METANET : a system for network problems study

Claude Gomez, Maurice Goursat

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 124

Programme 5
Automatique, Productique,
Traitement du Signal et des Données

METANET : A SYSTEM FOR NETWORK PROBLEMS STUDY

Claude GOMEZ
Maurice GOURSAT

Novembre 1990

RT_0124

METANET: a system for network problems study

METANET : un système pour l'étude des problèmes de réseaux

Claude Gomez Maurice Goursat

INRIA
Domaine de Voluceau
Rocquencourt - BP105
78153 Le Chesnay Cedex
FRANCE

Abstract

METANET is a software written in C++ and using X Window graphics for graph and network problems studies. It manages a library of FORTRAN programs in which the algorithms are encoded. A number of programs solving classical graph problems and minimal cost flow network problems are resident in METANET. Tools are given for generating new graphs and for graphically handling graphs. METANET includes BASILE which is a CACSD system for automatic control providing the user with a comprehensive language and interpreter for manipulating graph objects. METANET is an open system where the user can add its own programs.

Résumé

METANET est un logiciel pour la résolution des problèmes de réseaux et l'étude de nouvelles méthodes. Il est écrit en C++ et utilise l'environnement X Window. Les algorithmes résidents dans METANET sont écrits en FORTRAN. METANET fournit des outils graphiques pour la manipulation et la génération des graphes. METANET inclut BASILE, système de CAO pour l'automatique. Il dispose donc de son interpréteur et de son langage ainsi que de toutes ses fonctionalités, permettant ainsi de manipuler les objets calculés sur les réseaux. METANET est un système ouvert et l'utilisateur peut y inclure ses propres programmes.

1 Introduction

METANET is a system for network problems study. The aim of METANET is twofold: to use various solution methods and to try new methods.

METANET works on a computer under a UNIX system with X Window.

We use C [5] and C++ [6]. In fact, we use the GNU compilers **gcc** [13] and **g++** [14] made by Free Software Foundation. But you can use standard C and C++ compilers.

Graphics are of major interest for network problems and are have been developed in METANET by using the X Window System Version 11 Release 4 [11] with the Athena Widgets library [12].

The BASILE system ([1]) which is a CACSD system for automatic control developed at INRIA is included in METANET. This provides the user with a comprehensive language and interpreter together with the BASILE functionalities (see section 5.3).

METANET is an open system. You can add your own programs in METANET for testing new methods (see section 6.2).

On the computer, the location of the system is a directory called **metanet**. When we speak about a file in the file system of the computer, its path is relative to **metanet** directory.

2 Graph and network problems

We describe in this section the graph and network problems METANET can solve. The menus and functions used to solve them are described in sections 5.2.2 and 5.3.1.

2.1 Graph problems

METANET can directly solve the following graph problems:

- computing the connected and strong connected components of a graph.
- computing shortest paths from one node to another. There are two problems. The first one is to find a path compound by the minimum number of arcs between two nodes (arc number shortest path). The second one is to find a path of minimum length between two nodes (length shortest path). The arc lengths are given.
- searching for a circuit in a directed graph.
- computing the minimal weight spanning tree with a given node as its root. For that, to each arc can be given a weight. The total weight of this spanning tree, if it exists, is the sum of the weight of the arcs of the tree. This number must be minimum.

2.2 Flow problems

We consider a connected digraph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ where $\mathcal{N}$ and $\mathcal{A}$ are respectively the sets of nodes and arcs. There are n nodes and m arcs. With each node i is associated a real number b_i which can be positive (supply, source node), negative (demand, sink node) or zero (pure transshipment node).

A flow on $\mathcal{G}$ is a vector x having a value on each arc and verifying the first Kirchhoff law:

$$\sum_{i:(i,j)\in\mathcal{A}} x_{i,j} - \sum_{j:(j,i)\in\mathcal{A}} x_{j,i} = b_i \qquad \forall i \in \mathcal{N}$$

With each arc (i, j) (from node i to node j) are associated two numbers $l_{i,j}$ and $u_{i,j}$ which are respectively the lower and upper bounds for the flow. Moreover a cost $c_{i,j}$ is associated with each arc.

METANET can compute the maximum capacity path from node i to node j. It is a path for which the minimum capacity of all the arcs is maximum.

The general minimum linear cost network flow problem consists in finding a flow which minimizes the linear cost on the arcs and which verifies the capacity constraints. We call it problem $(\mathcal{P})$:

$$(\mathcal{P}) \quad \begin{cases} \min \sum_{(i,j)\in\mathcal{A}} c_{i,j}x_{i,j} \\ \sum_{i:(i,j)\in\mathcal{A}} x_{i,j} - \sum_{j:(j,i)\in\mathcal{A}} x_{j,i} = b_i \qquad \forall i \in \mathcal{N} \\ l_{i,j} \leq x_{i,j} \leq u_{i,j} \qquad \forall (i,j) \in \mathcal{A} \end{cases}$$

METANET can solve problem $(\mathcal{P})$and various problems related to problem $(\mathcal{P})$we describe below.

If we suppose that vector b is equal to 0, i.e. there are only pure transshipment nodes, we obtain problem $(\mathcal{P}_1)$.

Moreover, we can consider that the graph has only one source and one sink. If the graph has more than one source and one sink, it is possible to link all the sources to a single super source and to link all the sinks to a single super sink (this is the augmented graph).

With theses assumptions we can compute the maximum flow from the source to the sink according to the flow bounds on the arcs. This is problem $(\mathcal{P}_m)$. The value of this flow is called the value of the flow on the return arc (virtual arc from sink to node).

With theses assumptions, we can also impose the value of the flow on the return arc and solve problem $(\mathcal{P}_1)$for this flow value. This is problem $(\mathcal{P}_2)$.

All these flow problems are linear. We can consider a minimum quadratic cost network flow problem. We call it problem $(\mathcal{Q})$:

$$(\mathcal{Q}) \quad \begin{cases} \min \sum_{(i,j)\in\mathcal{A}} \Gamma_{i,j}(x_{i,j}) \qquad with \quad \Gamma_{i,j}(x_{i,j}) = \frac{1}{2} w_{i,j}[(x_{i,j} - \bar{x}_{i,j})^2] \\ \sum_{i:(i,j)\in\mathcal{A}} x_{i,j} - \sum_{j:(j,i)\in\mathcal{A}} x_{j,i} = b_i \qquad \forall i \in \mathcal{N} \\ l_{i,j} \leq x_{i,j} \leq u_{i,j} \qquad \forall (i.j) \in \mathcal{A} \end{cases}$$

where $w_{i,j}$ and $\bar{x}_{i,j}$ are given.

If we suppose that the vector b is equal to 0, i.e. there are only pure transshipment nodes, we obtain problem $(\mathcal{Q}_1)$.

3 Data structures

3.1 Variables

In a network problem, there are many variables denoting graph theory objects (trees, number of arcs, tail node of arc ...), physical flow objects (source, pressure, supply ...), computer science objects (pointer, error flag ...) together with mathematical objects (matrix line, coordinates, maximum ...). The number of variables is very high (a few hundred !). In METANET, we use many FORTRAN subroutines written by many various people who use their own names for these variables. Moreover, a variable name cannot have more than 6 letters in FORTRAN and it is often difficult to find the meaning of a variable from its name.

To get rid of these taxinomy problems which are not obvious problems (see Linné), we have adopted the following method. Each variable is characterized by its *definition*. This definition is

unique and is a string (a kind of object every high level language knows). This definition corresponds to an *abstract* variable.

These variables are global variables in METANET.

