

Appendix 1.

Notation and concepts

We define an ecological network as a directed graph (digraph) $G(V,E)$ with V vertices, the species/trophospecies or nutrient pools, and E oriented, weighted edges, the trophic exchanges, usually measured as grams of C for hectare for year (or other currencies, according to the aim of the research).

We represent the digraph as an adjacency matrix, that is to say a squared $V \times V$ matrix where every coefficient represents the flux of matter from row-vertex to column one. If the edge does not exist the coefficient will be 0.

A strongly connected component (SCC) of the graph is a maximal subset of vertices set $U \subseteq V$ where for any $u_1, u_2 \in U$, there is at least a directed path (sequence of vertices and edges) from u_1 to u_2 and one from u_2 to u_1 , so that u_1 and u_2 are reachable from each other. The vertices of any directed graph can be partitioned into n strongly connected components S_1, \dots, S_n where $V \geq n$. Strongly connected components do not share vertices with each other (disjoint sets).

The size of a strongly connected component $|S_i|$ is the number of vertices belonging to the i th of these components. A SCC with $|S_i|=1$ can be only a source (a vertex with just outgoing edges), a sink (just incoming edges) or a cross-vertex, that is a node connecting two or more SCCs. If the size of a strongly connected component is equal to the total number of nodes, $|S_i|=V$, the graph is said to be strongly connected. If a digraph encompasses more than one strongly connected component, it will have no Hamiltonian cycles, that is cycles that visit all the vertices in the graph; this is because cycles involving vertices in different SCC are not possible (Bang-Jensen and Gutin 2000); in fact all the cycles in the graph remain confined into its strongly connected components.

Aggregating vertices inside every SCC yields a directed acyclic graph (DAG). Because a DAG comprises only one-way edges, it is a pictorial representation of a hierarchy of dependencies involving graph components. For ecological flow networks such hierarchy pertains the transfer of mass/energy throughout the system.

According to the hierarchical nature of DAGs, a procedure called topological sorting can be applied so that one can order graph elements according to precedence constraints. Thus, given a directed acyclic graph $G'(V',E')$, a topological sort is a linear ordering of all its vertices from left to right, such as for every edge $(e_{v_1,v_2} \in E')$, v_1 precedes v_2 in the order of vertices. A topological sort must include at least a source and a sink.

Finding SCCs

In order to find SCCs in a digraph we should perform two depth first search – DFS visits: the first one on the digraph, the second

one on its transposed form, that is to say with all the edges with inverted directions. Exploring a graph in search of SCCs requires a calculation effort that is linearly proportional to the graph size, considering vertices and edges ($\Theta(V+E)$) (Tarjan 1972). The simplest algorithm for SCCs (Hopcroft and Tarjan 1973) has been used here. See below for a ready to use code.

Aggregating SCCs

Once obtained the SCCs, we should lump together all the vertices belonging to the same component. We utilized the algorithm proposed by Hirata (1978) that has been extended in Ulanowicz and Kemp (1979) work. Because of in our analysis we need just topological values (presence/absence of edges), the aggregation procedure is simplified. We may describe this process as an homomorphic mapping ϕ from digraph $G(V,E)$ to $G'(V',E')$ (Hirata and Ulanowicz, 1985), that is to say a surjective map of the graph. The aggregation process causes self-loops to be created: these self-loops (diagonal coefficients on the adjacency matrix) have been eliminated before running the topological sorting.

Topological sorting

Topological sorting orders, in linear time (Knuth 1997), the vertices and edges of a DAG in a simple and consistent way; hence, in the framework of ecological flow networks, it makes evident the fundamental linear pathways that energy and matter use to travel in the ecosystem. In other words, topological sort produces a chain of donors (such as resources) and consumers in a succession of steps that highlights the linear fundamental flux of matter, which enters the system and leaves it, after a certain time that is affected by cycling.

