MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un logiciel d'aide à la théorie des graphes

Nguyen, Thanh Lam

Award date:
1985

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE DAME DE LA PAIX - NAMUR

INSTITUT D'INFORMATIQUE

IMPLEMENTATION D'UN LOGICIEL D'AIDE A LA THEORIE DES GRAPHES

Mémoire présenté par

NGUYEN THANH LAM

en vue de l'obtention du grade de

Licencié et Maître en Informatique

Année Académique 1984-1985

Je tiens à exprimer ma profonde gratitude à Monsieur J.FICHEFET qui m'a permis de réaliser ce mémoire et m'a donné des conseils très précieux tout au long de mon travail.

Je voudrais témoigner toute ma vive reconnaissance à tous les Professeurs qui m'ont enseigné dans les deux années académiques 1983-84 et 1984-85.

Enfin, qu'il me soit permis de remercier infiniment toutes les personnes qui m'ont aidé positivement pour l'achèvement de ce projet.

TABLE DES MATIERES

Page

Page

Page

INTRODUCTION

La théorie des graphes est déjà connue à partir de 1736 [L. Euler] mais depuis 1960, favorisée par l'apparition des premiers calculateurs électroniques, on assiste à une véritable explosion des recherches et des applications. Elle intervient en fait chaque fois que l'on est amené à représenter une situation ou un problème par un schéma constitué par un ensemble de points reliés entre eux par des branches.

Le type des problèmes le plus classique est la représentation d'un réseau de communication : réseaux de route, réseaux de chemin de fer, réseaux téléphoniques, réseaux électriques, etc... et en général, la représentation d'une relation binaire qui soit algébrique, mécanique, chimique, sociologique, etc...

Dans une démarche pour l'étude d'un problème concernant la théorie des graphes, on peut distinguer deux étapes :

1- L'analyse du problème qui permet de schématiser le problème sous forme d'un graphe.

2- Le travail de résolution et d'interprétation des résultats.

Ici, nous n'allons aborder que la seconde étape sus-mentionnée. L'utilisateur doit alors lui-même formaliser le graphe correspondant au problème à résoudre. Le travail de cette étape est effectué en mode interactif. En effet, le dialogue avec l'utilisateur permet de maîtriser les solutions possibles tout en orientant l'analyse du problème d'après

ses préférences. De plus, on n'oublie pas d'offrir à l'utilisateur une facilité de l'introduction des données du graphe sous forme d'une conversation.

Le but poursuivi dans ce mémoire est la mise au point d'un logiciel d'aide à la théorie des graphes en mode interactif sur l'ordinateur VAX11/780 de l'Institut d'Informatique de Namur.

Dans la première partie et au premier chapitre, on va vous offrir une connaissance sur la théorie des graphes via les définitions et les concepts de base. Le second et dernier chapitre de cette partie consiste à exposer quelques problèmes les plus fréquents concernant le graphe tels que le problème du plus court chemin, le problème central de l'ordonnancement, le problème de parcours eulériens, le problème du Postier chinois et le problème du voyageur de commerce avec leurs algorithmes.

Dans la deuxième partie, on va aborder l'environnement du logiciel au premier chapitre, cela veut dire que sur quelle machine doit-on implémenter et de quel langage est-il traduit le logiciel. On trouve une vue globale du logiciel dans le second chapitre à l'aide d'un schéma des états du logiciel et leurs explications. Le troisième chapitre présentera les structures de données concernant la représentation d'un graphe, la cohérence des algorithmes et l'archivage d'un graphe sur un fichier. Le chapitre qui suit,précisera les modules du logiciel et leurs spécifications, du fait qui permet de savoir le fonctionnement intérieur du logiciel. Et on va terminer cette deuxième partie par un dernier chapitre qui donne à l'utilisateur une probabilité de faire une extension au logiciel.

PREMIERE PARTIE

L A T H E O R I E D E S G R A P H E S

INTRODUCTION

Dans la vie courante, étant donné un problème et après avoir lu ce problème, on a la tendance d'abstraire cette situation donnée en traçant, sur une feuille de papier, des points pouvant représenter des individus, des localités, des corps chimiques, etc..., reliés entre eux par des lignes ou des flèches symbolisant une certaine relation.

Cette représentation figurative présente un double avantage:

- elle permet de mettre en évidence la structure profonde de la situation donnée,
- au point de vue pratique, elle fournit une vision globale du problème, ce qui constitue un guide précieux pour l'intuition et le raisonnement.

Le langage des graphes permet de mettre en pratique cette idée.

Dans la première partie, on va vous faire connaître le graphe via les définitions et les concepts de base du premier chapitre. Le second chapitre donnera l'exposé de quelques problèmes concernant le graphe, les plus fréquents et leurs algorithmes . Ici, on n'a pas d'ambition de montrer tous les algorithmes pour un problème mais on en a sélectionné le plus simple et le plus compréhensible

CHAPITRE I

GENERALITES SUR LES GRAPHES

Nous allons formaliser les notions intuitives qui ont pour but de vous offrir une bonne connaissance sur graphe, permettant de suivre la suite du mémoire.

I.1 DEFINITIONS ET CONCEPTS DE BASE

I.1.1. Graphes orientés

Un graphe $G=[X,U]$ est déterminé par:

- un ensemble X des sommets ou des noeuds. Si $N = |X|$ est le nombre de sommets (de noeuds), on dit que G est d'ordre N.
- un ensemble U des arcs qui sont des couples ordonnés de sommets. Si $u = (x_i, x_j)$, $u \in U$ et $x_i, x_j \in X$, x_i est l'extrémité initiale de u et x_j l'extrémité terminale de u. On notera souvent $|U| = M$

Un arc $u = (x_i, x_j)$ dont les extrémités coïncident est appelé une boucle.

Un p-graphe est un graphe dans lequel il n'existe jamais plus de p arcs entre les deux sommets quelconques x_i, x_j pris dans cet ordre.

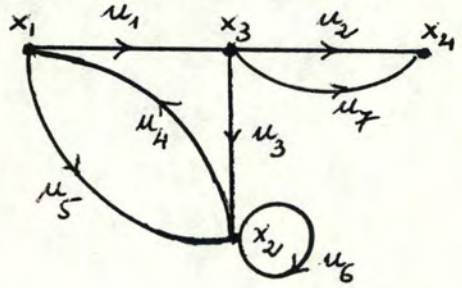


Fig. 1 Exemple d'un 2-graphe

$u_6 = (x_2, x_2)$ est une boucle. On a un 2-graphe car $u_2 = (x_3, x_4)$, $u_7 = (x_3, x_4)$.

I.1.2. Applications multivoques

Etant donné un graphe orienté $G = [X, U]$

On dit que x_j est un successeur de x_i s'il existe un arc $(x_i, x_j) \in U$.

L'ensemble des successeurs d'un sommet $x_i \in X$ est noté Γx_i

L'application $\Gamma : X \to \mathcal{P}(X)$

$$x \longmapsto \Gamma(x) = \Gamma_x = \{y \in X / (x,y) \in U\}$$

est appelée une application multivoque.

On dit que x_j est un prédécesseur de x_i, s'il existe un arc de la forme: (x_j, x_i)

L'ensemble des prédécesseurs de $x_i \in X$ peut alors être noté $\Gamma^{-1} x_i$

où $\Gamma^{-1} : X \to \mathcal{P}(X)$

$$x \longmapsto \Gamma^{-1}(x) = \Gamma_x = \{y \in X / (y,x) \in U\}$$

est l'application réciproque de Γ

Si le graphe G est un 1-graphe, on voit qu'il est parfaitement déterminé par la donnée de l'ensemble X et de l'application multivoque $\Gamma : X \to \mathcal{P}(X)$. Un tel graphe peut donc être aussi noté : $G = [X, \Gamma]$

En éliminant l'arc u_7 du graphe de la figure 1, on obtient un 1-graphe qui peut donc être représenté par une application multivoque Γ . Pour ce graphe, l'application Γ sera :

$$\Gamma_{x_1} = \{x_2, x_3\} , \Gamma_{x_2} = \{x_1, x_2\} , \Gamma_{x_3} = \{x_3, x_4\} , \Gamma_{x_4} = \{\}$$

I.1.3. Graphes non orientés

L'étude de certaines propriétés des graphes ne demande pas l'orientation des arcs. On s'intéresse simplement à l'existence ou à la non-existence d'un(ou de plusieurs)arcs entre deux sommets.

Pour transformer d'un graphe orienté à un graphe non orienté, à tout arc (x_i, x_j), on associe le couple non ordonné (x_i, x_j), qui est appelé l'arête (x_i, x_j). Graphiquement, l'arête (x_i, x_j) sera représentée par un segment (sans flèche) joignant deux sommets x_i et x_j

Etant donné un graphe non orienté $G = [X,U]$, l'ensemble U sera un ensemble d'arêtes, c'est à dire une famille finie de parties à deux éléments de X.

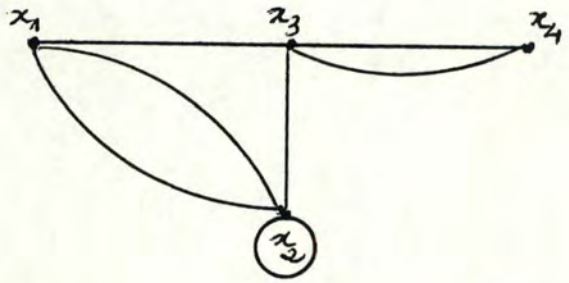


Fig.2- Graphe non orienté correspondant au graphe de la fig.1

Un multigraphe est un graphe pour lequel il peut exister plusieurs arêtes entre deux sommets x_i et x_j donnés.

Un graphe est dit simple si :

-il est sans boucle ,

-il n'y a jamais plus d'une arête entre deux sommets quelconques.

I.1.4. Principales définitions

- Deux arcs (arêtes) sont dits adjacents s'ils ont au moins une extrémité commune.

- Le demi-degré extérieur du x_i noté $d_G^+(x_i)$, est le nombre d'arcs ayant x_i comme extrémité initiale.

- Le demi-degré intérieur du x_i noté $d_G^-(x_i)$, est le nombre d'arcs ayant x_i comme extrémité terminale.

- Le degré du x_i, noté $d_G(x_i)$ est la somme de $d_G^-(x_i)$ et $d_G^+(x_i)$, c'est à dire

$$d_G(x_i) = d_G^-(x_i) + d_G^+(x_i)$$

- Etant donné $A \subset X$, on définit :

$$\omega^+(A) = \{(x,y) \in U / x \in A \text{ et } y \in X-A\}$$

$$\omega^-(A) = \{(x,y) \in U / x \in X-A \text{ et } y \in A\}$$

On note:

$$\omega(A) = \omega^+(A) + \omega^-(A)$$

Un ensemble d'arcs (ou d'arêtes) de la forme $\omega(A)$ est appelé un cocycle du graphe.

- Un graphe $G=[X,U]$ est dit symétrique si :

$$\forall (x_i, x_j) \in U \Rightarrow (x_j, x_i) \in U$$

- Un 1-graphe $G=[X,U]$ est dit antisymétrique si:

$$\forall (x_i, x_j) \in U \Rightarrow (x_j, x_i) \notin U$$

- Un graphe $G=[X,U]$ est dit complet si:

$$\forall (x_i, x_j) \in U \Rightarrow \exists (x_i, x_j) \in U \text{ ou } (x_j, x_i) \in U$$

- Etant donné $G=[X,U]$, $A \subset X$
  $G'=[A,U']$ est un sous-graphe engendré par A dont

$$U' = \{(x,y) \in U / x, y \in A\}$$

- Etant donné $G=[X,U]$, $V \subset U$
  $G'=[X,V]$ est un graphe partiel engendré par V.

- Etant donné un graphe simple $G = [X,U]$, le graphe complémentaire $G = [X,\bar{U}]$ dont

$$\bar{U} = \{(x,y) / \forall x,y \in X \text{ et } (x,y) \notin U\}$$

- Etant donné un graphe $G = [X,U]$ et $A \subset X$, $V \subset U$, le sous-graphe partiel engendré par A et V est le graphe partiel de G_A engendré par V.

I.2. MATRICES ASSOCIEES A UN GRAPHE

I.2.1. Matrice d'incidence sommets-arcs

La matrice d'incidence sommets-arcs d'un graphe $G = [X,U]$ est une matrice $A = [a_{iu}]$, $i = 1,\ldots,N$; $u = 1,\ldots,M$ à coefficients entiers 0, +1, -1 telle que chaque colonne correspond à un arc de G, et chaque ligne à un sommet de G; si $u = (x_i, x_j) \in U$, la colonne u a tous ses termes nuls, sauf :

$$a_{iu} = +1$$
$$a_{ju} = -1$$

D'une façon équivalente, si on considère une ligne i quelconque (correspondant au sommet x_i), alors

$$\omega^+(x_i) = \{u / a_{iu} = +1\}$$
$$\omega^-(x_i) = \{u / a_{iu} = -1\}$$

Exemple:

La matrice d'incidence sommets-arcs du graphe de la figure 3 est:

$$\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \begin{array}{c} \begin{array}{ccccc} u_1 & u_2 & u_3 & u_4 & u_5 \end{array} \\ \begin{pmatrix} +1 & +1 & 0 & 0 & +1 \\ -1 & 0 & +1 & 0 & 0 \\ 0 & -1 & -1 & +1 & 0 \\ 0 & 0 & 0 & -1 & -1 \end{pmatrix} \end{array}$$

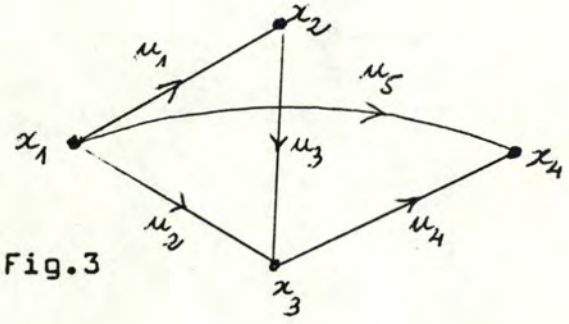


Fig.3

I.2.2. Matrice d'incidence sommets-arêtes

Soit un graphe $G = [X,U]$ où U est un ensemble d'arêtes.

La matrice d'incidence sommets-arêtes de G est une matrice à coefficients 0 ou 1, où chaque ligne i correspond à un sommet x_i de G, et chaque colonne à une arête u de G; de plus, si $u = (x_i, x_j)$, alors la colonne u a tous ses éléments nuls sauf:

$a_{iu} = +1$

$a_{ju} = +1$ (au lieu de -1 comme dans la matrice d'incidence sommets-arcs)

On dit qu'un graphe $G = [X,U]$ est biparti si l'ensemble des sommets X peut être partitionné en deux sous-ensembles X_1 et X_2 de telle sorte que, pour toute arête $(x_i, x_j) \in U$:

$$x_i \in X_1 \Rightarrow x_j \in X_2$$

$$x_i \in X_2 \Rightarrow x_j \in X_1$$

I.2.3. Matrice d'adjacence

Soit $G = [X,U]$ un 1-graphe, comportant éventuellement des boucles (mais pas plus d'une par sommet).

La matrice d'adjacence est une matrice $\mathcal{A}$ à coefficients 0 ou 1:

$$\mathcal{A} = [a_{ij}] \qquad \begin{matrix} i = 1, \ldots, N \\ j = 1, \ldots, M \end{matrix}$$

où chaque ligne correspond à un sommet de G, et où $a_{ij} = +1$ si et seulement si $(x_i,x_j) \in U$

Dans le cas non orienté, on peut aussi définir la matrice d'adjacence d'un graphe simple en considérant qu'à chaque arête (x_i,x_j), correspondent deux arcs (x_i,x_j) et (x_j,x_i). Dans ce cas, la matrice d'adjacence est symétrique.

I.2.4. Les différentes représentations d'un graphe

Pour décrire un graphe G, un certain nombre de représentations peuvent être utilisées. Il existe essentiellement deux grandes familles de représentations : la première utilise la matrice d'adjacence ou ses dérivées, la seconde la matrice d'incidence ou ses dérivées.

i/ A partir de la matrice d'adjacence

La matrice d'adjacence permet d'écrire soit des 1-graphes (orientés), soit des graphes simples (non orientés).

Etant donné un graphe $G = [X,U]$, $|X| = N$ et $|U| = M$, on doit avoir une matrice de N x N dimensions ayant M éléments non zéro.

Dans le cas des graphes peu denses ($M \ll N^2$ pour les graphes orientés, $M \ll 1/2\ N\ (\ N+1\ )$ pour les graphes non orientés) il y a donc une perte importante de places de mémoire, et alors il sera avantageux de décrire uniquement les éléments non nuls.

Pour cela, on utilisera deux tableaux a(.) de dimensions N+1 et b(.) de dimensions M (cas orienté) ou 2M (cas non orienté). Pour chaque sommet x_i, la liste

des successeurs de x_i est contenue dans le tableau b à partir de la case numéro a (x_i). On voit donc que l'ensemble des successeurs de x_i est contenu entre les cases $a(x_i)$ et $a(x_i+1)$ du tableau b(.) et l'on a :

$$d^+(x_i) = a(x_i+1) - a(x_i) \qquad \text{(cas orienté)}$$

$$d(x_i) = a(x_i+1) - a(x_i) \qquad \text{(cas non orienté)}$$

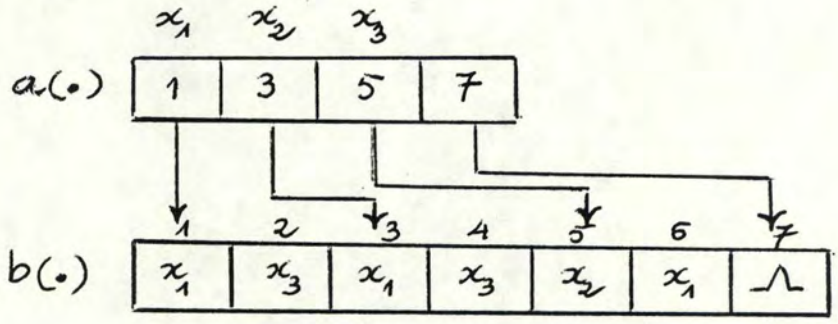
$$a(x_i) = \sum_{j=1,\, i-1} d^+(x_j) + 1$$

Cela se voit que dans le cas orienté, ceci revient à décrire par son application multivoque $x_i \rightarrow \Gamma x_i$.

Lorsqu'un graphe est pondéré, c'est à dire, l'on associe à chaque arc (ou arête) un poids p(u) qui peut être stocké dans un autre tableau p(.) en correspondance biunivoque avec b(.).

Exemple:

- Cas orienté : Considérons le 1-graphe de la figure 4



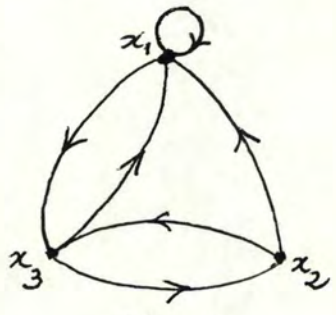


Fig. 4

- Cas non orienté : Considérons le graphe simple de la figure 5

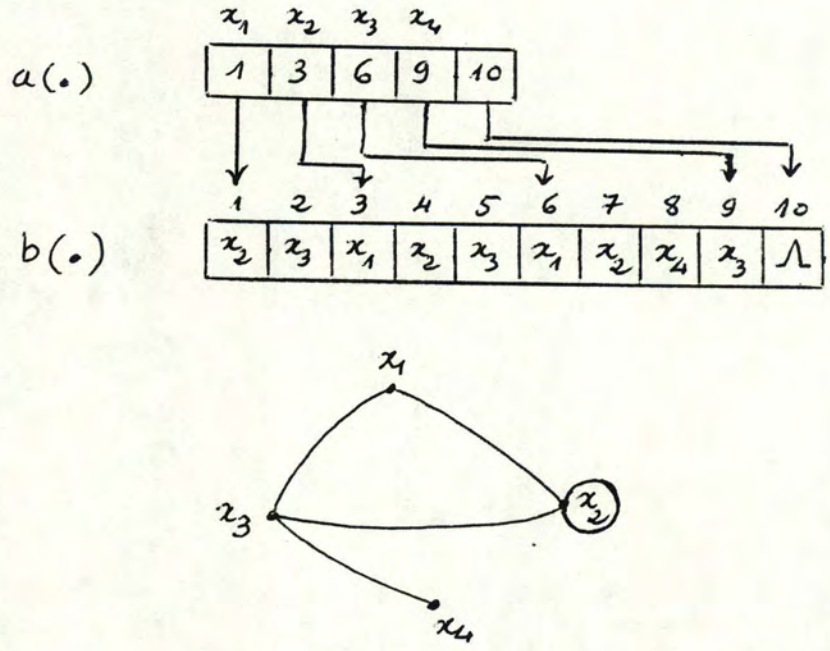


Fig. 5

ii/ A partir de la matrice d'incidence

La matrice d'incidence permet de décrire parfaitement la structure d'un multigraphe (sans boucle)

- Liste des arêtes

La première méthode consiste à définir deux tableaux a(.) et b(.) de dimensions M donnant pour chaque $u \in U$ les numéros a(u) et b(u) des extrémités. Dans le cas orienté, a(u) sera l'extrémité initiale et b(u) l'extrémité terminale. Ceci revient à décrire la matrice d'incidence colonne par colonne.

Contrairement à la matrice d'incidence, cette représentation permet de décrire des multigraphes comportant des boucles.

Exemple :

Considérons le multigraphe de la figure 6

	1	2	3	4	5	6	7	8
a(.)	x_1	x_3	x_1	x_2	x_2	x_3	x_2	x_2

b(.)	x_3	x_1	x_3	x_1	x_1	x_2	x_3	x_2

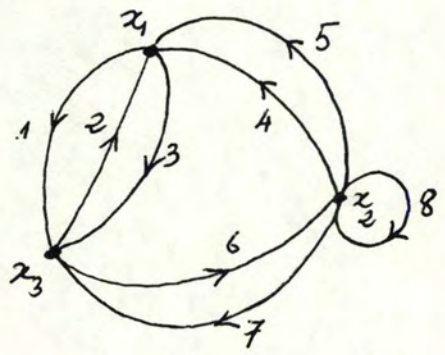


Fig. 6

Lorsque le graphe est pondéré, il suffit d'ajouter un tableau p(.) de dimensions M en correspondance biunivoque avec le tableau a(.)

- Liste des cocycles $w^+(x_i)$ ou $w(x_i)$

La deuxième méthode c'est qu'on peut décrire la matrice d'incidence ligne par ligne en indiquant pour chaque $x_i \in X$

- la liste $w^+(x_i)$ (cas orienté)
- la liste $w(x_i)$ (cas non orienté)

Pour cela, deux tableaux a(.) et b(.) sont utilisés:

$a(x_i)$ est l'adresse du commencement de la liste de cocycle de x_i dans le tableau b(.)

b(.) est le tableau des cocycles des sommets du graphe.