The definitions of the abstract variables used so far are given below.

```
# of a node of a circuit
# of arc for starting the search of a cycle
# of node to be labelled
# of the pseudo node to be expanded
arc array for adjacency description for directed graph
arc array for adjacency description for undirected graph
arc array for adjacency description of the reduced graph
arc array for adjacency description of the transitive closure
arc number shortest path from i to j
arc number shortest path length from i to j
arc type array
array of arc number shortest path length from i0 to node .
circuit
circuit logical
connected component # of node .
connecting edge # of node .
discretization step
end variable
error variable
eulerian chain array
head node array
integer demand node array
integer flow value on return arc
integer imposed value for the flow
integer maximal flow edge array
integer maximum capacity edge array
integer minimal cost flow edge array
integer minimal cost flow edge array (relax)
integer minimal cost imposed flow edge array
integer minimum capacity edge array
integer minimum cost value
integer quadratic cost function origin edge array
integer unitary cost edge array
integer working array for edges
integer working array for nodes
integer working array for pseudo-nodes
integer working variable
logical for searching circuit
logical working array for nodes
matrix of arc number shortest path lengths
matrix of previous node on arc number shortest paths
matrix of previous node on max capacity paths
matrix of previous node on shortest paths
max capacity path from i to j
```

maximum number of arcs
maximum number of arcs for undirected graph
maximum number of arcs of the transitive closure
maximum number of arcs of the transitive closure of reduced graph
maximum number of edges
maximum number of edges + 1
maximum number of nodes
maximum number of nodes + 1
maximum number of pseudo-nodes
minimal cost flow error variable
minimal cost imposed flow error variable
minimal quadratic cost flow error variable
minimal weight tree with root i0
modif logical
negative length circuit logical
no tree with root i0
node and pseudo node number
node array for adjacency description for directed graph
node array for adjacency description for undirected graph
node array for adjacency description of the transitive closure
node set of connected component i
node set of strong connected component i
number of arcs
number of arcs for undirected graph
number of arcs of the reduced graph
number of connected components
number of edges
number of edges + 1
number of nodes
number of nodes + 1
number of nodes of the reduced graph
number of strong connected components
odd cycle giving a pseudo node
origin for labelling
pointer array for adjacency description for directed graph
pointer array for adjacency description for undirected graph
pointer array for adjacency description of the reduced graph
pointer array for adjacency description of the transitive closure
precision for minimum quadratic cost flow (2**precision)
previous node of . in a circuit
previous node of . in minimal weight tree with root i0
previous node of . on a arc number shortest path from i0
previous node of . on a shortest path from i0
previous node of . on max capacity path from i0
primal variable array
rank function
real array of max capacity path value from i0 to node .
real array of shortest path length from i0 to node .

```
real length edge array
real matrix of arc lengths
real matrix of max capacity path values
real matrix of shortest path lengths
real max capacity path value from i to j
real maximum capacity edge array
real maximum weight matching value
real minimal quadratic cost flow edge array
real quadratic cost function weight arc array
real quadratic cost function weight edge array
real shortest path length from i to j
real unitary cost edge array
real weight arc array
real weight edge array
real working array for nodes
real working array for pseudo-nodes
shortest path from i to j
sink node #
source node #
starting node # i0
strong connected component # of node .
tail node array
value for infinity
variable for connectivity
```

The abstract variables are defined in an unique file as **graph_variable** structures (see section 3.2.

3.2 C++ Objects

Each abstract variable is described by a C++ structure called **graph_variable**:

```
enum ftype {FLOGICAL, FINT, FREAL, FDOUBLE, FINT2, FREAL2, FDOUBLE2};

struct graph_variable {
  int id;
  char* definition;
  char* description;
  int dim;
  int* dimensions_id;
  int value;
  int object;
  ftype type;
  union {
    flogical* l;
    fint* i;
    freal* r;
    fdouble* d;
    fint2* i2;
```

```
    freal2* r2;
    fdouble2* d2;
  };
  void (*compute) (graph_variable*);
  void (*display) (graph_variable*);
};
```

`id` is a unique integer which characterizes the variable. It is its identifier. This number is used in internal computations.

`definition` is the definition of the variable (see section 3.1).

`description` is the complete description of the variable.

`dim` is the number of dimensions of the variable, which can be 0 if it is a scalar.

`dimensions_id` is a vector with `dim` elements representing the identifiers of other scalar variables giving the number of dimensions.

`value` is 1 if a value has been assigned to the variable and 0 otherwise.

`object` is 1 if an object has been created for the variable (i.e. memory has been allocated for it) and 0 otherwise.

`type` is the type of the variable which can be:

- `FLOGICAL`: FORTRAN logical scalar or vector
- `FINT`: FORTRAN integer scalar or vector
- `FREAL`: FORTRAN real scalar or vector
- `FDOUBLE`: FORTRAN double precision scalar or vector
- `FINT2`: FORTRAN integer matrix
- `FREAL2`: FORTRAN real matrix
- `FDOUBLE2`: FORTRAN double precision matrix

`l`, `i`, `r`, `d`, `i2`, `r2` or `d2` is a pointer to the object containing the value of the variable.

`compute` is the address of a function which must be applied to the variable to compute its value.

`display` is the address of a function which must be applied to the variable to (possibly graphically) display the value of the variable.

Variables can be used by FORTRAN subroutines or by internal C functions of METANET. For coherency, the types of the variables used in C functions are the same as the ones used in FORTRAN subroutines in order to be able to mix them.

To understand the meaning of the various slots, we give below a description of the various cases which can appear in practice. We describe what to do with the slots according to the use of the variable in a FORTRAN subroutine or in a C function.

Output variable from a FORTRAN subroutine or from a C function: If the object has not yet been created (the slot `object` is 0), it is automatically created by using the slots `type`, `dim` and `dimensions_id`. After the FORTRAN subroutine has been executed, this variable has values and the slot `value` is 1.

Input variable in a FORTRAN subroutine or in a C function: If the object has not yet been created (the slot `object` is 0), it is automatically created by using the slots `type`, `dim` and `dimensions_id`. Then, if the object has not yet been given a value (the slot `value` is 0), we compute the values of the object by using the function pointed to by the slot `compute`.

Working variable in a FORTRAN subroutine: Do the same as an output variable from a FORTRAN subroutine but a new object is created every time. The slots `object` and `value` are not used.

Except for a working variable, only one object is created for an abstract variable.

The process of computing variables is recursive. Indeed, if necessary, other abstract variables can be computed for computing the value of an abstract variable. This is typically the case when we need to compute the dimensions of an array.

3.3 Graph object

Each graph is described by a C++ structure called **graph**. There is a global variable called **theGraph** which is a pointer to the **graph** structure of the graph under study.

There are two types of graph: directed or undirected. The differences between these two types of graphs are reflected by their graphic representation and their internal representation. But any type of graph can be considered as directed or undirected. Indeed, there are FORTRAN programs dealing with directed graphs and FORTRAN program dealing with undirected graphs and we could want to apply any type of program to any type of graph.

Notice that an edge is an undirected link between two nodes and that an arc is a directed edge from a tail node to a head node and that any type of graph is internally represented by its nodes and its arcs, never by its edges. This means that each edge of an undirected graph is internally represented by two arcs (see figure 2).

There are three abstract variables used by FORTRAN programs according to the type of graphs they deal with: `"number of arcs"`, `"number of edges"` and `"number of arcs for undirected graphs"`. `"number of arcs"` is the number of arcs in the internal representation of the graph. `"number of edges"` is the number of links in the real graph. `"number of arcs for undirected graphs"` is the number of arcs in the internal representation of the graph if we consider it as undirected.

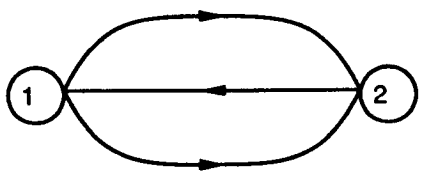
A directed graph is represented by its nodes and its arcs into a **graph** structure. `"number of edges"` equals `"number of arcs"` and `"number of arcs for undirected graphs"` equals 2 times `"number of arcs"`. See figure 1.

An undirected graph is represented by its nodes and its arcs into a **graph** structure. But now `"number of edges"` equals `"number of arcs"` divided by 2. Indeed an edge is internally represented by two arcs with opposite directions and with consecutive internal numbers $2p-1$ and $2p$. And `"number of arcs for undirected graphs"` equals `"number of arcs"`. See figure 2.