Programming SCCs, DAG and topological sort

Here below a complete software in C that accomplishes all the procedures performed in the paper is presented. The code requires GSL (GNU Scientific Library, <www.gnu.org/software/gsl>).

```

/*
-----
-- Strongly Connected -----
-- this is a sample code -----
-- allesina@msu.edu -----
-----
INPUT: Matrix File
OUTPUT: SHELL or FILE
-----
COMPILING:
gcc scc.c -o SCC -lgsl -lgslcblas -O2
-----
RUNNING:
./SCC test.txt
test.txt contains
1st line: nubmer of compartments --> N
2nd line...N+1th line Adjacency matrix
-----

// Libraries must include GSL
// http://sources.redhat.com/gsl/
*/
#include <stdio.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_permutation.h>
#include <gsl/gsl_sort_vector_double.h>

// Global Variables
int N; // number of vertices
gsl_matrix * M; // the adjacency matrix
//-----
// VARS STRONGLY CONNECTED COMPONENTS
//-----
gsl_vector * DFSC; // colour
gsl_vector * SCCS; // belonging SCC
gsl_vector * DFSD; // Discovery
gsl_vector * DFSF; // Finishing
gsl_permutation * DFSOrder;
int DFS_T; // Time
int SCC_Num; // SCC count
//-----
// VARS AGGREGATION
//-----
gsl_matrix * Agg; // the aggregation matrix
gsl_matrix * DAG; // Resulting DAG once the ScCs have been aggregated
//-----
// INPUT READING
//-----
int ReadFile(char * myfile)
{
    char *line = NULL;
    FILE *fp;
    int i,j,tmp;
    printf ("\nOpening: %s\n", myfile);
    if ((fp = fopen (myfile, "r")) == NULL)
    {
        printf ("Cannot open the file!!!\n");
        return 1;
    }
    fscanf(fp,"%d\n",&N);
    printf("Number of vertices: %d\n",N);
    M = gsl_matrix_calloc(N,N);
    for (i=0;i<N;i++)
    {
        for (j=0;j<N;j++)
        {
            fscanf(fp,"%d",&tmp);
            gsl_matrix_set(M,i,j,tmp);
        }
        fscanf(fp,"\n");
    }
    // print the adjacency matrix
    printf("\nAdjacency matrix\n");
}

```

```

    PrintMat(M);
    return 0;
}
//-----
// PRINT A MATRIX
//-----
int PrintMat(gsl_matrix * C)
{
    int i,j;
    for (i=0;i<N;i++)
    {
        for (j=0;j<N;j++)
        {
            printf("%1.0f\t", gsl_matrix_get(C,i,j));
        }
        printf("\n");
    }
    return 0;
}
//-----
// AGGREGATION INTO DAG
//-----
int AggMat(void)
{
    gsl_matrix * M1;
    gsl_matrix * M2;
    int i,j,k;
    double z;
    M1=gsl_matrix_calloc(N,N);
    M2=gsl_matrix_calloc(N,N);
    for(i=0;i<N;i++)
    {
        for (j=0;j<N;j++)
        {
            z=0;
            for (k=0;k<N;k++)
            {
                z=z+gsl_matrix_get(Agg,i,k)*gsl_matrix_get(M,k,j);
            }
            if (z>0)
            {
                z=1;
            }
            gsl_matrix_set(M1,i,j,z);
        }
    }
    for(i=0;i<N;i++)
    {
        for (j=0;j<N;j++)
        {
            z=0;
            for (k=0;k<N;k++)
            {
                z=z+gsl_matrix_get(M1,i,k)*gsl_matrix_get(Agg,j,k);
            }
            if (z>0)
            {
                z=1;
            }
            gsl_matrix_set(M2,i,j,z);
        }
    }
    // Remove Self Loops
    for(i=0;i<N;i++)
    {
        if (gsl_matrix_get(M2,i,i)==1)
        {
            gsl_matrix_set(M2,i,i,0);
        }
    }
    // Build Dag
    DAG=gsl_matrix_calloc(SCC_Num,SCC_Num);
    for(i=0;i<SCC_Num;i++)
    {
        for(j=0;j<SCC_Num;j++)
        {
            gsl_matrix_set(DAG,i,j,gsl_matrix_get(M2,i,j));
        }
    }
}

```

```

    }
    return 0;
}
//-----
// STRONGLY CONNECTED COMPONENTS
//-----
int DFS (int i,int tras)
{
    int k;
    DFS_T++;
    gsl_vector_set(DFSC,i,1);
    gsl_vector_set(DFSD,i,DFS_T);
    if (tras==1)
    {
        gsl_vector_set(SCCS,i,SCC_Num);
    }
    for (k=0;k<N;k++)
    {
        if(gsl_matrix_get(M,i,k)>0)
        {
            if (gsl_vector_get(DFSC,k)==0)
            {
                DFS(k,tras);
            }
        }
    }
    DFS_T++;
    if (tras ==0)
    {
        gsl_vector_set(DFSF,i,-DFS_T); //reverse
    }
    gsl_vector_set(DFSC,i,2);
    return 0;
}
int SCC (void)
{
    int i,l;
    printf ("\nSTRONGLY CONNECTED COMPONENTS\n");
    SCC_Num=0;
    DFSC= gsl_vector_calloc(N);
    DFSD=  gsl_vector_calloc(N);
    DFSF=  gsl_vector_calloc(N);
    SCCS=  gsl_vector_calloc(N);
    DFSOrder = gsl_permutation_calloc(N);
    DFS_T=0;
    for (i=0; i<(N); i++)
    {
        if (gsl_vector_get(DFSC,i)==0)
        {
            DFS(i,0);
        }
    }
    // order the nodes
    gsl_sort_vector_index(DFSOrder, DFSF);
    // repeat DFS search on the transpose ordering nodes by time
    gsl_matrix_transpose(M);
    // re initialize vectors
    DFSC=gsl_vector_calloc(N);
    DFSD=gsl_vector_calloc(N);
    DFS_T=0;
    // starts DFS
    for (i=0; i<(N); i++)
    {
        if (gsl_vector_get(DFSC,gsl_permutation_get(DFSOrder,i))==0)
        {
            DFS(gsl_permutation_get(DFSOrder,i),1);
            SCC_Num++;
        }
    }
    // Allocate Aggregation Matrix
    Agg=gsl_matrix_calloc(N,N);
    // lists the SCCS
    for (i=0; i<(SCC_Num); i++)
    {
        printf ("\nStrongly Connected Component #%d\n",i);
        for (l=0;l<(N);l++)

```

```

        {
            if (gsl_vector_get(SCCS,1)==i)
            {
                printf("%d\t",1);
                gsl_matrix_set(Agg,i,1,1);
            }
        }
    }
    // re-transpose the matrix
    gsl_matrix_transpose(M);
    printf ("\nNumber of Components:%d\n",SCC_Num);
    return 0;
}
//-----
// TOPOLOGICAL SORT
//-----
int Toposort(int i)
{
    int k,z;
    DFS_T++;
    gsl_vector_set(DFSC,i,1);
    gsl_vector_set(DFSD,i,DFS_T);
    for (k=0;k<SCC_Num;k++)
    {
        if(gsl_matrix_get(DAG,i,k)>0)
        {
            if (gsl_vector_get(DFSC,k)==0)
            {
                Toposort(k);
            }
        }
    }
    DFS_T++;
    gsl_vector_set(DFSC,i,2);
    //Print the component
    printf("[");
    for(z=0;z<N;z++)
    {
        if(gsl_vector_get(SCCS,z)==i)
        {
            printf("%d ",z);
        }
    }
    printf("]<--");
    return 0;
}
//-----
// MAIN
//-----
int main(int argc, char *argv[])
{
    int i,j;
    // reads the command line: filename.txt
    char *myfile;
    myfile = argv[1];
    printf("\nSelected file: %s\n", myfile);
    // reads the File
    j=ReadFile(myfile);
    // -----
    // start
    // -----
    // Strongly Connected Components
    j=SCC();
    // Aggregation into DAG
    printf("\nAGGREGATION MATRIX\n");
    PrintMat(Agg);
    // aggregate the components
    j=AggMat();
    printf("\nResulting Directed Acyclic Graph\n");
    for (i=0;i<SCC_Num;i++)
    {
        for (j=0;j<SCC_Num;j++)
        {
            printf("%1.0f\t", gsl_matrix_get(DAG,i,j));
        }
        printf("\n");
    }
}

```

```

// Topological Sort
DFSC= gsl_vector_calloc(SCC_Num);
DFSD=  gsl_vector_calloc(SCC_Num);
DFSF=  gsl_vector_calloc(SCC_Num);
DFS_T=0;
printf ("\n TOPOLOGICAL SORTING\n");
for (i=0; i<SCC_Num; i++)
{
    if (gsl_vector_get(DFSC,i)==0)
        {
            Toposort(i);
        }
}
printf("\n\n");
// -----
// stop
// -----
return 0;
}

```