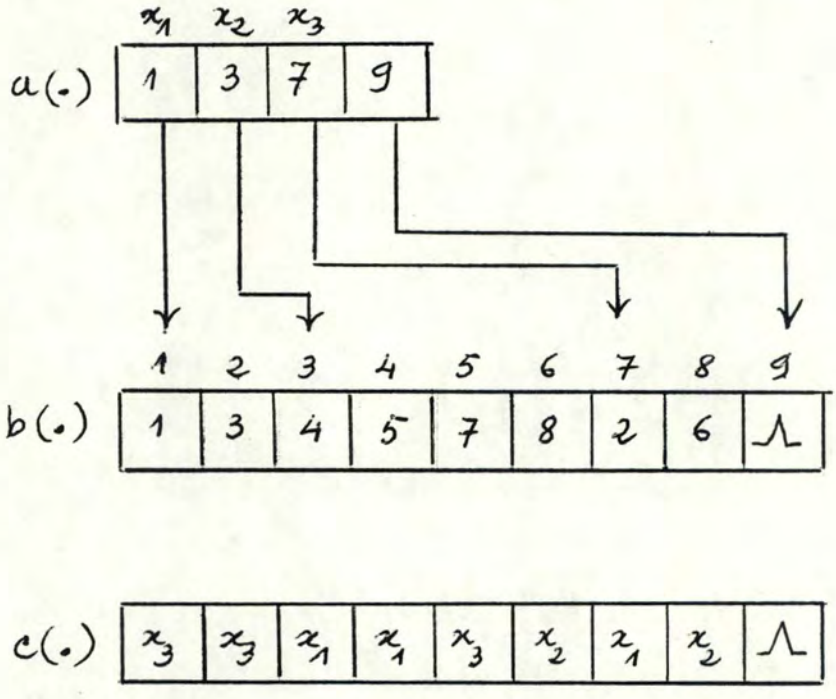
La liste des cocycles de x_i est commencée de $a(x_i)$ à $a(x_{i+1})-1$.

Pour savoir l'autre extrémité de l'arc (ou de l'arête), on pourra ajouter un troisième tableau c(.) en correspondance biunivoque avec b(.).

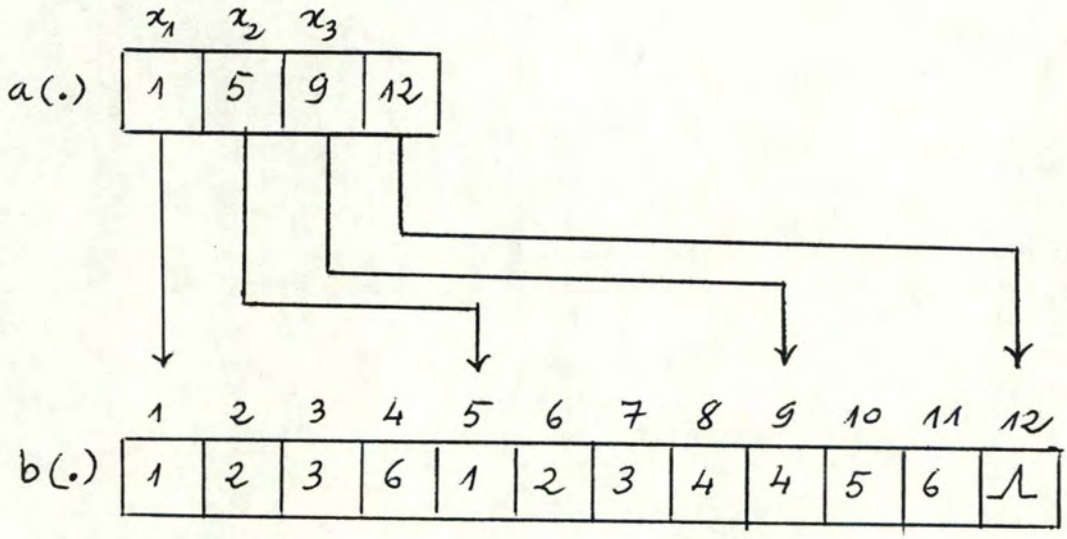
Là encore, si le graphe est pondéré, il suffira de disposer un tableau supplémentaire p(.) en correspondance biunivoque avec b(.) et c(.).

Exemple:

- Cas orienté: Reprenons le multigraphe orienté de la figure 6



- Cas non orienté: Considérons un multigraphe non orienté de la figure 7.



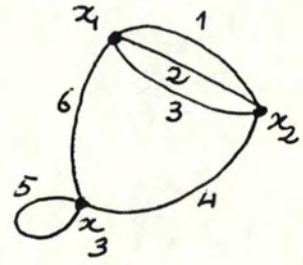


Fig. 7

I.3. CONNEXITE

I.3.1. Chaîne. Chaîne élémentaire. Cycle. Cycle élémentaire

Une chaîne de longueur q (de cardinalité q) est une séquence de q arcs :

$$L = \{u_1, u_2, \ldots, u_q\}$$

où chaque u_r, $2 \leq r \leqslant q-1$, ait une extrémité commune avec u_{r-1} et l'autre extrémité commune avec u_{r+1}.

Si $u_1 = (x_1, y_1)$ et $u_q = (x_q, y_q)$, alors x_1 et y_q sont appelés les extrémités de la chaîne L. On dit aussi que la chaîne L joint les sommets x_1 et y_q.

Exemple:

Considérons un 1-graphe de la figure 8

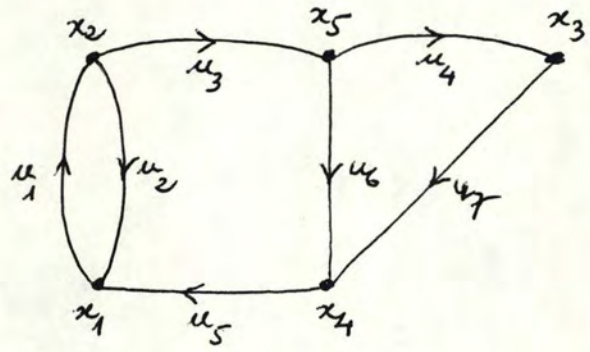


Fig. 8

$L = \{u_2, u_5, u_6, u_4\}$ est une chaîne allant de x_2 à x_3.

- Une chaîne élémentaire est une chaîne telle qu'en la parcourant, on ne rencontre pas deux fois le même sommet, c'est à dire

L est élémentaire $\Leftrightarrow \forall u = (x, y) \in L$, $d_G(x) \leqslant 2$ et $d_G(y) \leqslant 2$

- Un cycle est une chaîne dont les extrémités coïncident.

- Un cycle élémentaire est un cycle minimal, c'est à dire ne contenant strictement aucun autre cycle. En parcourant un cycle élémentaire, on ne rencontre pas deux fois le même sommet (sauf le sommet choisi comme origine du parcours).

I.3.2. Chemin. Chemin élémentaire. Circuit. Circuit élémentaire

- Un chemin de longueur q est une séquence de q arcs: $P = u_1, u_2, \ldots, u_q$ avec

$$u_1 = (x_o, x_1)$$
$$u_2 = (x_1, x_2)$$
$$\vdots$$
$$u_q = (x_{q-1}, x_q)$$

x_o : l'extrémité initiale du chemin P

x_q : l'extrémité terminale du chemin P

- Le chemin élémentaire, circuit et circuit élémentaire sont définis comme la chaîne élémentaire, cycle et cycle élémentaire en ajoutant la notion de l'orientation.

I.3.3. Connexité. Nombre de connexité

- Un graphe est dit connexe si:

$\forall x, y \in X$, alors

$$x \; R \; y \iff \begin{cases} - \text{ soit } x=y \\ - \text{ soit il existe une chaîne joignant } x \text{ et } y \end{cases}$$

R est une relation d'équivalence.

Le nombre p de classes d'équivalence distinctes est appelé le nombre de connexité du graphe.

Les sous-graphes $G_1, G_2, \ldots, G_p$ engendrés par les sous-ensembles $X_1, X_2, \ldots, X_p$ sont appelés les composants connexes du graphe G.

La vérification de la connexité d'un graphe est un des premiers problèmes de la théorie des graphes, par exemple la vérification de la connexité d'un réseau électrique, d'un réseau téléphonique, etc...

I.3.4. Point d'articulation. k-connexité. k-arête-connexité

Un point d'articulation d'un graphe est un sommet dont la suppression augmente le nombre de composantes connexes.

Un isthme est une arête dont la suppression a le même effet.

On dira qu'un multigraphe G connexe est k-arête-connexe s'il ne peut être déconnecté par l'élimination de moins de k arêtes.

I.3.5. Graphe fortement connexe.Composantes fortement connexes

Un graphe est dit fortement connexe, si
$\forall x,y \in X$, alors

$$xRy \Leftrightarrow \begin{cases} - \text{ soit } x=y \\ - \text{ soit il existe à la fois un chemin de } x \text{ à } y \text{ et un chemin de } y \text{ à } x. \end{cases}$$

R est une relation d'équivalence.

Le nombre q de classes d'équivalence distinctes est appelé le nombre de connexité forte du graphe.

Les sous-graphes G_1, G_2,..., G_q engendrés par les sous-ensembles X_1, X_2,..., X_q sont fortement connexes et sont appelés les composantes fortement connexes de G.

CHAPITRE II

QUELQUES PROBLEMES ET LEURS ALGORITHMES

En raison de ses larges applications, l'analyse de la théorie des graphes a été développée rapidement au cours des années récentes. De plus, une analyse successive dépend beaucoup de l'existence des algorithmes efficaces.

II.1. LE PROBLEME DU PLUS COURT CHEMIN

II.1.1. Définition et notation

Considérons un graphe sans boucle $G = [X,U]$; à chaque arc duquel est associé un nombre réel, ce nombre sera désigné par $l(u) = l_{ij}$ pour l'arc $u = (x_i,x_j)$ et appelé longueur de l'arc. Soit μ un chemin quelconque de G, on appellera longueur du chemin μ le nombre :

$$l(\mu) = \sum_{u \in \mu} l(u)$$

Le problème du plus court chemin entre deux sommets x_i et x_j sera de trouver un chemin μ de x_i à x_j dont la longueur totale:

$$l(\mu) = \sum_{u \in \mu} l(u)$$

est minimum.

Remarques importantes:

i/ Entre x_i et x_j (descendant de x_i) donnés, il existe un chemin de longueur finie au minimum si G est sans circuit de longueur strictement négative (circuit absorbant).

ii/ Il est clair que si l'on inverse le signe de la longueur de chacun des arcs d'un graphe, tout chemin primitivement de longueur minimale sera après cette inversion un chemin de longueur maximale (et inversement). Ainsi, tous les résultats et algorithmes qui n'impliquent aucune restriction sur le signe des longueurs, s'appliquent indifféremment aux plus courts comme aux plus longs chemins.

II.1.2. Divers aspects du problème

Pour différencier les divers aspects de ce problème, il suffit d'adopter l'un après l'autre les trois points de vue suivants:

i/ Le graphe G peut avoir, ou au contraire ne doit pas avoir de circuit: cette distinction est importante car l'absence de circuit permet de ranger les sommets du graphe de manière à ce que le calcul soit séquentiel; en outre on ne court pas le risque de rechercher la solution d'un problème qui n'en a pas.

ii/ La longueur des arcs est de signe quelconque ou au contraire elle a un signe déterminé.

iii/ La recherche des chemins de longueur minimale doit être entreprise entre deux sommets seulement.

II.1.3. Principaux algorithmes

Les algorithmes proposés pour la recherche des chemins de longueur minimale sont fort nombreux, mais ici nous allons présenter seulement deux parmi eux. En premier lieu, nous allons décrire le plus simple et efficace algorithme à résoudre ce problème dans le cas $l_{ij} \geq 0$ ($\forall$ i,j) et après c'est la méthode du cas

général de $l_{ij} \gtrless 0$ en supposant qu'il n'existe pas de circuit absorbant.

i/ Cas où $l_{ij} \geq 0$

Le plus efficace algorithme du chemin le plus court de s à t est donné par DIJKSTRA. Généralement, la méthode est basée sur l'affectation temporaire de labels aux noeuds, et chaque label d'un noeud est une borne supérieure de la longueur du chemin de s à ce noeud. Ces labels sont continuellement réduits par une procédure itérative jusqu'au moment où ils sont égaux à la longueur du plus court chemin de s au noeud donné. Les détails de cette méthode sont comme suit :

Algorithme DIJKSTRA ($l_{ij} \geq 0$)

Soit $\pi(x_i)$ le label du noeud x_i

Soit $\bar{S}$ l'ensemble des noeuds ayant le label temporaire.

Initialisation

Etape 1. $\pi(s) = 0$

$\bar{S} = X - \{s\}$

$\pi(x_i) = \infty \qquad \forall x_i \in \bar{S}$

$p = s$

Mise à jour des labels.

Etape 2. Faire pour tout $x_i \in \Gamma(p)$

$$\pi(x_i) = \min \left[\pi(x_i), \pi(p) + l(p, x_i) \right]$$

Fixant le label comme permanent.

Etape 3. Sélectionner $x_i^* \in \bar{S}$ tel que

$$\pi(x_i^*) = \min \left[\pi(x_i) \right]$$

Etape 4. $\bar{S} = \bar{S} - \{x_i^*\}$ et $p = x_i^*$

Etape 5.

- Dans le cas où le chemin de s à t est désiré.

Si p = t, l(p) est la longueur du plus court chemin demandé. Terminer.

Sinon aller à l'étape 2.

- Dans le cas où le chemin de s à tout noeud est demandé.

Si $|\bar{S}| = 0$, alors tout $\pi(x_i)$ est la longueur du plus court chemin de s à x_i. Terminer.

Sinon aller à l'étape 2.

ii/ Cas général ($\ell_{ij} \gtrless 0$)

La méthode est proposée par FORD, MOORE et BELLMAN de l'année 1950. Elle est itérative et basée aussi sur l'assignement du label au noeud, où à la fin de la k ième itération, les labels représentent les valeurs de leur plus court chemin (de s à tout noeud) ne contenant pas plus de k+1 arcs. La différence entre cette méthode et celle du DIJKSTRA c'est qu'aucun label n'est considéré comme final dans le traitement sauf au moment où tous les labels ont la valeur finale.

ii.1/ Algorithme du cas général [BELLMAN 1958]

Soit $\pi^k(x_i)$ le label du noeud x_i à la fin de (k+1) ième itération.

Initialisation

Etape 1. $S = \Gamma(\ ),\ k=0\ ,\ \pi^1(1) = 0$

$$\pi^1(x_i) = \begin{cases} \ell(s, x_i) & \forall x_i \in \Gamma(s) \\ \infty & \forall x_i \notin \Gamma(s) \end{cases}$$

Mise à jour des labels.

Etape 2. $T_i = \Gamma^{-1}(x_i) \cap S$

Faire pour tout $x_i \in \Gamma(s)$

$$\pi^{k+1}(x_i) = \min\left[\pi^k(x_i),\ \min_{x_j \in T_i}\{\pi^k(x_j) + \ell(x_j, x_i)\}\right]$$

(S: l'ensemble des noeuds ayant un chemin de s à eux ne contenant pas plus de k arcs)

Test de terminaison.

Etape 3.

(a) Si $k \leq n-1$ et $\pi^{k+1}(x_i) = \pi^k(x_i) \quad \forall x_i \in X$

alors la solution optimale sera obtenue et les labels seront les longueurs des plus courts chemins. Terminer.

(b) Si $k < n-1$ mais $\pi^{k+1}(x_i) \neq \pi^k(x_i) \quad \exists x_i \in X$

Aller à l'étape 4.

(c) Si $k = n-1$ et $\pi^{k+1}(x_i) \neq \pi^k(x_i) \quad \exists x_i \in X$

il existe un circuit de longueur négative.

Préliminaires de l'itération suivante.

Etape 4. Mise à jour de S

$$S = \{x_i / \pi^{k+1}(x_i) \neq \pi^k(x_i)\}$$

Etape 5. $k = k+1$

Aller à l'étape 2.

Les chemins pourraient être obtenus en ajoutant de plus un autre label $\theta^k(x_i)$ à chaque noeud x_i, où $\theta^k(x_i)$ est le noeud justement avant x_i sur le plus court chemin de s à x_i dans la k ième itération. On peut commencer par

$$\theta^1(x_i) = s \quad \forall x_i \in \Gamma(s)$$

et $$\theta^1(x_i) = 0 \quad \forall x_i \notin \Gamma(s)$$

Les labels $\theta^k(x_i)$ peuvent être mis à jour à l'étape 2:

$$\theta^{k+1}(x_i) = \theta^k(x_i) \text{ si } \pi^k(x_i) \leq \min_{x_j \in T_i} \{\pi^k(x_j) + l(x_j, x_i)\}$$

ou

$$\theta^{k+1}(x_i) = x_j \text{ si } x_j \in T_i / \pi^k(x_j) + l(x_j, x_i) \text{ est minimum.}$$

Si ∇ est le vecteur des labels θ à la fin de l'algorithme, le plus court chemin de s à x_i est:

$$S, \ldots, \theta^3(x_i), \theta^2(x_i), \theta(x_i), x_i$$

où $\theta^i(x_i) = \underbrace{\theta(\theta(\ldots \theta}_{i \text{ fois}}(x_i))\ldots)$

ii.2. Exemple

Considérons le graphe de la figure 9. Il nous demande de chercher les plus courts chemins de l'un aux autres ou de détecter le circuit négatif s'il existe.

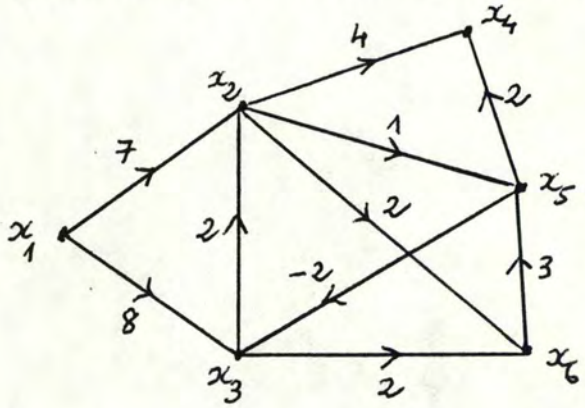


Fig. 9

L'algorithme procède comme suit :

Initialisation

Etape 1. $S = 1$, $S = \Gamma(x_i) = \{x_2, x_3\}$

$\pi^1(x_1) = 0$, $\pi^1(x_2) = 7$, $\pi^1(x_3) = 8$

$\pi^1(x_i) = \infty$ pour les autres si k=1

Première itération

Etape 2. $\Gamma(S) = \{x_2, x_4, x_5, x_6, x_3\}$. Alors pour x_2 :

$$T_2 = \{x_1, x_3\} \cap \{x_2, x_3\} = \{x_3\},$$

et $\pi^2(x_2) = \min[\pi^1(x_2), \{\pi^1(x_3) + \ell(x_3, x_2)\}]$

$= \min[7, 10] = 7$

pour x_4 : $T_4 = \{x_2, x_5\} \cap \{x_2, x_3\} = \{x_2\}$

et $\pi^2(x_4) = \min [\infty, 11] = 11$

pour x_5 : $T_5 = \{x_2, x_6\} \cap \{x_2, x_3\} = \{x_2\}$

et $\pi^2(x_5) = \min [\infty, (7+1)] = 8$

pour x_6 : $T_6 = \{x_2, x_3\} \cap \{x_2, x_3\} = \{x_2, x_3\}$

$\pi^2(x_6) = \min [\infty, \min\{(7+2), (8+2)\}] = 9$

pour x_3 : $T_3 = \{x_1, x_5\} \cap \{x_2, x_3\} = \phi$

$\pi^2(x_3) = 8$

Les labels $\pi^2(x_i)$ maintenant sont :

$[0, 7, 8, 11, 8, 9]$

Etape 3 (b) : aller à l'étape 4

Etape 4 : $S = \{x_4, x_5, x_6\}$

Etape 5 : $k=2$, aller à l'étape 2

Deuxième itération

Etape 2 : $\Gamma(S) = \{x_4, x_3, x_5, x_6\}$. Alors

pour x_4 : $T_4 = \{x_2, x_5\} \cap \{x_4, x_5, x_6\} = \{x_5\}$

$\pi^3(x_4) = \min [10, \{8+2\}] = 10$

pour x_3 : $T_3 = \{x_1, x_5\} \cap \{x_4, x_5, x_6\} = \{x_5\}$

$\pi^3(x_3) = \min [8, \{8-2\}] = 6$

pour x_5 : $T_5 = \{x_2, x_6\} \cap \{x_4, x_5, x_6\} = \{x_6\}$

$\pi^3(x_5) = \min [8, \{9+3\}] = 8$

pour x_6 : $T_6 = \{x_2, x_3\} \cap \{x_3, x_4, x_5, x_6\} = \{x_3\}$

$\pi^3(x_6) = \min [9, \{8+2\}] = 9$

Les labels $\pi^3(x_i)$ maintenant sont :

$[0, 7, 6, 10, 8, 9]$

Etape 3 (b) : Aller à l'étape 4

Etape 4 : $S = \{x_3, x_4\}$

Troisième itération

Etape 2 : $\Gamma(S) = \{x_2, x_6, x_4, x_3\}$

$T_2 = \{\}$, $\pi^4(x_2) = 7$

$T_6 = \{x_2, x_3\}$, $\pi^4(x_6) = 8$

$T_4 = \{x_2\}$, $\pi^4(x_4) = 10$

$T_3 = \{\}$, $\pi^4(x_3) = 6$

Les labels $\pi^4(x_i)$ maintenant sont

$$[0, 7, 6, 10, 8, 8]$$

Quatrième itération

Etape 2 : $\Gamma(S) = \{x_6\}$

$T_6 = \{\}$, $\pi^4(x_6) = 8$

Les labels $\pi^5(x_i)$ maintenant sont

$$[0, 7, 6, 10, 8, 8]$$

Etape 3. Terminer

Le vecteur de label $\pi^4(x_i)$ est le même que $\pi^5(x_i)$ et alors ces labels sont les longueurs du plus court chemin.

II.2. LE PROBLEME CENTRAL DE L'ORDONNANCEMENT

II.2.1. Définition et notation

Etant donné un objectif qu'on se propose d'atteindre et dont la réalisation suppose l'exécution préalble de multiples tâches, soumises à de nombreuses contraintes, déterminer l'ordre et le calendrier d'exécution des diverses tâches.

Ici nous étudierons le cas particulier le plus important (problème central de l'ordonnancement) où les seules contraintes sont des contraintes de successions dans le temps (l'exécution de la tâche j ne peut être commencée que lorsque la tâche i est achevée). On supposera donc que l'objectif à atteindre se décompose en travaux élémentaires ou tâches - chaque tâche i est caractérisée par sa durée d_i et par les contraintes qui la lient à d'autres tâches.

La représentation par un graphe d'un problème d'ordonnancement permettra une bonne appréhension globale du problème.

L'étude de ce graphe permettra alors d'identifier les tâches prioritaires et de détecter à temps pour prendre les mesures correctives nécessaires, les retards ou les dépassements de moyens.

II.2.2. Le graphe potentiels-tâches [B.ROY 1960]

A partir du projet donné on construit le graphe suivant :

- A chaque tâche i, on associe un sommet x_i du graphe.
- Si la tâche i doit précéder la tâche j, on définira un arc (x_i, x_j) de longueur d_{x_i}
- Le graphe doit être sans circuit.
- On ajoute au graphe deux tâches supplémentaires α et ω , correspondant respectivement la tâche de début et la tâche de fin de longueurs $d_\alpha = d_\omega = 0$

Le travail commençant à la date 0, on cherche un ordonnancement qui minimalise la durée totale du travail .

*La date au plus tôt t_i de début de la tâche i est:

$$t_i = \max_{x_j \in \Gamma^{-1} x_i} (t_j + d_j)$$

c'est à dire que t_i est égale à la longueur du plus long chemin $\ell(\alpha, x_i)$ de α à x_i

* La durée minimale du projet t_ω est la longueur du plus long chemin de α à ω .

* La date au plus tard T_i pour commencer la tâche i est:

$$T_i = \min_{x_j \in \Gamma x_i} (T_j - d_i)$$

avec $T_\omega = t_\omega$

ce qui montre que

$$T_i = t_\omega - \ell(x_i, \omega)$$

* La marge m_i de la tâche i est définie comme la différence entre la date au plus tôt et la date au plus tard:

$$m_i = T_i - t_i$$

* Les tâches critiques sont les tâches dont la marge est nulle.