The **graph** structure is described below.

```
struct graph {
  char name[MAXNAM];
  graph* un_graph;
  int directed;
  int node_number;
  int arc_number;
  int sink_number;
  int source_number;
  list sinks;
  list sources;
  list arcs;
  list nodes;
```

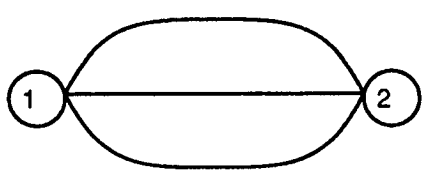
Number of arcs = 3
Number of edges = 3
Number of arcs for undirected graph = 6



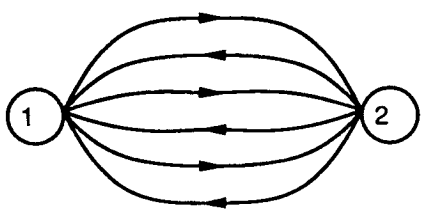
Graphic representation = Internal representation

Figure 1: Directed Graph

Number of arcs = 6
Number of edges = 3
Number of arcs for undirected graph = 6



Graphic representation



Internal representation

Figure 2: Undirected Graph

```
  int max_arc_user_number;
  int max_node_user_number;
  arc** arcArray;
  node** nodeArray;
  arc** userArcArray;
  node** userNodeArray;
};
```

name is the name of the graph.

un_graph is a pointer to the corresponding undirected graph obtained from this graph if it is directed.

directed is a flag which is 1 if the graph is a directed one and 0 if it is undirected.

node_number is the total number of nodes including sources and sinks.

arc_number is the total number of arcs (remember that an arc is a directed edge).

sink_number is the number of sink nodes.

source_number is the number of source nodes.

sinks is a list of the sink nodes pointers.

sources is a list of the source nodes pointers.

nodes is a list of pointers to **node** structures corresponding to the nodes of the graph in decreasing order of node internal numbers.

arcs is a list of pointers to **arc** structures corresponding to the arcs of the graph in decreasing order of arc internal numbers.

The slots **max_arc_user_number** and **max_node_user_number** are the maximum number of the arc user numbers and the maximum number of the node user numbers.

The **graph** structure uses in its slots four other structures described below.

arcArray, **nodeArray**, **userArcArray** and **userNodeArray** are arrays of pointers to the corresponding object. For instance **arcArray[i]** is a pointer to arc object with internal number $i+1$ and **userArcArray[i]** is a pointer to arc object with user number $i+1$. These arrays are redundant with the slots **nodes** and **arcs** but are used for efficiency.

3.3.1 Node object

Each node is described by a C++ structure called **node**.

```
enum node_type {PLAIN, SINK, SOURCE};

struct node {
  int number;
  int user_number;
  list connected_arcs;
  double demand;
  node_type type;
  int x, y;
};
```

number is the internal node number.

user_number is the user node number.

connected_arcs is a list of pointers to the arcs the node is connected to (the notion of connected arc is not a directed one).

`demand` is the demand of node wich can be positive or negative (supply).
`node_type` is the type of the node which can be **`PLAIN`**, **`SINK`** or **`SOURCE`**.
The other slots are used for graphic purposes.

3.3.2 Arc object

Each arc is described by a C++ structure called **`arc`**.

```
struct arc {
  int number;
  int user_number;
  node* head;
  node* tail;
  double unitary_cost;
  double minimum_capacity;
  double maximum_capacity;
  double length;
  double quadratic_weight;
  double quadratic_origin;
  double weight;
  int g_type;
  int x0, y0, x1, y1, x2, y2, x3, y3, xmax, ymax, xa0, ya0, xa1, ya1, xa2, ya2;
};
```

`number` is the internal arc number.
`user_number` is the user arc number.
`head` is a pointer to the head node of the arc.
`tail` is a pointer to the tail node of the arc.
`unitary_cost` is the unitary cost of the arc.
`minimum_capacity` is the minimum capacity of the arc.
`maximum_capacity` is the maximum capacity of the arc. When an arc has only a capacity as attribute, we use this slot.
`length` is the length of the arc.
`quadratic_weight` is the value w_u in the quadratic cost $\frac{1}{2}w_u[(\varphi_u - \bar{\varphi}_u)^2]$.
`quadratic_origin` is the value $\bar{\varphi}_u$ in the quadratic cost $\frac{1}{2}w_u[(\varphi_u - \bar{\varphi}_u)^2]$.
`weight` is a value on the arc. It can be used for computing a minimal weight tree. It can also be the resistance of the arc.
The other slots are used for graphic purposes.

3.4 FORTRAN Program Description

A family of FORTRAN programs is a number of FORTRAN subroutines with FORTRAN names for FORTRAN variables corresponding to abstract variables (see section 3.1). Usually, one FORTRAN variable corresponds to one abstract variable, but it is possible to have several FORTRAN variables corresponding to one abstract variable (it is the case of working arrays for instance). Typically a family of FORTRAN programs has been written by a same group of people and deals with a same type of network problems.

Each family of FORTRAN programs is described by two files. If <name> is the name of the family, there is a file called <name>-variable.desc describing variables and another one called <name>-subroutine.desc describing subroutines. Their syntax is explained below.

<name>-variable.desc is an ASCII file describing FORTRAN variables. Each variable is described by three lines plus a separator line:

```
name
definition
description
<separator line>
```

name is the FORTRAN name of the variable. **definition** is the definition of the variable (see section 3.1) and **description** is a comment or description.

This file only establishes a correspondence between **name** and **definition**.

<name>-subroutine.desc is an ASCII file describing FORTRAN subroutines. Each subroutine is described by five lines plus a separator line:

```
name
<list of input variable names in lexicographic order>
<list of output variable names in lexicographic order>
<list of working variable names in lexicographic order>
description
<separator line>
```

name is the FORTRAN name of the subroutine and **description** is its complete description. In the three other lines, a tabulation of 8 characters is used.

3.5 FORTRAN subroutine objects

The objects described below are not given by the user but are automatically generated by a program using the definition of the **graph_variable** objects (see section 3.2) together with the files *.desc described in section 3.4.

Each FORTRAN subroutine is described by a C++ structure called **fortran_subroutine**:

```
enum ptype {IN, OUT, WORK};

struct arg {
  ptype type;
  int id;
};

struct subroutine {
  char* family;
  char* name;
  char* description;
  void (*compute) (...);
  int n_arg;
  arg* args;
};
```

family is the family of the FORTRAN programs to which the variable belongs.
name is the FORTRAN name of the subroutine.
description is the complete description of the subroutine.
compute is a pointer to the FORTRAN subroutine,
n_arg is the number of arguments for the FORTRAN subroutine.
args is a pointer to a vector of **arg** objects describing the arguments:
type is the type of the argument which can be **IN** for input, **OUT** for output or **WORK** for working.
id if the identifier of the corresponding abstract variable.

4 Data Files

A graph named **g** is described by two data files: **g.graph** and **g.metanet**. The data files are in the directory **data**. The former is an ascii file and contains minimal data for a graph. It can be modified by the user in an editor. It is described below. The latter is a binary file created by METANET and contains complementary data (graphics, redundant data ...).

4.1 Structure of graph file

The **graph** file is an ascii file with the following structure:

```
GRAPH TYPE (0 = UNDIRECTED, 1 = DIRECTED) :
<one line with 0 or 1 according to the type of the graph>
NUMBER OF ARCS :
<one line with the number of arcs>
NUMBER OF NODES :
<one line with the number of nodes>
**************************************
DESCRIPTION OF ARCS :
ARC #, TAIL NODE #, HEAD NODE #
COST, MIN CAP, CAP, MAX CAP, LENGTH, Q WEIGHT, Q ORIGIN, WEIGHT
<a blank line>
<two lines for each arc>
**************************************
DESCRIPTION OF NODES :
NODE #, POSSIBLE TYPE (1 = SINK, 2 = SOURCE)
X, Y
DEMAND
<a blank line>
<three lines for each node>
```

When describing a node, no type means a plain node.

4.2 Modifying graphs

You can modify an existing file by using the Modify menu (see section 5.2.3). But it is also possible to do it by hand. For that, you have to modify the **graph** ascii file described above. Then, you have to use the "Load and Compute Graph" item of the Begin menu (see section 5.2.1) to update the **metanet** binary file.

4.3 Generating graphs

There are programs used to automatically generate graphs in the directory **gengraph**.
They are **make-graph-file**, **netgen** and **mesh**.

