ZASTOSOWANIA MATEMATYKI
APPLICATIONES MATHEMATICAE
XVI, 2 (1978)

ALGORITHMS 62-64

JADWIGA DZIKIEWICZ and M. M. SYSŁO (Wrocław)

GRAPH-THEORETIC ALGORITHMS FOR SPARSE MATRIX TRANSFORMATIONS

This paper presents three algorithms for transforming a given sparse matrix A into a block diagonal form and into a block upper triangular form by using only row and column permutations. These reduced forms of a matrix allow us to partition some numerical problems into those of smaller sizes. Graph-theoretic results are used in the presented algorithms.

1. THE PROBLEM AND ITS APPLICATION

Let A be a square matrix whose entries are in some field. Without loss of generality we may assume the non-zero elements of A to be equal to 1. Let P, Q, and R denote the permutation matrices. A matrix A is *decomposable* if there exist permutation matrices P and Q such that

$$PAQ = \begin{bmatrix} A_{11} & & & \\ & A_{22} & & 0 \\ 0 & & \ddots & \\ & & & A_{kk} \end{bmatrix}, \tag{1}$$

where A_{ii} $(i = 1, 2, \ldots, k)$ are square submatrices, otherwise A is *indecomposable*. Similarly, a matrix A is *reducible* if there exist permutation matrices P and Q such that

$$PAQ = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1k} \\ & A_{22} & \ldots & A_{2k} \\ 0 & & \ddots & \vdots \\ & & & A_{kk} \end{bmatrix}, \tag{2}$$

otherwise A is *irreducible*.

There are several numerical problems related to matrices which can be simplified if a matrix is decomposable or reducible. For instance:

(i) Computation of the determinant $\det(A)$ of a matrix A,

$$\det(A) = \det(PAQ) = \prod_{i=1}^{k} \det(A_{ii}).$$

(ii) Finding the inverse A^{-1} of a non-singular matrix A,

$$A^{-1} = Q(PAQ)^{-1}P.$$

A is non-singular if and only if every submatrix A_{ii} ($i = 1, 2, \ldots, k$) is non-singular. Thus the inverse of a given matrix can be expressed in terms of the inverses of submatrices A_{ii} ($i = 1, 2, \ldots, k$). If a matrix A is decomposable, then

$$(PAQ)^{-1} = \begin{bmatrix} A_{11}^{-1} & & & \\ & A_{22}^{-1} & & 0 \\ & 0 & \ddots & \\ & & & A_{kk}^{-1} \end{bmatrix},$$

and if A is reducible, then A^{-1} can be computed by iterating the following formula for $k = 2$:

$$(PAQ)^{-1} = \begin{bmatrix} A_{11}^{-1} & -A_{11}^{-1}A_{12}A_{22}^{-1} \\ 0 & A_{22}^{-1} \end{bmatrix}.$$

(iii) Solving the system of linear equations $Ax = b$.

The system of linear equations with a decomposable matrix A can be split into independent subsystems. Firstly we solve the system $(PAQ)y = Pb$ and then $x = Qy$.

(iv) Finding eigenvalues of a matrix A.

If a matrix A is decomposable or reducible by using P and $Q = P^{-1}$, then

$$\det(A - \lambda I) = \det(PAP^{-1} - \lambda I) = \prod_{i=1}^{k} \det(A_{ii} - \lambda I),$$

i.e., the set of eigenvalues of A is the union of the sets of eigenvalues of submatrices A_{ii} ($i = 1, 2, \ldots, k$). Let us write

$$PAP^{-1} = \begin{bmatrix} A_{11} & & & \\ & A_{22} & & 0 \\ & 0 & \ddots & \\ & & & A_{kk} \end{bmatrix}, \tag{3}$$

$$PAP^{-1} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1k} \\ & A_{22} & \cdots & A_{2k} \\ & 0 & \ddots & \vdots \\ & & & A_{kk} \end{bmatrix}. \tag{4}$$

The paper presents three procedures for finding transformations of forms (4), (2), and (1) (transformation (3) is a special form of (4)).