II.2.3. Les algorithmes

Dans les algorithmes correspondant au problème d'ordonnancement, on a utilisé une fonction particulière appelé " Rang ". La fonction Rang associé à un graphe sans circuit est obtenu en affectant, à chaque $x_i \in X$ un nombre entier positif $R(x_i)$ tel que:

- $R(x_0) = 0$ où x_0 est la racine du graphe ($|\Gamma^{-1} x_0| = 0$)
- $R(x_i)$ = le nombre d'arcs dans un chemin de cardinalité maximum entre x_0 et x_i.

L'algorithme pour obtenir cette fonction Rang est le suivant :

i/ Algorithme : Recherche de la fonction rang d'un graphe sans circuit.

Etape 1: Initialisation

Poser $d_i^- = |\Gamma^{-1} x_i| \quad \forall x_i \in X$,

$k = 0$

$S = X$

Etape 2 : La mise à jour de $R(x_i)$ et d_i^-

Soit S_k l'ensemble des sommets $x_i \in S$ tel que $d_i^- = 0$.

Pour tout $x_i \in S_R$, faire

- $R(x_i) = k$
- $\forall x_j \in \Gamma x_i$, $d_j^- = d_j^- - 1$

Etape 3 : Test de terminaison

$S = S - S_k$

$k = k + 1$

Si $|S| = 0$, terminer

Sinon aller à l'étape 2.

Les deux algorithmes suivants qui nous donnent respectivement la date au plus tôt et la date au plus tard d'un problème d'ordonnancement.

ii/ Algorithme : Recherche des dates au plus tard d'un problème d'ordonnancement.

Etape 1 :Initialisation

$t_\alpha = 0$

Etape 2 : Appeler la fonction rang du graphe.

Etape 3 : Prendre les sommets x_j par rang croissant et faire

$$t_j = \max_{x_i \in \Gamma^{-1} x_j} (t_i + d_i)$$

iii/ Algorithme : Recherche des dates au plus tard d'un problème d'ordonnancement.

Etape 1. Initialisation

Poser $T_\omega = t_\omega$

Etape 2. Prendre les sommets x_j par décroissance et faire

$$T_j = \min_{x_i \in \Gamma x_j} (T_i) - d_j$$

iv/ Exemple: cfr. à l'exemple d'utilisation dans l'annexe.

II.3. LE PROBLEME DE PARCOURS EULERIENS

II.3.1. Définitions

Soit $G = [X,U]$ un graphe non orienté.

Une chaîne eulérienne est une chaîne empruntant une fois et une fois seulement chaque arête de G.

Un cycle eulérien est une chaîne eulérienne dont les extrémités coïncident.

II.3.2. Problème (le problème des ponts de KOENIGSBERG)

La ville de Koenigsberg est traversée par la rivière Pregel qui coule de part et d'autre de l'île de Kneiphof et possède 7 ponts (Fig.10-a)

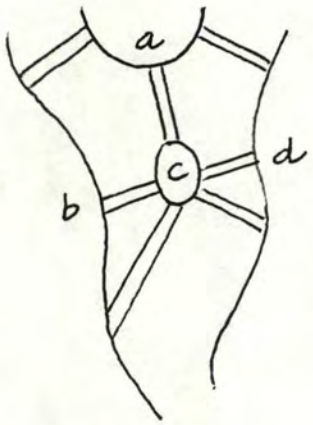


Fig. 10-a

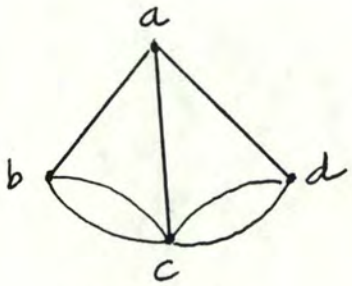


Fig.10-b

Un piéton pourrait-il, en se promenant, traverser chaque pont une fois et une fois seulement ?

Le problème équivaut à la recherche d'une chaîne eulérienne dans le multigraphe de la fig. 10-b où l'on associe un sommet à chacune des quatre régions a, b, c, d et une arête à chaque pont.

II.3.3. Algorithme de recherche d'une chaîne eulérienne

i/ Théorème

Un multigraphe $G = [X,U]$ connexe, admet une chaîne eulérienne si et seulement si le nombre de sommets de degré impair est 0 ou 2 [12]

Un graphe G satisfaisant au théorème sera dit eulérien

ii/ Algorithme de recherche d'une chaîne eulérienne

L'algorithme que nous allons présenter se déduit directement de la démonstration du théorème précédent.

On construit la chaîne L d'origine a et n'empruntant pas deux fois la même arête. Si toutes les arêtes de G n'ont pas été utilisées, soit G_1 le graphe partiel engendré par les arêtes restantes. On choisit un sommet x_1 de G_1 de degré plus grand que 0 dans G_1, et on construit suivant le même principe un cycle μ_1 d'origine x_1 et n'empruntant pas deux fois la même arête de G_1. Lorsqu'on revient à x_1, la chaîne L est augmentée du cycle μ_1. Si toutes les arêtes de G_1 ont été utilisées, c'est terminé. Sinon on fait la même façon à G_2 et ainsi de suite jusqu'à épuisement des arêtes de G.

Quelques notions :

* $\forall u \in U$, $\sigma(u)$: le numéro de l'arête suivant immédiatement l'arête u.

* $\forall x \in X$, $\omega(x)$: l'ensemble des arêtes incidentes de x

et $|\omega(x)|$: le degré de x.

* $u=(x,y)\in\omega(x)$, $\delta_x(u)=y$ le numéro de l'autre extrémité de l'arête u.

Algorithme (1)

(a)(Initialisation)

$\sigma(u)=0 \quad \forall u\in U$

$\alpha(x)=|\omega(x)| \quad \forall x\in X$

$v=0$ (* nombre d'arêtes dans la chaîne *)

$\bar{u}=|U|+1$ (* $\bar{u}$=dernière arête empruntée*)

$\forall x\in X$, $\varphi(x)$ est le numéro de la dernière arête utilisée pour arriver au sommet x;

$\varphi(a)=|U|+1$; $\varphi(x)=0 \quad \forall x\in X\setminus\{a\}$

$\bar{\bar{u}}=-1$

$x=a$; $t=b$ (* a,b sont les extrémités de la chaîne*)

(b) (Ajouter une arête à la chaîne)

Si $\alpha(x)=0$ aller en (d). Sinon

Soit u l'arête numéro $\alpha(x)$ dans le cocycle $\omega(x)$

Faire : $\alpha(x)\leftarrow\alpha(x)-1$

Si: $u=\bar{u}$, ou si $\sigma(u)\neq 0$, retourner en (b). Sinon

$y=\delta_x(u)$

$\varphi(y)=u$

$\sigma(\bar{u})=u$ (* mettre la dernière arête après l'arête u venant d'insérer dans la chaîne *)

$v=v+1$ (* incrémenter le nombre d'arêtes de la chaîne *)

$x=y$

Si x = t, aller en (c)

Sinon: $\bar{u}$ = u, et retourner en (b)

(c) $\sigma(u)=\bar{\bar{u}}$

Si v = M Fin. Sinon:

(d) Sélectionner un sommet z quelconque tel que $\alpha(k) \neq 0$

Poser: $x = z$; $t = z$

$\bar{u} = \varphi(x)$

$\bar{\bar{u}} = \sigma(\bar{u})$

Retourner en (b)

II.4. LE PROBLEME DU " POSTIER CHINOIS " (NON ORIENTE)

II.4.1. Définition et notation

Soit $G = [X,U]$ un graphe connexe non orienté et associons à chaque arête $u \in U$ un nombre $l(u) \geq 0$.

Le problème du " Postier chinois " (non orienté) est de déterminer un cheminement entre deux sommets a et b donnés dans G utilisant chaque arête de G au moins une fois, et de longueur totale minimale (MEI KO KWAN 1962).

On peut toujours supposer que a et b du cheminement sont confondus. Dans le cas contraire, il suffirait de rajouter à G une arête $u = (a,b)$ tel que $l(u) = 0$: à tout parcours fermé de longueur totale minimale sur $G' = [X, U \cup \{(a,b)\}]$, correspond un parcours entre a et b sur $G = [X,U]$ et de même longueur.

Si G est eulérien, alors le cycle eulérien de G sera une solution optimale du problème.

En général, G n'est pas eulérien, et par suite, il existe un certain nombre de sommets $Ximp \subset X$ de degré impair.

II.4.2. Théorème

i/ Lemme 1

X_{imp} est de cardinalité paire. [12]

Le problème revient alors à rajouter un certain nombre d'arêtes au graphe G initial de manière à le rendre eulérien tout en minimalisant la somme des longueurs des arêtes supplémentaires.

On conviendra de n'ajouter une arête $u' = (x_i, x_j)$ que s'il existe déjà $u = (x_i, x_j)$ dans G et $l(u') = l(u)$. L'arête u' est appelée une copie de u.

ii/ Lemme 2

Soit U' l'ensemble des copies d'arêtes de G tel que $G' = (X, U+U')$ est eulérien.

Soit x_i^1 un sommet de degré impair dans G. Alors l'ensemble U' contient une chaîne élémentaire joignant x_i^1 à un autre sommet $x_j \neq x_i^1$, $x_j \in X_{imp}$ [12]

iii/ Lemme 3

Soit $x_i, x_j \in X_{imp}$ et

$$L' = \{u'_1, u'_2, \ldots, u'_k\} \quad (u_i' \in U')$$

et $L = \{u_1, u_2, \ldots, u_k\}$ $(u_i \in U' / u_i'$ est une copie de $u_i)$

Alors L est une chaîne de longueur minimale entre x_i et x_j dans G. [ ]

Considérons maintenant le graphe non orienté complet $\mathcal{K}(X_{imp})$ construit sur X_{imp} et où chaque arête (x_i, x_j) $(x_i \in X_{imp}, x_j \in X_{imp})$ est munie d'un poids p_{ij} égal à la longueur de la plus courte chaîne entre x_i et x_j dans G et cette longueur peut être calculée à l'aide

d'un algorithme du plus court chemin.

A chaque arête de $\mathcal{K}(X_{imp})$ correspond ainsi une chaîne de G.

iv/ Théorème

A une solution optimale du " Postier chinois " on peut faire correspondre un couplage parfait de poids minimum dans le graphe $\mathcal{K}(X_{imp})$ et réciproquement. [12]

D'après le théorème, la résolution du problème de " Postier chinois " non orienté se ramène à la détermination d'un couplage parfait de poids minimum [12] .

II.4.3. Solution

Pour résoudre le problème du " Postier chinois " on doit effectuer des tâches suivantes :

- Etablir l'ensemble X_{imp} des sommets du graphe de degré impair
- Déterminer les longueurs de toutes les plus courtes chaînes entre les couples de sommets de X_{imp} dans G.
- Construire le graphe complet $\mathcal{K}(X_{imp})$ où le poids d'une arête (x_i, x_j) est la longueur de la plus courte chaîne entre x_i et x_j $(x_i \in X_{imp}, x_j \in X_{imp})$
- Faire un couplage parfait de poids minimum sur $\mathcal{K}(X_{imp})$
- Chercher les chaînes (*) de G correspondant au couplage

Enfin la solution optimale du problème est donc obtenue en ajoutant au graphe initial les arêtes des chaînes (*)

II.5. LE PROBLEME DU VOYAGEUR DE COMMERCE (NON ORIENTE)

II.5.1. Enoncé

Un représentant de commerce doit rendre visite à n clients $x_1, x_2, \ldots, x_n$ en partant d'une ville x_o et revenir à son point de départ. Il connaît les distances d_{oj} qui séparent le dépôt x_o de chacun de ses clients x_j, ainsi que la distance d_{ij} séparant deux clients quelconques x_i et x_j ($D = d_{ij}$) .

Dans quel ordre doit-il rendre visite à ses clients pour que la distance totale parcourue soit minimale ? Ce problème revient à rechercher un cycle hamiltonien de longueur totale minimale dans le graphe complet G construit sur l'ensemble des sommets

$$X = \{x_o, x_1, \ldots, x_n\}$$

les arêtes étant munies des longueurs d_{ij} .

II.5.2. Solution

Pour résoudre le problème, on propose d'utiliser les techniques d'énumération basées sur une exploration par séparation et évaluation [cfr. chap. 11, 12], [7], [14]

Algorithme de Little

Etape 1. Recherche d'une évaluation par défaut [cfr. 2.1. chap. 11 , 12]

- Calculer

$$h^i = \min_i d_{ij} \geqslant 0$$

-Soustraire h^i à tous les éléments d'une même ligne i de D.

- Calculer

$$h_j = \min_j d_{ij} \geqslant 0$$

- Soustraire h_j à tous les éléments d'une même colonne j de D (* La nouvelle matrice obtenue D' a tous les éléments positifs $d'_{ij} \geqslant 0$ et contient au moins

un zéro par ligne et par colonne *)

- Calculer l'évaluation par défaut de l'ensemble des circuits hamiltoniens

$$ev(\Omega) = h = \sum_i h^i + \sum_j h_j$$

Etape 2. La procédure par séparation et évaluation séquentielle (S.E.S.) [chap. 11, 12]

- Calculer les θ_{ij} correspondant aux $d'_{ij} = 0$

$$\theta_{ij} = \alpha_i + \beta_j$$

où $\alpha_i = \min_{k \neq i} d'_{kj} \geq 0$

et $\beta_j = \min_{l \neq j} d'_{il} \geq 0$

- Séparer le couple (i,j) qui maximise θ_{ij}
- Calculer $ev(\bar{ij}) = ev(\Omega) + \theta_{ij}$

et $ev(ij) = ev(\Omega) + \theta_{ji}$

- Simplifier la matrice D' par élimination de la ligne i et de la colonne j. On fait de plus d'_{ij} = HIGHVALR pour éliminer le circuit possible

n ← n-1 (n est le nombre de noeuds du graphe)

Si n = 1 alors FIN

Sinon aller à l'étape 1

Remarquons que la matrice $D = [d_{ij}]$ est non symétrique à cause des sens uniques.

DEUXIEME PARTIE

LE LOGICIEL D'AIDE A LA THEORIE DES GRAPHES

G R A P H L O G

INTRODUCTION

Le but de ce mémoire est de créer un logiciel d'aide à la théorie de graphe appelé GRAPHLOG, qui a l'objectif de nous offrir les facilités à résoudre les problèmes pouvant représenter en graphe.

Après vous avoir fait connaître quelques notions de graphe et avoir élaboré les problèmes les plus fréquents et leurs algorithmes, nous allons étudier les outils qui seront utilisés à l'implémentation du logiciel et sa structure.

Dans cette deuxième partie, nous allons aborder l'environnement du logiciel au premier chapitre, c'est à dire sur quelle machine implémentons-nous et de quel langage est-il traduit le logiciel.

Le second chapitre donnera une vue globale du logiciel à l'aide d'un schéma des états du logiciel et leurs explications.

Dans le troisième chapitre, nous présenterons les structures de données concernant la représentation d'un graphe, la cohérence des algorithmes et l'archivage d'un graphe sur un fichier.

Le chapitre suivant va préciser les modules du logiciel et leurs spécifications, du fait qui nous permet de savoir le fonctionnement intérieur du logiciel.

Et nous terminerons cette deuxième partie en donnant à l'utilisateur une probabilité de faire une extension au logiciel.

CHAPITRE I

L'ENVIRONNEMENT DU LOGICIEL

La construction efficace d'un logiciel demande des outils impeccables. Ainsi, on a décidé d'implémenter le logiciel sur le VAX-11/780 de l'Institut d'Informatique de Namur et de le traduire par le langage VAX11-PASCAL.

I.1. LE VAX-11 / 780

Le VAX-11/780 est un ordinateur qui est produit par le DIGITAL.

Son architecture et son système d'exploitation VAX/VMS sont conçus et crées ensemble. On est certain que le VAX architecture rehaussait l'efficacité du système d'exploitation VAX/VMS et que le système d'exploitation donne l'avantage au VAX processeur.

Le VAX processeur offre 32-bit d'adressage virtuel, une gestion sophistique de mémoire et une mécanique de protection et tous ceux-ci sont complètement exploités par le VAX/VMS.

De plus, le VAX peut fournir un système de multiutilisateur pour des applications larges et compliquées et il peut compiler et exécuter concurremment des énormes programmes.

Le VAX architecture est aussi préparé pour augmenter la performance du programme. Il donne un ensemble d'instructions puissantes et de nombreux types de données. Le résultat c'est que le compilateur va générer les codes compacts et efficaces et les faire rapidement. Du fait, le temps d'exécution d'une appli-

cation sera moins long et plus performant. Le VAX est encore compatible avec le PDP-11.

D'après ce qui précède, on a tiré quelques avantages lorsqu'on implémente le logiciel GRAPHLOG sur le VAX :

- Le 32-bit adressage virtuel permet d'éviter le problème de limite de l'espace d'adresse du programme.
- Le système multiutilisateur facilite l'exécution du logiciel.
- Le temps d'exécution du programme est efficace.
- Enfin car il est compatible au PDP-11 donc on peut implémenter le logiciel aussi sur le PDP-11 qui est disponible à l'Institut d'Informatique de Namur.

I.2. LE VAX11-PASCAL

On a choisi le VAX11-PASCAL pour traduire le logiciel car il permet de décrire la structure de données du graphe sous forme de pointeur. Ce fait nous donne une facilité à introduire les données. De plus la structure de pointeurs permet aussi de créer les listes et la plupart des algorithmes de graphe en ont besoin.

Un autre avantage de Pascal c'est qu'il offre les types SET et ARRAY qui sont indispensables dans les algorithmes de graphe et permet aux utilisateurs de créer ses propres types convenables en faisant la combinaison des types disponibles de Pascal.

Le VAX11-PASCAL permet d'avoir des accès par clé sur le fichier organisé indexé. Ce fait est très intéressant car les autres Pascal ne permettent que l'accès séquentiel ou direct qui oblige les utilisateurs soit à perdre beaucoup de temps (accès séquentiel), soit à devoir savoir la position de l'enregistrement

accédé (accès direct). Avec la permission d'accès par clé, l'utilisateur peut facilement consulter l'enregistrement via sa clé (un nom logique) et le modifier.

Le VAX11-PASCAL donne un autre avantage c'est de compiler séparément les modules et de les linker en un seul module avant de l'exécuter..Il permet aussi les procédures des modules séparés de pouvoir appeler entre eux. Ce fait nous aide à moduler notre logiciel.

Enfin le langage Pascal est un langage bien structuré et le plus proche du pseudo-langage, permettant de traduire aisément les algorithmes de graphe.

Néanmoins, le VAX11-PASCAL a aussi quelques inconvénients:

- Le cardinal du type SET est limité, donc le nombre de noeuds d'un graphe est limité ($\leq$ 255)
- Il ne permet pas de lire correctement des chiffres de l'écran. (On doit créer soi-même les fonctions de lecture des nombres)
- Il est moins rapide que le langage C.

CHAPITRE II

LA CONFIGURATION DU LOGICIEL

GRAPHLOG est un logiciel concernant l'étude de la théorie des graphes. Il demande l'utilisateur d'avoir une bonne notion sur la théorie des graphes. De plus, GRAPHLOG nous donne une facilité à l'examen d'un graphe et nous permet aussi de résoudre certains problèmes tels que le plus court chemin, l'ordonnancement, la recherche d'un chemin eulérien, etc... Mais la représentation figurative du problème doit être réalisée par l'utilisateur c'est à dire après avoir lu l'énoncé du problème, c'est l'utilisateur lui même qui va abstraire cet énoncé en traçant sur une feuille de papier des points pouvant représenter des individus, des localités, des corps chimiques, etc..., reliés entre eux par des lignes ou des flèches symbolisant une certaine relation.

GRAPHLOG a été développé pour travailler uniquement en mode interactif; il existe des commandes qui permettent de changer les états du logiciel. A chaque état, l'utilisateur peut réaliser un certain nombre de traitements sur le graphe.

Ce chapitre est décomposé en deux paragraphes : le premier concerne le schéma des états du logiciel et le deuxième se rapporte aux spécifications d'état.

II.1. LE SCHEMA DES ETATS DU LOGICIEL

La configuration du GRAPHLOG est schématisée à la figure 11. Le superviseur a pour but de contrôler les états du logiciel GRAPHLOG. Le superviseur nous permet de faire les changements d'état adéquats suivant un

scénario déterminé pendant l'exécution choisi par l'utilisateur.

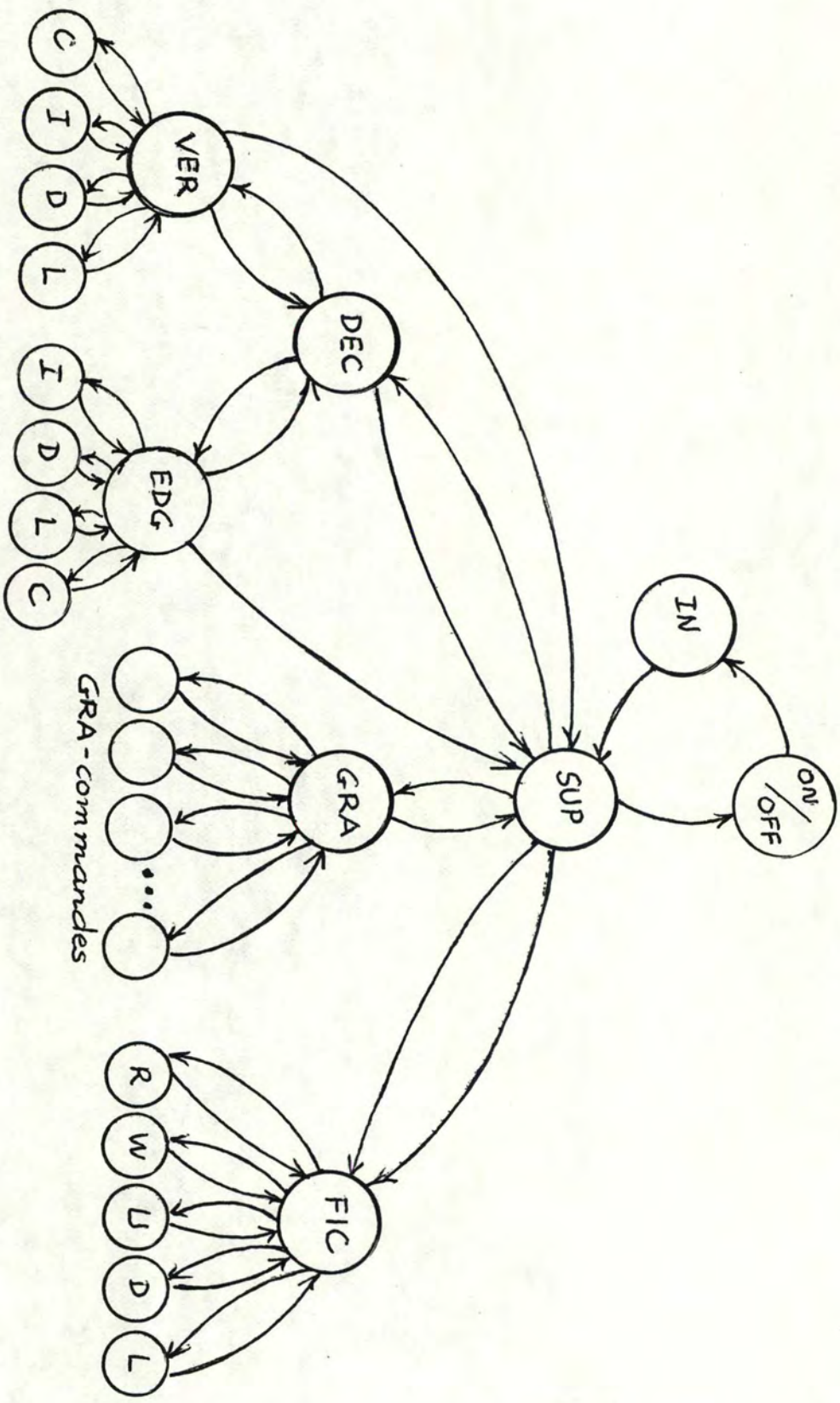


Fig. 11 Schéma général des états dans GRAPHLOG

A tout moment GRAPHLOG est caractérisé soit par un état stable, soit par un état instable.