4.3.1 Making a template graph file

make-graph-file is used to created a template graph file. It asks for the graph name, verifies if it is not the name of an existing file, and asks for the number of arcs, nodes, sources and sinks. Then a **graph** file is created in the **data** directory and you can modify it by hands.

4.3.2 Netgen

netgen is a modification of the famous **netgen** program for generating graphs ([10]). First you can use **seed** for generating a random number in the file **FOR011.DAT** used by **netgen**, but this is not necessary (there is already one).

netgen displays the following line :

```
INPUT NODES,NSORC,NSINK,DENS,MINCST,MAXCST,ITSUP
```

and you have to put the number of nodes, the number of pure sources, the number of pure sinks, the density of the graph (an integer for an approximate number of arcs), the minimum cost, the maximum cost and the maximum supply for the whole network.

Then **netgen** displays the following line :

```
INPUT NTSORC,NTSINK,BHICST,BCAP,MINCAP,MAXCAP
```

and you have to put the number of transshipment source nodes, the number of transshipment sink nodes, the percentage of arcs with a cost, the percentage of capacitated arcs and the minimum and maximum arc upper capacities (the lower capacity is always 0).

An ascii file describing the network is created (**NETGEN.DAT**) and you have to use the program **netgen2graph** to create a **graph** file in the **data** directory. This saved graph can be the original graph or the corresponding augmented graph (only one source and one sink). **netgen2graph** asks the question.

netgen computes no coordinates for the nodes. When METANET loads such a graph, before computing the **metanet** file, it has to compute values for the coordinates of the nodes. The algorithm is at the present time a very simple one and it will be improved in the future.

4.3.3 Mesh

mesh is a program derived from a finite element one. It resembles **netgen** and uses a subpart of it for generating the network but it gives a triangulation of the plan from which we have the node coordinates. The generated graph is planar.

mesh displays the following line :

```
INPUT NODES,NSORC,NSINK,MINCST,MAXCST,ITSUP,OTSUP
```

and you have to put the number of nodes, the number of pure sources, the number of pure sinks, the minimum cost, the maximum cost and the maximum supply and the maximum demand for the whole network.

Then **netgen** displays the following line :

```
INPUT BHICST,BCAP,MINCAP,MAXCAP
```

and you have to put the percentage of arcs with a cost, the percentage of capacitated arcs and the minimum and maximum arc upper capacities (the lower capacity is always 0).

An ascii file describing the network is created (**MESH.DAT**) and you have to use the program **mesh2graph** to create a **graph** file in the **data** directory. This saved graph can be the original

graph or the corresponding augmented graph (only one source and one sink). **netgen2graph** asks the question.

Figure 3 summarizes everything about data files, generating graph files and concerned programs.

5 Using METANET

To activate METANET, you only have to issue a **metanet** command to your system.

There are two windows. The first one is the interpreter window (see section 5.3). The second one is a graphic window with menus (see sections 5.1 and 5.2). See also appendix A where a sample session of METANET is given.

5.1 Using the mouse in the graphic window

All mouse buttons are equivalent.

The mouse can be used to highlight an object (arc or node) in the graphic window. For that, you only have to click once in the object. For a curbed arc, you have to click in the middle of the arc. If another object was already highlighted, it becomes non highlighted. When you click in an highlighted object, it is no longer highlighted.

When you are in the Modify menu (see section 5.2.3), the mouse can be used to modify the graph. When you click in a place where there is no object, a new node is created. If you click in a node and another node is highlighted, then a new arc is created from the latter node to the former node. You can also move a node by clicking in it and moving the mouse while the button is down.

5.2 Menus

There are three levels of menus. We call them the Begin menu, the Study menu and the Modify menu.

5.2.1 Begin Menu

This menu is displayed when METANET is activated. The items are described below.

Quit For quitting METANET.

Load Graph When this item is activated, a list of all the graphs existing in the data directory (see section 4) is displayed and you can choose one of them. Then it is displayed and we are in the Study menu. If the graph you are loading has no **metanet** data file (see section 4), then a message is issued and METANET will first create the missing file. In this case, if the nodes have no coordinates, a message is issued and METANET will compute them.

Load and Compute Graph It is the same thing as Load Graph menu, but the binary file **<graph_name>.metanet** is recomputed from the ascii file **<graph_name>.graph**. This item must be activated when you have modified the data in the ascii file.

New Graph This is for creating a new graph. METANET prompts for its name which must be different from the names of already existing graphs. Then we are in the Modify menu.

Delete Graph For deleting a graph.

Copy Graph For copying a graph.

Rename Graph For renaming a graph.

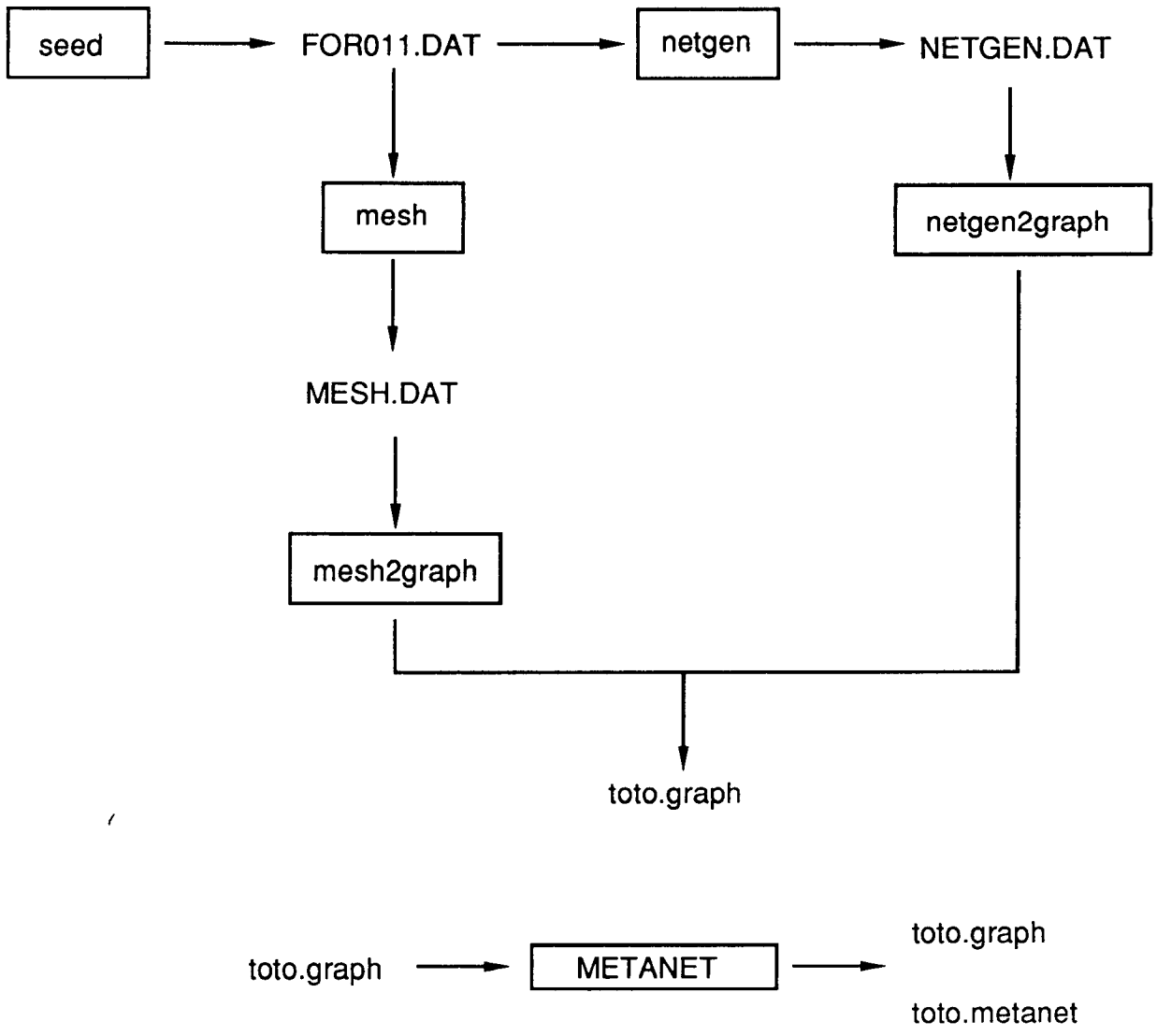


Figure 3: Data files

5.2.2 Study Menu

This menu is the standard menu displayed when we want to use METANET to perform computations.

