L_{11} & \\ l_{12}^T & l_{22} \end{array}\right]\left[\begin{array}{cc} L_{11}^T & l_{12} \\ & l_{22} \end{array}\right]=\left[\begin{array}{cc} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{array}\right],$$

- L_{11} and A_{11} are $(n-1)$-by-$(n-1)$
- $L_{11}L_{11}^T = A_{11}$,
- $L_{11}l_{12} = a_{12}$,
- $l_{12}^T l_{12} + l_{22}^2 = a_{22}$.

Up-looking Cholesky

- solve $L_{11}L_{11}^T = A_{11}$ for L_{11}
- solve $L_{11}l_{12} = a_{12}$ for l_{12}
- $l_{22} = \sqrt{a_{22} - l_{12}^T l_{12}}$

```
csn *cs_chol (const cs *A, const css *S)
{
    double d, lki, *Lx, *x, *Cx ;
    int top, i, p, k, n, *Li, *Lp, *cp, *pinv, *s, *c, *parent, *Cp, *Ci ;
    cs *L, *C, *E ;
    csn *N ;
    if (!CS_CSC (A) || !S || !S->cp || !S->parent) return (NULL) ;
    n = A->n ;
    N = cs_calloc (1, sizeof (csn)) ;       /* allocate result */
    c = cs_malloc (2*n, sizeof (int)) ;     /* get int workspace */
    x = cs_malloc (n, sizeof (double)) ;    /* get double workspace */
    cp = S->cp ; pinv = S->pinv ; parent = S->parent ;
    C = pinv ? cs_symperm (A, pinv, 1) : ((cs *) A) ;
    E = pinv ? C : NULL ;           /* E is alias for A, or a copy E=A(p,p) */
    if (!N || !c || !x || !C) return (cs_ndone (N, E, c, x, 0)) ;
    s = c + n ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    N->L = L = cs_spalloc (n, n, cp [n], 1, 0) ;    /* allocate result */
    if (!L) return (cs_ndone (N, E, c, x, 0)) ;
    Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (k = 0 ; k < n ; k++) Lp [k] = c [k] = cp [k] ;
```

```
for (k = 0 ; k < n ; k++)      /* compute L(:,k) for L*L' = C */
{
    /* --- Nonzero pattern of L(k,:) ---------------------------------- */
    top = cs_ereach (C, k, parent, s, c) ;     /* find pattern of L(k,:) */
    x [k] = 0 ;                                /* x (0:k) is now zero */
    for (p = Cp [k] ; p < Cp [k+1] ; p++)      /* x = full(triu(C(:,k))) */
    {
        if (Ci [p] <= k) x [Ci [p]] = Cx [p] ;
    }
    d = x [k] ;                        /* d = C(k,k) */
    x [k] = 0 ;                        /* clear x for k+1st iteration */
    /* --- Triangular solve --------------------------------------------- */
    for ( ; top < n ; top++)    /* solve L(0:k-1,0:k-1) * x = C(:,k) */
    {
        i = s [top] ;                  /* s [top..n-1] is pattern of L(k,:) */
        lki = x [i] / Lx [Lp [i]] ; /* L(k,i) = x (i) / L(i,i) */
        x [i] = 0 ;                    /* clear x for k+1st iteration */
        for (p = Lp [i] + 1 ; p < c [i] ; p++)
        {
            x [Li [p]] -= Lx [p] * lki ;
        }
        d -= lki * lki ;               /* d = d - L(k,i)*L(k,i) */
        p = c [i]++ ;
        Li [p] = k ;                   /* store L(k,i) in column i */
        Lx [p] = lki ;
    }
```

```
        /* --- Compute L(k,k) ----------------------------------------- */
        if (d <= 0) return (cs_ndone (N, E, c, x, 0)) ; /* not pos def */
        p = c [k]++ ;
        Li [p] = k ;                /* store L(k,k) = sqrt (d) in column k */
        Lx [p] = sqrt (d) ;
    }
    Lp [n] = cp [n] ;               /* finalize L */
    return (cs_ndone (N, E, c, x, 1)) ; /* success: free E,s,x; return N */
}
```

```
int cs_ereach (const cs *A, int k, const int *parent, int *s, int *w)
{
    int i, p, n, len, top, *Ap, *Ai ;
    if (!CS_CSC (A) || !parent || !s || !w) return (-1) ;   /* check inputs */
    top = n = A->n ; Ap = A->p ; Ai = A->i ;
    CS_MARK (w, k) ;                        /* mark node k as visited */
    for (p = Ap [k] ; p < Ap [k+1] ; p++)
    {
        i = Ai [p] ;                        /* A(i,k) is nonzero */
        if (i > k) continue ;               /* only use upper triangular part of A */
        for (len = 0 ; !CS_MARKED (w,i) ; i = parent [i]) /* traverse up etree*/
        {
            s [len++] = i ;                 /* L(k,i) is nonzero */
            CS_MARK (w, i) ;                /* mark i as visited */
        }
        while (len > 0) s [--top] = s [--len] ; /* push path onto stack */
    }
    for (p = top ; p < n ; p++) CS_MARK (w, s [p]) ;    /* unmark all nodes */
    CS_MARK (w, k) ;                        /* unmark node k */
    return (top) ;                          /* s [top..n-1] contains pattern of L(k,:)*/
}
```