Il est dans un état instable lorsqu'il est en train d'exécuter une opération quelconque.

Il est dans un état stable, soit à la fin d'une opération, soit dans l'attente d'une requête rentrée par l'utilisateur.

A un état stable quelconque du GRAPHLOG, selon la touche frappée par l'utilisateur, GRAPHLOG peut après avoir réalisé l'opération demandée, soit aller vers un autre état stable, soit retourner à son état initial.

Quand l'utilisateur fait exécuter GRAPHLOG, il peut se trouver dans les états suivants :

1. IN : Initiation
2. SUP : Superviseur
3. DEC : Declaration
4. GRA : Sélection de traitement sur graphe
5. FIC : Fichier
6. VER : Vertex
7. EDG : Edge
8. GRA - Commandes

II.2. LES SPECIFICATIONS DES ETATS DU LOGICIEL

II.2.1. Etat IN

Avant l'exécution du GRAPHLOG, l'utilisateur est dans l'état OFF. Pour passer à l'état IN, il faut faire exécuter GRAPHLOG via VAX/VMS par la commande.

- RUN GRAPHLOG

L'état IN est stable lorsque la fonction initiation de l'écran (INITPROG) est réalisée.

Pour passer de l'état IN à l'état SUP, l'utilisateur doit taper <Return> (fig. 12)

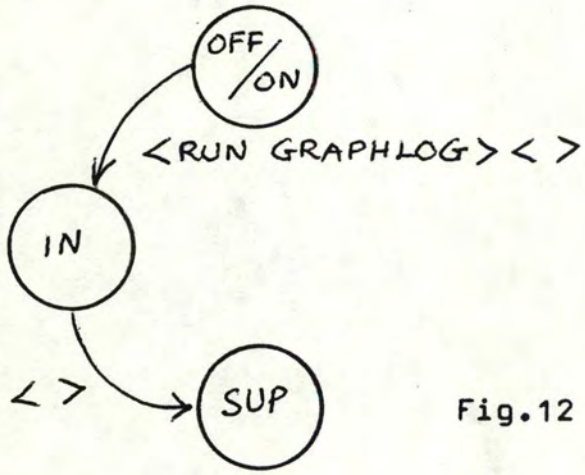


Fig.12

N.B. Les touches à frapper sont mis dans <...>
<>:taper sur la touche RETURN

II.2.2. Etat SUP

Lorsqu'on est dans l'état SUP, l'utilisateur pourrait, soit sortir du logiciel, c'est à dire revenir à l'état OFF, soit sélectionner un état suivant concernant ce qu'il veut. (Fig. 13)

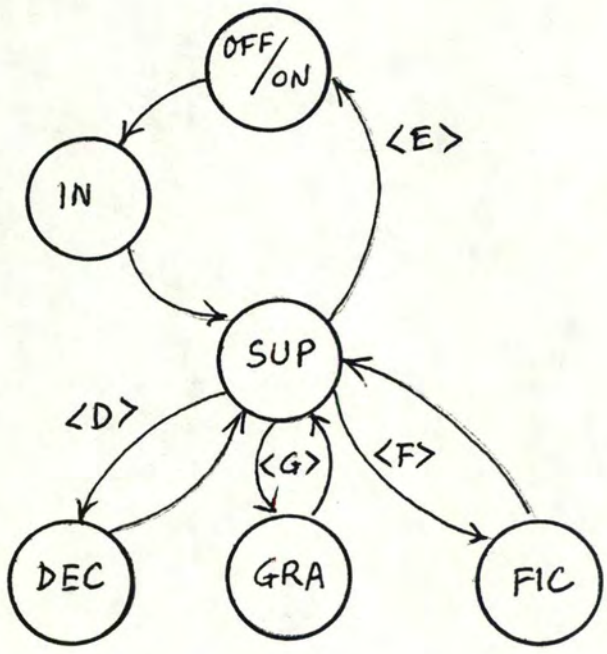


Fig. 13

N.B. On reste toujours à l'état SUP lorsqu'on frappe sur <X> tel que $X \notin \{D, G, F, E\}$

II.2.3. Etat DEC

Dans l'état DEC, on ne peut que soit quitter l'état pour revenir à l'état SUP, soit entrer un parmi les deux états suivants: l'état VER ou (exclusif) l'état EDG. (fig. 14)

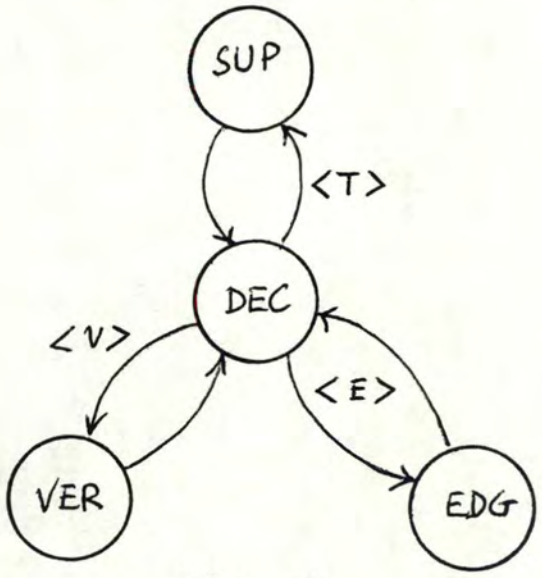


Fig. 14

La fonction de DEC est de permettre à l'utilisateur de déclarer un graphe (inexistant) ou de modifier un graphe (existant).

II.2.4. Etat VER

Dans cet état, on peut réaliser un certain nombre d'opérations sur les noeuds. Les opérations sont réalisées en mode interactif et peuvent être réexécutées autant de fois que l'utilisateur le désire en utilisant la même méthode. La sortie de cet état consiste à revenir à l'état DEC ou à rentrer directement à l'état SUP. (fig.15)

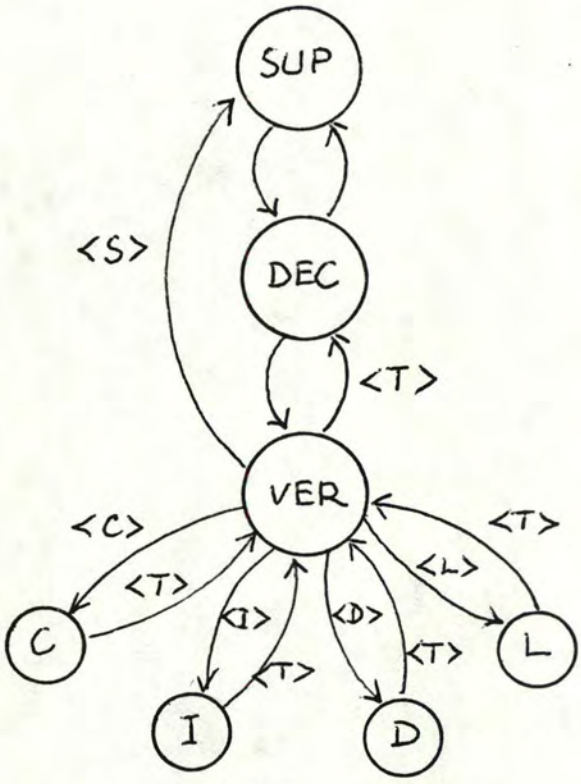


Fig.15

Les opérations possibles sont :

C : Création d'un nouveau graphe

I : Insérer un noeud à un graphe

D : Supprimer un noeud d'un graphe

L : Lister tous les noeuds du graphe

L'Etat VER fait l'objet d'un module particulier : cfr. Module du traitement de noeud du graphe.

II.2.5. Etat EDG

Cet état ressemble à l'état VER, mais les opérations qui le concernent, sont faites sur les arcs (ou les arêtes) au lieu d'être faites sur les noeuds. Une autre différence est que dans cet état, il n'y a pas d'opération de création parce qu'il est impossible de créer un graphe avec les arcs sans avoir les noeuds, mais par contre, c'est acceptable pour les noeuds. (fig.16)

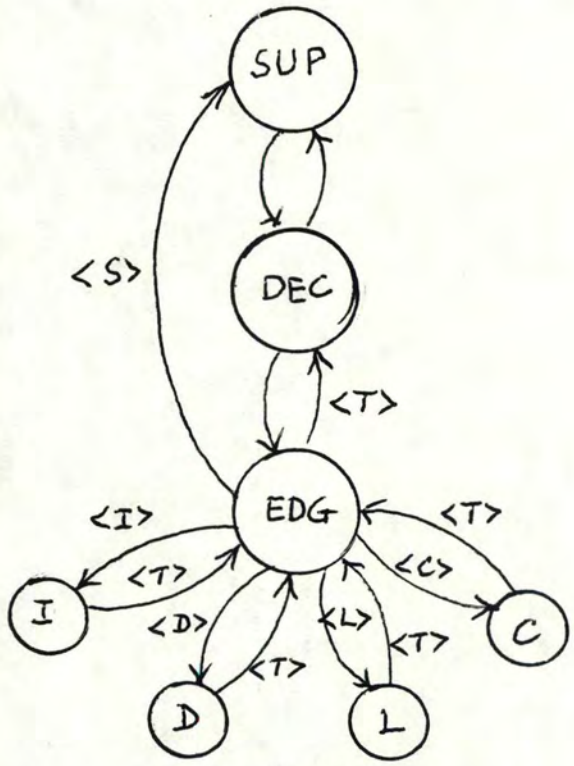


Fig.16

Les opérations possibles sont :

< I > : Insérer un arc à un graphe
avec son poids s'il y en a

< D > : Supprimer un arc d'un graphe

< C > : Changer la valeur du poids d'un arc donné

< L > : Lister tous les arcs du graphe
avec leur poids.

Etat EDG fait l'objet d'un module particulier : cfr. module du traitement d'arc du graphe du chapitre IV

II.2.6. Etat FIC

Lorsqu'on est dans cet état, on peut demander l'exécution d'un certain nombre d'opérations sur les fichiers. Selon la décision de l'utilisateur, soit on reste dans cet état pour réaliser une telle opération, soit on en sort pour revenir à l'état SUP. (Fig.17)

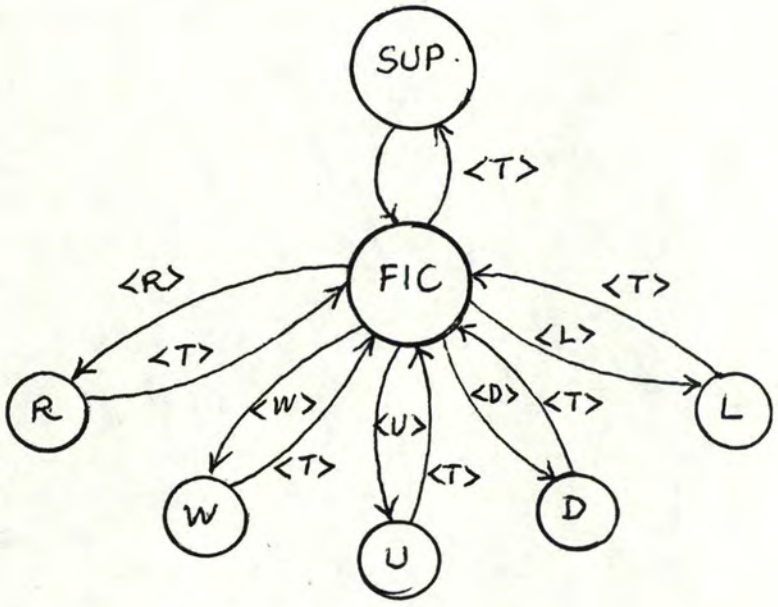


Fig.17

Les opérations possibles sont :

< R > : Lire un graphe archivé dans un fichier

< W > : Ecrire un graphe sur un fichier

< U > : Modifier un graphe du fichier

< D > : Supprimer un graphe du fichier

< L > : Lister tous les noms du graphe dans le fichier.

Etat FIC fait l'objet d'un module particulier. cfr. Module d'archivage sur fichier dans le chapitre IV.

II.2.7. Etat GRAP

L'état GRAP est un état le plus fréquent que l'on reste dedans. Quand on est dans cet état, à l'aide d'un menu de choix, on peut étudier facilement la théorie de graphe via les GRA-commandes. Une GRA-commande se ressemble à une conversation entre l'utilisateur et l'ordinateur, et GRAP a donc pour rôle d'un interprète des requêtes de l'utilisateur.

Suivant la touche frappée, l'une des actions suivantes est faite :

- Exécution de la requête demandée et retour à l'état GRAP.

- Sortir l'état GRAP et retourner à l'état SUP. (Fig.18)

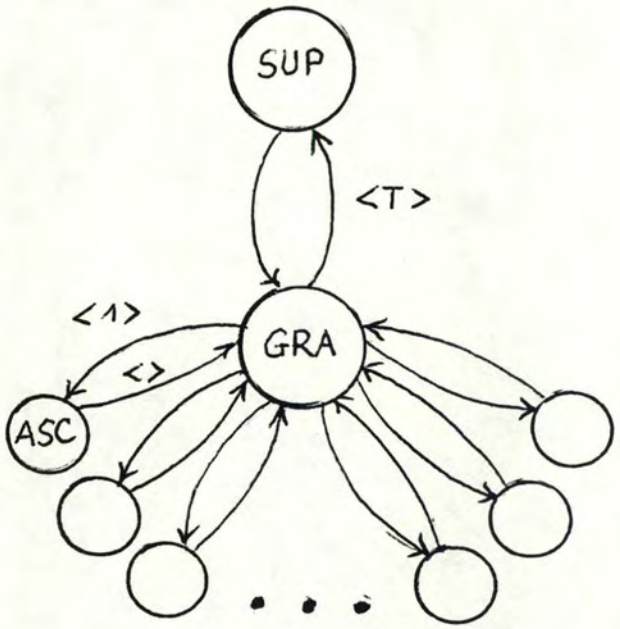


Fig.18

Les opérations possibles sont :

1-ANTISYMETRIC	12-EXIST-CIRCUIT	23-STRONG-COMPONENTS
2-ASCENDANT	13-EXIST-LOOP	24-SHORTEST-PATH
3-CHAIN-EULERIAN	14-EXIST-PATH	25-SUCCESSORS
4-COMPLETE	15-EXIST-TREE	26-SYMETRIC
5-CONNECTED	16-ISOLED-POINTS	27-TRAVEL-COMMERCIAL
6-COMPONENTS	17-NONDIRECT-FORM	28-TERMINATE
7-COMPLEMENT	18-ORDONNANCEMENT	
8-DEGREE	19-ORDRE	
9-DEGREE-IN	20-PREDECESSOR	
10-DEGREE-OUT	21-POSTIER-CHINOIS	
11-DESCENDANT	22-STRONGLY-CONNECTED	

CHAPITRE III

LA STRUCTURE DE DONNEES

Nous avons vu au chapitre II de la première partie qu'un graphe peut-être représenté soit par une matrice d'adjacence, soit par une matrice d'incidence. Ici, nous décidons de représenter un graphe à partir de sa matrice d'adjacence, et pour faciliter cette représentation, chaque noeud du graphe peut-être masqué par un nombre entier.

Dans le premier paragraphe, nous allons discuter sur la représentation du graphe sur l'ordinateur et le deuxième donnera les concepts des structures intermédiaires tels que la liste des noeuds, les vecteurs d'arcs, etc... Enfin, pour terminer ce chapitre, on vous propose une structure de données du graphe archivé sur un fichier.

III.1. LA REPRESENTATION DU GRAPHE SUR L'ORDINATEUR

Nous avons vu dans I.2.4. de la première partie, une matrice d'adjacence peut-être représentée sous forme de deux vecteurs (ou trois vecteurs dans le cas où l'on s'intéresse à étudier un problème concernant le poids d'arcs (ou d'arêtes). Ici, on ne peut pas représenter un graphe sous forme de trois vecteurs car il y a quelques inconvénients sur les opérations de déclaration d'un graphe.

En effet, le premier inconvénient est que le langage PASCAL ne permet que de déclarer un vecteur avec une longueur fixe , donc on doit prévoir une taille maximum convenable. Ce fait nous pose un problème de

gaspillage de place de mémoire. Comme on sait qu'à tout moment du logiciel, le graphe doit être existant, c'est à dire les trois vecteurs sont toujours dans la mémoire et si le nombre de sommets et le nombre d'arcs sont trop petitspar rapport à la taille maximum du vecteur, on a donc gaspillé de place de mémoire.

Le second inconvénient, c'est avec cette représentation, on a des difficultés dans les traitements de mise à jour d'un graphe telles que l'insertion d'un noeud (ou d'un arc) et la suppression d'un noeud (ou d'un arc).

A cause des inconvénients sus-mentionnés et avec la permission du langage PASCAL, on propose une autre représentation de la matrice d'adjacence du graphe : c'est la notion de pointeur. D'abord, avec cette notion on ne perd pas de place de mémoire, car elle permet d'allouer dynamiquement les places de mémoire, c'est à dire le nombre de places réservées est justement égal au nombre de sommets et d'arcs. Cela nous fait éviter le gaspillage de place de mémoire. D'ailleurs, les opérations de l'insertion et de la suppression peuvent être réalisées facilement via cette représentation.

La figure 19 nous donne une vue globale de la structure de donnée d'un graphe dans la mémoire. A partir de la cellule HEAD, on peut consulter les informations sur graphe via les pointeurs. Le langage PASCAL permet de décrire cette structure comme suit :

```
Type setvertex  = ^ pointerver
     pointerver = record
                    current: setvertex
                    nomint : integer
                    nomext : message
```

```
link     : setvertex
colink0 : matadj
end ;
```

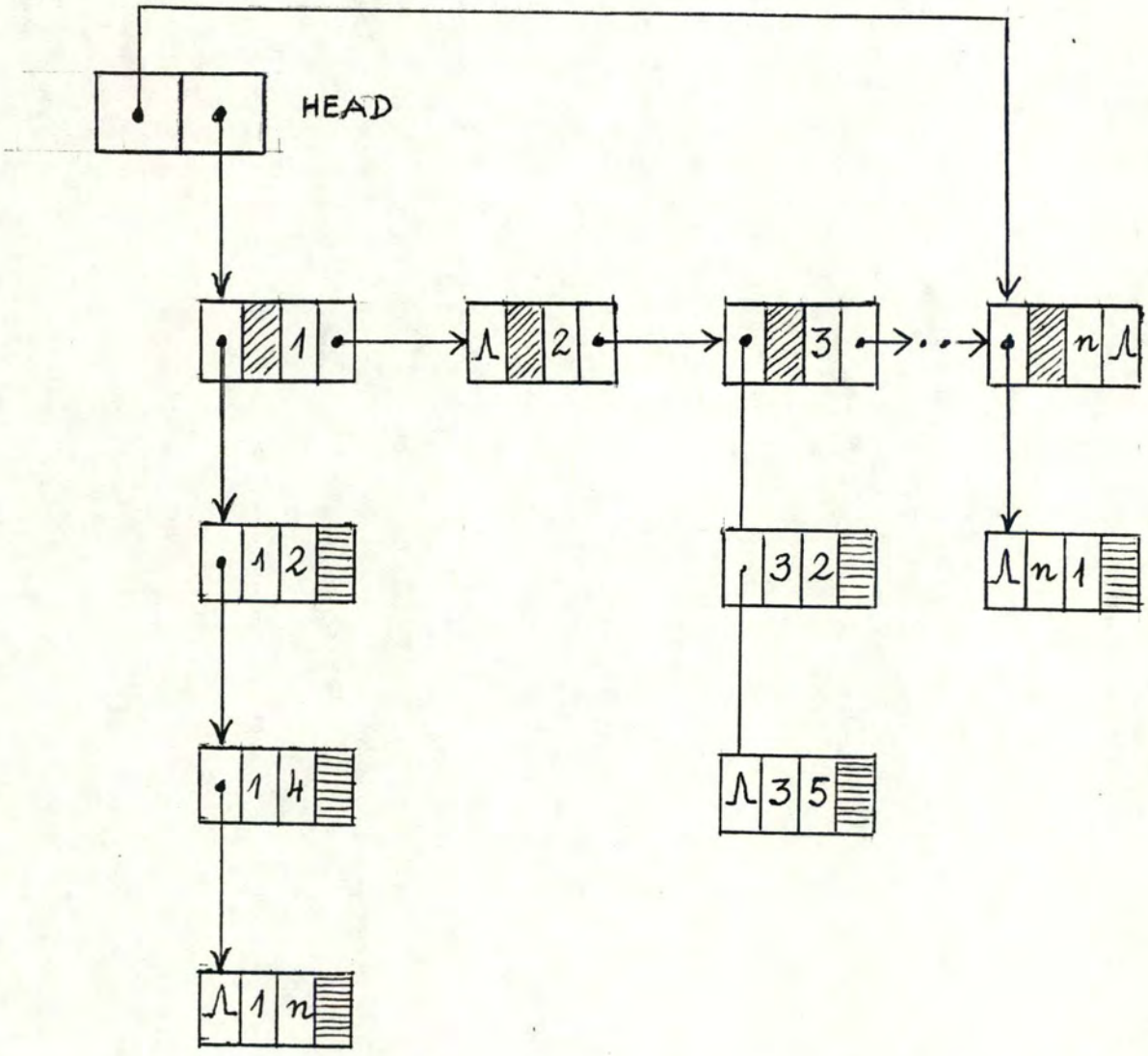


Fig. 19 La représentation de la matrice d'adjacence

: nom externe du noeud

: le poids de l'arc

```
Type   matadj     =  ^. pointeredg
       pointeredg =  record
                     initial : integer
                     final   : integer
                     poids   : real
                     colink  : matadj
                     end,
```

Dans la cellule HEAD, il y a deux pointeurs :

-le pointeur LINK donne l'adresse de la première cellule horizontale, correspondant à un noeud.

-le pointeur CURRENT donne l'adresse de la dernière cellule horizontale.

Avec ces deux pointeurs, on peut savoir tout de suite si un graphe a seulement un noeud, c'est à dire HEAD ^.LINK = HEAD ^.CURRENT, au lieu de consulter la première cellule via HEAD ^. LINK et voir si le link de cette cellule est-elle NIL. De plus, ce fait permet de traiter directement quelques opérations sur la dernière cellule sans passer les précédentes à l'aide des pointeurs.

Chaque cellule horizontale de la structure représente un noeud du graphe. Elle contient quatre items suivants :

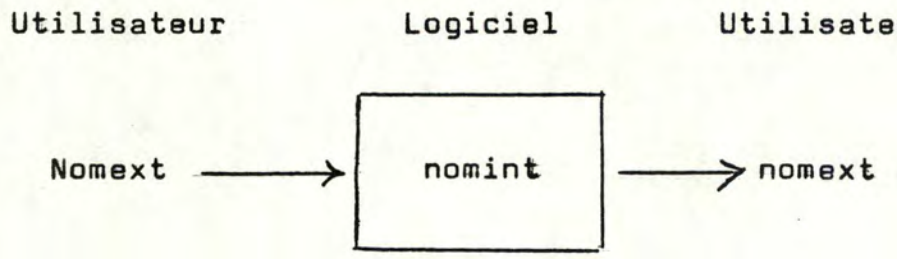
- nomint : c'est le nom interne de ce noeud qui est représenté par un nombre entier donné suivant l'ordre d'entrée de ce noeud.

- nomext : c'est le nom externe de ce noeud qui est donné par l'utilisateur.

- le pointeur LINK donne l'adresse de la suivante.

- le pointeur COLINKO donne l'adresse de la première cellule verticale, correspondant à un arc d'origine de ce noeud.