Quit For quitting the Study menu and entering the Begin menu.

Compute Variable This is to perform a computation by menu. When this item is activated, a list of all the abstract variables we can compute by menu is displayed and the user must choose one of them. The list of these variables is given below.

```
arc number shortest path from i to j
arc number shortest path length from i to j
circuit
integer flow value on return arc
integer maximal flow edge array
integer minimal cost flow edge array
integer minimal cost flow edge array (relax)
integer minimal cost imposed flow edge array
integer minimum cost value
max capacity path from i to j
minimal weight tree with root i0
node set of connected component i
node set of strong connected component i
number of connected components
number of strong connected components
rank function
real max capacity path value from i to j
real minimal quadratic cost flow edge array
real shortest path length from i to j
shortest path from i to j
```

They correspond to the problems described in section 2. We have the same functionalities in the interpreter (see section 5.3.1).

Object Attributes If an object (arc or node) is highlighted, its attributes are printed in the interpreter window. Otherwise the attributes of the graph are displayed.

Find Node Used to highlight a node. METANET prompts for a node number. If the node is not displayed, the graph moves to display it.

Find Arc Used to highlight an arc. METANET prompts for an arc number. If the arc is not displayed, the graph moves to display it.

Modify Graph This item permits to enter the Modify menu.

5.2.3 Modify Menu

This menu is displayed when we want to modify an existing graph or we want to create a new one.

Quit For quitting the Modify menu. If the graph has been modified but not saved, METANET asks you if you want to save it. If you answer yes, you are again in the Modify menu and you can activate the Save Graph menu.

Delete Object If there is an highlighted object (arc or node), delete it.

Number Object If there is an highlighted object (arc or node), METANET prompts for a number for it. If there is no highlighted object and no number has be given to arcs and nodes, METANET asks you if you want to use internal numbers (automatically given by METANET every time an arc or a node is created) for nodes and arcs. Otherwise, if there are nodes and/or arcs without number, METANET asks you if you want to automatically number nodes and arcs. In this case, each non numbered arc or node is displayed and you can give a number to it.

Object Attributes If an object (arc or node) is highlighted, its attributes are printed in the interpreter window. Moreover a window appears and you can interactively modify attributes of the object. Otherwise the attributes of the graph are displayed.

Find Node Used to highlight a node. METANET prompts for a node number. If the node is not displayed, the graph moves to display it.

Find Arc Used to highlight an arc. METANET prompts for an arc number. If the arc is not displayed, the graph moves to display it.

Create Source If there is an highlighted node, it becomes a source.

Create Sink If there is an highlighted node, it becomes a sink.

Remove Source/Sink If there is an highlighted source or sink, it becomes a plain node.

Save Graph This item is used to save the graph. If there are nodes and/or arcs that have not been numbered, METANET issues a message and you are again in the Modify menu. Then you have to use the Number Object item. When the graph is saved, METANET always renumber nodes and arcs internal numbers in order to have consecutive numbers.

Rename and Save Graph The same as the item Save Graph, but METANET prompts for a new name for it.

5.3 The interpreter

The interpreter window provides the user with a comprehensive MATLAB ([8]) like language. In fact, we have in this window the BASILE system ([1]) which is a CACSD system for automatic control developed at INRIA.

Most of the variables you can compute by using the Compute Variable item of the Study menu can be computed in BASILE interpreter. But now, you can get the results as BASILE expressions and do computations on them.

5.3.1 METANET functions in BASILE

We describe below the new functions in BASILE.

See section 2 for the description of the various network problems.

A path or a tree is a BASILE row of arc numbers and a node set is a BASILE row of node numbers.

`printg(n)` prints characteristics of the loaded graph with more or less informations according to the value of `n` (0, 1 or 2 where 2 corresponds to maximum information).

`printa(i,n)` prints characteristics of arc `i` with more or less informations according to the value of `n` (0, 1 or 2 where 2 corresponds to maximum information).

`printn(i,n)` prints characteristics of node `i` with more or less informations according to the value of `n` (0, 1 or 2 where 2 corresponds to maximum information).

`showp(p)` highlights the path `p` in the graphic window.

`showns(s)` highlights the node set `s` in the graphic window.

`connum()` returns the number of connected components of the graph.

`concom(n)` returns the set of nodes in the n^{th} connected component.

`sconnum()` returns the number of strong connected components of the graph.

`sconcom(n)` returns the set of nodes in the n^{th} strong connected component.

`ansp(i,j)` returns the arc number shortest path from node `i` to node `j`.

`anspl(i,j)` returns the number of arcs in the arc number shortest path from node `i` to node `j`.

`lsp(i,j)` returns the length shortest path from node `i` to node `j`.

`lspl(i,j)` returns the total length of the shortest path from node `i` to node `j`.

`circuit()` returns a circuit of the graph if there is one.

`wstree(i)` returns the minimal weight spanning tree with node `i` as its root.

`maxcpp(i,j)` returns the maximum capacity path from node `i` to node `j`.

`maxcpv(i,j)` returns the value of the maximum capacity in the maximum capacity path from node `i` to node `j`.

`mcfar()` returns the array of flows for minimum cost network flow problem $(\mathcal{P})$by using a relaxation method (see [3]).

`mcfa()` returns the array of flows for minimum cost network flow problem $(\mathcal{P}_1)$.

`mfa()` returns the maximal flow array, solving problem $(\mathcal{P}_m)$.

`micfa(c)` returns the array of flows for minimum imposed cost network flow problem $(\mathcal{P}_2)$. `c` is the given value of the imposed flow.

`mqcfa(p)` returns the array of flows for quadratic minimum cost network flow problem $(\mathcal{Q})$. `p` is a negative integer corresponding to the precision of the algorithm which is 2^p (discretization step).

`mcfv()` returns the value of the computed cost for problems $(\mathcal{P})$, $(\mathcal{P}_1)$, $(\mathcal{P}_2)$and $(\mathcal{Q})$.

`rafv()` returns the value of the flow on the return arc for problems $(\mathcal{P}_2)$and $(\mathcal{P}_m)$.

6 Internals

6.1 Using FORTRAN

We describe here the way METANET uses FORTRAN subroutines.

First of all, all the abstract variables (see section 3.1) are uniquely defined as **graph_variable** structures in C++ (see section 3.2). There are two global variables: **nVariables** which is the number of abstract variables and **variables** which is a vector of pointers to the **graph_variable** objects. These objects and this vector are created in the file **abstract-variables.c**. This file is *the reference* for the definition of the abstract variables.

There are programs which automatically add or remove abstract variables (see section 6.2).

All FORTRAN subroutines of all families are loaded in METANET and linked to METANET. With the definition of the **graph_variable** objects (see section 3.2) together with the files *.desc described in section 3.4 a program automatically generates a file containing a function creating and computing **subroutine** objects corresponding to FORTRAN subroutines (see section 6.2.1). This file is **subroutines.c**.

There are two global variables: **nSubroutines** is the number of subroutines and **subroutines** is a vector of pointers to the **subroutine** objects.

Then, the steps to execute the FORTRAN subroutines are described below.

1. By looking to the **subroutine** structure of the FORTRAN subroutine, we get the abstract variable identifiers of the variables which appear in the calling sequence of the subroutine.
2. For each graph variable of the calling sequence of the subroutine, we create the corresponding object and/or compute its value if necessary (see section 3.2).
3. Then we can call the FORTRAN subroutine with good C++ objects as arguments and execute it.

The above process is recursive because other FORTRAN subroutines can be called when computing values of abstract variables for input.

6.2 Maintaining METANET

There are programs for maintaining METANET for a user who wants to modify METANET internals (variables) and to manage subroutine families.

These programs are in the directory **src/maintain**.

6.2.1 Adding or removing

There are programs that automatically manage the abstract variables and FORTRAN subroutines in METANET.

- **add-variable <filename>** adds abstract variables described in file <filename>. Each variable is described by three lines plus a separator line:

```
name type dimensions dim1 dim2 ... dimn
definition
description
<separator line>
```

definition is the definition of the variable (see section 3.1) and **description** is a comment or description.

