DSR-graph, positive feedback loops and injectivity
November 2014

Initializations

```
> restart :
> interface(rtablesize = 40) :
> with(ListTools) :
> with(LinearAlgebra) :
> with(GraphTheory) :
> with(combinat) :
> _Envsignum0 := 0 :
```

Execute all the comands in this section below. You can select it and then press the symbol "!" above. This will execute the selected region.

To execute before proceeding

Step 1. Procedures to create the DSR-graph from the stoichiometric matrix

Find the matrix Z from the stoichiometric matrix A:

```
> findZ := proc(A)
    local Z, n, m, i, j :
      n := Dimension(A)[1] :
      m := Dimension(A)[2] :
      Z := Matrix(n, m) :
   for i from 1 to n by 1 do
     for j from 1 to m by 1 do
        if A[i,j] < 0 then Z[i,j] := z[i,j]; end if;      ### what is the z?
     end do;
   end do:
    return(Z) :
    end proc:
```

```
>
```

Find the DSR graph from labels, A and Z

```
> ##Create signed DSR graph: entries are two matrices and the labels of the nodes
```

```
createDSRgraphsigned := proc(mynodes, A, Z)
  local G, n, m, Adj, varsZ, Zsign, varsA, Asign, X:
  n := Dimension(A)[1]: m := Dimension(A)[2]:
  X := Transpose(Z):
  varsZ := indets(X):
  Zsign := subs(seq(varsZ[i] = 1, i = 1..numelems(varsZ)), X):

 Adj := Matrix(n + m, n + m):
 Adj[[n + 1..n + m], [1..n]] := Transpose(map(signum, A)):
 Adj[[1..n], [n + 1..n + m]] := Transpose(Zsign):

 G := GraphTheory[Graph](mynodes, Adj, weighted = true):
 return(G):
end proc:
```

Find the DSR graph from labels and A and return the list of edges:

```
> findedgesDSR := proc(mynodes, A)
    local G, Z:
    Z := findZ(A):
   G := createDSRgraphsigned(mylabels, A, Z):
   return(Edges(G, weights)):
  end proc:
```

```
>
```