Le rôle des deux items nomint et nomext peut figurer comme suit :



Chaque cellule verticale de la structure représente un arc d'origine du noeud correspondant à la cellule horizontale de même colonne. Elle contient aussi quatre items :

- initial : c'est le nom interne d'extrémité initial de l'arc.
- final : c'est le nom interne d'extrémité final de l'arc.
- poids : c'est la valeur du poids de l'arc (dans le cas du graphe pondéré)
- le pointeur COLINK donne l'adresse de la suivante.

Reamarquons que si la valeur du pointeur est égal à NIL, cela signifie que c'est la dernière cellule.

Cette structure nous donne une facilité de réaliser les opérations de mise à jour d'un graphe. Les opérations sont les suivantes :

- Ajouter ou supprimer un noeud n'importe où. [cfr. le module de traitement de noeud du graphe]
- Ajouter ou supprimer un arc n'importe où. [cfr. le module de traitement d'arc du graphe]

III.2. LES STRUCTURES DE DONNEES INTERMEDIAIRES ADAPTIVES AUX ALGORITHMES DES GRAPHES

Les structures de donnée intermédiaires que l'on aperçoit souvent sont : pile, ensemble et vecteur. Ici, on ne parle pas des structures d'ensemble et vecteur car le langage PASCAL de VAX11/VMS donne toutes ces structures [ ] . Donc il reste seulement la structure de pile à étudier.

Comme nous savons en mathématique que dans certaines applications on a besoin d'une liste, étant une séquence finie, d'éléments d'un ensemble donné. La plus simple implémentation d'une liste est illustrée dans la figure 20.

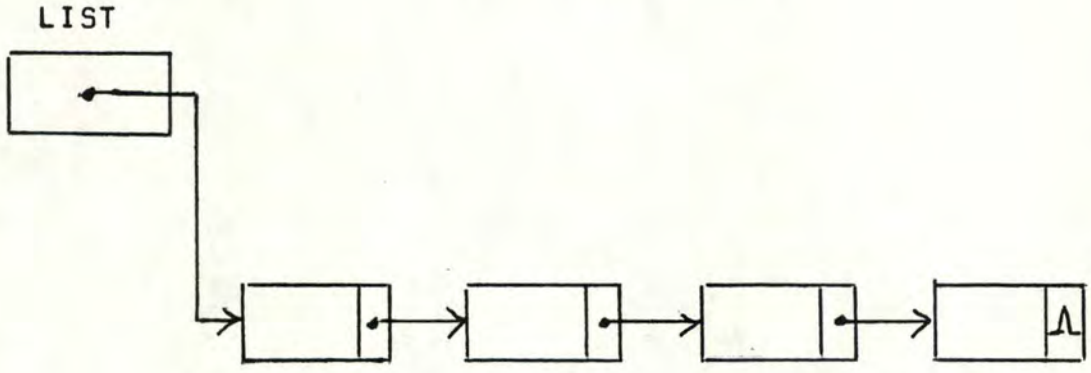


Fig. 20

Chaque élément de la liste consiste en deux allocations de mémoire. La première contient les informations nécessaires et la seconde contient un pointeur à l'élément suivant. Le programme peut donc accéder au dernier élément au moyen de passer séquentiellement tous les éléments précédents à l'aide des pointeurs. Par conséquent, la structure d'une liste est entièrement une structure linéaire.

Une pile est une liste du type LIFO (last-in,

La première opération que l'on fait sur une liste c'est de la créer. Dans notre logiciel, la procédure CREATSTA s'occupe de faire ça. Sa fonction est de demander une allocation pour l'élément STACK et de faire l'initialisation suivante :

STACK^. node ← 0

STACK^. nextnode ← nil

Les opérations d'addition et de suppression ne peuvent être exécutées que sur [•] de la figure 21 . Les procédures qui réalisent ces opérations sont :

La procédure PUSHDOWN permet d'ajouter un élément au bout de la pile. Son principe est comme suit :

- Demander une nouvelle allocation pour le nouveau élément.
- ELEMENT ^. nextnode ← STACK ^. nextnode
- ELEMENT ^. node ← nom du nouveau élément
- STACK^. nextnode ← l'adresse de la nouvelle allocation

La fonction POPTOP permet de supprimer un élément de la pile (bien sûr c'est au bout) et le mettre dans la variable POPTOP. Son principe est comme suit :

- Mettre ELEMENT à supprimer dans POPTOP
- STACK ^. nextnode ← ELEMENT ^. nextnode
- Supprimer ELEMENT.

Une fonction supplémentaire que nous proposons pour tester si une pile est vide ou non, c'est la fonction EMPTYSTA. Selon sa valeur est TRUE ou FALSE, on peut savoir la situation de la pile. A l'aide de cette fonction, on peut éviter de faire une POPTOP sur une pile vide, ce qui concerne une erreur impardonnable.

first-out) c'est à dire toute opération d'addition ou de suppression est faite d'un bout de la liste.

Dans notre logiciel, on a utilisé une pile de noeuds. Le langage PASCAL donne la description de cette pile comme suit :

```
Type  Stackver    = ^pointerlist
      Pointerlist = record
                      node     = interger
                      nextnode = stackver
                      end,
```

Maintenant nous allons considérer les opérations sur la pile (figure 21)

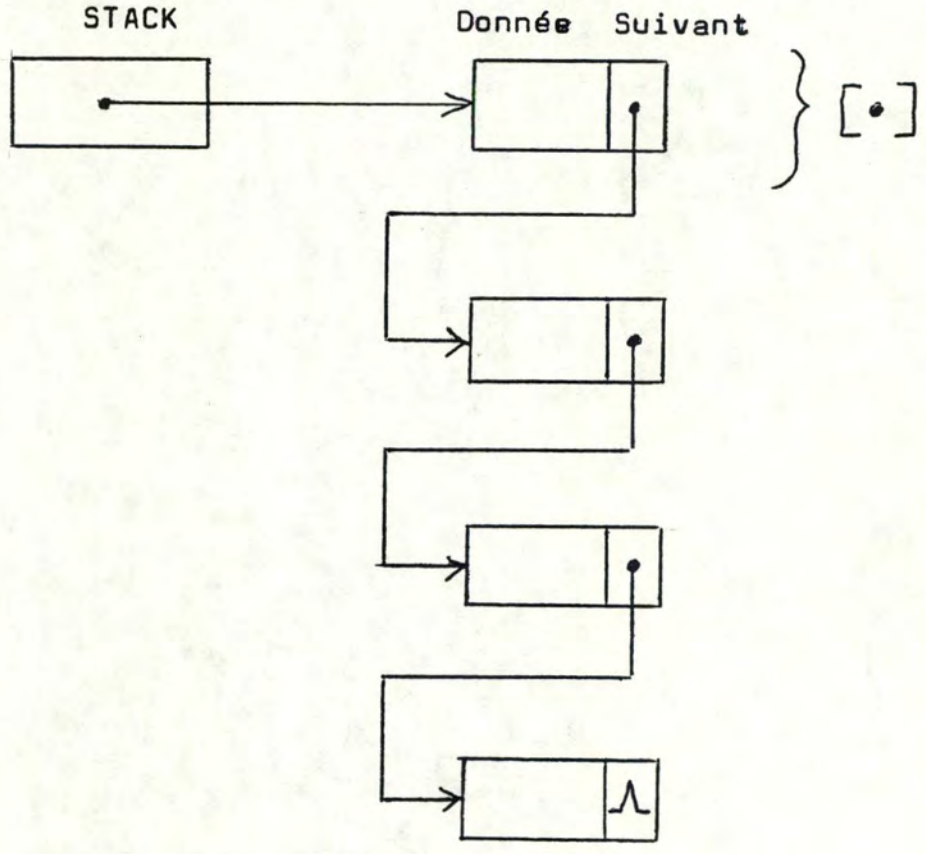


Fig. 21. La configuration d'une pile de quatre éléments.

III.3. LA STRUCTURE DE DONNEES DU GRAPHE SUR LES FICHIERS

Le logiciel GRAPHLOG permet d'archiver un graphe sur un fichier. Comme nous savons que c'est impossible d'avoir une structure du type de pointeur sur les fichiers, on doit alors accepter la représentation d'un graphe par les vecteurs. Cela donnera une gaspillage inévitable de places sur disque en devant déclarer les vecteurs en taille maximale.

Ici, on a utilisé deux fichiers : FILEVER.DAT et FILEEDG.DAT, dont leur structure est montrée dans la figure 22, à cause de la structure de donnée d'un noeud est différente de la structure de donnée d'un arc.

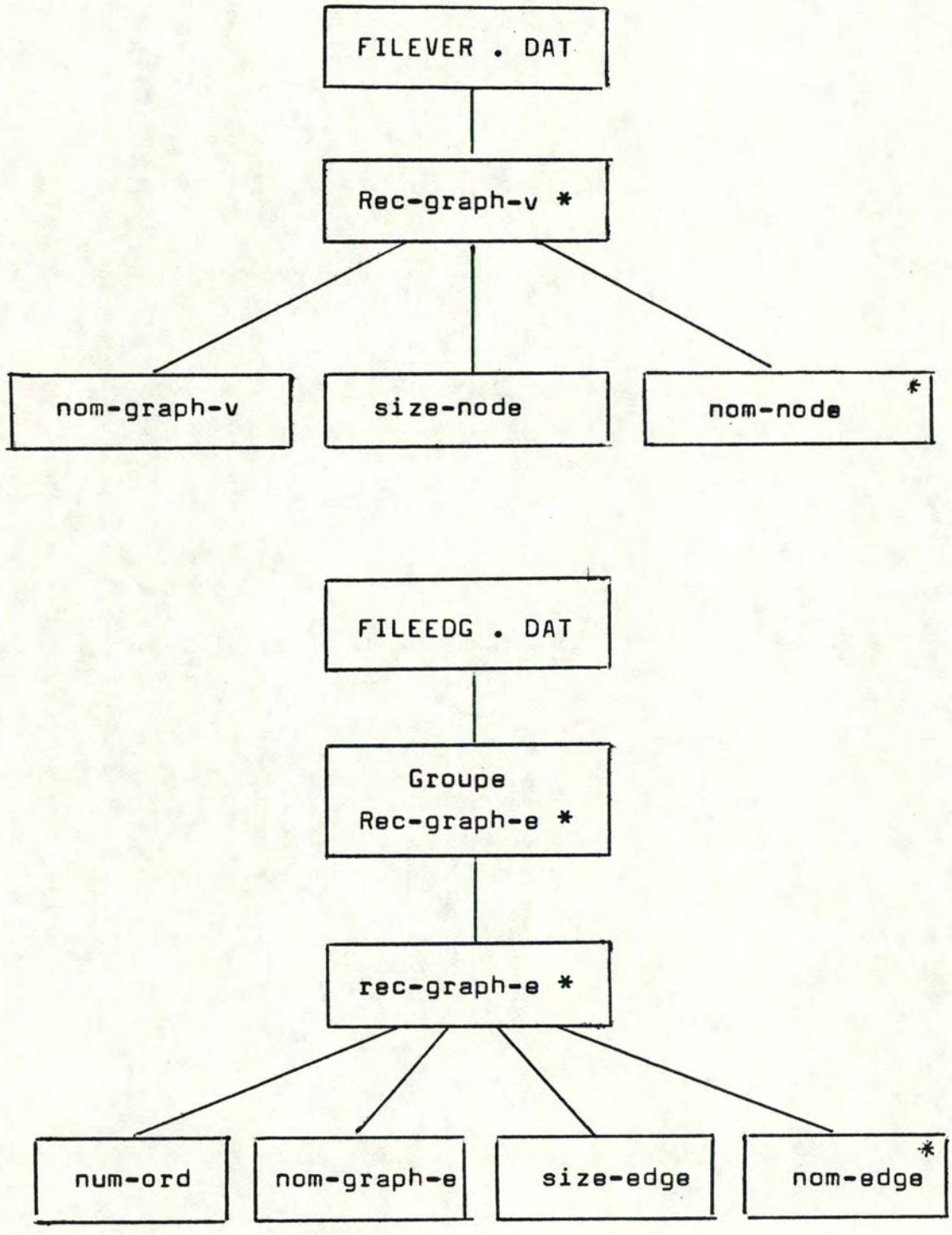


Fig.22 La représentation des deux fichiers :
FILEVER.DAT et FILEEDG.DAT

NB. L'astérisque dans les cadres indique une répétition de l'élément dans le cadre pendant une certaine fois.

Dans le fichier FILEVER, le nom node est un vecteur de taille maximale (255 noeuds) dont chaque élément est le nom externe du noeud, et l'index du vecteur représente le nom interne du noeud. Ce fichier est un fichier indexé permettant d'accéder par clé, dont la clé est le nom du graphe : nom-graph-v.

Dans le fichier FILEEDGE; au lieu de représenter la matrice d'adjacence d'un graphe comme un enregistrement du fichier, on a proposé que chaque enregistrement du fichier est une séquence de 255 arcs. Cela donne un avantage c'est qu'on ne gaspille pas de places du disque. Si le nom-edge est un vecteur de taille maximale (255 arcs) dont chaque élément a une structure suivante :

- nom d'extrémité initiale (nom interne)
- nom d'extrémité finale (nom interne)
- le poids de l'arc.

Alors le nombre de places gaspillées est calculé comme suit :

$$N_G = 255 - [\text{size-edge} \ (\ \text{mod}\ 255\ )]$$

où size-edge est le nombre d'arcs du graphe.

Avec cette structure, on a le problème d'accès. Comme nous savons que le fichier FILEEDGE est un fichier indexé permettant d'accéder par clé et que chaque enregistrement est une séquence de 255 arcs. Par conséquent, on ne peut pas prendre le nom du graphe comme clé. Pour résoudre ce pooblème, on ajoute en plus une autre clé: num-ord, étant l'ordre de l'enregistrement appartenant à un graphe, et on va accéder à un graphe via deux clés :

- nom du graphe
- et num-ord

Cette structure donnera les facilités à réaliser une certaine opération sur le fichier, telle que :

- lecture d'un graphe archivé
- écriture d'un graphe à archiver
- mise à jour d'un graphe archivé
- supprimer un graphe archivé

[cfr. Module du traitement de fichier]

CHAPITRE IV

LES MODULES DU LOGICIEL

La structure du logiciel est modulaire, ce qui permet aux certains modules d'être utilisés par plusieurs autres modules. Chaque module du logiciel a une fonction bien déterminée et a fortement caché l'information. Cela donnera un intérêt à l'utilisateur à maintenir et modifier le logiciel.

On a basé sur l'analyse d'un graphe pour décomposer le logiciel :

- Les modules concernant la représentation d'un graphe.
- Les modules concernant l'étude de la théorie de graphe.

De plus, à cause de l'utilisation de notre logiciel est en mode interactif, on doit alors ajouter certains modules tels que :

- Module de la gestion d'écran
- Module de la gestion de fichier

Avant de décrire en détails les modules, on propose un schéma de la décomposition du logiciel en module, dans la figure 23, permettant aux lecteurs d'avoir une idée générale sur la découpe du logiciel.

Remarquons que les procédures d'un module sont exposées brièvement dans ce chapitre. Par conséquent, pour avoir les détails d'une procédure, l'utilisateur peut la consulter dans l'annexe.

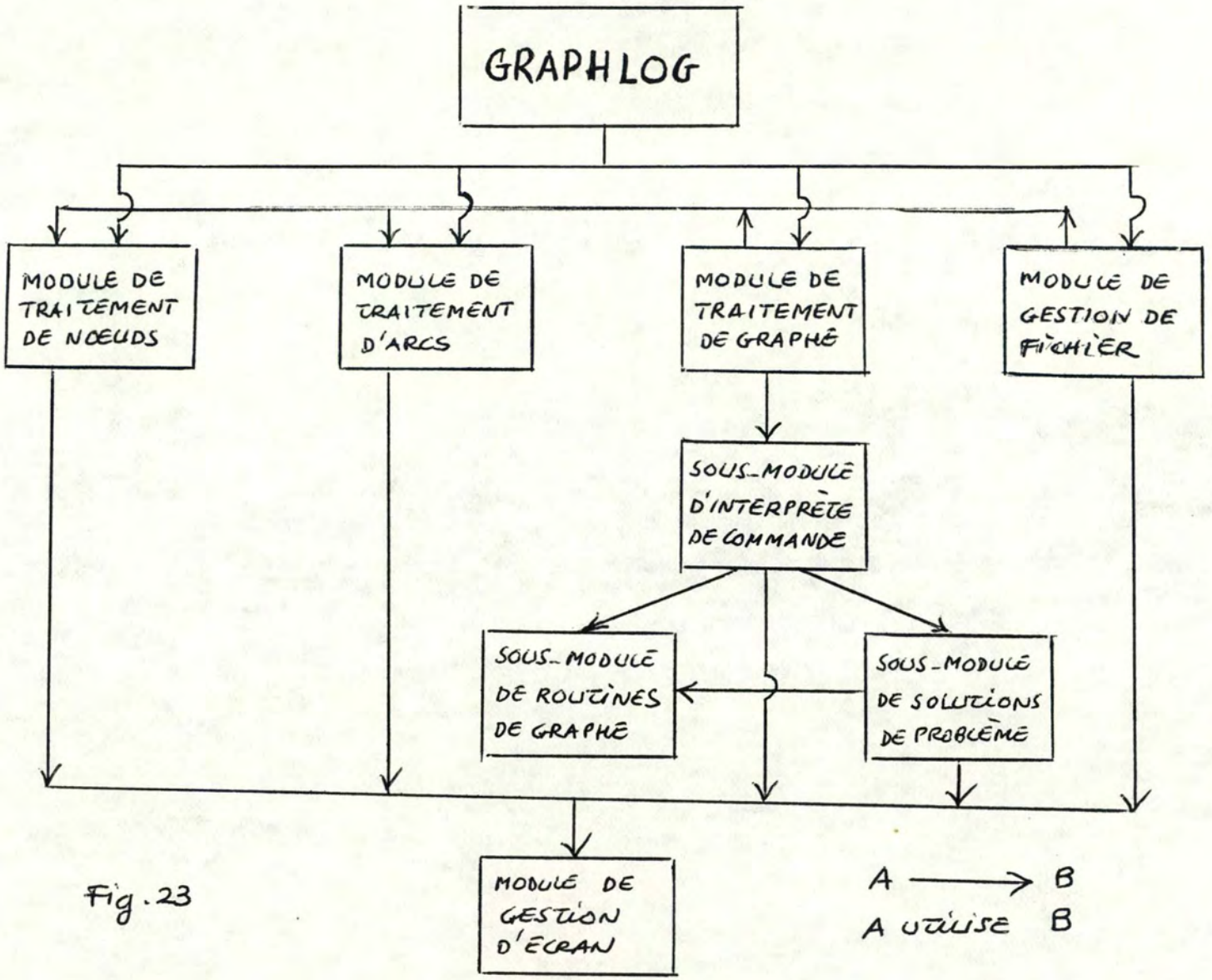

GRAPHLOG
MODULE DE TRAITEMENT DE NŒUDS
MODULE DE TRAITEMENT D'ARCS
MODULE DE TRAITEMENT DE GRAPHE
MODULE DE GESTION DE FICHIER
SOUS-MODULE D'INTERPRÈTE DE COMMANDE
SOUS-MODULE DE ROUTINES DE GRAPHE
SOUS-MODULE DE SOLUTIONS DE PROBLÈME
MODULE DE GESTION D'ÉCRAN
A → B
A UTILISE B

Fig. 23

IV.1. MODULE DE GESTION D'ECRAN

IV.1.1. Objectif

Rappelons que notre logiciel est exécuté en mode interactif, donc la formalité de l'écran joue un rôle assez important. Le but de ce module est d'aider les utilisateurs de former facilement l'écran à volonté et d'avoir aisément la réception de donnée. Pour parvenir à ce but, le module produit alors certaines primitives en basant sur les procédures disponibles du système VAX/VMS sous-mentionnées :

- function SCR$ERASE-PAGE (% immed ligne, column)
- procedure LIB$ERASE-PAGE
- procedure LIB$ERASE-LINE
- function SCR$SET-CURSOR (% immed ligne, column)
- procedure LIB$STOP (% immed i)

IV.1.2. Les procédures du module

Selon son but, on peut diviser les procédures en deux groupes suivants :

- Groupe de gestion d'écran
- Groupe de réception de donnée de l'écran.

i/ Groupe de gestion d'écran

Nom de procédure	Description
- HOME	: permet de positionner le curseur au bout en haut à gauche de l'écran.
- BLANK	: permet d'effacer tout l'écran sans faire attention à la position du curseur.
- CLREOS	: permet d'effacer tout l'écran en bas à partir de la position du curseur à ce moment là.
- GOTOXY	: permet de mettre le curseur à la position qu'on veut.

- SAUTPAGE : c'est une procédure spéciale coneernant l'affichage des listes sur l'écran. Elle permet de tourner la page c'est à dire d'effacer l'écran et de positionner le curseur à une position prédéterminée, après avoir affiché sur l'écran un nombre désirable de lignes.
- WAIT : permet de mettre en repos le logiciel dans un moment déterminé.
- CLREOL : permet d'effacer tout le reste d'une ligne à partir de la position du curseur à ce moment là.

ii/ Groupe de réception de donnée de l'écran

Comme nous savons que le logiciel GRAPHLOG est écrit en PASCAL et certainement le PASCAL nous donne l'inconvénient dans la lecture des nombres de l'écran. Donc pour résoudre ce problème, on a proposé deux procédures suivantes :

- LIRENUM : permet de lire un nombre entier correct de l'écran, c'est à dire elle peut détecter des erreurs syntaxiques et de traduire un string de chiffres en nombre entier.
- LIRENUMR : permet de lire un nombre réel correct de l'écran.

IV.2. MODULE DE TRAITEMENT DE NOEUD DU GRAPHE

IV.2.1. Objectif

L'ensemble des noeuds est l'un parmi les deux composants immanquables du graphe. En basant sur la structure de donnée du graphe [cfr. III.1 de la deuxième partie] , ce module permet de créer l'ensemble des

noeuds, d'ajouter ou de supprimer un élément de l'ensemble. De plus, pour offrir un avantage aux utilisateurs dans la vérification de l'ensemble des noeuds du graphe, ce module est capable de lister tous les éléments de l'ensemble sous forme nom externe et nom interne de l'élément.

IV.2.2. Les procédures du module

Pour pouvoir cacher l'information au plus possible, on a décidé de découper le module en deux parties :

- Les opérations primitives telles que la création, l'insertion, la suppression et le parcours de la liste.
- L'interface entre l'utilisateur et le module.

i/ Les opérations primitives

Nom de procédure	Description
-NEWSET	: permet de créer un initial ensemble vide, consistant seulement la cellule HEAD de l'ensemble, [cfr. III.1 de la 2ème partie]
- EMPTY	: permet de savoir si l'ensemble des noeuds est vide ou non.
- SEARCH	: permet de chercher l'adresse d'un élément de l'ensemble en donnant simplement son nom interne. [cfr. III.1. de la 2ème partie]
- INSERT	: permet d'insérer un nouveau élément dans l'ensemble.
- DELETEV	: permet de supprimer un élément de l'ensemble.

- SIZE : permet de savoir le nombre d'éléments de l'ensemble.
- MEMBER : permet de vérifier si un élément, en donnant son nom externe, appartient à un ensemble.
- LIST : permet d'afficher sur l'écran la liste d'éléments sous la forme suivante :
  nom externe, nom interne
- NAMEXT : permet de donner le nom externe d'un élément lorsqu'on a son nom interne.

Remarquons que les procédures INSERT et DELETEV sont capables de détecter les erreurs sémantiques telles que :

- Insérer un élément après un élément inexistant
- Supprimer un élément inexistant.

ii/ L'interface entre l'utilisateur et le module

Ce sont les procédures concernant les opérations de l'état VER [cfr. II.2.4. de la 2ème partie] qui jouent le rôle :

- de former l'écran convenable à l'opération
- de faire la conversation avec les utilisateurs
- et de détecter l'erreur syntaxique.

Nom de procédure	Description
- CREATNODE	: c'est l'interface de la création d'un ensemble de noeuds en faisant un nombre de fois d'insertion d'un élément.

- INSERNODE : c'est l'interface de l'insertion d'un élément.
- DELETNODE : c'est l'interface de suppression d'un élément.
- IENTERNODE : c'est l'interface de l'état VER du logiciel GRAPHLOG [cfr. II.2.4 de la 2ème partie]
- INITDECL : c'est l'interface de l'état DEC du logiciel GRAPHLOG [cfr. II.2.3 de la 2ème partie]

IV.3. MODULE DE TRAITEMENT D'ARC DU GRAPHE

IV.3.1. Objectif

Comme nous savons que l'ensemble des arcs (ou d'arêtes) peut se représenter sous forme d'une matrice d'adjacence et que dans ce logiciel, une matrice est figurée via la structure de pointeur. Ce module permet aux utilisateurs de réaliser certaines opérations courantes sur les arcs du graphe telles que :

- lister les arcs d'un graphe
- ajouter un arc à un graphe
- supprimer un arc existant d'un graphe
- changer le poids d'un arc.

De plus, ce module fournit aussi quelques primitives qui seront utilisées au module de traitement de graphe, telles que :

- le nombre d'arcs d'un graphe
- l'exisrence d'un arc d'un graphe
- la valeur du poids d'un arc

IV.3.2. Les procédures du module

Comme dans le module du traitement de noeud, ce module se décompose aussi en deux parties :

- Les opérations primitives
- L'interface entre l'utilisateur et le module.

i/ Les opérations primitives

Nom de procédure	Description
- LISTEDGE	: permet d'afficher sur l'écran la liste des arcs sous la forme suivante : nom externe de l'extrémité initiale ⇒ nom interne de l'extrémité finale : poids
- EDGEXIST	: permet de savoir si un arc, en donnant les deux noms internes à ses deux extrémités, appartient à un graphe.
-SIZEEDGE	: donne le nombre d'arcs du graphe
- INSEREDGE	: permet d'ajouter un nouveau arc au graphe
- DELETEDGE	: permet de supprimer un arc du graphe
- VALUE	: donne la valeur du poids d'un arc donné
- CHANGEVALUE	: permet de changer la valeur du poids d'un arc donné

Remarquons que les procédures INSEREDGE et

DELETEDGE sont capables de détecter les erreurs sémantiques telles que :

- Insérer un arc ayant les extrémités inexistantes
- Supprimer un arc inexistant

ii/ L'interface entre l'utilisateur et le module

Ce sont les procédures concernant les opérations de l'état EDG [cfr. II.2.5. de la 2ème partie] ayant le rôle identique à celui du module précédent.

Nom de procédure	Description
- INSERARCS	: L'interface de l'insertion d'un arc
- DELETARCS	: L'interface de la suppression d'un arc
- CHANGEV	: L'interface de changer la valeur du poids d'un arc donné.
- IENTEREDGE	: L'interface de l'état EDG du logiciel GRAPHLOG qui permet de réaliser l'opération choi de cet état telle que : l'insertion, la suppression, le changement de valeur, ...

Toutes les procédures du premier groupe du module de traitement de noeuds et du module de traitement d'arcs pourront être utilisées par le module du traitement de graphe fig.23 , mais les procédures du deuxième groupe ne s'emploient que dans leur propre partie.

IV.4. MODULE DE GESTION DE FICHIER

IV.4.1. Objectif

Pour éviter aux utilisateurs de déclarer de nouveau le même graphe chaque fois lorsqu'ils en ont besoin, ce module permet de sauver le graphe après avoir fini de l'analyser et de rappeler aisément. De plus , à l'aide de ce module, on peut faire une mise à jour des informations du graphe et supprimer un graphe sur le fichier lorsqu'on ne l'intéresse plus. Ce module donne aussi la liste de graphe du fichier pour faire rappeler le nom du graphe voulu aux utilisateurs. C'est possible de changer simplement le nom du graphe en faisant une combinaison d'opérations suivantes :

- lire le graphe via son nom
- écrire ce graphe sous son nouveau nom
- supprimer l'ancien graphe via son ancien nom.

IV.4.2. Les procédures du module

Les procédures de ce module se scindent aussi en deux groupes: le premier groupe est l'ensemble des opérations primitives et le deuxième l'ensemble des interfaces concernant ces opérations.

i/ Les opérations primitives

Nom de procédure	Description
- CREATFILE	: permet de créer les fichiers du graphe au premier moment ou de faire l'ouverture des fichiers lorsqu'ils existent

- READFILEV : permet de lire un record du fichier FILEVER.DAT [cfr. III.3. de la 2ème partie]
- READFILEE : permet de lire un record du fichier FILEEDG.DAT [cfr. III.3. de la 2ème partie]
- WRITEFILEV : permet d'écrire un record au fichier FILEVER.DAT
- WRITEFILED : permet d'écrire un record au fichier FILEEDG.DAT
- UPDATEFILE : permet de faire la mise à jour de quelques informations sur un graphe, c'est à dire après avoir lu le graphe du fichier , faire certaines opérations du module de traitement de noeuds ou du module de traitement d'arcs, on peut écrire de nouveau ce graphe étant modifié sous le même nom.
- DELRECFILE : permet de supprimer un graphe de fichiers
- CLOSEFILE : la fermeture des fichiers

Les procédures UPDATEFILE et DELRECFILE ont la possibilité de détecter les erreurs sémantiques telles que :

- mise à jour un graphe inexistant (et pour le corriger, on peut faire une écriture au lieu d'une mise à jour)
- supprimer un graphe inexistant

ii/ Les interfaces

ii/ Les interfaces

La plupart des interfaces de ce module, c'est :
- de nous demander le nom du graphe qui est considéré comme clé pour travailler sur les fichiers
- d'afficher les messages
  - soit c'est fait
  - soit c'est refusé à cause des erreurs sémantiques ou syntaxiques.

Nom de procédure	Description
- INITREAD	: l'interface de la lecture d'un graphe du fichier.
- INITWRITE	: l'interface de l'écriture d'un graphe au fichier.
- INITUPDATE	: l'interface de la mise à jour d'un graphe du fichier.
- INITDELREC	: l'interface de la suppression d'un graphe du fichier.
- INITLIST	: effacer sur l'écran la liste de noms du graphe du fichier correctement, c'est à dire elle s'occupe du fait de tourner la page lorsque la liste est plus longue que la taille de l'écran.
- INITFILE	: l'interface de l'état FIC du logiciel qui permet de réaliser les opérations désirées telles que : Read, Write, Update, Delete, list, Terminate [cfr. II.2.6. de la deuxième partie] en appelant les autres procédures du modules.

IV.5. MODULE DE TRAITEMENT DU GRAPHE

C'est le plus grand module du logiciel et sa fonction est d'analyser un graphe. On a décomposé ce module en trois sous-modules :

- Interprète de commandes
- Solutions du problème
- Routines du graphe

pour pouvoir l'étudier facilement.

IV.5.1. Sous-module de routines du graphe

i/ Objectif

Le but de ce sous-module facilite les utilisateurs à analyser un graphe ou à écrire eux-mêmes un programme concerné (on verra plus de détails dans le chapitre V). Ce module est basé fortement sur les notions que l'on a vues au premier chapitre de la première partie. Evidemment, ici on n'énumère pas toutes les procédures du module, mais on peut les consulter dans l'annexe pour bien les utiliser.

ii/ Les procédures du module

Nom de procédure	Description
- ASCENDANTS	: l'ensemble des ascendants du noeud demandé.
- DESCENDANTS	: l'ensemble des descendants du noeud demandé.
- SUC.	: l'ensemble des successeurs du noeud demandé.
- PRED.	: l'ensemble des prédécesseurs du noeud demandé.
- COMP-CONNEXE	: le composant connexe correspondant au noeud demandé.

- COMP-FORT-CONNEXE : le composant fortement connexe correspondant au noeud demandé.
- ISOLE : l'ensemble des noeuds isolés du graphe.
- COMPLET : détecter si le graphe est-il complet.
- CONNEXE : détecter si le graphe est-il connexe.
- FORT-CONNEXE : détecter si le graphe est-il fortement connexe.
- ANTI-SYMETRIQUE : détecter si le graphe est-il anti-symétrique.
- SYMETRIQUE : détecter si le graphe est-il symétrique.
- EXISTCHEMIN : détecter s'il existe un chemin d'un noeud à un autre.
- EXISTE-CIRCUIT : détecter s'il existe un circuit d'un noeud à lui-même.
- EXISTE-BOUCLE : chercher tous les noeuds du graphe ayant un boucle.
- EXISTE-ARBRE : détecter si le graphe est-il un arbre.
- DMOINS : donner le degré moins d'un noeud donné.
- DPLUS : donner le degré plus d'un noeud donné.
- DEGRE : donner le degré d'un noeud donné.
- ORDRE : donner le nombre de noeuds d'un graphe donné.
- COMPLEMENTAIRE : trouver le complément d'un graphe.

- TRANSFORM-NON-ORIENT : transformer un graphe de devenir un graphe non orienté.

IV.5.2. Sous-module des solutions du problème

i/ Objectif

Ce sous-module donne un privilège à la résolution des problèmes du graphe.Mais il ne peut donner que des solutions des problèmes que l'on a examinés au 2ème chapitre de la première partie. Comme l'on a vu dans la figure 23, ce sous-module utilise la plupart des procédures du sous-module précédent.

ii/ Les procédures du module

Dans le deuxième chapitre de la peemière partie, on a exposé seulement cinq problèmes les plus courants, on a alors cinq grandes procédures correspondantes sous-mentionnées :

- PLUSCRTCHEM : pour le problème du plus court chemin.
- ORDONNAN : pour le problème de l'ordonnancement.
- CHAINEULER : pour le problème de chaîne eulérienne.
- POSTCHI : pour le problème de postier chinois.
- VOYACOM : pour le problème de voyage de commerce.

IV.5.3. Sous-module de l'interprète de commande

i/ Objectif

Ce sous-module a pour but d'interpréter les commandes requises par l'utilisateur et d'exécuter ces commandes. Ainsi il devrait :

- occuper soi-même l'interface entre les utilisateurs et les sous-modules plus les modules précédents.

- et appeler les procédures disponibles des autres modules et sous-modules.

ii/ Les procédures du module

Ici, on ne liste pas toutes les procédures du module car ses fonctions se ressemblent. Les fonctions principales sont :

- de faire la conversation avec l'utilisateur.
- de détecter les erreurs syntaxiques et sémantiques.
- d'appeler les procédures du sous-module de routines et du sous module de solutions concernant la demande de l'utilisateur.

CHAPITRE V

L'EXTENSION DU LOGICIEL

Le logiciel GRAPHLOG offre aux utilisateurs un pouvoir de greffer leurs propres programmes en utilisant les procédures disponibles du logiciel. Ce fait est réalisable car le logiciel est modulaire et de plus le langage PASCAL VAX/VMS permet de créer et compiler les modules séparément. Remarquons que chaque module correspond à un fichier et par conséquent la liste est comme suit :

Nom du module	Nom du fichier correspondant
- Module de gestion d'écran	ECRAN.PAS
- Module de traitement de noeuds	VERTEX.PAS
- Module de traitement d'arcs	EDGE.PAS
- Module de gestion de fichier	FICHIER.PAS
- Sous-module d'interprète de commande	COMGRAPH.PAS
- Sous-module de solutions de problème	PROBGRAPH.PAS
- Sous module de routines	ROUTGRAPH.PAS

et le programme principal du logiciel est mis dans le fichier GRAPHLOG.PAS .

Ici, on ne s'intéresse qu'à ajouter les programmes concernant les traitements sur graphe, sans nécessairement savoir la représentation du graphe. Pour cela, l'utilisateur doit connaître simplement la variable globale ENSBLE et certaines procédures du module de traitement de noeuds et du module de traitement d'arcs [cfr. Chap. IV de la 2ème partie]

Exemple

Etant donné un graphe G = [X,U] et un sous-ensemble S de X.

L'instruction suivante

$\forall x \in S \equiv$ forall x in S (langage VERTEX)

est traduite en PASCAL comme suit :

```
for x : = 1 to SIZE ( ENSBLE ) do
if x in S then
   begin
      •
      •
      •
   end;
```

d'où S est un sous-ensemble de [1 ... 255]

On peut classer les programmes d'utilisateurs en deux types :

- Les routines de base du graphe.
- Les solutions des problèmes concernant la théorie des graphes.

Selon le type du programme, on prend, soit le fichier ROUTGRAPH.PAS, soit le fichier PROBGRAPH.PAS pour l'y insérer. La partie d'interface du programme est mise dans le fichier COMGRAPG.PAS. Enfin, toutes les nouvelles procédures ou fonctions doivent être mentionnées dans le fichier GRAPHLOG.PAS .

Remarquons que tous les travaux sus-dits sont effectués à l'aide d'un logiciel disponible sur le VAX : EDIT .

Pour être prêt à exécuter le logiciel, on devrait faire les deux commandes suivantes l'une après l'autre :

```
$ COMPILE
$ LINKG
```

L'exemple suivant illustre tout ce qui précède aux utilisateurs.

Exemple

On veut ajouter une procédure concernant la détection d'un graphe antisymétrique.

Via le logiciel EDIT du VAX/VMS, on va créer les procédures suivantes :

- La fonction ANTI-SYMETRIQUE dans le fichier ROUTGRAPH.PAS :

```
[ GLOBAL ] function ANTI-SYMETRIQUE ( g: setvertex ) :
boolean ; label 10 ;
ver x,y : integer ;
    begin
      anti-symétrique : = false ;
      for x : = 1 to size ( g ) do
      for y : = 1 to size ( g ) do
if ( edgexist (g,x,y) ) and ( edgexist (g,y,x) )
then goto 10 ;
      anti-symétrique : = true ;
      10 : end ;
```

- La procédure INITANT dans le fichier COMGRAPH.PAS :

```
[ GLOBAL ] procedure INITANT ;
ver tiep : char ;
    begin
      home ;
      clreos ;
      gotoxy ( 10,25 ) ;
      if not ( trong ) then
      begin
      if anti-symétrique ( ensble )
```

```
then writeln (' + THE GRAPH IS ANTISYMETRIC ! ')
else writeln (' + THE GRAPH IS NOT ANTISYMETRIC ! ') ;
read ( tiep ) ;
      end ;
end ;
```

On doit ajouter dans le fichier GRAPHLOG.PAS :

```
[EXTERNAL]  function ANTI-SYMETRIQUE ( g : setvertex :
            boolean ; EXTERN ;
[EXTERNAL]  procedure INITANT ;
            EXTERN ;
```

CONCLUSION

Le but du mémoire est de créer un logiciel d'aide à la théorie des graphes.

Bien que le but visé ait été atteint, il faudrait cependant souligner les limites du logiciel et les difficultés à surmonter tout au long de la réalisation du travail.

Comme la théorie des graphes est tellement vaste et ses applications sont multiformes, alors notre logiciel est limité dans certains points de vue. Il ne nous permet qu'à résoudre les cinq problèmes figurés dans la première partie et certains tests sur le graphe. Néanmoins, grâce à sa structure modulaire, notre logiciel fournit à l'utilisateur des possibilités d'extension, ce qui lui permettra d'agir aisément à son propre besoin.

Au départ, nous avons dû investir beaucoup de temps pour trouver une structure de données adaptable au graphe. Cette structure doit être cohérente aux algorithmes et en même temps satisfaisante à la manipulation de l'utilisateur sur les informations du graphe. De plus, on a aussi dû penser à archiver les informations du graphe sur un fichier. Ce travail est relativement facile grâce aux avantages du langage disponible VAX11-PASCAL.

Une autre difficulté qui nous demande pas mal de temps est que la plupart des algorithmes sont écrits d'une manière assez "littéraire". Bien sûr, l'exposé "littéraire" est beaucoup plus clair pour la compréhension de l'algorithme mais la mise en oeuvre sur l'ordinateur nous exige beaucoup de travaux.

De plus, rappelons que notre logiciel est exécuté en mode interactif, donc on doit prévoir toutes les erreurs syntaxiques et sémantiques. Pour ce faire, on a crée plusieurs fonctions et procédures pour la couche d'interface entre le logiciel et l'utilisateur.

Le logiciel est traduit en VAX11-PASCAL, par conséquent, il a aussi certains points faibles :

- Du point de vue dimension des graphes que l'on peut traiter, nous sommes actuellement limités à 255 noeuds et 255 x 255 arcs (ou arêtes).
- Le poids des arcs (ou arêtes) ne peut pas dépasser 9.999.999,99. Mais ceci n'est pas vraiment une contrainte, puisqu'on a une possibilité d'augmenter la valeur du poids en modifiant les constantes du programme.

En dernier lieu, il faut remarquer que notre logiciel peut-être implémenté sur le PDP-11 car la famille de VAX est compatible avec le PDP-11.

BIBLIOGRAPHIE

[1] AHO , HOPCROFT , ULLMAN
The design and analysis of computer algorithms.
Addison-Wesley publishing company.

[2] BELLMORE , M. and NEMHAUSER , G.L.
The travelling salesman problem - a survey
Ops. Res. , 16 , p.538 - 1968

[3] BERGE , C
The theory of graphs.
Methuen , London - 1962.

[4] BERNARD HARRIS
Graph theory and its applications.
Academic Press - New York , London - 1970

[5] BERNARD ROY , MICHEL HORPS
Arrangements remarquables d'arcs ou d'arêtes d'un graphe.
Dunod - Paris - 1970

[6] EDMONDS J. , JOHNSON E.L. (1973)
Matching , Euler tours , and the chinese postman.
Math. Programming , 5 , 1 , pp. 88-124.

[7] FICHEFET J.
La théorie des graphes.
Notes de cours.
Institut d'Informatique , Namur.

[8] HARARY F. (1959)
Graph theory.
Addison Wesley.

[9] JOHNSON D.B. (1973)
A note on Dijkstra's shortest path algorithm.
Journal ACM , 20 , PP. 385-388

[10] LAWLER E.L. , WOOD D.E. (1966)
Branch and bound methods : a survey.
Opns. Res. , 14 , pp. 699-719

[11] LITTLE J.D.C. , MURTY K.G. ,
SWEENEY D.W. , KAREL C. (1963).
An Algorithm for the travelling - Salesman Problem.
Opns. Res. , 11 , nº 2 , pp. 201-222.

[12] MICHEL GONGRAN , MICHEL MINOUX.
Graphes et algorithmes
Editions Eyrolles - 1979

[13] NICHOLSON , T.A.J.
Finding the shortest route between two points in a network.
The Computer Jl. , 9 , p.275 - 1966.

[14] NICOS CHRISTOFIDES.
Graph Theory - An algorithmic approach.
Academic Press - NewYork , London , San Francisco - 1975.

[15] NICOS CHRISTOFIDES. (1972)
Bounds for the travelling - Salesman problem.
Operations Research , 20 , pp.1044-1056.

[16] RICHARD BELLMAN , KENNETH L. COOKE ,
JO ANN LOCKETT.
Algorithms graphs and computers
Academic Press - NewYork and London - 1970

[17] RONALD C. READ.
Graph Theory and Computing.
Academic Press - NewYork and London - 1972.

[18] STRICKER , R. (1970)
Public sector vehicle routing - The Chinese postman problem.
Ph. D. Thesis , Electrical Eng. Dept. , M.I.T.

[19] F. GRIZE et A. STROHMEIER
SARTEX - Manuel de référence du langage
Version 2.0 - Juin 1983.

[20] DIGITAL EQUIPMENT CORPORATION
VAX - Software handbook.
1982.

[21] DIGITAL EQUIPMENT CORPORATION
VAX - Architecture handbook

[22] HENRY M. LEVY , RICHARD H. ECKHOUSE , JR.
Computer programming and architecture
the VAX-11.
Digital Press - April 1980

[23] DIGITAL EQUIPMENT CORPORATION
Programming in VAX-11 PASCAL
July 1983

[24] DIGITAL EQUIPMENT CORPORATION
VAX-11 PASCAL - Language Reference Manual
October 1982

FACULTES UNIVERSITAIRES NOTRE DAME DE LA PAIX - NAMUR

INSTITUT D'INFORMATIQUE

IMPLEMENTATION D'UN LOGICIEL D'AIDE A LA THEORIE DES GRAPHES

(ANNEXE)

Mémoire présenté par

NGUYEN THANH LAM

en vue de l'obtention du grade de

Licencié et Maître en Informatique

Année Académique 1984-1985

1 - MANUEL DE L'UTILISATEUR ET EXEMPLE D'UTILISATION

2 - TYPES DE DONNEES ET VARIABLES GLOBALES

3 - SPECIFICATIONS DE PROCEDURE

A - MANUEL DE L'UTILISATEUR

1- Convention concernant l'utilisation du logiciel

< > = Appuyer sur la touche RETURN du clavier.
<X> = Frapper sur la touche X du clavier.
▨ = Position du curseur sur l'écran.

2- Exécution du logiciel

- Pour déclencher le logiciel GRAPHLOG, il suffit de taper la commande ∅ RUN GRAPHLOG < >

Suivant cette commande, une image apparaît sur l'écran et le logiciel désiré est prêt à utiliser.

- Pour continuer le logiciel, on doit taper < >

3- Les menus principaux

Dès le début de l'exécution, le logiciel affiche sur l'écran le premier menu.

SUPERVISOR : D(eclare), E(nd), F(ile), G(raph) ▨

- Pour traiter les données d'un graphe: <D > (1)
- Pour traiter les fichiers du graphe : <F > (2)
- Pour étudier le graphe : <G > (3)
- Pour terminer le logiciel : <E >

Suivant la touche tapée, on voit sur l'écran les menus suivants :

(1) DECLARE: V(ertex), E(dge), T(erminate)

==========

. Pour traiter les noeuds d'un graphe: <V> (4)

. Pour traiter les arcs d'un graphe : <E> (5)

. Pour retourner au SUPERVISOR : <T>

(2) FILE: R(ead), W(rite), U(pdate), D(eleterec),
L(ist), T(erminate)

==========

. Pour lire un graphe archivé dans
les fichiers : <R>

- Le logiciel demande le nom du graphe du fichier à l'utilisateur qui veut le lire.

ENTER THE NAME OF GRAPH :

- Le nom du graphe doit être majuscule et ne peut pas dépasser 20 caractères.

- Après avoir fini la lecture, le message suivant apparaît sur l'écran :

Readfile was done ! < >

. Pour écrire un graphe sur les fichiers: <W>

. Pour mettre à jour un graphe existant
dans les fichiers: <U>

. Pour supprimer un graphe existant
dans les fichiers: <D>

Reamarquons que les opérations sus-dites se font comme des opérations de lecture.

. Pour lister tous les noms du graphe
sur les fichiers: <L>

. Pour retourner au SUPERVISOR: <T>

(3)

****
* CHOIX DU MENU *
****

1-ANTISYMETRIC	12-EXIST-CIRCUIT	23-STRONG-COMPONENTS
2-ASCENDANT	13-EXIST-LOOP	24-SHORTEST-PATH
3-CHAIN-EULERIAN	14-EXIST-PATH	25-SUCCESSORS
4-COMPLETE	15-EXIST-TREE	26-SYMETRIC
5-CONNECTED	16-ISOLED-POINTS	27-TRAVEL-COMMERCIAL
6-COMPONENTS	17-NONDIRECT-FORM	28-TERMINATE
7-COMPLEMENT	18-ORDONNANCEMENT	
8-DEGREE	19-ORDRE	
9-DEGREE-IN	20-PREDECESSOR	
10-DEGREE-OUT	21-POSTIER-CHINOIS	
11-DESCENDANT	22-STRONGLY-CONNECTED	

L'utilisateur peut choisir une opération désirée sur le menu et réaliser cette opération en tapant le numéro correspondant à l'opération. Suivant l'opération, le logiciel demande l'utilisateur d'introduire certaines informations. Le résultat serait affiché sur l'écran après que l'opération soit finie. Et pour retourner au menu du choix, il faut taper< > . Pour retourner au SUPERVISOR, on doit taper le numéro correspondant à l'opération TERMINATE.

(4) VERTEX: C(reate), I(nsert), D(elete), L(ist),
T(erminate), S(upervisor)

==========

. Pour créer un nouveau graphe : <C>

- Le logiciel demande l'utilisateur d'introduire le nom des noeuds du graphe.

ENTER THE NODE :

- Le nom du noeud doit être majuscule et ne peut pas dépasser 30 caractères.

- Si on veut continuer de l'introduire, alors on tape < >

Sinon on tape <T>

- Après avoir fini l'introduction du noeud, une liste de tous les noeuds enregistrés apparaît sur l'écran et l'utilisateur doit taper<> pour continuer.

. Pour insérer un noeud au graphe existant: <I>

- Le logiciel demande à l'utilisateur le numéro du noeud pour insérer le nouveau noeud derrière le noeud demandé.

ENTER THE NUMBER : ▨

- Après avoir fait entrer le numéro, le logiciel demandera le nom du nouveau noeud.

ENTER THE NODE : ▨

- L'opération sera terminée en affichant sur l'écran le message :

Insert was done ! Continue ?

- Si l'utilisateur veut continuer, alors il tape <>, sinon <T>

. Pour supprimer un noeud du graphe: <D>

- Le logiciel demande simplement le numéro du noeud à supprimer et le reste se fait comme l'insertion d'un noeud.

. Pour lister tous les noeuds du graphe: <L>
. Pour retourner au DECLARE : <T>
. Pour retourner au SUPERVISOR : <S>

(5) EDGE: I(nsert), D(elete), L(ist), T(erminate), S(upervisor) ▨

==========

. Pour insérer un arc au graphe : <I>

- Le logiciel demande à l'utilisateur le numéro de l'extrémité initiale de l'arc à insérer:

ENTER THE NODE INITIAL : ▨

et le numéro de l'extrémité finale :

ENTER THE NODE FINAL : ▨

et la valeur du poids de l'arc :

ENTER THE POUND OF EDGE : ▨

- Dans le cas où l'arc n'a pas de poids, on tape simplement < > .

- Après avoir fini l'introduction d'un arc, un message apparaît sur l'écran:

Insert was done ! Continue ? ▨

- Si l'utilisateur veut continuer, alors il tape< > sinon <T>

. Pour supprimer un arc du graphe : <D>

- Le logiciel demande simplement les numéros des deux extrémités de l'arc à supprimer.

. Pour lister tous les arcs du graphe: <L>

. Pour retourner au DECLARE : <T>

. Pour retourner au SUPERVISOR : <S>

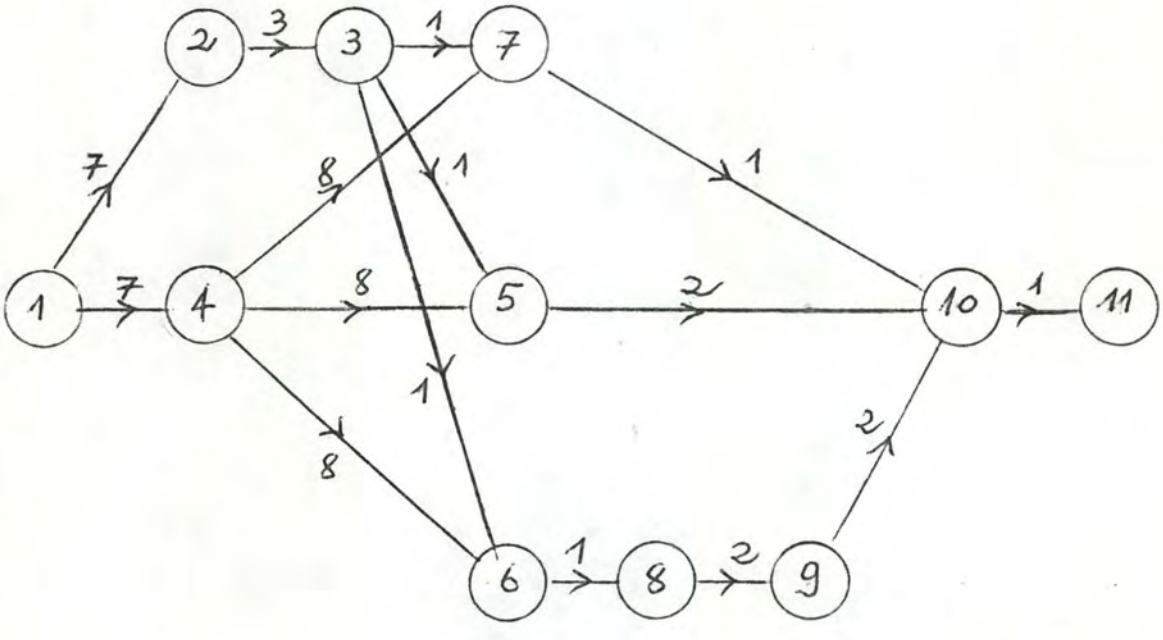
B - EXEMPLE D'UTILISATION

Afin d'illustrer l'utilisation du logiciel, considérons le problème de la construction d'un pavillon.

En effet, la construction d'un pavillon demande la réalisation d'un certain nombre de tâches. Le tableau suivant représente les différentes tâches avec leurs relations d'antériorité.

Code de la tâche		Durée (en semaines)	Tâches antérieures.
1	Travaux de maçonnerie	7	-
2	Charpente de la toiture	3	1
3	Toiture	1	2
4	Installation sanitaire et électrique	8	1
5	Façade	2	4,3
6	Fenêtres	1	4,3
7	Aménagement du jardin	1	4,3
8	Travaux de plafonnage	3	6
9	Mise en peinture	2	8
10	Emménagement	1	5,7,9

Le travail commençant à la date 0, on cherche un ordonnancement qui minimalise la durée totale du travail, donc la date de fin des travaux. Pour cela, on construit le graphe suivant :



Dans les pages suivants, on présente l'évolution d'exécution du logiciel pour résoudre le problème d'ordonnancement.

- Declencher le logiciel

*<RUN GRAPHLOG> < >

```
================================================================================

SUPERVISOR:     D(eclare),E(nd),F(ile),G(raph) <D>< >
================================================================================
```

- Declarer les noeuds

```
DECLARE:     V(ertex),E(dge),T(erminate) <V> < >
================================================================================

VERTEX:     C(reate),I(nsert),D(elete),L(ist),T(erminate),S(upervisor) <C>< >


ENTER THE NODE :    TRAVAUX DE MACONNERIE
ENTER THE NODE :    CHAPENTE DE LA TOITURE
ENTER THE NODE :    TOITURE
ENTER THE NODE :    INSTALLATIONS SANIT ET ELECT
ENTER THE NODE :    FACADE
ENTER THE NODE :    FENETRES
ENTER THE NODE :    AMENAGEMENT DU JARDIN
ENTER THE NODE :    TRAVAUX DE PLAFONNAGE
ENTER THE NODE :    MISE EN PEINTURE
ENTER THE NODE :    EMMENAGEMENT
ENTER THE NODE :    FIN

< >

ENTER THE NODE :
TRAVAUX DE MACONNERIE                1
CHAPENTE DE LA TOITURE               2
TOITURE                              3
INSTALLATIONS SANIT ET ELECT         4
FACADE                               5
FENETRES                             6
AMENAGEMENT DU JARDIN                7
TRAVAUX DE PLAFONNAGE                8
MISE EN PEINTURE                     9
EMMENAGEMENT                        10
FIN                                 11

< >


================================================================================

VERTEX:     C(reate),I(nsert),D(elete),L(ist),T(erminate),S(upervisor) <I>< >
================================================================================
```

- Déclarer les arcs

```
DECLARE:     V(ertex),E(dge),T(erminate) <E><>
=============================================================================

EDGE:      I(nsert),D(elete),C(hange),L(ist),T(erminate),S(upervisor) <I><>


ENTER THE NODE INITIAL :<1>
ENTER THE NODE FINAL :<2>
ENTER THE POUND OF EDGE :<7>
insert was done! Continue? <>


ENTER THE NODE INITIAL :<1>
ENTER THE NODE FINAL :<4>
ENTER THE POUND OF EDGE :<7>
insert was done! Continue? <>


ENTER THE NODE INITIAL :<2>
ENTER THE NODE FINAL :<3>
ENTER THE POUND OF EDGE :<3>
insert was done! Continue? <>


ENTER THE NODE INITIAL :<3>
ENTER THE NODE FINAL :<7>
ENTER THE POUND OF EDGE :<1>
insert was done! Continue? <>


ENTER THE NODE INITIAL :<3>
ENTER THE NODE FINAL :<5>
ENTER THE POUND OF EDGE :<1>
insert was done! Continue? <>


ENTER THE NODE INITIAL :<3>
ENTER THE NODE FINAL :<6>
ENTER THE POUND OF EDGE :<1>
insert was done! Continue? <>


ENTER THE NODE INITIAL :<4>
ENTER THE NODE FINAL :<7>
ENTER THE POUND OF EDGE :<8>
insert was done! Continue? <>


ENTER THE NODE INITIAL :<4>
ENTER THE NODE FINAL :<5>
ENTER THE POUND OF EDGE :<8>
insert was done! Continue? <>


ENTER THE NODE INITIAL :<4>
ENTER THE NODE FINAL :<6>
ENTER THE POUND OF EDGE :<8>
insert was done! Continue? <>
```

```
ENTER THE NODE INITIAL :<7>
ENTER THE NODE FINAL :<10>
ENTER THE POUND OF EDGE :<1>
insert was done! Continue?  < >


ENTER THE NODE INITIAL :<5>
ENTER THE NODE FINAL :<10>
ENTER THE POUND OF EDGE :<2>
insert was done! Continue?  < >


ENTER THE NODE INITIAL :<6>
ENTER THE NODE FINAL :<8>
ENTER THE POUND OF EDGE :<1>
insert was done! Continue?  < >


ENTER THE NODE INITIAL :<8>
ENTER THE NODE FINAL :<9>
ENTER THE POUND OF EDGE :<2>
insert was done! Continue?  < >


ENTER THE NODE INITIAL :<9>
ENTER THE NODE FINAL :<10>
ENTER THE POUND OF EDGE :<2>
insert was done! Continue?  < >


ENTER THE NODE INITIAL :<10>
ENTER THE NODE FINAL :<11>
ENTER THE POUND OF EDGE :<1>
insert was done! Continue? <T>
```

```
EDGE:     I(nsert),D(elete),C(hange),L(ist),T(erminate),S(upervisor) <L>< >
TRAVAUX DE MACONNERIE          => CHAPENTE DE LA TOITURE          :      7.00
TRAVAUX DE MACONNERIE          => INSTALLATIONS SANIT ET ELECT    :      7.00
CHAPENTE DE LA TOITURE         => TOITURE                         :      3.00
TOITURE                        => AMENAGEMENT DU JARDIN           :      1.00
TOITURE                        => FENETRES                        :      1.00
TOITURE                        => FACADE                          :      1.00
INSTALLATIONS SANIT ET ELECT   => AMENAGEMENT DU JARDIN           :      8.00
INSTALLATIONS SANIT ET ELECT   => FENETRES                        :      8.00
INSTALLATIONS SANIT ET ELECT   => FACADE                          :      8.00
FACADE                         => EMMENAGEMENT                    :      2.00
FENETRES                       => TRAVAUX DE PLAFONNAGE           :      1.00
AMENAGEMENT DU JARDIN          => EMMENAGEMENT                    :      1.00
TRAVAUX DE PLAFONNAGE          => MISE EN PEINTURE                :      2.00
MISE EN PEINTURE               => EMMENAGEMENT                    :      2.00
EMMENAGEMENT                   => FIN                             :      1.00
```

```
EDGE:     I  sert),D(elete),C(hange),L(ist),T(erminate),S(upervisor) <S>< >
```

- Archiver le graphe sur le fichier

```
SUPERVISOR:     D(eclare),E(nd),F(ile),G(raph) <F><>
================================================================================

FILE:     R(ead),W(rite),U(pdate),D(elete_rec),L(ist),T(erminate) <W><>
ENTER THE NAME OF GRAPH :
ORDONNANCEMENT
writefile was done !  <>
================================================================================

FILE:     R(ead),W(rite),U(pdate),D(elete_rec),L(ist),T(erminate) <T><>
================================================================================
```

SUPERVISOR: D(eclare),E(nd),F(ile),G(raph) <G><>

* * * * * * * *
* CHOIX DU MENU *
* * * * * * * *

... 19- ORDONNANCEMENT
...

>18 <

TACHES	DT +TOT	DT +TARD	MARGE	CRITIQUE
TRAVAUX DE MACONNERIE	0	0	> 0 <	*** 1***
CHAPENTE DE LA TOITURE	7	11	> 4 <	
TOITURE	10	14	> 4 <	
INSTALLATIONS SANIT ET ELECT	7	7	> 0 <	*** 2***
FACADE	15	18	> 3 <	
FENETRES	15	15	> 0 <	*** 3***
AMENAGEMENT DU JARDIN	15	19	> 4 <	
TRAVAUX DE PLAFONNAGE	16	16	> 0 <	*** 4***
MISE EN PEINTURE	18	18	> 0 <	*** 5***
EMMENAGEMENT	20	20	> 0 <	*** 6***
FIN	21	21	> 0 <	*** 7***

* * * * * * * *
* CHOIX DU MENU *
* * * * * * * *

... 28- TERMINATE

>28 <

==========

SUPERVISOR: D(eclare),E(nd),F(ile),G(raph) <E><>

LES TYPES DE DONNEE ET LES VARIABLES GLOBALES

I-LES CONSTANTS DU PROGRAMME.

```
maxlong = 30       : la longueur maximale d'une chaine de caracteres de
                     type message;
maxchiff= 4        : la longueur maximale d'un nombre entier;
highval=9999       : la valeur maximale d'un nombre entier;
highvalr=2**15     : la valeur maximale d'un nombre reel;
maxsothuc=15       : la longueur maximale d'un nombre reel;
maxrang=255        : la taille maximale des rangs du graphe;
maxstring = 20     : la longueur maximale d'une chaine de caracteres de
                     type de string;
blanche = '                                '    : le blanc correspondant
                                                  a un nom de type message;
blanchestr = '                                  '  : le blanc correspondant a un nom
                                                     de type string;
```

II-LES TYPE DE DONNEE.

```
string = packed array[1..maxstring] of char;
message = packed array[1..maxlong] of char;
resulset= set of 0..255;
rang = array[1..maxrang] of integer;
rangr = array[1..maxrang] of real;
arc = record
        n_init : integer;
        n_fin : integer;
        n_pd  : real;
      end;
fvertex = record
            nom_graph_v : [KEY(0)] string;
            size_node : integer;
            nom_node : array[1..maxrang] of message;
          end;
fedge = record
          num_ord : [KEY(0)] integer;
          nom_graph_e : [KEY(1)] string;
          size_edge : integer;
          nom_edge :  setarc;
        end;
stackver = ^pointerlist;
pointerlist = record
                node : integer;
                nextnode : stackver;
              end;
matadj = ^pointeredg;
pointeredg = record
               initial : integer;
               final   : integer;
               poid    : real;
               colink: matadj
             end;
setvertex = ^ pointerver;
pointerver = record
               current: setvertex;
               nomint : integer;
               nomext : message;
               link   : setvertex;
               colink0: matadj
               end;
```

III-LES VARIABLES GLOBALES.

```
filever : file of fvertex    : le fichier de noeuds du graphe;
recver  : fvertex            : l'enregistrement du fichier de noeuds;
fileedg : file of fedge      : le fichier d'arcs du graphe;
recedg  : fedge              : l'enregistrement du fichier d'arcs;
ensble:setvertex             : le graphe;
i,compt:integer              : les compteurs;
command,mess:message         : les donnees entrees;
newcell:setvertex            : une nouvelle cellule;
stoop,trong,xong:boolean     : les swichts de booleens;
com:char                     : un swicht de caractere;
newelem:matadj               : un nouveau element;
```

LES SPECIFICATIONS DE PROCEDURE DU LOGICIEL

****

LES PROCEDURES DU MODULE DE GESTION FICHIER

****

1-Procedure CREATFILE.

-Arguments: -

-Precondition: -

-Sorties: + fichier des noeuds : FILEVER.DAT,
+ fichier des arcs : FILEEDG.DAT.

-Postcondition: + les deux fichiers sont indexes et permettent les acces par cle.

-Fonction:

+ Ce procedure permet de creer deux fichiers :fichier des noeuds et fichier des arcs concernant au graphe.Ce procedure est traiter uniquement une seule fois au moment d'implementer le logiciel la premiere fois .

2-Procedure READFILEV.

-Arguments: + la cle du record du fichier FILEVER.DAT a lire.

-Preconditions: + la cle du record ne peut pas etre plus longue que vingt caracteres.

-Sorties: + le record est lu.

-Postconditions: + le record est mis dans la memoire dans la variable recver.

-Fonction:

+ Ce procedure permet de lire les noeuds d'un graphe archive dans le fichier FILEVER.DAT en donnant le nom de ce graphe,considere comme cle d'acces.

3-Procedure READFILEE.

-Arguments: + la cle du record du fichier FILEEDG.DAT a lire.

-Preconditions: + la cle du record ne peut pas etre plus longue que vingt caracteres.

-Sorties: + le record est lu.

-Postconditions: + le record est mis dans la memoire dans la variable recedg.

-Fonction:

+ Ce procedure permet de lire les arcs(ou aretes) d'un graphe archive dans le fichier FILEEDG.DAT en donnant le nom du graphe,considere comme cle d'acces.

4-Procedure WRITEFILEV.

-Arguments: -

-Preconditions: -

-Sorties: + le fichier FILEVER.DAT.

-Postconditions: + la variable recver est ecrite dans le fichier FILEVER.DAT

-Fonction:

+ Ce procedure permet d'ecrire tous les noeuds d'un graphe sur le fichier FILEVER.DAT a l'aide de la variable recver.

5-Procedure WRITEFILEE.

-Arguments: -

-Preconditions: -

-Sorties: + le fichier FILEEDG.DAT.

-Postconditions: + la variable recedg est ecrite sur le fichier FILEEDG.DAT

-Fonction:

+ Ce procedure permet d'ecrire tous les arcs(ou les aretes) d'un graphe sur le fichier FILEEDG.DAT a l'aide de la variable recedg.

6-Procedure UPDATEFILE.

-Arguments: + la cle du record a mettre a jour.

-Preconditions: + la cle du record ne peut pas etre plus que vingt caracteres.

-Sorties: + les deux fichiers FILEVER.DAT,FILEEDG.DAT.
+ les messages a l'ecran.

-Postconditions: + si le record n'existe pas alors afficher sur l'ecran le message d'erreur,
+ sinon
les records des deux fichiers ayant cette cle sont mis a jour,
afficher un message de dire cette mise a jour est faite.

-Fonction:

+ Ce procedure permet de mettre a jours d'un graphe archive dans les deux fichiers FILEVER.DAT,FLILEEDG. DAT en donnant le nom de ce graphe,considere comme cle d'acces.Dans le cas le record a mettre a jour existe,il le fait et en suite il va afficher sur l'ecran le message :'updatefile was done',par contre il ne fait qu'afficher le message de refus.

7-Procedure DELRECFILE.

-Arguments: + la cle du record a suprimer.

-Preconditions: + la cle du record ne peut pas etre plus longue que vingt caracteres.

-Sorties: + les deux fichiers FILEVER.DAT,FILEEDG.DAT.
+ les messages a l'ecran.

-Postconditions: + si le record n'existe pas alors afficher sur l'ecran le message d'erreur,
+ sinon
les records des deux fichiers ayyant cette cle sont suprimes,
afficher un message de dire cette supression est faite.

-Fonction:

+ Ce procedure permet de suprimer d'un graphe archive dans les deux fichiers FILEVER.DAT,FILEEDG.DAT en donnant le nom de ce graphe, considere comme cle d'acces.Dans le cas la record a suprimer existe, il lefait et ensuite il va afficher sur l'ecran un message de dire c' est fait,et par contre il ne fait qu'afficher le message de refus.

8-Procedure CLOSEFILE.

-Arguments: -

-Preconditions: -

-Sorties: + les deux fichiers FILEVER.DAT,FILEEDG.DAT.

-Postconditions: + les deux fichiers sont fermes.

-Fonction:

+ Ce procedure permet de cloturer deux fichiers: FILEVER.DAT, FILEEDG.DAT.

9-Procedure INITREAD.

-Arguments: -

-Preconditions: -

-Sorties: + soit le graphe est lu,
soit c'est un message d'erreur.

-Postconditions: + si le nom du graphe entre de l'ecran est correct
c'est donc la representation du graphe sous la forme
de pointeur,
les deux fichiers FILEVER.DAT et FILEEDG.DAT sont fermes.

-Fonction:

+ C'est l'interface de realiser d'une lecture de fichier.Il s'occupe les taches suivantes:

* demander le nom du graphe (interractif),
* ouvrir les fichiers,
* detecter l'erreurs semantiques:
si le nom du graphe existe,alors
faire la lecture des noeuds et des arcs;
sinon afficher sur l'ecran le message d'erreur.
* fermer les fichiers.

10-Procedure INITWRITE.

-Arguments: -

-Preconditions: -

-Sorties: + soit le graphe est ecrit,
soit c'est un message d'erreur.

-Postconditions: + les fichiers sont mise-a-jour en ajoutant de
nouveau graphe si le nom du graphe inexistant

-Fonction:
+ C'est l'interface de realiser une ecriture de fichier.Il s'occupe les taches suivantes:

* demander le nom du graphe (interractif),
* ouvrir les fichiers,
* detecter les erreurs semantiques:
si le nom du graphe inexistant,
alors ecrire le graphe sur le fichier,
sinon demander a l'utilisateur de faire une update
au lioeu d'une ecriture,
* fermer les fichiers.

11-Procedure INITUPDATE.

-Arguments: -

-Preconditions: -

-Sorties: + soit le graphe est modifie sur les fichiers,
soit c'est un message d'erreur.

-Postconditions: + les fichiers sont mise-a-jour en modifiant le graphe existant sur les fichiers.

-Fonction:
+ C'est l'interface de realiser d'une mise-a-jour de fichier.Il s'occupe les taches suivantes:

* demander le nom du graphe (interractif),
* ouvrir les fichiers,
* detecter les erreurs semantiques:
si le nom du graphe existant,
alors faire la modification de ce graphe,
sinon demander a l'utilisateur de creer le graphe,
* fermer les fichiers.

12-Procedure INITDELREC.

-Arguments: -

-Preconditions: -

-Sorties: + soit le graphe est supprime,
soit c'est un message d'erreur.

-Postconditions: + les fichiers sont mise-a-jour en supprimant le graphe existant demande.

-Fonction:
+ C'est l'interface de realiser d'une suppression d'un enregistrement de fichier.Il s'occupe les taches suivantes:

* demander le nom du graphe (interractif),
* ouvrir les fichiers,
* detecter les erreurs semantiques:
si le nom du graphe existe,
alors supprimer le graphe,
sinon afficher sur l'ecran un message d'erreur.
* fermer les fichiers.

13-Procedure INITLIST.

-Arguments: -

-Preconditions: -

-Sorties: + la liste des nom de graphe etant dans les fichiers.

-Postconditions: -

-Fonction:
+ L'affichage sur l'ecran la liste des nom de graphe existant dans les fichiers.

14-Procedure INITRFILE.