Left-looking Cholesky

```
function L = chol_left (A)
n = size (A,1) ;
L = zeros (n) ;
for k = 1:n
    L (k,k) = sqrt (A (k,k) - L (k,1:k-1) * L (k,1:k-1)') ;
    L (k+1:n,k) = (A (k+1:n,k) - L (k+1:n,1:k-1) * L (k,1:k-1)') / L (k,k) ;
end
```

$$\begin{bmatrix} L_{11} & & \\ l_{12}^T & l_{22} & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} & L_{31}^T \\ & l_{22} & l_{32}^T \\ & & L_{33}^T \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{31}^T \\ a_{12}^T & a_{22} & a_{32}^T \\ A_{31} & a_{32} & A_{33} \end{bmatrix}$$

- $l_{22} = \sqrt{a_{22} - l_{12}^T l_{12}}$
- $l_{32} = (a_{32} - L_{31} l_{12}) / l_{22}$

Left-looking Cholesky

```
function L = chol_left (A)
n = size (A,1) ;
L = sparse (n,n) ;
a = sparse (n,1) ;
for k = 1:n
    a (k:n) = A (k:n,k) ;
    for j = find (L (k,:))
        a (k:n) = a (k:n) - L (k:n,j) * L (k,j) ;
    end
    L (k,k) = sqrt (a (k)) ;
    L (k+1:n,k) = a (k+1:n) / L (k,k) ;
end
```

Supernodal Cholesky

```
function L = chol_super (A,s)
n = size (A) ;
L = zeros (n) ;
ss = cumsum ([1 s]) ;
for j = 1:length (s)
    k1 = ss (j) ;
    k2 = ss (j+1) ;
    k = k1:(k2-1) ;
    L (k,k) = chol (A (k,k) - L (k,1:k1-1) * L (k,1:k1-1)')' ;
    L (k2:n,k) = (A (k2:n,k) - L (k2:n,1:k1-1) * L (k,1:k1-1)') / L (k,k)' ;
end
```

Supernodal Cholesky

1. A symmetric update, `A(k,k)-L(k,1:k1-1)*L(k,1:k1-1)'`. In the sparse case, `A(k,k)` is a dense matrix. `L(k,1:k1-1)` represents the rows in a subset of the descendants of the `j`th supernode. The update from each descendant can be done with a single dense matrix multiplication.
2. A dense Cholesky factorization, `chol`.
3. A sparse matrix product, `A(k2:n,k)-L(k2:n,1:k1-1)*L(k,1:k1-1)'`, where the two `L` terms come from the descendants of the `j`th supernode.
4. A dense triangular solve `(...)/L(k,k)'` using the `k`th diagonal block of `L`.