- **remove-variable <id> ...** removes one or more abstract variable described by its unique identifier <id> (see section 3.1). Before removing these variables, the program verifies if they are used as arguments in FORTRAN subroutines.
- **add-family <family name>** adds a new family of subroutines in METANET. This program transforms the family FORTRAN variables appearing in the description files of the family (see section 3.4) into abstract variables. These abstract variables must have been created before.
- **remove-family <family name>** removes all the subroutines of a family from METANET. This program is useful and necessary when we have modified a family and we want to add it again in METANET.
- **remove-subroutine <subroutine> ...** removes one or more subroutine described by its name.

After using these programs, METANET must be recompiled.

6.2.2 Adding new functions

It is possible for the user to add new functions in the interpreter. This is the way for the user to test its own programs in METANET and to add its own family of FORTRAN subroutines.

The process is described below.

First, you define a new FORTRAN family. We suppose that the name of this family is **myfamily**. You create a directory named **myfamily** in the directory **lib** and put the FORTRAN subroutines in it. After having compiled them, you create a library with them named **libmyfamily.a** and put it in the **lib** directory. This new library must be known by METANET makefiles and you must add its name in the **FAMILIES** macro in the file **src/make.incl**.

Then, you create in the directory **lib/myfamily** the description files of the family (see 3.4), i.e. **myfamily-variables.desc** and **myfamily-subroutines.desc**.

The second step is to make this family known by METANET. For that you use the programs described in 6.2.1. You use **add-variable** to add your new variables and then **add-family** to add your new family.

The third step is to create the functions that correspond in the interpreter to the FORTRAN subroutines of the family.

If the outputs of your FORTRAN subroutine do not correspond to any METANET variable, you are obliged to create new ones. But if they correspond, for instance to test a subroutine improving an already existing algorithm in METANET, it is unnecessary to create new METANET variables. But it is highly recommended to do it. The reason is that a METANET variable is only computed once in METANET and then you cannot use two functions computing the same variable.

To create the functions, you have to go in the **src/basile** directory and describe the new functions in the file **matusr.desc** and add them in the **fundef** file. The syntax in the file **matusr.desc** is explained in the user guide of INTERBAS which comes with [1]. Then, the new function must be defined in the file **src/basile.c** and the computation of the new variables must be settled at the end of the file **src/function-variables.c**.

Then go to **src** directory and issue a **make** command.

In the next version of METANET, the whole process will be improved in a more user-friendly manner.

7 FORTRAN families

7.1 General family

The FORTRAN programs of this family are in [9].

`compc` for computing the number of connected components
`compfc` for computing the number of strong connected components
`arbor`, `prim` for computing a minimal weight tree
`chcm` for computing a max capacity path
`pccsc`, `ford`, `johns`, `dijkst`, `pcchna` and `floyd` are various programs for shortest path problems
`flomax` for maximum flow problems
`frmtrs` for computing transitive closure
`fcirc` for finding a circuit
`frang` for computing the rank function
`kilter` and `busack` for minimum cost flow problems
`floqua` for minimum quadratic cost flow problem

7.2 Relax family

The FORTRAN programs of this family are for solving minimum cost network flow problems by relaxation methods (see [3], [4]).

A Sample sessions of METANET

A.1 Shortest path

In this session, we load into METANET the directed graph called "mesh100". This graph has been generated using the program **mesh** (see section 4.3.3).

Figure 4 shows METANET graphic window with graph "mesh100" loaded.

We compute the arc number shortest path from node 57 to node 42 and display the path.

Figure 6 shows METANET graphic window with displayed shortest path.

The session in the interpreter window is:

```
<>p=ansp(57,42)


 p          =

!   87.    111.    145.    173.    167.    161.    171.    195. !
<>showp(p)
<>
```

A.2 Strong connected components

In this session, we load into METANET the directed graph called "netgen10". This graph has been generated using the program **netgen** (see section 4.3.2).

Figure 5 shows METANET graphic window with graph "netgen10" loaded.

We compute the number of strong connected components of the graph and then display the strong connected component number 2.

Figure 7 shows METANET graphic window with displayed strong connected component number 2.

The session in the interpreter window is:

```
<>sconnum()


 ans        =

    3.
<>s=sconcom(2)
 s          =

!   4.    5.    7.    2.    3.    8.    6.    9. !
<>showns(s)
<>
```

A.3 Maximal flow

We also use the directed graph called "netgen10" (see figure 5).

We compute the maximal flow array with "mfa" command. Then, by using BASILE language we construct the path "p" corresponding to this flow (where the flow is not zero) and we display it on the graph.

Figure 8 shows METANET graphic window with displayed path.

The session in the interpreter window is:

```
<>a=mfa()

 a         =


        colonnes    1 a   10

!   66.    3.    97.    0.    100.    0.    0.    0.    100.    0. !

        colonnes   11 a   21

!   0.    0.    0.    66.    0.    3.    0.    0.    0.    0.    0. !

        colonnes   22 a   30

!   100.    0.    0.    0.    0.    0.    0.    0.    66. !
<><m,n>=size(a)
 n         =

    30.
 m         =

    1.
<>j=0; for i=1:n, if a(i) <> 0 then j=j+1; p(1,j)=i; end end
<>p
 p         =

!   1.    2.    3.    5.    9.    14.    16.    22.    30. !

<>showp(p)
<>
```

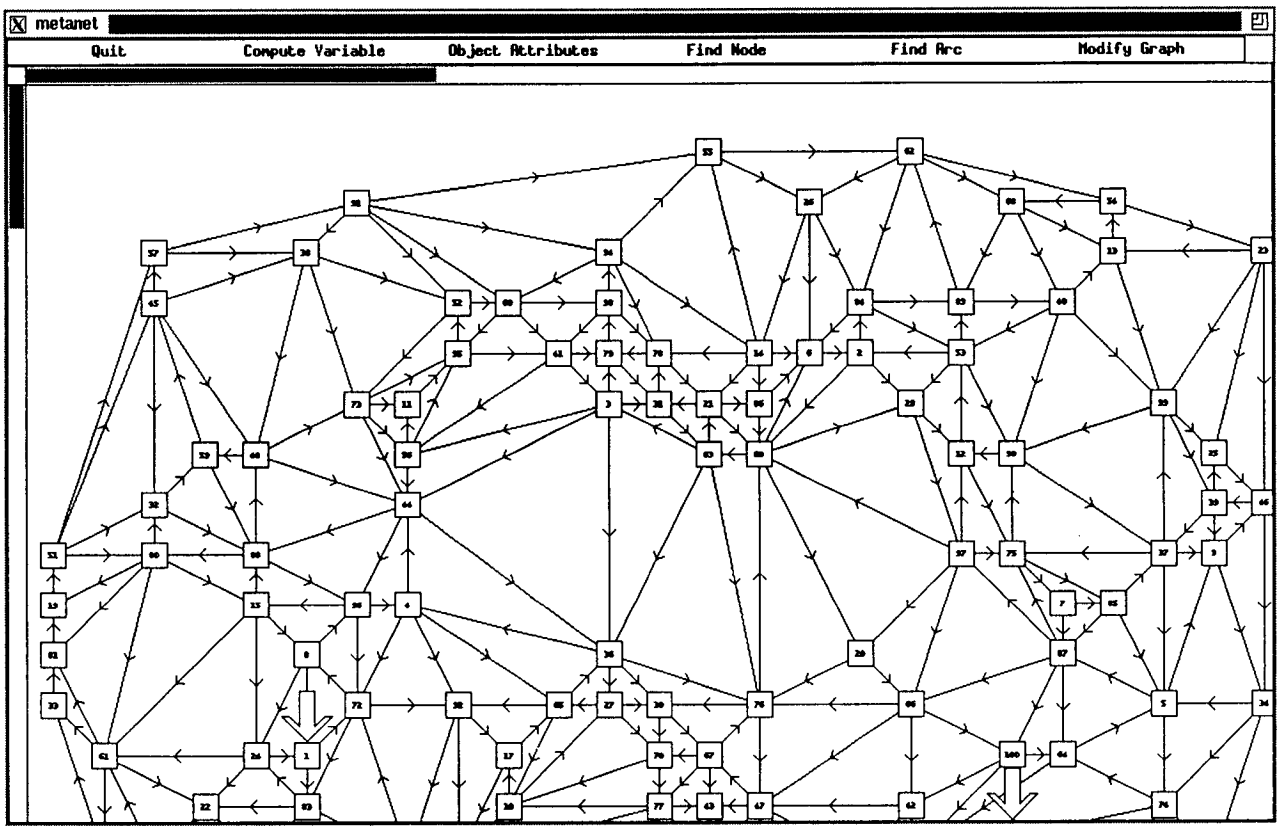


Figure 4: Graph "mesh100"