-Arguments: -

-Preconditions: -

-Sorties: + revenir a l'etat FIC.

-Postconditions: + l'operation demande dans l'etat FIC est fait.

-Fonction:
+ C'est l'interface de realiser d'un choix d'operation de l'etat FIC.

****

LES PROCEDURES DU MODULE DE TRAITEMENT DE NOEUDS

****

1-Procedure NEWSET.

-Arguments: + ens : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + l'ensemble n'existe pas.

-Sorties: + l'ensemble des noeuds et d'acrs .

-Postconditions: + l'ensemble est cree mais vide .

-Fonction:
+ C'est de creer un ensemble vide de noeuds en demandant simplement un pointeur qui pointe vers lui meme.

2-Procedure EMPTY.

-Arguments: + ens : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + l'ensemble doit etre existe.

-Sorties: + une valeur de boolean.

-Postconditions: + cette valeur est TRUE si l'ensemble n'est pas vide, par contre c'est FALSE.

-Fonction:
+ C'est de verifier si l'ensemble de noeuds est il vide.

3-Procedure SEARCH.

-Arguments: + num: le nom interne du noeud,type integer;
ens1: l'ensemble de noeuds et d'arcs du graphe,
type setvertex.

-Preconditions: + num est correct de point de vue syntaxique,
ens1 n'est pas vide.

-Sorties: + l'adresse de l'element num demande.

-Postconditions: + si l'element existe ,alors
SEARCH = l'adresse de cet element,
sinon SEARCH = nil.

-Fonction:
+ C'est de chercher l'adresse d'un element en donnant son nom interne.
cette fonction est faite en consultant sequentiellement les elements de
l'ensemble de noeuds via les pointeurs.

4-Procedure INSERT.

-Arguments: + num: le nom interne du noeud precedent que l'on veut
inserer le nouveau apres,type integer;
value: le nom externe du nouveau noeud,type message;
ens1: l'ensemble de noeuds et d'arcs du graphe,
type setvertex.
duoc: une variable qui montre si l'insertion est elle
bien faite,type boolean;

-Preconditions: + num et value sont corrects au point de vue syntaxique,
ens1 existe.

-Sorties: + l'element demande est insere.

-Postconditions: + si duoc = TRUE
l'ensemble = l'ensemble U {nouveau element insere}
sinon
l'ensemble = l'ensemble

-Fonction:
+ C'est d'inserer un noeud dans l'ensemble de noeuds suivant d'un
ordre desire.

5-Procedure DELETV.

-Arguments: + num: le nom interne du noeud desire a supprimer,type integer;
ens: l'ensemble de noeuds et d'arcs du graphe, type setvertex;
duoc: une variable qui montre si la suppression est elle bien faite,type boolean;

-Preconditions: + num et value sont corrects au point de vue syntaxique, ens1 existe.

-Sorties: + l'element demande est supprime.

-Postconditions: + si duoc = TRUE
l'ensemble = l'ensemble \ {element supprime}
sinon
l'ensemble = l'ensemble

-Fonction:
+ C'est de supprimer un noeud dans l'ensemble de noeuds en donnant son nom interne.

6-Procedure SIZE.

-Arguments: + ens: l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + ens doit etre existe.

-Sorties: + le nombre de noeuds du graphe.

-Postconditions: + SIZE = le nombre de noeuds.

-Fonction:
+ C'est de donner le nombre de noeuds du graphe.

7-Procedure MEMBER.

-Arguments: + value : le nom externe du noeud,type message;
ens: l'ensemble de noeuds et d'arcs du graphe,
type setvertex;

-Preconditions: + value est correct au point de vue de syntax,
ens existe.

-Sorties: + une valeur booleenne.

-Postconditions: + MEMBER = TRUE si l'element demande existe,par contre
MEMBER = FALSE.

-Fonction:
+ C'est de verifier si un element demande est il existe dans l'ensemble de noeuds du graphe en donnant le nom externe du noeud.

8-Procedure LIST.

-Arguments: + ens: l'ensemble de noeuds et d'arcs du graphe,
type setvertex.

-Preconditions: + ens doit etre existe.

-Sorties: + la liste de noeuds du graphe.

-Postconditions: -

-Fonction:
+ c'est de lister une liste de noeuds du graphe en donnant le nom externe et le nom interne du noeud.

9-Procedure NAMEXT.

-Arguments: + ens: l'ensemble de noeuds et d'arcs du graphe,
type setvertex;
numint: le nom interne du noeud donne,type integer.

-Preconditions: + numint est correct au point de vue de syntax,
ens existe.

-Sorties: + le nom externe du noeud demande.

-Postconditions: + NAMEXT = le nom externe du noeud s'il existe,par contre
NAMEXT = blanche.

-Fonction:
+ C'est de donner le nom externe d'un noeud demande lorsqu'on entre son nom interne.

10-Procedure CREATNODE.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est l'interface de realiser d'une creation de noeud d'un graphe. il s'occupe les taches suivantes:
* demander a l'utilisateur d'entrer les noeuds (interractif),
* inserer les noeuds dans l'ensemble de noeuds,
* detecter les erreurs syntaxiques,
* lister les noeuds inseres.

11-Procedure INSERNODE.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est l'interface de realiser d'une insertion de noeud a une position desiree.Il s'occupe les taches suivantes:
* demander a l'utilisateur le nom interne du noeud precedent du noeud qu'on veut entrer (interractif),
* detecter les erreurs syntaxiques et semantiques,
* inserer le noeud demande a la bonne position.

12-Procedure DELETNODE.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est l'interface de realiser d'une suppression de noeud du graphe. Il s'occupe les taches suivantes:
* demander a l'utilisateur le nom interne du noeud voulant supprimer (interractif),
* detecter les erreurs syntaxiques et semantiques,
* supprimer le noeud demande .

13-Procedure IENTERNODE.

-Arguments: + stp : une switch qui permet d'aller a l'etat SUP
selon sa valeur,type boolean

-Preconditions: + stp = FALSE.

-Sorties: + la valeur de stp.

-Postconditions: + soit stp = FALSE si on veut rentrer dans l'etat SUP
soit stp = TRUE si on veut rentrer dans l'etat DEC

-Fonction:
+ C'est l'interface de realiser d'un choix d'operation de l'etat VER.

14-Procedure INITDEC.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est l'interface de realiser d'un choix d'operation de l'etat DEC.

****

LES PROCEDURES DU MODULE DE TRAITEMENT D'ARCS

****

1-Procedure LISTEDGE.

-Arguments: + ens : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + ensemble doit etre existe.

-Sorties: + la liste d'arcs du graphe.

-Postconditions: -

-Fonction:
+ C'est de lister une liste d'arcs du graphe en donnant le nom externe de ses deux extremites et son poid (s'il existe).

2-Procedure EDGEXIST.

-Arguments: + ens: l'ensemble de noeuds et d'arcs du graphe, type setvertex,
numinit: le nom interne de l'extremite initiale, type integer,
numfin: le nom interne de l'extremite finale, type integer.

-Preconditions: + numinit et numfin doivent etre corrects au point de vue de syntax et l'ensemble existe.

-Sorties: + une valeur booleenne.

-Postconditions: + EDGEXIST = TRUE si l'arc demande existe,
EDGEXIST = FALSE si l'arc demande n'existe pas.

-Fonction:
+ C'est de verifier l'existence d'un arc demande .

3-Procedure SIZEEDGE.

-Arguments: + ens: l'ensemble de noeuds et d'arcs du graphe,
type setvertex.

-Preconditions: + ens doit etre existe.

-Sorties: + le nombre d'arcs du graphe.

-Postconditions: + SIZEEDGE = le nombre d'arcs

-Fonction:
+ C'est de donner le nombre d'arcs du graphe.

4-Procedure INSEREDGE.

-Arguments: + numinit : le nom interne de l'extremite initiale,
type integer;
numfin : le nom interne de l'extremite finale,
type integer;
ens : l'ensemble de noeuds et d'arcs du graphe,
type setvertex,
duoc : une variable qui montre si l'insertion est elle
bien faite,type boolean.

-Preconditions: + numinit et numfin sont corrects au point de vue de syntax
ens existe

-Sorties: + l'arc demande est insere.

-Postconditions: + si duoc = TRUE
l'ensemble = l'ensemble U {nouveau arc}
si duoc = FALSE
l'ensemble = l'ensemble.

-Fonction:
+ C'est d'inserer d'un arc dans l'ensemble d'arcs suivant d'un ordre
desire.

5-Procedure DELETEDGE.

-Arguments: + numinit : le nom interne de l'extremite initiale, type integer;
numfin : le nom interne de l'extremite finale, type integer;
ens : l'ensemble de noeuds et d'arcs du graphe, type setvertex,
duoc : une variable qui montre si la supression est elle bien faite,type boolean.

-Preconditions: + numinit et numfin sont corrects au point de vue de syntax
ens existe

-Sorties: + l'arc demande est supprime.

-Postconditions: + si duoc = TRUE
l'ensemble = l'ensemble \ {arc a supprimer}
si duoc = FALSE
l'ensemble = l'ensemble.

-Fonction:
+ C'est de supprimer d'un arc du graphe lorsqu'on donne le nom de ses deux extremites.

6-Procedure VALUE.

-Arguments: + i : le nom interne de l'extremite initiale, type integer;
j : le nom interne de l'extremite finale, type integer;
ens : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + i et j sont corrects au point de vue de syntax
ens existe

-Sorties: + la valeur de poid de l'arc demande.

-Postconditions: + VALUE = le poid de l'arc demande s'il existe
par contre VALUE = 0.

-Fonction:
+ C'est de donner la valeur du poid d'un arc du graphe lorsqu'on donne le nom interne de ses deux extremites.

7-Procedure CHANGEVALUE.

-Arguments: + i : le nom interne de l'extremite initiale, type integer;
j : le nom interne de l'extremite finale, type integer;
ens : l'ensemble de noeuds et d'arcs du graphe, type setvertex,
pds : la valeur du poid que l'on veut remplacer avec sa valeur ancienne,type real.

-Preconditions: + i,j et pds sont corrects au point de vue de syntax
ens existe

-Sorties: + l'ensemble de noeuds et d'arcs dugraphe.

-Postconditions: + l'ensemble = l'ensemble telque
le poid de l'arc demande = pds.

-Fonction:
+ C'est de changer la valeur du poid d'un arc donne.

8-Procedure INSERARCS.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est l'interface de realiser d'une insertion d'un arc.Il s'occupe les taches suivantes:
* demander a l'utilisateur le nom interne des deux extremites de l'arc qu'on veut inserer et son poid (interractif),
* detecter les erreurs syntaxiques et semantiques,
* inserer l'arc dans l'ensemble des arcs du graphe.

9-Procedure DELETARCS.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est l'interface de realiser d'une suppression d'un arc du graphe. Il s'occupe les taches suivantes:
* demander a l'utilisateur le nom interne des deux extremites d'arc voulant supprimer (interractif),
* detecter les erreurs syntaxiques et semantiques,
* supprimer l'arc demande .

10-Procedure CHANGEV.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est l'interface de realiser d'un changement de valeur du poid d'un arc du graphe.Il s'occupe les taches suivantes:
* demander a l'utilisateur le nom interne des deux extremites de l'arc voulant changer de poids et la valeur du nouveau poids (interractif),
* detecter les erreurs syntaxiques et semantiques,
* changer la valeur du poids d'arc demande .

11-Procedure IENTEREDGE.

-Arguments: + stp : une switch qui permet d'aller a l'etat SUP
selon sa valeur,type boolean

-Preconditions: + stp = FALSE.

-Sorties: + la valeur de stp.

-Postconditions: + soit stp = FALSE si on veut rentrer dans l'etat SUP
soit stp = TRUE si on veut rentrer dans l'etat DEC

-Fonction:
+ C'est l'interface de realiser d'un choix d'operation de l'etat EDG.

****

LES PROCEDURES DU MODULE DE GESTION D'ECRAN

****

1-Procedure BLANK.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est d'effacer tout l'ecran et de positionner le curseur au debut a gauche de l'ecran.

2-Procedure HOME.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est de positionner le curseur au debut a gauche de l'ecran.

3-Procedure CLREOL.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est d'effacer une ligne de l'ecran a partir de la position du curseur a ce moment la.

4-Procedure CLREOS.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
C'est d'effacer l'ecran du bas a partir de la position du curseur a ce moment la.

5-Procedure GOTOXY.

-Arguments: + ligne : la position horizontale que l'on veut poser le curseur,type integer;
cologne : la position verticale que l'on veut poser le curseur,type integer.

-Preconditions: + ligne et cologne sont corrects au point vue de syntax.

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est de positionnner le curseur a la position (ligne,cologne) de l'ecran.

6-Procedure WAIT.

-Arguments: -

-Preconditions: -

-Sorties: -

-Postconditions: -

-Fonction:
+ C'est d'arreter le programme dans certain temps et le reprendre apres

7-Procedure SAUTPAGE.

-Arguments: + sohang : le numero du ligne que l'on vient d'afficher sur l'ecran,type integer.

-Preconditions: + sohang est correct au point de vue de syntax.

-Sorties: + sohang

-Postconditions: + 4 < sohang < 20

-Fonction:
+ C'est d'effacer l'ecran a partir de la ligne 5 et de positionner le curseur a (5,1) de l'ecran.

8-Procedure LIRENUM.

-Arguments: + nombre : le nombre que l'on veut lire de l'ecran, type integer;
corr : une variable booleenne qui montre si le nombre vient de lire est il correct,type boolean.

-Preconditions: -

-Sorties: - nombre et corr.

-Postconditions: + si corr = TRUE ,alors nombre = le nombre lu par contre nombre = 0.

-Fonction:
+ C'est de detecter l'erreur au point de vue syntaxique du nombre entier venant de taper sur l'ordinateur.

9-Procedure LIRENUMR.

-Arguments: + nombrer : le nombre que l'on veut lire de l'ecran, type real;
corr : une variable booleenne qui montre si le nombre vient de lire est il correct,type boolean.

-Preconditions: -

-Sorties: - nombrer et corr.

-Postconditions: + si corr = TRUE ,alors nombrer = le nombre reel lu par contre nombrer = 0.

-Fonction:
+ C'est de detecter l'erreur au point de vue syntaxique du nombre reel venant de taper sur l'ordinateur.

****

LES PROCEDURES DU SOUS-MODULE DE ROUTINES DE GRAPHE

****

1-Procedure MAXINT.

-Arguments: + num1 : le premier nombre entier,type integer;
num2 : le deuxieme nombre entier,type integer.

-Preconditions: + num1 et num2 sont corrects au point de vue de syntax.

-Sorties: + MAXINT.

-Postconditions: + MAXINT >= num1 et num2.

-Fonction:
+ C'est de donner le maximun des deux nombres entiers.

2-Procedure MININT.

-Arguments: + num1 : le premier nombre entier,type integer;
num2 : le deuxieme nombre entier,type integer.

-Preconditions: + num1 et num2 sont corrects au point de vue de syntax.

-Sorties: + MININT.

-Postconditions: + MININT <= num1 et num2.

-Fonction:
+ C'est de donner le minimun des deux nombres entiers.

3-Procedure MAXREL.

-Arguments: + numr1 : le premier nombre reel,type real;
numr2 : le deuxieme nombre reel,type real.

-Preconditions: + numr1 et numr2 sont corrects au point de vue de syntax.

-Sorties: + MAXREL.

-Postconditions: + MAXREL >= numr1 et numr2.

-Fonction:
+ C'est de donner le maximun des deux nombres reels.

4-Procedure MINREL.

-Arguments: + numr1 : le premier nombre reel,type real;
numr2 : le deuxieme nombre reel,type real.

-Preconditions: + numr1 et numr2 sont corrects au point de vue de syntax.

-Sorties: + MINREL.

-Postconditions: + MINREL <= numr1 et numr2.

-Fonction:
+ C'est de donner le minimun des deux nombres reels.

5-Procedure ASCENDANTS.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud, type integer.

-Preconditions: + v est correct au point de vue de syntax,
g existe.

-Sorties: + ASCENDANTS, type resulset.

-Postconditions: + ASCENDANTS = { noeuds/ ils sont accessibles a v }

-Fonction:
+ C'est de donner l'ensemble des ascendants du noeud demande v.

6-Procedure DESCENDANTS.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud, type integer.

-Preconditions: + v est correct au point de vue de syntax,
g existe.

-Sorties: + DESCENDANTS, type resulset

-Postconditions: + DESCENDANTS = { noeuds/ ils sont accessibles de v }

-Fonction:
+ C'est de donner l'ensemble des descendants du noeud demande v.

7-Procedure SUC.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud, type integer.

-Preconditions: + v est correct au point de vue de syntax,
g existe.

-Sorties: + SUC, type resulset.

-Postconditions: + SUC = { noeud y / (v,y) est un arc du graphe }

-Fonction:
+ C'est de donner l'ensemble des sucesseurs du noeud demande v.

8-Procedure PRED.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud, type integer.

-Preconditions: + v est correct au point de vue de syntax,
g existe.

-Sorties: + PRED, type resulset.

-Postconditions: + PRED = { noeud y / (y,v) est un arc du graphe }

-Fonction:
+ C'est de donner l'ensemble des predescesseurs du noeud demande v.

9-Procedure COMP_CONNEXE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud, type integer.

-Preconditions: + v est correct au point de vue de syntax,
g existe.

-Sorties: + COMP_CONNEXE, type resulset.

-Postconditions: + COMP_CONNEXE = { noeud y / il existe un chemin de v a y }

-Fonction:
+ C'est de donner le composant connexe correspondant au noeud demande v

10-Procedure COMP_FORT_CONNEXE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud, type integer.

-Preconditions: + v est correct au point de vue de syntax,
g existe.

-Sorties: + COMP_FORT_CONNEXE, type resulset.

-Postconditions: + COMP_FORT-CONNEXE =
{ noeud y / il existe un chemin de v a y
et un chemin de y a v }

-Fonction:
+ C'est de donner le composant fortement connexe correspondant au noeud demande v.

11-Procedure ISOLE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + ISOLE, type resulset.

-Postconditions: + ISOLE = { noeud x / degree de x = 0 }

-Fonction:
+ C'est de donner l'ensemble de noeuds isoles du graphe.

12-Procedure COMPLET.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + COMPLET, type boolean.

-Postconditions: + COMPLET = TRUE si le graphe est complet, par contre COMPLET = FALSE.

-Fonction:
+ C'est de detecter si le graphe est il complet.

13-Procedure CONNEXE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + CONNEXE, type boolean.

-Postconditions: + CONNEXE = TRUE si le graphe est connexe, par contre CONNNEXE = FALSE.

-Fonction:
+ C'est de detecter si le graphe est il connexe.

14-Procedure FORT_CONNEXE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + FORT_CONNEXE, type boolean.

-Postconditions: + FORT_CONNEXE = TRUE si le graphe est fortement connexe, par contre FORT_CONNNEXE = FALSE.

-Fonction:
+ C'est de detecter si le graphe est il fortement connexe.

15-Procedure ANTI_SYMETRIQUE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + ANTI_SYMETRIQUE,type boolean.

-Postconditions: + ANTI_SYMETRIQUE = TRUE si le graphe est anti-symetrique, par contre ANTI_SYMETRIQUE = FALSE.

-Fonction:
+ C'est de detecter si le graphe est il anti-symetrique.

16-Procedure SYMETRIQUE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + SYMETRIQUE,type boolean.

-Postconditions: + SYMETRIQUE = TRUE si le graphe est symetrique, par contre SYMETRIQUE = FALSE.

-Fonction:
+ C'est de detecter si le graphe est il symetrique.

19-Procedure EXISTE_BOUCLE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
sv : l'ensemble de noeuds ayant un boucle, tyep resulset.

-Preconditions: + g existe.

-Sorties: + EXISTE_BOUCLE,type boolean et sv.

-Postconditions: + si EXISTE_BOUCLE = TRUE ,alors sv <> { }
si non sv = { }.

-Fonction:
+ C'est de checher tous les noeuds du graphe ayant un boucle.

20-Procedure EXISTE_ARBRE.

-Arguments: + g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + EXISTE_ARBRE,type boolean.

-Postconditions: + EXISTE_ARBRE = TRUE si le graphe est un arbre
par contre EXISTE_ARBRE = FALSE.

-Fonction:
+ C'est de detecter si le graphe est il un arbre.

21-Procedure DMOINS.

-Arguments: - g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud qu'on veut savoir son degre moins.

-Preconditions: + v est correct au point de vue de syntax, g existe.

-Sorties: + DMOINS,type integer.

-Postconditions: + DMOINS = le degre moins de v.

-Fonction:
+ C'est de donner le degre moins d'un noeud donne v.

22-Procedure DPLUS.

-Arguments: - g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud qu'on veut savoir son degre moins.

-Preconditions: + v est correct au point de vue de syntax, g existe.

-Sorties: + DPLUS,type integer.

-Postconditions: + DPLUS = le degre plus de v.

-Fonction:
+ C'est de donner le degre plus d'un noeud donne v.

23-Procedure DEGRE.

-Arguments: - g : l'ensemble de noeuds et d'arcs du graphe, type setvertex;
v : le nom interne du noeud qu'on veut savoir son degre moins.

-Preconditions: + v est correct au point de vue de syntax, g existe.

-Sorties: + DEGRE,type integer.

-Postconditions: + DEGRE = le degre de v.

-Fonction:
+ C'est de donner le degre d'un noeud donne v.

24-Procedure ORDRE.

-Arguments: - g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + ORDRE,type integer.

-Postconditions: + ORDRE = le nombre de noeuds du graphe.

-Fonction:
+ C'est de donner le nombre de noeuds d'un graphe donne g.

25-Procedure COMPLEMENTAIRE.

-Arguments: - g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe.

-Sorties: + g .

-Postconditions: + g = le complement de l'ancien graphe g.

-Fonction:
+ C'est de trouver le complement de l'ancien graphe g.

26-Procedure TRANSFORM_NON_ORIENT.

-Arguments: - g : l'ensemble de noeuds et d'arcs du graphe, type setvertex.

-Preconditions: + g existe et oriente.

-Sorties: + g .

-Postconditions: + g = le graphe non oriente correspondant a l'ancien graphe g.

-Fonction:
+ C'est de transformer l'ancien graphe g de devenir un graphe non oriente.

27-Procedure CREASTA.

-Arguments: + listev : liste de noeuds sous la forme d'un stack, type stackver.

-Preconditions: + listev inexistant.

-Sorties: + listev.

-Postconditions: + listev est cree mais vide.

-Fonction:
+ C'est de creer une liste vide de noeuds.

28-Procedure PUSHDOWN.

-Arguments: + listev : liste de noeuds sous la forme d'un stack, type stackver.
v : le nom interne du noeud que l'on veut mettre dans le stack, type integer.

-Preconditions: + v est correct au point de vue de syntax, listev existe.

-Sorties: + listev.

-Postconditions: + listev =listev + [v].

-Fonction:
+ C'est d'ajouter un noeud au stack.

29-Procedure POPTOP.

-Arguments: + listev : liste de noeuds sous la forme d'un stack, type stackver.
v : le nom interne du noeud que l'on veut mettre dans le stack,type integer.

-Preconditions: + v est correct au point de vue de syntax, listev existe.

-Sorties: + listev et POPTOP.

-Postconditions: + listev =listev - [v] et POPTOP = v.

-Fonction:
+ C'est de supprimer un noeud du stack et le mettre dans POPTOP.

30-Procedure EMPTYSTA.

-Arguments: + listev : liste de noeuds sous la forme d'un stack, type stackver.

-Preconditions: + listev existe.

-Sorties: + EMPTYSTA,type boolean.

-Postconditions: + EMPTYSTA = TRUE si listev est vide, par contre EMPTYSTA = FALSE.

-Fonction:
+ C'est de detecter si le stack est il vide.