2. PRELIMINARY DEFINITIONS

Let $D = \langle X, U\rangle$ be a *directed graph* (*digraph*), where X is the set of vertices and U is the set of arcs, i.e., the set of ordered pairs of vertices.

Let $A = (a_{ij})$ $(i, j = 1, 2, \ldots, n = |X|)$ denote the *adjacency matrix* of D with elements a_{ij} defined by

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in U, \\ 0 & \text{otherwise}. \end{cases}$$

The *transitive closure* $\check{D} = \langle X, \check{U}\rangle$ of D is a digraph such that $(i, j) \in \check{U}$ if and only if there is a path in D going from the vertex i to the vertex j. Let $\check{A} = (\check{a}_{ij})$ denote the adjacency matrix of $\check{D}$.

A digraph D is *strongly connected* if any two vertices of D are mutually reachable by directed paths. A *strongly connected component* of D is a maximal strongly connected subdigraph.

The *condensation* $D^* = \langle X^*, U^*\rangle$ of D is that digraph whose vertices correspond to the strongly connected components of D and whose arcs are induced by those of D.

A digraph $D = \langle X, U\rangle$ is *bipartite* if there exists a partition $S \cup T$ of X such that $S \cap T = \emptyset$ and if $(s, t) \in U$, then either $s \in S$ and $t \in T$ or $t \in S$ and $s \in T$.

3. METHOD USED

3.1. Procedure *matrixpermp***.** Procedure *matrixpermp* finds the permutation matrix P which reduces a given matrix A to the block triangular form (4).

Regard A as the adjacency matrix of a digraph D. The set of labelled vertices of D consists of n elements such that the i-th vertex of D corresponds to the i-th row and to the i-th column of A.

An algorithm for finding whether A is reducible is based on the following statements (see [6], [7], and [12] for details).

LEMMA 1. *A matrix A is irreducible to form* (4) *if and only if the digraph D associated with A is strongly connected.*

LEMMA 2. *Each vertex of D is in exactly one strongly connected component of D.*

LEMMA 3. *A matrix A is decomposable to form* (4) *if and only if the condensation D^* of D has at least two vertices and no arc.*

The algorithm can be described as follows.

Step 1. Find the strongly connected components of D. The depth-first search technique has been used in this step (see [4] and [14]).

```
procedure matrixpermp(n,p,nrp,f,np);
 value n;
 integer n;
 Boolean f;
 integer array p,nrp,np;
 comment (1) The procedure finds the permutation
  matrix P which reduces a given matrix A to the
  block triangular form (4);
 begin
  integer i,j,l,m,s;
  integer array c[1:n];
  procedure strongconnect(l,c);
   integer l;
   integer array c;
   comment (2) The procedure strongconnect is a re-
    alization of Tarjan's method [14] for finding
    the strongly connected components of the digraph
    D associated with A.
     Data for procedure strongconnect: n,p[1:m],nrp
    [0:n] are the global variables of procedure ma-
    trixpermp.
     Results of procedure strongconnect:
         l - the number of strongly connected compo-
             nents of D,
    c[1:n] - array which contains vertex numbers of
             consecutive strongly connected compo -
             nents of D. The number of the first ver-
             tex of each component is negative;
   begin
    integer i,j,m,s;
```

```
integer array lab,num,stos[1:n];
procedure dfs(v);
 value v;
 integer v;
 begin
  integer j,k,k1,k2,w,w1;
  lab[v]:=num[v]:=i:=i+1;
  s:=s+1;
  stos[s]:=v;
  k:=nrp[v];
  for w1:=nrp[v-1]+1 step 1 until k do
   begin
    w:=p[w1];
    k2:=num[w];
    if k2=0
     then
     begin
      dfs(w);
      k1:=lab[w];
      if k1<lab[v]
       then lab[v]:=k1
     end k2=0
     else
      if k2<num[v]
       then
       begin
        for j:=1 step 1 until s do
         if stos[j]=w
          then
          begin
```

```
                              if k2<lab[v]
                                then lab[v]:=k2;
                              go to Y
                            end j;
Y:                      end k2<num[v]
              end w1;
            k1:=num[v];
            if lab[v]=k1
              then
              begin
                l:=l+1;
                for j:=s step -1 until 1 do
                  begin
                    k2:=stos[j];
                    if num[k2]≥k1
                      then
                      begin
                        m:=m+1;
                        c[m]:=k2
                      end num[k2]≥k1
                      else go to L
                  end j;
L:              c[m]:=-c[m];
                s:=j
              end lab[v]=k1
          end dfs;
        for i:=1 step 1 until n do
          num[i]:=0;
        i:=l:=m:=s:=0;
        for j:=1 step 1 until n do
```

```
      if num[j]=0
        then dfs(j)
  and strongconnect;
comment (3) Step 1. Finding strongly connected
  components of D;
f:=true;
strongconnect(l,c);
if l=1
  then
  begin
    f:=false;
    go to FIN
  and l=1
  else
  begin
    integer k,h,h1,r1,v,w,w1,z;
    integer array a,b[1:l],nrc[0:l];
    Boolean array now[1:l,1:l];
    comment (4) Step 2. Computation of the adja-
      cency matrix now[1:l,1:l] of the condensation
      D* of D;
    for i:=1 step 1 until l do
      for j:=1 step 1 until l do
        now[i,j]:=false;
    nrc[0]:=j:=0;
    for i:=1 step 1 until n do
      if c[i]<0
        then
        begin
          j:=j+1;
```

```
   nrc[j]:=i;
   c[i]:=-c[i]
  end i;
for i:=1 step 1 until l do
 begin
  r1:=nrc[i];
  s:=nrc[i-1]+1;
  m:=i-1;
  for j:=1 step 1 until m,i+1 step 1 until l do
   begin
    h1:=nrc[j];
    k:=nrc[j-1]+1;
    for z:=s step 1 until r1 do
     begin
      w1:=nrp[c[z]];
      for w:=nrp[c[z]-1]+1 step 1 until w1 do
       begin
        v:=p[w];
        for h:=k step 1 until h1 do
         if v=c[h]
          then
          begin
           now[i,j]:=true;
           go to H
          end h
       end  w
     end z;
H:     end j
 end i;
comment (5) Step 3. Computation of the linear
```

```
     ordering a[1:l] of vertices of D* ;
    for i:=1 step 1 until l do
     begin
      s:=0;
      for j:=1 step 1 until l do
       if now[j,i]
        then s:=s+1;
      b[i]:=s;
      a[i]:=i
     end i;
    for k:=1 step 1 until l do
     begin
      for h:=k step 1 until l do
       begin
        j:=a[h];
        if b[j]=0
         then
         begin
          a[h]:=a[k];
          a[k]:=j;
          for i:=k+1 step 1 until l do
           begin
            s:=a[i];
            if now[j,s]
             then b[s]:=b[s]-1
           end i;
          go to K
         end b[j]=0
       end h;
K:   end k;
```

```
    comment (6) Step 4. Computation of the permu-
     tation np[1:n] from the linear ordering a[1:l]
     and from array c[1:n] containing vertex num-
     bers of consecutive strongly connected compo-
     nents of D;
    m:=0;
    for j:=1 step 1 until l do
     begin
      k:=nrc[a[j]-1]+1;
      h:=nrc[a[j]];
      for i:=k step 1 until h do
       begin
        m:=m+1;
        np[m]:=c[i]
       end i
     end j
   end;
FIN:
  end matrixpermp
```

Step 2. Find the condensation D^* of D.

Step 3. Find the linear ordering of vertices of D^*. The method of Marimont (see [9] and [10]) has been used in this step.

Step 4. Relabel the vertices of the consecutive strongly connected components of D using the ordering found in Step 3.

Step 1 and Steps 2-4 of procedure *matrixpermp* need $O(m)$ and $O(l^2+m)$ operations, respectively, where m and l are the numbers of arcs and the strongly connected components of the digraph associated with a given matrix, respectively. The space which is used by the procedure is bounded by $4n+3l+l^2+c$, where c is a constant which does not depend on data.

3.2. Procedure ***matrixpermpq2*****.** Procedure *matrixpermpq2* reduces a matrix A to form (2). It is assumed that A is a non-singular matrix since problems (i) and (ii) which can be simplified if A is reducible to form (2) are trivial in the opposite case.

```
procedure matrixpermpq2(n,p,nrp,f,np,nq);
 value n;
 integer n;
 Boolean f;
 comment (1) The procedure finds the permutation
  matrices P and Q which reduce a given matrix
  A to the block triangular form (2);
 integer array p,nrp,np,nq;
 begin
  integer i,j,k1,l,ll,l1,l2,m,m1,s;
  Boolean ff,ad;
  integer array c,k,r,sol[1:n];
  comment (2) The declaration of procedure ma-
   trixpermpp is to be inserted in this place.
   The procedure matrixpermpp is a modification
   of procedure matrixpermp consisting in ex-
   changing the 40-th and the 135-th rows of it
   to  k:=nrp[v]-1 and w1:=nrp[c[z]]-1,resp.;
  comment (3) Steps 1 and 2. Determination of
   a maximal transversal of matrix A and trans-
   formation of A to AR;
  f:=true;
  for j:=1 step 1 until n do
   k[j]:=sol[j]:=0;
  l:=0;
  m1:=-n-1;
  for i:=1 step 1 until n do
   begin
    for j:=1 step 1 until n do
     begin
```

```
      l2:=nrp[i];
      for l1:=nrp[i-1]+1 step 1 until l2 do
       begin
        l1:=p[l1];
        if l1≥j
         then
         begin
          ad:=l1=j;
          go to FD1
         end
       end l1;
      ad:=false;
FD1:  if ad
       then
       begin
        if sol[j]=0
         then
         begin
          sol[j]:=i;
          l:=l+1;
          r[i]:=0;
          go to NEXTI
         end sol[j]=0
       end ad
     end j;
    r[i]:=m1;
NEXTI:
   end i;
  if l=n
   then go to FUN;
```

```
ITER:
  for i:=1 step 1 until n do
   begin
    k1:=r[i];
    if k1<0
     then
     begin
      r[i]:=-k1;
      for j:=1 step 1 until n do
       if k[j]=0
        then
        begin
         l2:=nrp[i];
         for ll:=nrp[i-1]+1 step 1 until l2 do
          begin
           l1:=p[ll];
           if l1≥j
            then
            begin
             ad:=l1=j;
             go to FD2
            end
          end ll;
         ad:=false;
FD2:     if ad
          then
          begin
           if sol[j]=0
            then go to EX;
           k[j]:=-i;
```

```
           ff:=true
          end ad
        end j
     end k1<0
   end i;
  if ff
   then
   begin
    for j:=1 step 1 until n do
     begin
      k1:=k[j];
      if k1<0
       then
       begin
        k[j]:=-k1;
        r[sol[j]]:=-j;
        ff:=false
       end k1<0
     end j
   end ff
   else go to FUN;
  go to if ff then FUN else ITER;
EX:
  for sol[j]:=i while r[i]≤n do
   begin
    j:=abs(r[i]);
    i:=k[j]
   end sol[j];
  r[i]:=0;
  l:=l+1;
```

```
  if l=n
   then go to FUN1;
  for i:=1 step 1 until n do
   r[i]:=if r[i]≤n then 0 else m1;
  for j:=1 step 1 until n do
   k[j]:=0;
  go to ITER;
FUN:
  if l<n
   then
   begin
    f:=false;
    go to FIN
   end;
FUN1:
  for i:=1 step 1 until n do
   r[sol[i]]:=i;
  for i:=1 step 1 until n do
   begin
    l2:=nrp[i];
    for j:=nrp[i-1]+1 step 1 until l2 do
     if p[j]=r[i]
      then
      begin
       for l1:=j+1 step 1 until l2 do
        p[l1-1]:=p[l1];
       go to EP
      end j;
EP:end i;
  l:=nrp[n];
```

```
for i:=1 step 1 until l do
 p[i]:=sol[p[i]];
comment (4) Step 3. Computation of the permu-
 tation np;
matrixpermpp(n,p,nrp,f,np);
comment (5) Computation of the permutation nq;
if f
 then
 for i:=1 step 1 until n do
  nq[i]:=r[np[i]];
FIN:
 end matrixpermpq2
```

Let B be the bipartite digraph associated with A such that the se of vertices is $S \cup T = \{s_1, s_2, \ldots, s_n\} \cup \{t_1, t_2, \ldots, t_n\}$ and a pair $(s_i, t_j$ is an arc of B if and only if $a_{ij} \neq 0$.

Reduction (2) of A is related to the reduction of the bipartite digrap associated with A (see [1]-[3] and [8] for details).

If a matrix A is non-singular, then the reduction PAQ is in fac carried out by two steps $P(AR)P^{-1}$, where R is a permutation matri such that the matrix AR has non-zero diagonal elements, hence $Q = RP^{-}$ (see [2]).

The algorithm for finding the permutation matrices P and Q reducing A to form (2) is as follows.

Step 1. Find a maximal transversal of B, i.e., a maximal numbe of non-zero elements of A such that no two of them belong to the sam row or to the same column of A. The algorithm of Ford and Fulkerso has been used in this step (see [5] and [9]).

If a maximal transversal has cardinality less than n, then the matrix A cannot be reduced to form (2). The matrix A is singular in this case.

Step 2. Find AR, i.e., permute the columns of A until the elements of the maximal transversal occupy the main diagonal.

Step 3. Apply procedure *matrixpermp* to the matrix AR.

Procedure *matrixpermpq2* needs $O(n^3)$ and $O(m)$ operations in Step 1 and Step 2, respectively, and the space is bounded by $4n$ in these two steps. Time and space complexity of Step 3 has been evaluated above.

```
procedure matrixpermpq3(n,p,nrp,f,np,nq);
 value n;
 integer n;
 Boolean f;
 integer array p,nrp,np,nq;
 comment (1) The procedure finds the permutation
  matrices P and Q which decompose a given matrix
  A to the block diagonal form (1);
 begin
  integer i,k,l,m;
  k:=m:=0;
  for i:=1 aten 1 until n do
   begin
    l:=nrp[i];
    k:=l-k;
    m:=m+k×(k-1);
    k:=l
   end i;
  if m=0
   then
   begin
    f:=false;
    go to fin
   end m=0;
  k:=n×(n-1);
  if m>k
   then m:=k;
  begin
   integer i1,j,j1,l,l1;
   integer array cn,cnb,ncv,np1[1:n],nrp1[0:n],p1[1:m];
```

```
integer procedure conrec(p1,nrp1,cn,cv,ncv);
 integer array p1,nrp1,cn,cv,ncv;
 comment (2) The procedure conrec is a realiza-
  tion of Tarjan's method [14] for finding the
  connected components of the simple graph G
  associated with A. The structure of G is sto-
  red in arrays p1[0:m] and nrp1[0:n], where
  m=min(n×(n-1),(d[1]×(d[1]-1)+...+d[n]×(d[n]-
  1))) and d[i] is the number of non-zero
  elements in the i-th row of A;
 begin
  integer f,g,j,l;
  integer array number[1:n];
  procedure dfs(v);
   integer v;
   begin
    integer i,i1,w;
    l:=number[v]:=l+1;
    cn[v]:=g;
    f:=f+1;
    cv[l]:=v;
    i1:=nrp1[v];
    for i:=nrp1[v-1]+1 step 1 until i1 do
     begin
      w:=p1[i];
      if number[w]=0
       then dfs(w)
     end i;
   end dfs;
  g:=l:=0;
```

```
  for j:=1 step 1 until n do
   number[j]:=0;
  for j:=1 step 1 until n do
   if number[j]=0
    then
    begin
     f:=0;
     g:=g+1;
     dfs(j);
     ncv[g]:=f
    end j;
  conrec:=g
 end conrec;
comment (3) Finding the elements of arrays p1
 and nrp1 (see comment (2));
nrp1[0]:=m:=0;
for k:=1 step 1 until n do
 begin
  l:=k-1;
  for j:=1 step 1 until l,k+1 step 1 until n do
   begin
    for i:=1 step 1 until n do
     begin
      l1:=nrp[i];
      for i1:=nrp[i-1]+1 step 1 until l1 do
       if p[i1]≥k
        then
        begin
         if p[i1]≠k
          then go to FD2
```

```
             else
             for j1:=nrp[i-1]+1 step 1 until l1 do
              if p[j1]≥j
               then
               begin
                if p[j1]=j
                 then
                 begin
                  m:=m+1;
                  p1[m]:=j;
                  go to FD3
                 end p[j1]=j
                 else go to FD2
               end p[j1]≥j
           end i1;
FD2:    end i;
FD3:  end j;
     nrp1[k]:=m
    end k;
   comment (4) Computation of the permutation nq;
   k:=conrec(p1,nrp1,cn,nq,ncv);
   comment (5) Computation of the permutation np;
   if k=1
    then
    begin
     f:=false;
     go to fin
    end k=1
    else f:=true;
   for i:=1 step 1 until n do
```

```
 np[i]:=np1[i]:=0;
l:=0;
for i:=1 step 1 until k do
 begin
  cnb[i]:=l+1;
  l:=l+ncv[i]
 end i;
j:=1;
for i:=1 step 1 until n do
 begin
  j1:=nrp[i];
  if j1≥j
   then
   begin
    np1[i]:=1;
    l:=cn[p[j]];
    k:=cnb[l];
    np[k]:=i;
    cnb[l]:=k+1
   end j1≥j;
  j:=j1+1
 end i;
j:=1;
for i:=1 step 1 until n do
 if np1[i]=0
  then
  begin
   for j:=j step 1 until n do
    if np[j]=0
     then
```

```
                  begin
                    np[j]:=i;
                    go to nexti
                  end j;
      nexti:
                end i
          end;
      fin:
         end matrixpermpq3
```

3.3. Procedure *matrixpermpq3***.** Procedure *matrixpermpq3* finds the permutation matrices P and Q which decompose a given matrix A to the block diagonal form (1).

Let G denote the digraph associated with A whose vertices correspond to columns of A and two vertices are adjacent if there exists at least one row of A having non-zero elements in both of the columns. Let

$$C = A^{\mathrm{T}} * A = \bigcup_{k=1}^{n} a_{ki} \cap a_{kj},$$

where A is regarded as a Boolean matrix. It is easily seen that C is the adjacency matrix of G, i.e., (i, j) is an arc of G if and only if $c_{ij} \neq 0$. C is a symmetric matrix so that G is a symmetric digraph (i.e., simple graph).

It is easy to prove the following lemma (see also [7] and [15]).

LEMMA 4. (a) *Vertices of G (i.e., columns of A) which belong to a particular diagonal block are connected by paths of length n or less with each other, i.e., a partition of columns of A into diagonal block submatrices is determined uniquely by the partition of vertices of G into connected components.*

(b) *If $a_{ij} = 1$, then the i-th row of A belongs to the diagonal block containing the j-th column of A.*

The main steps of the algorithm are the following:

Step 1. Find the connected components of G. The depth-first search technique has been used in this step.

Step 2. Order columns of A according to the partition obtained in Step 1.

Step 3. Order rows of A (see Lemma 4 (b)).

The algorithm has the running time bounded by $O(n^3)$ and the space bounded by

$$6n+\min\left\{n^2, \sum_{i=1}^{n} d_i(d_i-1)\right\}+c,$$

where d_i $(i=1,2,\ldots,n)$ is the number of non-zero elements in the i-th row of A.

4. DATA AND RESULTS

Data are the same for all procedures.

n — dimension of a matrix A;

p[*1*: *m*], *nrp*[*0*: *n*] — these two arrays are the list representation of the matrix A, i.e., m is the number of non-zero elements of A and the column numbers of non-zero elements belonging to the row k are located in the increasing order in $p\,[nrp\,[k-1]+1\colon nrp\,[k]]$; we assume that $nrp\,[0]=0$ and one can see that $m=nrp\,[n]$.

Results:

f — Boolean variable which has the value **true** if the matrix A can be transformed and **false** otherwise;

np[*1*: *n*] — integer array such that $P=(e_{i_1}, e_{i_2}, \ldots, e_{i_n})^{\mathrm{T}}$, where $i_j = np\,[j]$ and e_i denotes the i-th unit vector;

nq[*1*: *n*] — integer array such that $Q=(e_{j_1}, e_{j_2}, \ldots, e_{j_n})$, where $j_l = nq[l]$.

5. EXAMPLES AND REMARKS

Example 1.

$$A_1 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

The matrix A_1 can be reduced to form (4) by procedure *matrixpermp* using

$$P_1 = (e_5, e_4, e_1, e_2, e_6, e_3)^{\mathrm{T}}.$$

Thus

$$P_1A_1P_1^{-1} = \left[\begin{array}{ccc|c|cc} 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ \hline & & & 0 & 0 & 1 \\ \cline{4-6} & 0 & & & 0 & 1 \\ & & & & 1 & 0 \end{array}\right].$$

Notice that procedure *matrixpermpq2* applied to the matrix A_1 fails to find this reduction. Generally, procedure *matrixpermpq2* is not a relaxation of procedure *matrixpermp*.

Example 2.

$$A_2 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}.$$

The digraph corresponding to the matrix A_2 is strongly connected. Hence, for no permutation P does PA_2P^{-1} reduce. Using procedure *matrixpermpq2* we obtain permutation matrices

$$R_2 = (e_2, e_1, e_4, e_3) \quad \text{and} \quad P_2 = (e_1, e_3, e_2, e_4)^{\mathrm{T}},$$

and hence $Q_2 = (e_2, e_4, e_1, e_3)$. Finally,

$$P_2A_2Q_2 = \left[\begin{array}{c|c|cc} 1 & 1 & 0 & 1 \\ \hline 0 & 1 & 0 & 1 \\ \cline{2-4} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{array}\right].$$

Example 3.

$$A_3 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Despite $\det(A_3) = 0$, the matrix A_3 has the maximal transversal of cardinality 4. In fact, procedure *matrixpermpq2* can reduce any matrix A which satisfies $\mathrm{per}(A) \neq 0$, where per denotes the $(+)$-determinant (permanent) of a matrix.

For the matrix A_3, procedure *matrixpermpq2* finds

$$P_3 = (e_1, e_3, e_2, e_4)^{\mathrm{T}}, \quad Q_3 = (e_2, e_4, e_1, e_3),$$

and

$$P_3A_3Q_3 = \left[\begin{array}{c|cc|c} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right].$$

The same result can be obtained by applying procedure *matrixpermpq3* to the matrix A_3.

Example 4.

$$A_4 = \begin{bmatrix} A_1 & 0 \\ 0 & A_3 \end{bmatrix}.$$

The matrix A_1 can be reduced by using procedure *matrixpermp*, and the matrix A_3 can be reduced by using procedure *matrixpermpq2* and decomposed by using *matrixpermpq3*.

Procedure *matrixpermp* reduces A_4 to the form

$$\begin{bmatrix} P_1A_1P_1^{-1} & 0 \\ 0 & A_3 \end{bmatrix},$$

and procedure *matrixpermpq3* decomposes A_4 to the form

$$\begin{bmatrix} A_1 & 0 \\ 0 & P_3A_3Q_3 \end{bmatrix},$$

however procedure *matrixpermpq2* fails.

Notice that reduction of form (2) can be found by using procedure *matrixpermpq2* to submatrices A_1 and A_3.

References

[1] A. L. Dulmage and N. S. Mendelsohn, *Coverings of bipartite graphs*, Canad. J. Math. 10 (1958), p. 517-534.

[2] — *On the inversion of sparse matrices*, Math. Comput. 16 (1962), p. 494-496.

[3] — *Two algorithms for bipartite graphs*, SIAM J. Appl. Math. 11 (1963), p. 183-194.

[4] J. Dzikiewicz, *Algorithmus 26, An algorithm for finding the transitive closure of a digraph*, Computing 15 (1975), p. 75-79.

[5] L. R. Ford and D. R. Fulkerson, *Flows in networks*, Princeton University Press, N. J., 1962 (*Przeplywy w sieciach*, PWN, Warszawa 1969).

[6] F. Harary, *A graph theoretic method for the complete reduction of a matrix with a view toward finding its eigenvalues*, J. Mat. Physics 39 (1959), p. 104-111.

[7] — *A graph theoretic approach to matrix inversion by partitioning*, Numer. Math. 4 (1962), p. 128-135.

[8] D. M. Johnson, A. L. Dulmage and N. S. Mendelsohn, *Connectivity and reducibility of graphs*, Canad. J. Math. 14 (1962), p. 529-539.

[9] J. Kucharczyk and M. M. Sysło, *Algorytmy optymalizacji w języku ALGOL 60*, PWN, Warszawa 1975.

[10] R. B. Marimont, *A new method of checking the consistency of precedence matrices*, J. ACM 6 (1959), p. 164-171.

[11] M. M. Sysło, *Algorithm 36, Transitive closure of a graph*, Zastosow. Matem. 14 (1974), p. 477-480.

[12] — *Stosowana teoria grafów, I. Zastosowanie teorii grafów w metodach numerycznych*, Matematyka Stosowana 5 (1975), p. 69-87.

[13] — and J. Dzikiewicz, *Computational experiences with some transitive closure algorithms*, Computing 15 (1975), p. 33-39.

[14] R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Computing 1 (1972), p. 146-160.

[15] R. P. Tewarson, *Row-column permutation of sparse matrices*, Comput. J. 10 (1967), p. 300-305.

INSTITUTE OF MATERIAL SCIENCE AND TECHNICAL MECHANICS
TECHNICAL UNIVERSITY OF WROCŁAW
50-370 WROCŁAW

INSTITUTE OF COMPUTER SCIENCE
UNIVERSITY OF WROCŁAW
50-384 WROCŁAW

Received on 5. 2. 1977

ALGORYTMY 62-64

JADWIGA DZIKIEWICZ i M. M. SYSŁO (Wrocław)

TRANSFORMACJE MACIERZY RZADKICH METODAMI TEORII GRAFÓW

STRESZCZENIE

Praca zawiera opisy trzech algorytmów dla transformacji kwadratowej macierzy rzadkiej do diagonalnej lub górnej trójkątnej postaci blokowej przy użyciu tylko permutacji wierszy i kolumn. Zredukowane w ten sposób macierze pozwalają zmniejszyć wymiary wielu problemów analizy numerycznej, jak np. obliczanie wartości wyznacznika, odwracanie macierzy, obliczanie wartości własnych oraz rozwiązywanie układów równań liniowych. Podstawowe kroki algorytmów mają swoje uzasadnienie na gruncie teorii grafów.

Dane (jednakowe dla wszystkich procedur):

n — wymiar macierzy A;

$p[1:m]$, $nrp[0:n]$ — tablice zawierające listową reprezentację macierzy A, tzn. m jest liczbą niezerowych elementów macierzy A, a numery kolumn, w których znajdują się niezerowe elementy k-tego wiersza, umieszczone są w rosnącym porządku w $p[nrp[k-1]+1 : nrp[k]]$; zakładamy, że $nrp[0] = 0$, i widać, że $m = nrp[n]$.

Wyniki:

f — zmienna boolowska, która przyjmuje wartość **true**, jeśli istnieje odpowiednia transformacja macierzy A, wartość **false** zaś w przeciwnym razie;

$np[1:n]$ — tablica całkowita taka, że $P = (e_{i_1}, e_{i_2}, \ldots, e_{i_n})^{\mathrm{T}}$, gdzie $i_j = np[j]$, a e_i oznacza i-ty wektor jednostkowy;

$nq[1:n]$ — tablica całkowita taka, że $Q = (e_{j_1}, e_{j_2}, \ldots, e_{j_n})$, gdzie $j_l = nq[l]$.