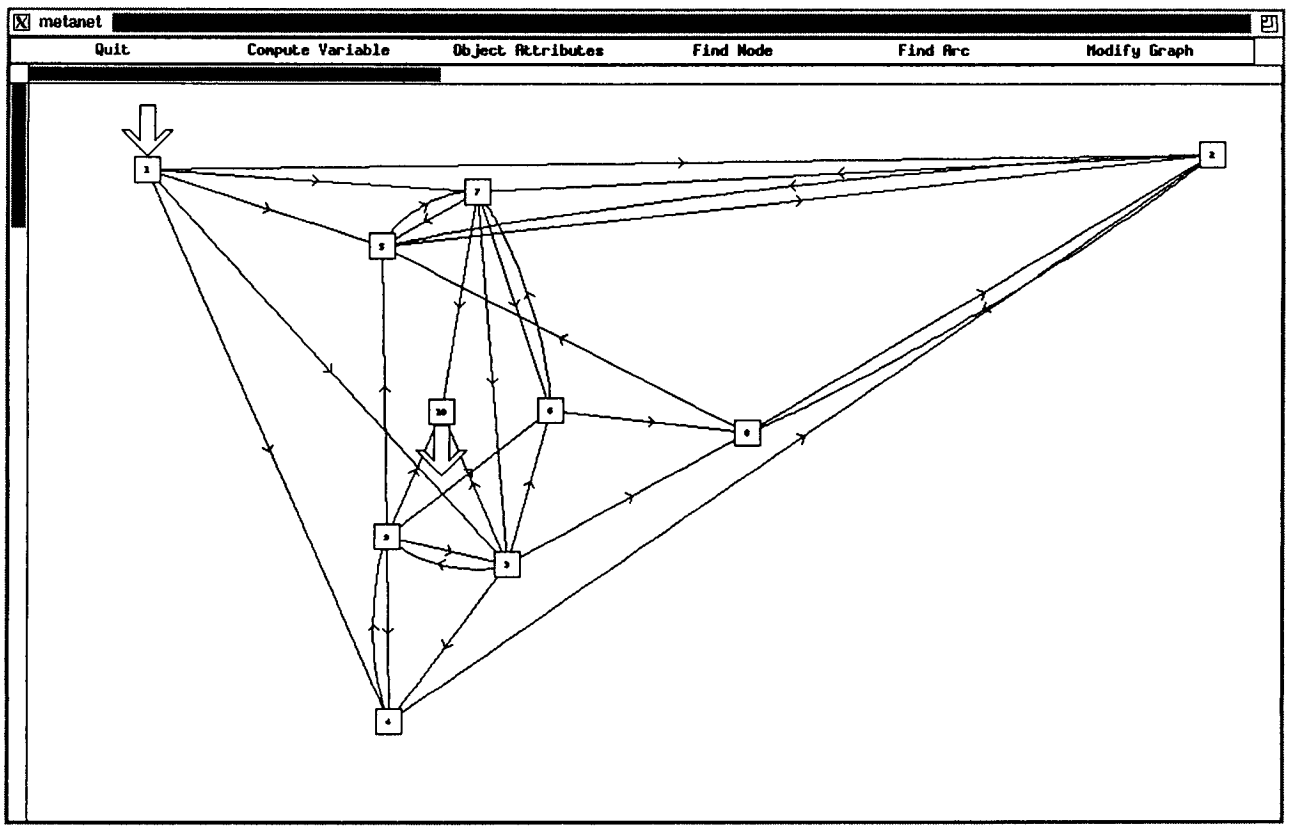


Figure 5: Graph "netgen10"