Step 2. Procedures to test for multistationarity

Procedures

```
> ## compute Mtilde determinant
  computdet := proc(N, X)
   global Mt:
    local M, F, i, bigdet, n1, s1:
    n1 := Dimension(N)[1]: s1 := Rank(N):
    M := N.X:
    Mt := M:
```

```
    if s1 < n1 then
      F
        := ReducedRowEchelonForm(Transpose(Matrix([op(NullSpace(Transpose(
       N)))])))):
      for i from 1 by 1 to Dimension(F)[1] do
          Mt[ArrayTools[SearchArray](F[i])[1]] := F[i]:
      end do:
    end if:
    bigdet := expand(Determinant(Mt)):
    return(bigdet):
    end proc:
```

```
> ## compute Mtilde determinant
    computdet2 := proc(N, X)
     global Mt:
      local M, F, i, bigdet, n1, s1:
      n1 := Dimension(N)[1]: s1 := Rank(N):
      M := N.X:
      Mt := M:
     if s1 < n1 then
       F
         := ReducedRowEchelonForm(Transpose(Matrix([op(NullSpace(Transpose(
        N)))])))):
       for i from 1 by 1 to Dimension(F)[1] do
          Mt[ArrayTools[SearchArray](F[i])[1]] := F[i]:
       end do:
     end if:
     bigdet := expand(Determinant(Mt)):
     return(bigdet, Mt):
     end proc:
```

```
>
```

```
> ## injectivity check
    injective := proc(N, X)
     local det, signs, i, l, k:
     det := computdet(N, X):
     signs := ListTools[MakeUnique](map(sign, [coeffs(det)])):
      if det ≠ 0 then
          i := (-1)^(numelems(signs) + 1):
     else i := 0: end if:
      return(i, det):
     end proc:
```

```
> ## injectivity check
    injectivePW := proc(N, X)
     local det, signs, i, n, m, X1, l, k:
     n := Dimension(N)[1]: m := Dimension(N)[2]:
```

```
    X1 := DiagonalMatrix(Vector([seq(k_i, i = 1..m)])).X
          .DiagonalMatrix(Vector([seq(l_i, i = 1..n)])):

    det := computdet(N, X1):
    signs := ListTools[MakeUnique](map(sign, [coeffs(det)])):
     if det ≠ 0 then
         i := (-1)^(numelems(signs) + 1):
    else i := 0: end if:
     return(i, det):
    end proc:
```

```
> findV := proc(A)
     local V, n, m, i, j:
     n := Dimension(A)[1]:
     m := Dimension(A)[2]:
      V := Matrix(n, m):
    for i from 1 to n by 1 do
      for j from 1 to m by 1 do
         if A[i,j] < 0 then V[i,j] := -A[i, j]; end if;
      end do;
    end do:
     return(Transpose(V)):
     end proc:
```

```
> isinjective := proc(A)
     local V, i, det:
     V := findV(A):
     i, det := injectivePW(A, V):
     if i = 1 then return(1):
     else
       return(0):
     end if:
    end proc:
```

```
> ## compute Mtilde determinant with F given
    computdetF := proc(N, X, F)
     global Mt:
     local M, i, bigdet, n1, s1, sp:
     n1 := Dimension(N)[1]: s1 := Rank(N): sp := Dimension(X)[2]:
     M := N.X:
    Mt := Matrix(sp, sp):
     Mt[1..Dimension(F)[1]] := F:
     Mt[Dimension(F)[1] + 1..sp] := M:
```

```
    bigdet := expand(Determinant(Mt)) :
    return(bigdet) :
end proc:
```

```
> ## compute Mtilde determinant with F given
computdetF2 := proc(N, X, F)
  local M, i, bigdet, n1, s1, sp, Mt :
  n1 := Dimension(N)[1] : s1 := Rank(N) : sp := Dimension(X)[2] :
  M := N.X :
  Mt := Matrix(sp, sp) :
   Mt[1..Dimension(F)[1]] := F :
   Mt[Dimension(F)[1] + 1..sp] := M :

  bigdet := expand(Determinant(Mt)) :
  return(bigdet, Mt) :
end proc:
```

```
> ## injectivity check with F given
injectivePWF := proc(N, X, F)
  local det, signs, i, n, m, X1, l, k :
  n := Dimension(X)[2] : m := Dimension(X)[1] :
  X1 := DiagonalMatrix(Vector([seq(k_i, i = 1..m)])).X
      .DiagonalMatrix(Vector([seq(l_i, i = 1..n)])) :

  det := computdetF(N, X1, F) :
  signs := ListTools[MakeUnique](map(sign, [coeffs(det)])) :
  if det ≠ 0 then
     i := (-1)^(numelems(signs) + 1) :
  else i := 0 : end if:
  return(i, det) :
end proc:
```

```
> ## injectivity check with F given
injectivePWF2 := proc(N, X, F)
  local det, signs, i, n, m, X1, l, k, Mt :
  n := Dimension(X)[2] : m := Dimension(X)[1] :
  X1 := DiagonalMatrix(Vector([seq(k_i, i = 1..m)])).X
      .DiagonalMatrix(Vector([seq(l_i, i = 1..n)])) :

  det, Mt := computdetF2(N, X1, F) :
  signs := ListTools[MakeUnique](map(sign, [coeffs(det)])) :
  if det ≠ 0 then
     i := (-1)^(numelems(signs) + 1) :
```

```
    else i := 0 : end if:
     return(i, det, Mt) :
    end proc:


> ## check if after gauss reduction the system becomes injective: check all
    gaussinjallpos := proc(B, V, F)
        local m, myset, n, i, det, bigset, seenset, B2, V2, B1, myset2, control, control2 :
       m := Dimension(B)[2] : n := Dimension(B)[1] :
        myset := {seq(i, i = 1 ..n)} : control := 0 :
        bigset := {seq(i, i = 1 ..m)} :
       seenset := [myset] :

      while myset ≠ FAIL and control = 0 do
        B2 := B[.., [op(myset), op(bigset minus myset)]] :
        V2 := V[[op(myset), op(bigset minus myset)],..] :
         i, det, B1 := gaussinj(B2, V2, F) :
        if i = 1 then control := 1 : end if:  ## injective found
        if i = 2 then control := 2 : end if:  ## no positive steady states
        myset2 := { } :
        for i from 1 by 1 to Dimension(B1)[1] do
            myset2 := {op(myset2), ArrayTools[SearchArray](B1[i])[1]} :
        end do:
        seenset := [op(seenset), myset2] :
        seenset := [op(seenset), myset] :
        seenset := ListTools[MakeUnique](seenset) :
        while member(myset, seenset) and myset ≠ FAIL do
          myset := nextcomb(myset, m) :
       end do:
      end do:
      control2 := 0 :
      if control = 1 then
          while myset ≠ FAIL and control = 1 do
            B2 := B[.., [op(myset), op(bigset minus myset)]] :
            B1 := ReducedRowEchelonForm(B2) :
            control2 := gausssamesign(B1) :
           if control2 = 1 then control := 2 : end if:
          seenset := [op(seenset), myset] :
           seenset := ListTools[MakeUnique](seenset) :
          while member(myset, seenset) and myset ≠ FAIL do
            myset := nextcomb(myset, m) :
          end do:
        end do:
      end if:
      return(control) :
    end proc:
>
```

```
> ## check if after gauss reduction the system becomes injective: check all
  gausssignall := proc(B)
     local m, myset, n, i, det, bigset, seenset, B2, V2, B1, myset2, control, control2 :
    m := Dimension(B)[2] : n := Dimension(B)[1] :
     myset := {seq(i, i = 1 ..n)} :
     bigset := {seq(i, i = 1 ..m)} :

   control2 := 0 :
      while myset ≠ FAIL and control2 = 0 do
        B2 := B[ .., [op(myset), op(bigset minus myset)]] :
        B1 := ReducedRowEchelonForm(B2) :
        control2 := gausssamesign(B1) :
        myset := nextcomb(myset, m) :
     end do:
  return(control2) :
  end proc:
>

> ## check if after gauss reduction the system becomes injective
  gaussinj := proc(N, V, F)
    local p, mysigns, control2, myset, B, i, m, n, maxcols, B1, B3, V3, j, maxcols2, count,
      indiceslist, mypos, B4, V4, control, M, Mt, det, signs, sp, det2, l, k :
  m := Dimension(N)[2] : n := Dimension(N)[1] : sp := Dimension(V)[2] :
  B1 := N :
  maxcols := numelems(ArrayTools[SearchArray](B1)) − n :
  B3 := Matrix(Dimension(B1)[1], maxcols) :
  V3 := Matrix(maxcols, sp) :

  count := 1 : indiceslist := [ ] :
  for i from 1 by 1 to Dimension(B1)[1] do
      maxcols2 := ArrayTools[SearchArray](B1[i]) :
      for j from 2 by 1 to numelems(maxcols2) do
        control := 0 :
        mypos := ListTools[Search](maxcols2[j], indiceslist) :
        if mypos ≠ 0 then
          ##print(i, j, mypos) :
           if Equal(V[maxcols2[j]] − V[maxcols2[1]], V3[mypos]) then
                B3[i, mypos] := -B1[i, maxcols2[j]] : control := 1 :
           end if:
        end if:
        if control = 0 then
            V3[count] := V[maxcols2[j]] − V[maxcols2[1]] :
            indiceslist := [op(indiceslist), maxcols2[j]] :
            B3[i, count] := -B1[i, maxcols2[j]] :
            count := count + 1 :
        end if:
      end do:
```

```
    end do:
    B4 := SubMatrix(B3, [1..Dimension(B3)[1]], [1..count - 1]):
    V4 := SubMatrix(V3, [1..count - 1], [1..Dimension(V3)[2]]):

    M := B4.DiagonalMatrix(Vector([seq(k_i, i = 1..Dimension(V4)[1])])).V4
      .DiagonalMatrix(Vector([seq(l_i, i = 1..Dimension(V4)[2])])):
    Mt := Matrix(sp, sp):
    Mt[1..Dimension(F)[1]] := F:
    Mt[Dimension(F)[1] + 1..sp] := M:

    ##injectivity test
    det := Determinant(Mt):
    det2 := collect(det, indets(det),'distributed'):
    signs := ListTools[MakeUnique](map(sign, [coeffs(det2)])):
    if det2 ≠ 0 then
      i := (-1)^(numelems(signs) + 1):
    else i := 0: end if:
    return(i, det, B1):
end proc:

> ## check if after gauss reduction the system becomes injective and return also the
    matrices
gaussinjV := proc(N, V, F)
  local p, mysigns, control2, myset, B, i, m, n, maxcols, B1, B3, V3, j, maxcols2, count,
    indiceslist, mypos, B4, V4, control, M, Mt, det, sp, signs, det2:
m := Dimension(N)[2]: n := Dimension(N)[1]: sp := Dimension(V)[2]:
B1 := N:
maxcols := numelems(ArrayTools[SearchArray](B1)) - n:
B3 := Matrix(Dimension(B1)[1], maxcols):
V3 := Matrix(maxcols, sp):

count := 1: indiceslist := []:
for i from 1 by 1 to Dimension(B1)[1] do
    maxcols2 := ArrayTools[SearchArray](B1[i]):
    for j from 2 by 1 to numelems(maxcols2) do
      control := 0:
      mypos := ListTools[Search](maxcols2[j], indiceslist):
      if mypos ≠ 0 then
        ##print(i, j, mypos):
        if Equal(V[maxcols2[j]] - V[maxcols2[1]], V3[mypos]) then
          B3[i, mypos] := -B1[i, maxcols2[j]]: control := 1:
        end if:
      end if:
      if control = 0 then
        V3[count] := V[maxcols2[j]] - V[maxcols2[1]]:
        indiceslist := [op(indiceslist), maxcols2[j]]:
        B3[i, count] := -B1[i, maxcols2[j]]:
```

```
            count := count + 1 :
        end if:
      end do:

     end do:
    B4 := SubMatrix(B3, [1..Dimension(B3)[1]], [1..count - 1]) :
    V4 := SubMatrix(V3, [1..count - 1], [1..Dimension(V3)[2]]) :

    M := B4.DiagonalMatrix(Vector([seq(h_i, i = 1..Dimension(V4)[1])])).V4
      .DiagonalMatrix(Vector([seq(l_i, i = 1..Dimension(V4)[2])])) :
    Mt := Matrix(sp, sp) :
    Mt[1..Dimension(F)[1]] := F :
    Mt[Dimension(F)[1] + 1..sp] := M :
     ##injectivity test
    det := Determinant(Mt) :
    det2 := collect(det, indets(det),'distributed') :
     signs := ListTools[MakeUnique](map(sign, [coeffs(det2)])) :
     if det2 ≠ 0 then
        i := (-1)^(numelems(signs) + 1) :
     else i := 0 : end if:
     return(i, det, B4, V4) :
  end proc:


> ## check if after gauss reduction the system becomes injective: check all
  gaussinjall := proc(B, V, F, myred)
      local m, myset, i, n, k, det, bigset, seenset, B2, V2, B1, myset2, myset3, control,
       control2 :
      m := Dimension(B)[2] : n := Dimension(B)[1] :
       myset := {seq(i, i = 1..n)} : control := 0 :
       bigset := {seq(i, i = 1..m)} :
      seenset := [myset] :

    while myset ≠ FAIL and control = 0 do
       B2 := B[.., [op(myset), op(bigset minus myset)]] :
       V2 := V[[op(myset), op(bigset minus myset)],..] :
        B1 := ReducedRowEchelonForm(B2) :

        myset2 := { } :
        for k from 1 by 1 to n do
            myset2 := {op(myset2), ArrayTools[SearchArray](B1[k])[1]} :
       end do:
        myset3 := {op([op(myset), op(bigset minus myset)][[op(myset2)]])} :
       if not member(myset3, seenset) then
            i, det, B1 := gaussinj(B1, V2, F) :
            if i = 1 then control := 1 : end if:  ## injective found
       end if:
       seenset := [op(seenset), myset3] :
```

```
    seenset := [op(seenset), myset]:
    seenset := ListTools[MakeUnique](seenset):

    ##find new subset, filtered by seenset and the already known independent columns
   control2 := 0:
   while control2 = 0 do
      control2 := 1:
      myset := nextcomb(myset, m):
     if myset ≠ FAIL then
        if member(myset, seenset) then control2 := 0:
        else
         k := 1:
         while k ≤ numelems(myred) and control2 = 1 do
            if subset(myred[k], myset) then control2 := 0: end if:
            k := k + 1:
          end do:
       end if:
     end if:
   end do:
 end do:  ##end big do, for myset and control

 return(control):
end proc:
>


> ## check if after gauss reduction the system becomes injective: check bistable
 Bistablecheck := proc(B)
    local m, n, j, k, control, control2, myrow, myvec, signvec:
   m := Dimension(B)[2]: n := Dimension(B)[1]:
   j := 1:
   control2 := 0:
   control := 2:
   while control2 = 0 and j ≤ n do
      myrow := convert(ArrayTools[SearchArray](B[j]), list):
     if numelems(myrow) ≤ 1 then control2 := 1: control := 3:
     else
        myvec := convert(B[j][myrow[2..numelems(myrow)]], list):
        signvec := ListTools[MakeUnique](map(sign, myvec)):
        if signvec ≠ [-1] then control2 := 1: control := 0:
        else
           k := 2:
           while control2 = 0 and k ≤ numelems(myrow) do
                if numelems(ArrayTools[SearchArray](B[.., myrow[k]])) ≠ 1
    then control2 := 1: control := 0: end if:
                k := k + 1:
          end do:
        end if:
```

```
        end if:
        j := j + 1 :
    end do:
    return(control) :
    end proc:
>
```

```
> ## check if after gauss reduction the system becomes injective: check bistable
    Bistablecheck2 := proc(B)
      local m, n, j, k, control, control2, myrow, myvec, signvec, disjsets, nonzerocols,
       totalcard :
      m := Dimension(B)[2] : n := Dimension(B)[1] :
      j := 1 : disjsets := [ ] :
      control2 := 0 :
      control := 2 :
      while control2 = 0 and j ≤ n do
          myrow := convert(ArrayTools[SearchArray](B[j]), list) :
          if numelems(myrow) ≤ 1 then control2 := 1 : control := 3 :
          else
              myvec := convert(B[j][myrow[2 ..numelems(myrow)]], list) :
              signvec := ListTools[MakeUnique](map(sign, myvec)) :
              if signvec ≠ [-1] then control2 := 1 : control := 0 :
              else k := 2 :
                   while control2 = 0 and k ≤ numelems(myrow) do
                       nonzerocols := ArrayTools[SearchArray](B[.., myrow[k]]) :
                       if numelems(nonzerocols) > 1 then disjsets := [op(disjsets),
      convert(nonzerocols, list)] :
                       end if:
                       k := k + 1 :
                   end do:
              end if:
          end if:
          j := j + 1 :
      end do:
      disjsets := MakeUnique(disjsets) :
      totalcard := 0 :
      for k from 1 by 1 to numelems(disjsets) do totalcard := totalcard
          + numelems(disjsets[k]) : end do:
      if numelems(MakeUnique(Flatten(disjsets))) ≠ totalcard then control := 0 : end
          if:

      return(control) :
      end proc:
```

```
> ## check if after gauss reduction the system becomes injective: check all
    gaussinjallBi := proc(B, V, F, myred)
```

```
local m, myset, i, n, k, det, bigset, seenset, B2, V2, B1, myset2, myset3, control,
 control2 :
m := Dimension(B)[2] : n := Dimension(B)[1] :
myset := {seq(i, i = 1 ..n)} : control := 0 :
bigset := {seq(i, i = 1 ..m)} :
seenset := [myset] :

while myset ≠ FAIL and control = 0 do
  B2 := B[ .., [op(myset), op(bigset minus myset)]] :
  V2 := V[[op(myset), op(bigset minus myset)],..] :
  B1 := ReducedRowEchelonForm(B2) :

  myset2 := { } :
  for k from 1 by 1 to n do
      myset2 := {op(myset2), ArrayTools[SearchArray](B1[k])[1]} :
  end do:
  myset3 := {op([op(myset), op(bigset minus myset)][[op(myset2)]])} :
  if not member(myset3, seenset) then
      i, det, B1 := gaussinj(B1, V2, F) :
      if i = 1 then control := 1 : end if:  ## injective found
      if i = -1 or i = 0 then
          control := Bistablecheck(B1) :
      end if:
  end if:
  seenset := [op(seenset), myset3] :
  seenset := [op(seenset), myset] :
  seenset := ListTools[MakeUnique](seenset) :

  ## find new subset, filtered by seenset and the already known independent columns
 control2 := 0 :
 while control2 = 0 do
    control2 := 1 :
    myset := nextcomb(myset, m) :
   if myset ≠ FAIL then
     if member(myset, seenset) then control2 := 0 :
     else
      k := 1 :
      while k ≤ numelems(myred) and control2 = 1 do
        if subset(myred[k], myset) then control2 := 0 : end if:
        k := k + 1 :
       end do:
     end if:
   end if:
  end do:
end do:  ##end big do, for myset and control

return(control) :
end proc:
```

```
>
```

```
> gaussinjallBi2 := proc(B, V, F, myred)
    local m, myset, i, n, k, det, bigset, seenset, B2, V2, B1, myset2, myset3, control,
     control2, lastseen, myseen :
    m := Dimension(B)[2] : n := Dimension(B)[1] :
    myset := {seq(i, i = 1 ..n)} : control := 0 :
    bigset := {seq(i, i = 1 ..m)} :
    seenset := [ ] :
    myseen := [ ] :

    while myset ≠ FAIL and control = 0 do
      B2 := B[ .., [op(myset), op(bigset minus myset)]] :
      V2 := V[[op(myset), op(bigset minus myset)],..] :
      B1 := ReducedRowEchelonForm(B2) :
      lastseen := myset : myseen := [op(myseen), myset] :
      myset2 := { } :
      for k from 1 by 1 to n do
          myset2 := {op(myset2), ArrayTools[SearchArray](B1[k])[1]} :
      end do:
      myset3 := {op([op(myset), op(bigset minus myset)][[op(myset2)]])} :
      if not member(myset3, seenset) then
          i, det, B1 := gaussinj(B1, V2, F) :
          if i = 1 then control := 1 : end if:  ## injective found
          if i = -1 or i = 0 then
                control := Bistablecheck2(B1) :
          end if:
      end if:
      seenset := [op(seenset), myset3] :
      seenset := [op(seenset), myset] :
      seenset := ListTools[MakeUnique](seenset) :

      ## find new subset, filtered by seenset and the already known independent columns
      control2 := 0 :
      while control2 = 0 do
          control2 := 1 :
          myset := nextcomb(myset, m) :
          if myset ≠ FAIL then
              if member(myset, seenset) then control2 := 0 :
              else
                k := 1 :
                while k ≤ numelems(myred) and control2 = 1 do
                    if subset(myred[k], myset) then control2 := 0 : end if:
                    k := k + 1 :
                end do:
              end if:
          end if:
```

```
    end do:
  end do:  ##end big do, for myset and control

  return(control, myset, B1, lastseen, myseen):
end proc:
```

```
> isinjectiveextended := proc(A)
    local V, M, F, i, n1, s1, B, toexclude, myred, control, myset, B1, lastseen, myseen:
    myred := {}:
    V := findV(A):
    n1 := Dimension(A)[1]: s1 := Rank(A):
    toexclude := []:
    if s1 < n1 then
      F
      := ReducedRowEchelonForm(Transpose(Matrix([op(NullSpace(Transpose(
      A)))]))):
      for i from 1 by 1 to Dimension(F)[1] do
        toexclude := [op(toexclude), ArrayTools[SearchArray](F[i])[1]]:
      end do:
    else
      F := []:
    end if:
    B := SubMatrix(A, [op({seq(i, i = 1 .. n1)} minus {op(toexclude)})], [1
      ..Dimension(A)[2]]):
    if Dimension(B)[1] < Dimension(B)[2] then
      control, myset, B1, lastseen, myseen := gaussinjallBi2(B, V, F, myred):
      return(control):
    else
      return(0):
    end if:
  end proc:
```

```
>
```

Step 3. Finding the positive feedback loops for multistationary networks

Auxiliary procedures

```
> addlist := proc(mylist, myaddlist)
    local i, newlist:
```

```
    newlist := [op(mylist), op(myaddlist)]:
    return newlist:
end proc:
```

This procedure computes the polynomial $p_{A,Z}$ in the main text. The input are the matrices A and Z (in the function denoted N and X).

```
> ## compute Mtilde determinant
computdetS := proc(N, X)
 global Mt:
  local M, F, i, bigdet, n1, s1:
  n1 := Dimension(N)[1]: s1 := Rank(N):
  M := N.Transpose(X):
  Mt := M:
  if s1 < n1 then
    F
      := ReducedRowEchelonForm(Transpose(Matrix([op(NullSpace(Transpose(
     N)))])))):
   for i from 1 by 1 to Dimension(F)[1] do
      Mt[ArrayTools[SearchArray](F[i])[1]] := F[i]:
   end do:
  end if:
  bigdet := expand(Determinant(Mt)):
  return(bigdet):
 end proc:
```

This function returns the list of monomials that have the wrong sign. The input are the determinant and the wrong sign.

```
> ##Given a determinant and a wrong sign, return the list of wrong monomials
badterms := proc(deter, mysign)
    local vars, coeflist, monomlist, coeflistsign, wterms, i:
     vars := indets(deter):
    coeflist := [coeffs(deter, vars, 't')]:
     monomlist := [t]:
   coeflistsign := map(sign, coeflist):
    wterms := []:
    for i from 1 by 1 to numelems(coeflistsign) do
        if mysign = coeflistsign[i] then wterms := [op(wterms), monomlist[i]]: end
     if:
    end do:
   return(wterms):

 end proc:
```

Given a monomial on the entries of a matrix Amatrix, this function finds a matrix from Amatrix

such that the variables in the monomial become 1 and the rest are zero.

```
> ##Find submatrix of a matrix corresponding to a monomial
findmatrix := proc(wmonom, Amatrix)
    local vars, wvarscomp, Anew, i, Anew1 :
    vars := indets(wmonom) :
    wvarscomp := indets(Amatrix) minus vars :
    Anew := subs(seq(wvarscomp[i] = 0, i = 1 ..numelems(wvarscomp)), Amatrix) :
    Anew1 := subs(seq(vars[i] = 1, i = 1 ..numelems(vars)), Anew) :
   return(Anew1) :
  end proc:
```

Find the two submatrices of A and Z corresponding to the monomial, and make A symbolic by introducing a new variable x.

```
> ## Find the two matrices A,Z corresponding to a monomial
twomatrices := proc(wmonom)
    global A, Z :
    local vars, wvarscomp, Zembed, i, row, col, Aembed, Aembedx, nZ, X :
   X := Transpose(Z) :
    vars := indets(wmonom) :
    wvarscomp := indets(X) minus vars :
    Zembed := subs(seq(wvarscomp[i] = 0, i = 1 ..numelems(wvarscomp)), X) :
    row, col := ArrayTools[SearchArray](Zembed) :
    Zembed := Zembed[convert(row, list), convert(col, list)] :
     nZ := numelems(vars) :
    Aembed := A[convert(col, list), convert(row, list)] :
   Aembed := map(signum, Aembed) :
   row, col := ArrayTools[SearchArray](Aembed) :
   Aembedx := Matrix(nZ, nZ) :
   for i from 1 by 1 to numelems(col) do
     Aembedx[row[i], col[i]] := Aembed[row[i], col[i]]·x_i :
   end do:
   return(Aembedx, Zembed) :
end proc:
```

Extract the indices of the species and the reaction in the given monomial in the variables of Z

```
> ## Extract indices of species and reaction in the monomial from Z
extractsr := proc(wmonom)
    global Z :
    local vars, wvarscomp, Zembed, i, row, col, X :
    X := Transpose(Z) :
     vars := indets(wmonom) :
    wvarscomp := indets(X) minus vars :
     ## indeterminates not in the monomial
    Zembed := subs(seq(wvarscomp[i] = 0, i = 1 ..numelems(wvarscomp)), X) :
     ##set the entries of the indeterminates not in the monomial to zero
```

```
    row, col := ArrayTools[SearchArray](Zembed) :
     ##find the nonzero entries of the resulting matrix.
    return(row, col) :  ##return the species and reaction indices
  end proc:
```

```
> ##Create DSR graph: entries are two matrices and the labels of the nodes
  createDSRgraph := proc(mynodes, A, Z)
    local G, n, m, Adj, varsZ, Zsign, varsA, Asign, X:
    n := Dimension(A)[1] : m := Dimension(A)[2] :
    X := Transpose(Z) :
    varsZ := indets(X) :
    Zsign := subs(seq(varsZ[i] = 1, i = 1 ..numelems(varsZ)), X) :
    varsA := indets(A) :
    Asign := subs(seq(varsA[i] = 1, i = 1 ..numelems(varsA)), A) :

   Adj := Matrix(n + m, n + m) :
   Adj[[n + 1 ..n + m], [1 ..n]] := Transpose(map(signum, Asign)) :
   Adj[[1 ..n], [n + 1 ..n + m]] := Transpose(Zsign) :

    G := GraphTheory[Graph](mynodes, Adj, weighted = true) :
    return(G) :
  end proc:
>
```

This function selects the subgraphs that give rise to the monomials with the wrong sign.

```
> ## Select the subgraphs that correspond to the wrong terms of A and Z
  graphlist := proc(mydet)
   global A, Z :
   local srlist, row, col, Gsub, s, wsign, wrongterms, k, wcurrent, Aembedx, Zembed,
       detZ, detAx, wsignA, wrongtermsA, wcurrentA, j, Aembedx1, mynodes :
  Gsub := [ ]:
   srlist := [ ]:
   s := Rank(A) :
  wsign := (-1)^(s + 1) :                    ## find wrong sign
  wrongterms := badterms(mydet, wsign) :
       ## select the monomials with the wrong sign
   for k from 1 by 1 to numelems(wrongterms) do
       ## for each such monomial, find the associated subgraph
      wcurrent := wrongterms[k] :
      row, col := extractsr(wcurrent) :
       ## find the indices of the species and the reactions in the monomial
      mynodes := [seq(S_col[i], i = 1 ..numelems(col)), seq(R_row[i], i = 1
        ..numelems(row))] :
      Aembedx, Zembed := twomatrices(wcurrent) :
       ## the returned Zembed is giving half of the edges of the subgraphs
```

```
    detZ := subs(seq(indets(Zembed)[i] = 1, i = 1 ..numelems(indets(Zembed))),
     Determinant(Zembed)) :
    detAx := expand(Determinant(Aembedx)) :
     wsignA := wsign·detZ :
    wrongtermsA := badterms(detAx, wsignA) :
     ## select the monomials with the wrong sign of the subsystem
     for j from 1 by 1 to numelems(wrongtermsA) do
        wcurrentA := wrongtermsA[j] :
        Aembedx1 := findmatrix(wcurrentA, Aembedx) :
     ## find the other half of the edges of the subgraphs
        Gsub := [op(Gsub), createDSRgraph(mynodes, Aembedx1, Zembed)] :
     end do:
  end do:
  return(Gsub) :  ##return the list of graphs
 end proc:
```

Given a list of edges that form a loop, the function returns the edges ordered such that connected they form the loop.

```
> ## Order the edges to have a loop
 orderedge := proc(myedges)
    local orderededges, endpoint, total, control, k :
   orderededges := [myedges[1]] :
   endpoint := myedges[1][1][2] :
   total := numelems(myedges) :
  while numelems(orderededges) < total do
     control := 0 : k := 2 :
     while control = 0 do
       if endpoint = myedges[k][1][1] then
          orderededges := [op(orderededges), myedges[k]] :
          control := 1 :
         endpoint := myedges[k][1][2] :

       end if:
       k := k + 1 :
     end do:
  end do:
   return(orderededges) :
 end proc:
```

Find the sequence of signs of the loop

```
> ##Extract the  sequence  of signs of a loop
 extractsign := proc(orderededges)
    local graphsign, i :
    graphsign := [ ] :
   for i from 1 by 1 to numelems(orderededges) do
     graphsign := [op(graphsign), orderededges[i][2]] :
```

```
    end do:
    return(graphsign):
  end proc:
```

Given a list of graphs, we find the positive feedback loops that they contain and return the sign pattern of each positive feedback loop as well (as those given in Table 1 in the main text).

```
> ##Find the positive feedback loops in the list of graphs
  positivefeed := proc(Gsub)
    local selected, j, mygraph, Gsubcomp, k, mycomp, newgraph, wedges, myprod, i,
     signcycle:
    selected := []:
    signcycle := []:
    Gsubcomp := []:
    for j from 1 by 1 to numelems(Gsub) do
      mygraph := Gsub[j]:
      Gsubcomp := ConnectedComponents(mygraph):
      for k from 1 by 1 to numelems(Gsubcomp) do
        mycomp := Gsubcomp[k]:
        newgraph := InducedSubgraph(mygraph, mycomp):
        wedges := Edges(newgraph, weights):
        myprod := mul(wedges[i][2], i = 1..numelems(wedges)):
        if myprod = 1 then              ##if the loop is positive, select it
          selected := [op(selected), [op(wedges)]]:
        end if:
      end do:
    end do:
    selected := ListTools[MakeUnique](selected):
    for k from 1 by 1 to numelems(selected) do
      selected[k] := orderedge(selected[k]):
      signcycle := [op(signcycle), extractsign(selected[k])]:
    end do:
    return(selected, signcycle):
  end proc:
```

The main procedure to find the positive loop is the following:

```
> ##main program: find the positive loops
  findloops := proc( )
    global A, Z:
    local Gsub, selected, signcycle, mydet:
    mydet := computdetS(A, Z): ## find the polynomial p_{A, Z}
```

```
    Gsub := graphlist(mydet) :
      ## find the list of subgraphs corresponding to the wrong signs
    selected, signcycle := positivefeed(Gsub) :
      ## find the positive feedback loops and their sign pattern
     return(selected) :
   end proc:
```

The second main procedure of the method is the function that draws the selected positive feedback loops. It requires a list with the names of the nodes (see the examples below)

```
> ## draw the positive feedback loops
drawloops := proc(selected, speciesord)
  local loops, i, vertices, speciesdic, selected2 :
 loops := [ ]:
 speciesdic := {seq(S_i = speciesord[i], i = 1 ..numelems(speciesord))} :
 selected2 := subs(speciesdic, selected) :
for i from 1 by 1 to numelems(selected2) do
     vertices := ListTools[MakeUnique]([seq(op(selected2[i][j][1]), j = 1
     ..numelems(selected2[i]))]) :
    loops := [op(loops), Digraph(vertices, {op(selected2[i])})] :
 end do:
DrawGraph(loops, style = circle);

 end proc:
```

A is the stoichiometric matrix of the network.

Step 1.
Create the DSR-graph and find if there is a positive feedback loop through the competition.

To create the DSR-graph, do:

```
> findedgesDSR(mylabels, A)
```

where mylabels contains the names of the nodes.

Step 2
The next step is to find out which of the selected networks can have multiple steady states.
Here are a few tests to run before using the toolbox.

Injectivity test:

```
> isinjective(A)
```

If the answer is 1, then classify the network as NOT MULTISTATIONARY: there can be either none or

one positive steady state
If the answer is 0, then apply the next test.

At this point, if there are not many networks one can use the toolbox. If there are many networks, one can apply the next test. It takes though some time to run.

```
> isinjectiveextended(A)
```

If the answer is 1, then classify the network as NOT MULTISTATIONARY.
If the answer is 2, then classify the network as MULTISTATIONARY.
If the answer is 3, then the network does not have positive steady states.
If the answer is 0, then use the toolbox.

Step 3
For the networks that are multistationary, decide whether multistationarity can be attributed to competition, and find the positive feedback loops underlying multistationarity.

For that, we do:

```
> Z := findZ(A) : n := Dimension(A)[1] : m := Dimension(A)[2] : s := Rank(A) :
> selected := findloops( )
```

The procedure returns the list of positive feedback loops that underly multistationarity.
You can draw the loops after giving a label to the species, see the examples below.

The test could also be applied to the non-multistationary networks. Some of them would return some loops, and for some of them nothing would be returned.

Step 4.
The next step would be to understand what causes multistationarity. If it relates to the loops, then we should test if the networks that do not have multistationarity have that loop structure in their DSR-graph.

This part is still open.

Tricks:
If network N is multistationary and N' is another network that is identical to N **but has extra reactions** that do not change the dimension of the stoichiometric space, i.e. the rank of N' is the same as the rank of N, then N' is also multistationary.

Example 1

Consider the network
A+B< -> C
D+B->E ->A+F
A->D
F->B

The stoichiometric matrix A is:

```
> A := Transpose(Matrix([[-1,-1,1,0,0,0],-[-1,-1,1,0,0,0],[0,-1,0,-1,1,0],[1,0,
    0,0,-1,1],[-1,0,0,1,0,0],[0,1,0,0,0,-1]]));
```

$$A := \begin{bmatrix} -1 & 1 & 0 & 1 & -1 & 0 \\ -1 & 1 & -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix} \quad \textbf{(2.1)}$$

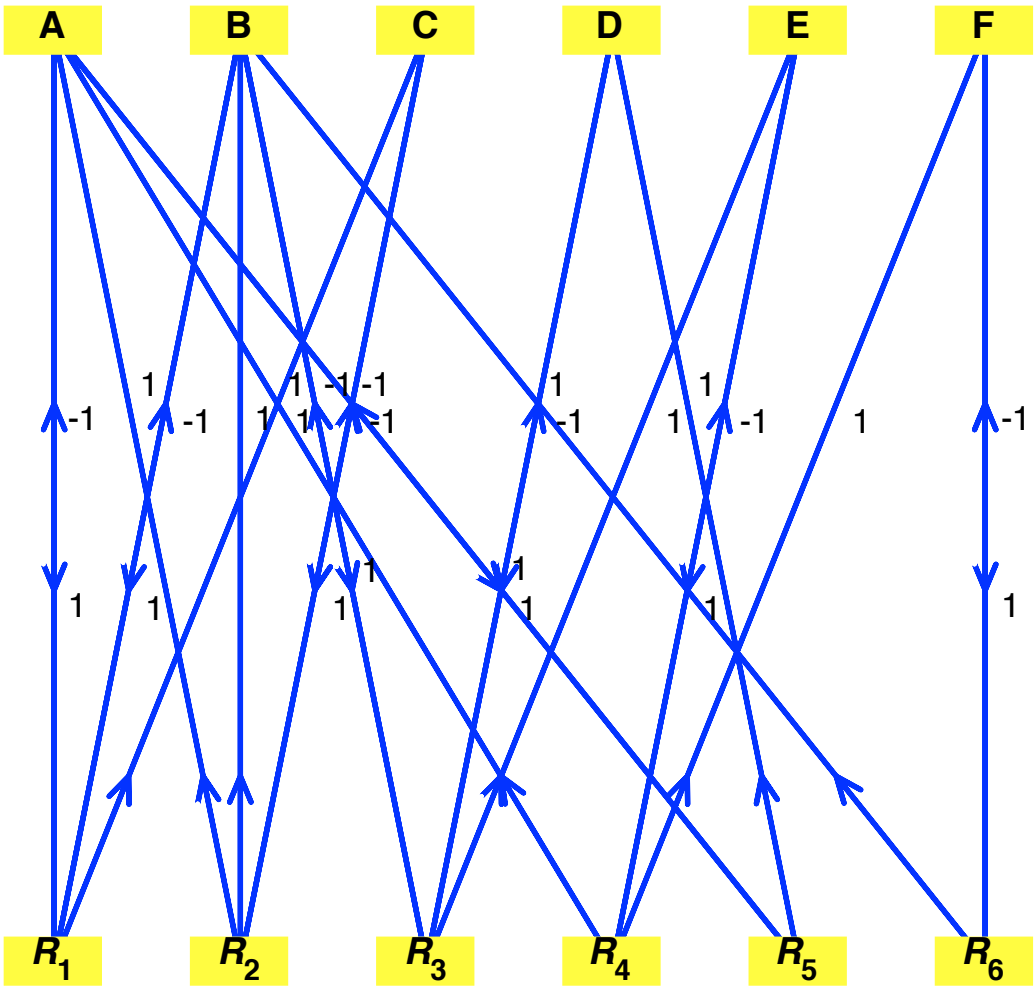
Step 1.
Create the DSR-graph and find if there is a positive feedback loop through the competition.

```
> mylabels := ["A", "B", "C", "D", "E", "F", seq(R[i], i = 1..6)]:
> findedgesDSR(mylabels, A)
```

$$\{[["A", R_1], 1], [["A", R_5], 1], [["B", R_1], 1], [["B", R_3], 1], [["C", R_2], 1], [["D", R_3], 1], [["E", R_4], 1], [["F", R_6], 1], [[R_1, "A"], -1], [[R_1, "B"], -1], [[R_1, "C"], 1], [[R_2, "A"], 1], [[R_2, "B"], 1], [[R_2, "C"], -1], [[R_3, "B"], -1], [[R_3, "D"], -1], [[R_3, "E"], 1], [[R_4, "A"], 1], [[R_4, "E"], -1], [[R_4, "F"], 1], [[R_5, "A"], -1], [[R_5, "D"], 1], [[R_6, "B"], 1], [[R_6, "F"], -1]\} \quad \textbf{(2.2)}$$

If one wants to draw the graph, then one does:

```
> G := createDSRgraphsigned(mylabels, A, findZ(A)):
> DrawGraph(G)
```



Step 2

Injectivity test:

```
> isinjective(A)
```

$$0 \qquad \textbf{(2.3)}$$

Because the answer is 0, we apply the next test.

```
> isinjectiveextended(A)
```

$$2 \qquad \textbf{(2.4)}$$

Because the answer is 2, we classify the network as MULTISTATIONARY.

Step 3

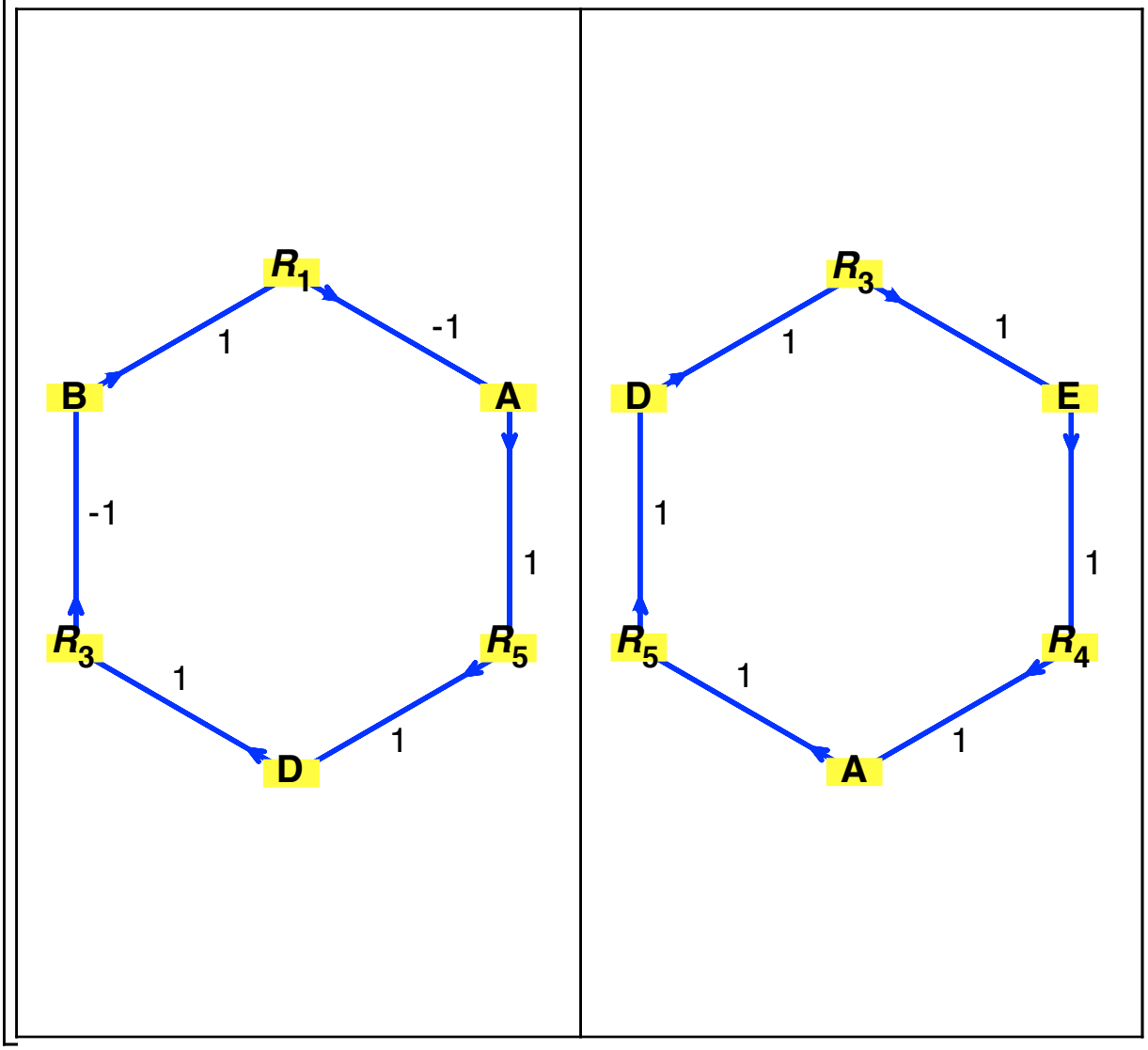
For the networks that are multistationary, decide whether multistationarity can be attributed to competition, and find the positive feedback loops underlying multistationarity.

```
> Z := findZ(A) : n := Dimension(A)[1] : m := Dimension(A)[2] : s := Rank(A) :
> selected := findloops( )
```

$$selected := [[[[R_1, S_1], -1], [[S_1, R_5], 1], [[R_5, S_4], 1], [[S_4, R_3], 1], [[R_3, S_2], -1], [[S_2, R_1], 1]], [[[R_3, S_5], 1], [[S_5, R_4], 1], [[R_4, S_1], 1], [[S_1, R_5], 1], [[R_5, S_4], 1], [[S_4, R_3], 1]]] \quad (2.5)$$

These are the positive feedback loops. You can draw the loops after giving a label to the species:

```
> speciesord := ["A", "B", "C", "D", "E", "F"] :
> drawloops(selected, speciesord)
```



Example 2

A+B< -> C
D+B->G ->D+F
A->D
G->C
F->B

The stoichiometric matrix A is:

```
> A := Transpose(Matrix([[-1,-1,1,0,0,0],-[-1,-1,1,0,0,0],[0,-1,0,-1,0,1],[0,0,
      0,1,1,-1],[-1,0,0,1,0,0],[0,0,1,0,0,-1],[0,1,0,0,-1,0]]));
```

$$A := \begin{bmatrix} -1 & 1 & 0 & 0 & -1 & 0 & 0 \\ -1 & 1 & -1 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 & -1 & 0 \end{bmatrix} \tag{3.1}$$

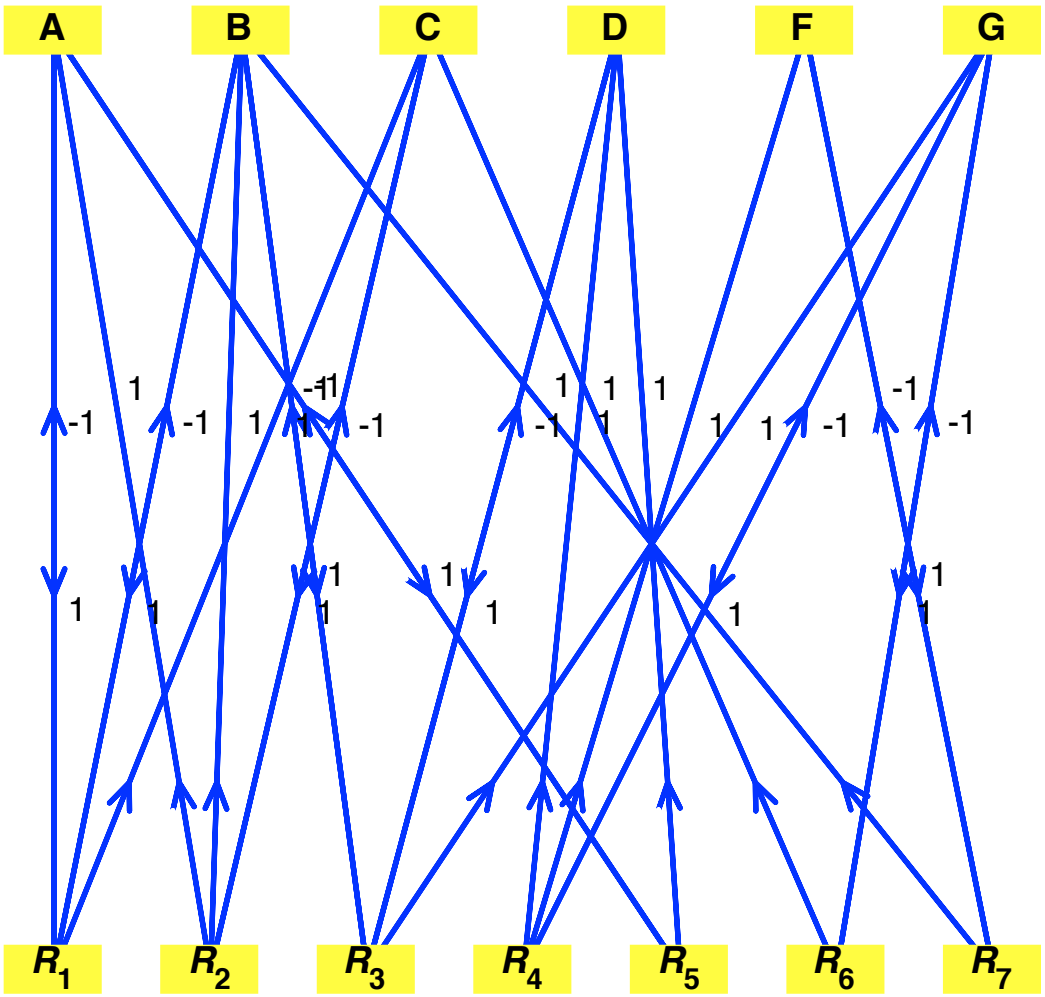
Step 1.
Create the DSR-graph and find if there is a positive feedback loop through the competition.

```
> mylabels := ["A","B","C","D","F","G",seq(R[i],i=1..7)]:
> findedgesDSR(mylabels,A)
```

$$\{[["A",R_1],1],[["A",R_5],1],[["B",R_1],1],[["B",R_3],1],[["C",R_2],1],[["D",R_3],1],[["F",R_7],1],[["G",R_4],1],[["G",R_6],1],[[R_1,"A"],-1],[[R_1,"B"],-1],[[R_1,"C"],1],[[R_2,"A"],1],[[R_2,"B"],1],[[R_2,"C"],-1],[[R_3,"B"],-1],[[R_3,"D"],-1],[[R_3,"G"],1],[[R_4,"D"],1],[[R_4,"F"],1],[[R_4,"G"],-1],[[R_5,"A"],-1],[[R_5,"D"],1],[[R_6,"C"],1],[[R_6,"G"],-1],[[R_7,"B"],1],[[R_7,"F"],-1]\} \tag{3.2}$$

If one wants to draw the graph, then one does:

```
> G := createDSRgraphsigned(mylabels,A,findZ(A)):
> DrawGraph(G)
```



Step 2

Injectivity test:

```
> isinjective(A)
```

$$0 \qquad \textbf{(3.3)}$$

Because the answer is 0, we apply the next test.

```
> isinjectiveextended(A)
```

$$0 \qquad \textbf{(3.4)}$$

Because the answer is 0, you should use the toolbox.
The toolbox says YES, the network admits multiple steady states

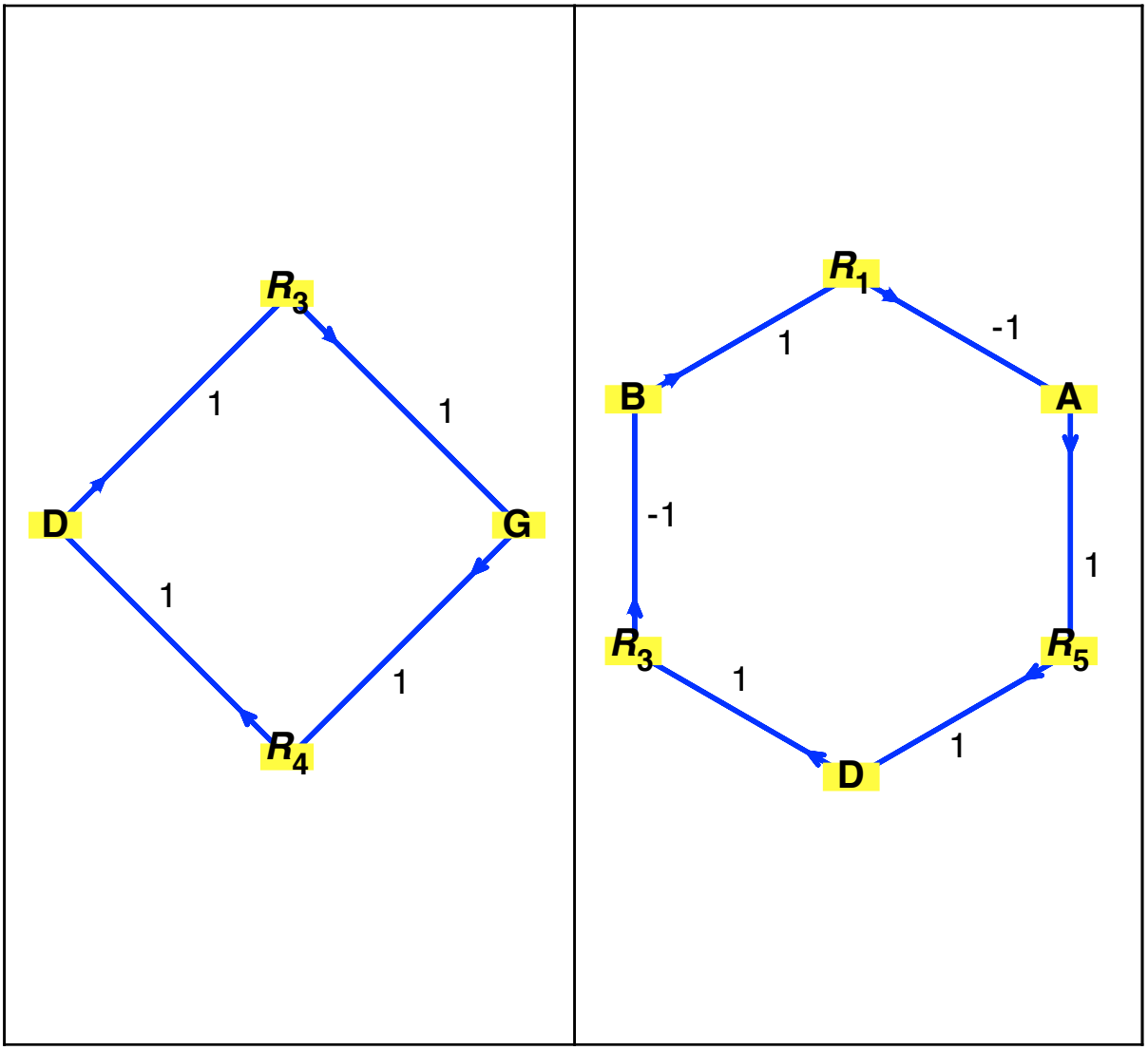
Step 3
We find the positive feedback loops.

```
> Z := findZ(A) : n := Dimension(A)[1] : m := Dimension(A)[2] : s := Rank(A) :
> selected := findloops( )
```

$$selected := [[[[R_3, S_6], 1], [[S_6, R_4], 1], [[R_4, S_4], 1], [[S_4, R_3], 1]], [[[R_1, S_1], -1], [[S_1, R_5], 1], [[R_5, S_4], 1], [[S_4, R_3], 1], [[R_3, S_2], -1], [[S_2, R_1], 1]]] \quad (3.5)$$

These are the positive feedback loops. You can draw the loops after giving a label to the species:

```
> speciesord := ["A", "B", "C", "D", "F", "G"] :
> drawloops(selected, speciesord)
```



Example 3

Consider the network
A+B< -> C

D+B->E ->A+F
A->D
F->B

The stoichiometric matrix A is:

```
> A := Transpose(Matrix([[-1,-1,1,0,0,0],-[-1,-1,1,0,0,0],[0,-1,0,-1,1,0],[1,0,
    0,0,-1,1],[-1,0,0,1,0,0],[0,1,0,0,0,-1]]));
```

$$A := \begin{bmatrix} -1 & 1 & 0 & 1 & -1 & 0 \\ -1 & 1 & -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix} \quad \textbf{(4.1)}$$

Step 1.
Create the DSR-graph and find if there is a positive feedback loop through the competition.

```
> mylabels := ["A","B","C","D","E","F",seq(R[i],i=1..6)]:
> findedgesDSR(mylabels,A)
```

$$\{[["A",R_1],1],[["A",R_5],1],[["B",R_1],1],[["B",R_3],1],[["C",R_2],1],[["D",R_3],1],[["E",R_4],1],[["F",R_6],1],[[R_1,"A"],-1],[[R_1,"B"],-1],[[R_1,"C"],1],[[R_2,"A"],1],[[R_2,"B"],1],[[R_2,"C"],-1],[[R_3,"B"],-1],[[R_3,"D"],-1],[[R_3,"E"],1],[[R_4,"A"],1],[[R_4,"E"],-1],[[R_4,"F"],1],[[R_5,"A"],-1],[[R_5,"D"],1],[[R_6,"B"],1],[[R_6,"F"],-1]\} \quad \textbf{(4.2)}$$

If one wants to draw the graph, then one does:

```
> G := createDSRgraphsigned(mylabels,A,findZ(A)):
> DrawGraph(G)
```

Step 2

Injectivity test:

```
> isinjective(A)
```

$$0 \quad \textbf{(4.3)}$$

Because the answer is 0, we apply the next test.

```
> isinjectiveextended(A)
```

$$2 \qquad \textbf{(4.4)}$$

Because the answer is 2, we classify the network as MULTISTATIONARY.

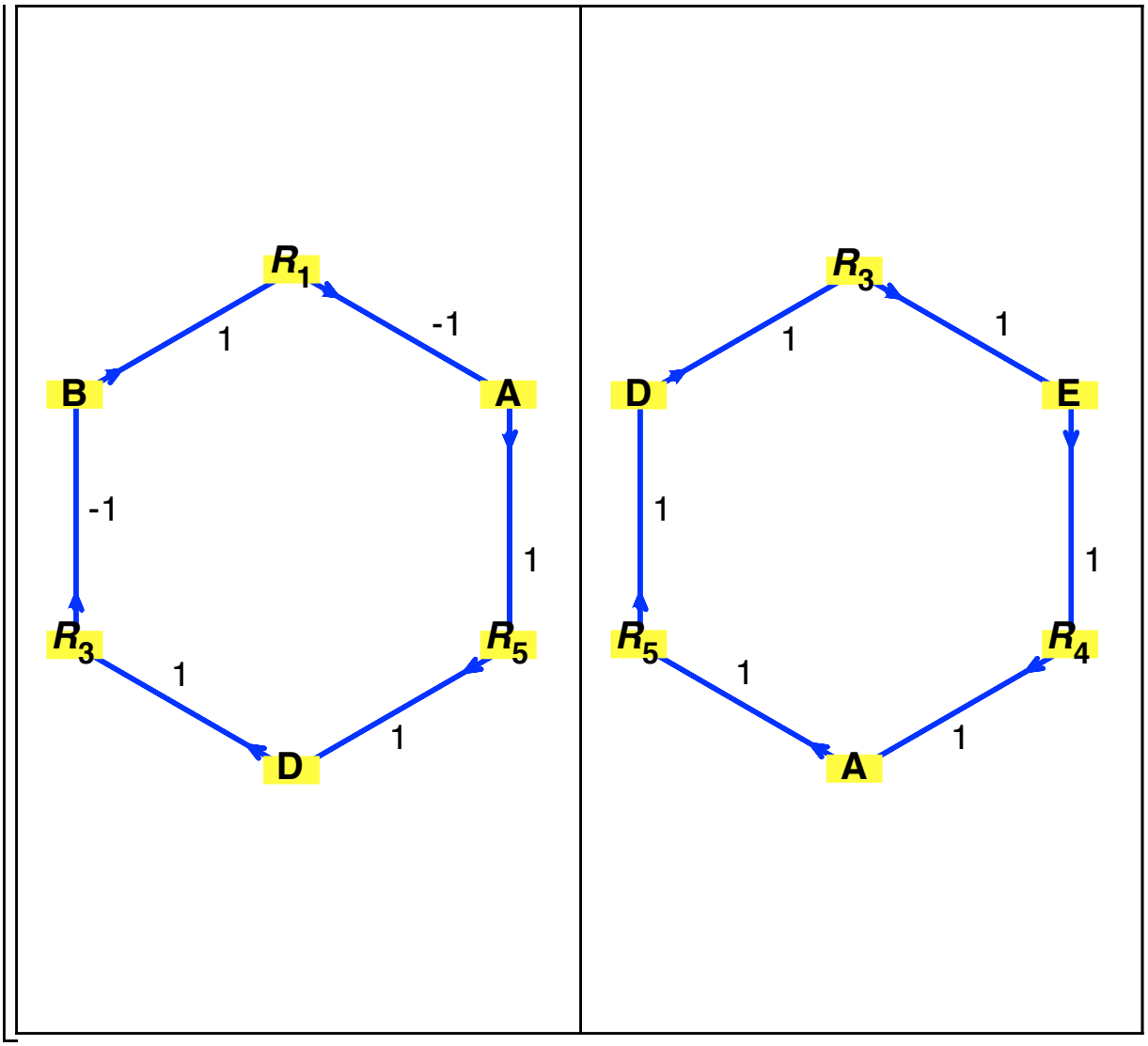
Step 3
For the networks that are multistationary, decide whether multistationarity can be attributed to competition, and find the positive feedback loops underlying multistationarity.

```
> Z := findZ(A) : n := Dimension(A)[1] : m := Dimension(A)[2] : s := Rank(A) :
> selected := findloops( )
```

$$selected := [[[[R_1, S_1], -1], [[S_1, R_5], 1], [[R_5, S_4], 1], [[S_4, R_3], 1], [[R_3, S_2], -1], [[S_2, R_1], 1]], [[[R_3, S_5], 1], [[S_5, R_4], 1], [[R_4, S_1], 1], [[S_1, R_5], 1], [[R_5, S_4], 1], [[S_4, R_3], 1]]] \qquad \textbf{(4.5)}$$

These are the positive feedback loops. You can draw the loops after giving a label to the species:

```
> speciesord := ["A", "B", "C", "D", "E", "F"] :
> drawloops(selected, speciesord)
```




```
> tA := ImportMatrix("enumeration/4species/testNoninjective_13_35.csv")
```

$$tA := \begin{bmatrix} 1 & 0 & -1 & -1 \\ -1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & -1 & 0 & 0 \end{bmatrix} \tag{4.6}$$

```
> isinjectiveextended(tA)
Error, (in gaussinj) invalid input: LinearAlgebra:-Dimension
expects its 1st argument, A, to be of type {Matrix, Vector} or
coercible via `~Simplify`, but received []
>
```