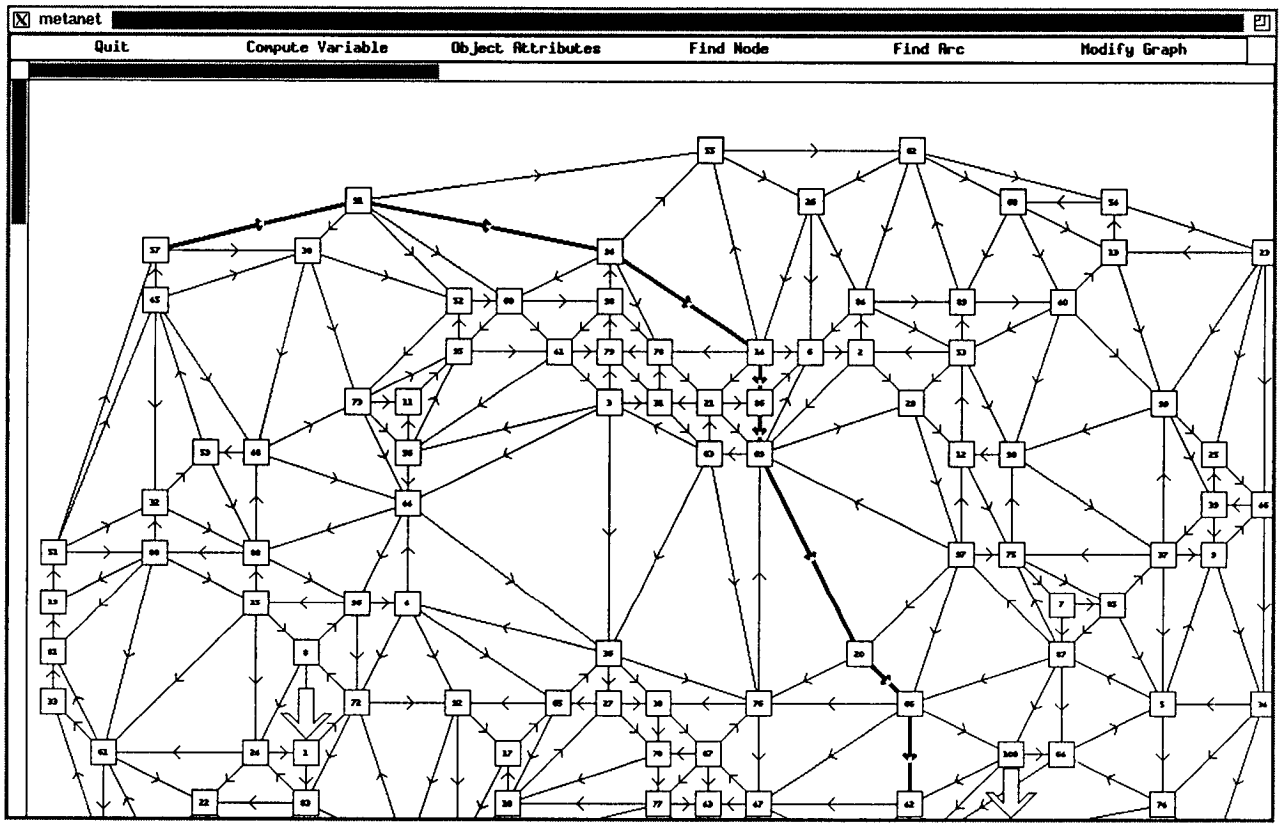


Figure 6: Shortest path in graph "mesh100" from node 57 to node 42

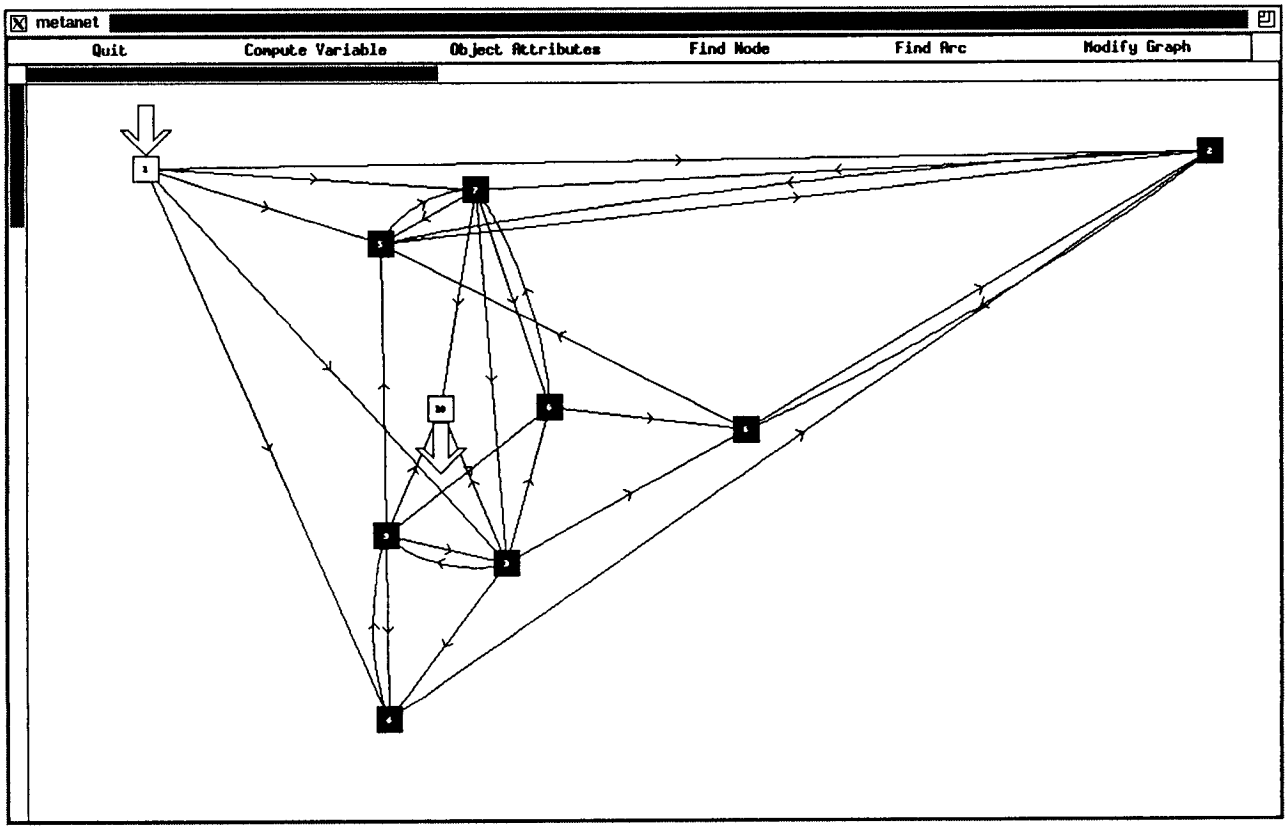


Figure 7: Strong connected component 2 in graph "netgen10"

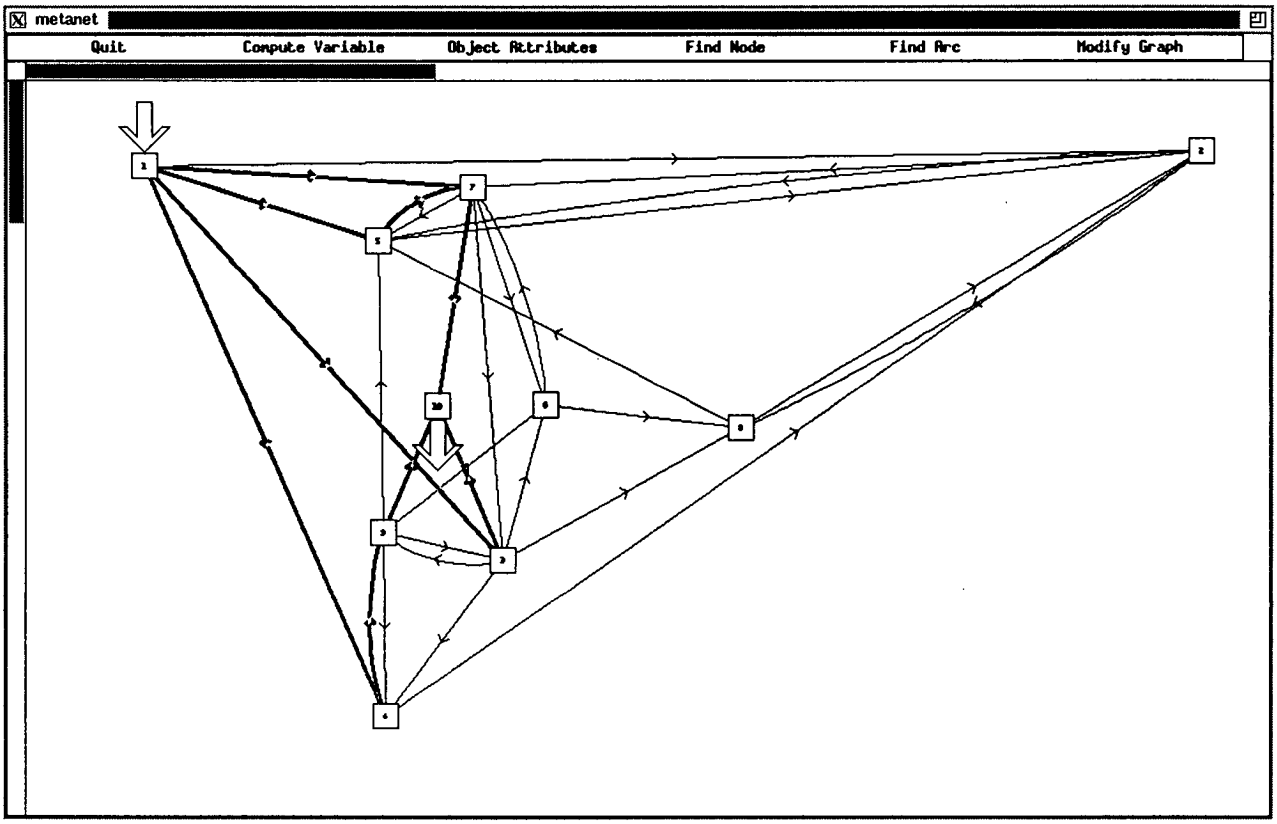


Figure 8: Path for maximal flow in graph "netgen10"

References

[1] François Delebecque, Carlos Klimann and Serge Steer, BASILE, guide de l'utilisateur, INRIA 1989.

[2] Claude Berge, Graphes, Gaulthier-Villars 1973.

[3] Dimitri P. Bertsekas and Paul Tseng, Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems, Operations Research, Vol. 26, No. 1, January-February 1988.

[4] Dimitri P. Bertsekas and John N. Tsitsiklis, Parallel and Distributed Computation, Prentice Hall 1989.

[5] Brian W. Kernighan and Dennis M. Ritchie, the C Programming Language, Prentice Hall 1978.

[6] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley 1986.

[7] Michel Gondran and Michel Minoux, Graphes et algorithmes, Eyrolles 1985.

[8] Cleve Moler, MATLAB User's Guide. Technical report CS81–1, Department of Computer Science, University of New Mexico, 1982.

[9] Georges Bartnik and Michel Minoux, Graphes algorithmes logiciels, Dunod 1986.

[10] D. Klingman, A. Napier and J. Stutz, NETGEN: A Program for Generating Large Scale Capacitated Assignment, Transportation and Minimum Cost Flow Network Problems, Management Science, Vol. 20, No. 5, January 1974.

[11] James Gettys, Robert W. Scheifler and Ron Newman, X Window System, Xlib - C Language X Interface, X Version 11 Release 4, MIT, 1989.

[12] Chris D. Peterson, X Window System, Athena Widget Set - C Langage Interface, X Version 11 Release 4, MIT, 1989.

[13] Richard M. Stallman, Using and Porting GNU CC, September 1989.

[14] Michael D. Tiemann, User's Guide to GNU C++, August 1989.

Contents

List of Figures

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique