UNIVERSITE DE NANCY I

U. E. R. DE MATHEMATIQUES

gestion statique de mémoire dans un système de programmation modulaire

thèse

pour l'obtention
du titre de
docteur ingénieur

soutenue le 10 mars 1973

par

jacques dendien

jury :

M. C.	PAIR	Président
M. M.	DEPAIX	Examinateurs
M. J. C.	DERNIAME	

UNIVERSITE DE NANCY I
U. E. R. DE MATHEMATIQUES

gestion statique de mémoire dans un système de programmation modulaire

thèse

pour l'obtention
du titre de
docteur ingénieur

soutenue le 10 mars 1973

par

jacques dendien

jury :

M. C.	PAIR	Président
M. M.	DEPAIX	Examinateurs
M. J. C.	DERNIAME	

J'exprime ma reconnaissance à Monsieur PAIR, Directeur de l'Institut Universitaire de Calcul Automatique, pour l'intérêt qu'il a montré pour ce travail, et pour l'honneur qu'il me fait en présidant le Jury.

Je remercie vivement Monsieur DEPAIX qui a bien voulu accepter de participer au Jury.

Mes remerciements vont aussi à Monsieur DERNIAME, qui assure la direction du projet CIVA dans lequel s'intégre ce travail, mené à terme grâce aux conseils précieux qu'il m'a prodigués et aux nombreuses idées dont il m'a fait part.

J'exprime toute ma gratitude à Monsieur FIEGEL, pour l'efficacité et la rapidité avec lesquelles il a réalisé les travaux de programmation.

Enfin, je remercie Madame MARCHAND et Mademoiselle LANG, qui ont apporté tout leur soin à la réalisation effective de cette thèse.

INTRODUCTION

I - APERCU DU ROLE DU PRE-EDITEUR DE LIENS

II - OPTIONS FONDAMENTALES

APERCU DU ROLE DU PRE-EDITEUR DE LIENS

L'étude présentée s'inclut dans le projet CIVA. Le but de ce projet est de fournir un ensemble d'outils permettant de passer, sans heurt de l'analyse à l'exploitation d'une chaîne de traitement, donc essentiellement, de définir un langage unique permettant de décrire à la fois les résultats de l'analyse, les programmes et leur exploitation.

La description des actions est modulaire, celle des informations également. L'unité de description d'actions s'appelle module. L'unité de description d'informations s'appelle classe.

Ces unités sont rédigées et compilées, dans une large mesure, indépendamment les unes des autres, ce qui nécessite, avant l'exécution, des travaux préliminaires assurant les liaisons entre les différentes unités.

Nous nous plaçons dans le cas d'applications importantes et répétitives. Dans ces conditions, nous cherchons avant tout à assurer une exécution rapide, peut-être au prix de traitements relativement longs pour assurer le passage à l'exécution.

Un aspect important de ce travail de préparation est la prévision de l'implantation en mémoire des variables. Un second aspect également important, est la prévision de l'implantation des textes objets et la mise en oeuvre de techniques de recouvrement.

Ce travail de préparation s'insère entre la compilation et l'édition de liens. Le compilateur, pour chaque classe ou module, produit un texte objet dans lequel certaines adresses sont définies par rapport à des références externes au programme compilé. La définition deces références, et la résolution définitive des adresses, ne peut se faire que grâce à un éditeur de liens, programme chargé de faire un tout cohérent de textes objets épars.

Habituellement, un éditeur de liens implante les textes objets de manière consécutive. A La rigueur des systèmes d'exploitation plus évolués sont dotés d'un éditeur de liens capable de traiter une structure de recouvrement donnée explicitement par le biais de cartes de commande. Il n'y a jamais de recherche automatique d'une structure de recouvrement visant à minimiser la taille du programme exécutable produit par le compilateur ; la responsabilité d'une éventuelle optimisation échoit

en fin de compte au programmeur. Ce dernier ne peut intervenir que dans des cas simples, mettant en jeu des possibilités de recouvrement assez pauvres. De plus l'effort supplémentaire ainsi demandé au programmeur, le distrait de sa tâche principale et exige de sa part une connaissance particulière du système d'exploitation la machine qu'il utilise (particularités de l'éditeur de liens et du langage de commande).

C'est pourquoi il semble intéressant d'automatiser ce travail. Il faut dégager la responsabilité du programmeur dans la préparation de l'édition de liens et déterminer une implantation satisfaisante des informations, de manière à minimiser l'encombrement de la mémoire centrale.

Il est clair que ces fonctions ne sauraient être confiées au compilateur dans le cas d'un langage de programmation modulaire. En effet, les compilations des différentes unités sont séparées. Le compilateur ne connaît que l'unité qu'il est en train de traiter : il agit à un niveau microscopique. Au contraire, la tâche que nous proposons d'effectuer se situe à un niveau macroscopique. Une vue d'ensemble de toutes les unités constituant un tout exécutable est nécessaire.

Cette tâche doit donc être confiée à un programme spécifique servant de trait d'union entre le compilateur et un éditeur de liens classique. Nous appellerons ce programme un pré-éditeur de liens.

CIVA est un langage de programmation modulaire offrant des possibilités de recouvrement extrêmement riches, tant entre les informations variables qu'entre les textes objets. Nous étudierons donc les possibilités offertes par un pré-éditeur de liens pour chacun de ces deux types d'informations.

II - OPTIONS FONDAMENTALES

Le langage n'admet pas d'appels récursifs de procédures. Comme nous bornons notre étude à l'implantation des textes objets et des variables de taille fixe, qui sont des informations dont l'encombrement est connu à la fin de la compilation, il semble opportun, moyennant l'hypothèse de non récursivité, d'étudier les possibilités d'implantation statique.

Cette option est justifiée par le fait que nous nous intéressons à des applications répétitives, pour lesquelles une exécution rapide est requise. On sait bien que les méthodes de gestion dynamique de mémoire, dont le mérite est de conduire à un encombrement instantané minimum de la mémoire, amènent en contrepartie un ralentissement de l'exécution. Ce ralentissement est dû à la mise en oeuvre des algorithmes, souvent complexes, qu'elles nécessitent, et par la lenteur des accès aux informations qui ne peut plus se faire de manière directe, mais par des mécanismes d'adressage indirect ou d'indexation.

Une gestion statique de mémoire balaie totalement ces inconvénients dans la mesure où l'on garantit un accès par adressage direct aux informations. Elle n'est cependant possible que pour des objets dont la taille est fixe pendant l'exécution et connue à la compilation.

En revanche, s'il est vrai qu'une gestion dynamique "ne coûte que si l'on s'en sert", la réalisation d'une implantation statique est un investissement à priori. C'est pourquoi, l'hypothèse d'applications répétitives est très importante, car elle permet d'espérer un amortissement rapide du coût des traitements préliminaires, d'autant plus longs que l'on désire une gestion de mémoire plus fine.

Il est donc intéressant de prévoir une méthode de gestion statique graduelle, et non pas une optimisation à outrance, peut être inutile, parfois néfaste.

Dans cette optique, nous pouvons déjà discerner deux grandes familles d'applications suivant que l'on utilise ou non une mémoire secondaire.

Les techniques de recouvrement peuvent conduire à l'utilisation d'une mémoire secondaire, nécessaire pour stocker temporairement des informations qui ne résident pas en permanence en mémoire centrale. Pour mettre en oeuvre un recouvrement sur des informations variables, qui naissent ou meurent au cours de l'exécution, l'emploi d'une mémoire secondaire peut être évité : il suffit de conserver, en mémoire centrale une représentation de ces informations pendant toute leur durée de vie. Lorsque la durée de vie d'une information cessera, la place occupée par sa représentation deviendra vacante, et récupérable pour la représentation d'une autre information naissant ultérieurement.

Par contre, une telle méthode n'est pas applicable pour les informations constantes : si leur représentation cesse d'exister en mémoire centrale à un mo donné de l'exécution, il faut bien que leur valeur soit retrouvée sur une mémoi secondaire lors d'une utilisation ultérieure.

Le fait qu'il s'agit d'informations constantes rend utile des opérations de sauvegarde depuis la mémoire centrale vers la mémoire secondaire : il suffit de disposer sur cette dernière d'une version originale des constantes.

Il suffira, lorsque la présence en mémoire centrale d'une constante devienc nécessaire, de tirer une copie de la version originale.

Nous limiterons notre étude à ces deux techniques de recouvrement : sur les variables, sans emploi de mémoire secondaire, et sur les constantes avec des échanges à sens unique, depuis la mémoire secondaire vers la mémoire centrale.

Nous écarterons des solutions mettant en jeu des échanges dans les deux sens (nécessaires lorsqu'une information variable est chassée de la mémoire centrale avant expiration de sa durée de vie). De telles solutions amènent en effet à des échanges fréquents entre mémoire principale et mémoire secondaire, et ralentiss les exécutions de manière considérable.

Si nous considérons un programme exécutable CIVA (nous dirons aussi unité de traitement) tel que le recouvrement mis en oeuvre sans l'aide de mémoire seconda amène à un encombrement acceptable de la mémoire centrale, il serait ridicule, même nuisible, de pousser plus avant la réduction de l'encombrement : on mettrai ainsi en oeuvre une mémoire secondaire dont l'emploi pourrait être évité.

Le but du pré-éditeur de liens n'est donc pas à partir de textes objets donn de fournir un encombrement aussi restreint que possible, mais plutôt à partir d' taille de mémoire donnée, de fournir une structure de recouvrement compatible av cette taille, sans traitement excessif.

Une méthode raisonnable de gestion statique de mémoire doit donc tenir compt du fait que l'utilisation de la mémoire secondaire ne doit rester qu'un pis-alle Ceci peut sembler mener à une disjonction totale entre les traitements relatifs

textes objets. Une telle disjonction impliquerait la mise en oeuvre d'algorithmes différents, spécifiques aux deux cas, et, par conséquent, un redoublement du coût des traitements.

Il serait donc particulièrement agréable de disposer d'une méthode d'implantation cohérente, qui puisse prendre en charge les deux cas apparemment disjoints. Cette cohérence ne peut être atteinte que si nous dégageons une parenté entre les accès aux variables et aux textes objets. Cette parenté, comme nous le verrons, découle de la structure même d'une unité de traitement. Cette structure est définie par les liens et les interactions existant entre modules et classes.

En résumé, nous chercherons d'abord à montrer qu'il est possible d'adopter une méthode de gestion statique. Ainsi, les informations seront atteintes par adressage direct ce qui garantira des gains de temps appréciables à l'exécution.

D'autre part, nous chercherons une méthode d'implantation graduée, ne conduisant pas à des optimisations excessives. Enfin, les algorithmes mis en oeuvre doivent résoudre aussi bien les problèmes soulevés par l'implantation des variables que ceux soulevés par l'implantation des textes objets.

Puisque nous nous intéressons aux possibilités de recouvrement entre variables et textes objets, il s'impose pour régler le cas des variables, d'étudier les problèmes de portée des identificateurs de variables, de durée d'existence des représentations des variables en mémoire centrale. Quant aux textes objets, il sera nécessaire pour dégager les possibilités de recouvrement, d'étudier les liaisons pouvant exister dans une même unité de traitement, entre tous les textes engendrés par le compilateur.

Le module est l'unité d'action, la liaison fondamentale entre des modules est l'ordre d'appel de module à un autre.

La classe est l'unité de description d'informations. Un module ou une classe, seront liés à une autre classe par le pouvoir qu'ils auront d'accéder aux informations décrites dans cette classe. Ce pouvoir est déterminé par l'emploi d'une

directive que nous allons étudier. C'est la directive "utilise".

Nous étudierons l'unité de traitement d'un point de vue macroscopique. En examinant sa structure, nous verrons que les données de base de notre problème sont fournies par l'examen des déclarations d'identificateurs, des ordres d'appel d'un module à un autre, et l'emploi de la directive "utilise".

1ère PARTIE

RELATIONS ENTRE MODULES ET CLASSES

1 - DECLARATIONS.

2 - DIRECTIVE "Utilise".

3 - APPELS DE MODULES.

4 - DOMAINE DE VALIDITE D'UN IDENTIFICATEUR.
- 4. 1 Définition
- 4. 2 Domaine de validité d'un identificateur en fonction de sa déclaration
- 4. 3 Echanges d'informations entre les modules

5 - INFORMATIONS DEVANT FIGURER SIMULTANEMENT EN MEMOIRE CENTRALE A UN MOMENT DONNE DE L'EXECUTION.
- 5. 1 Informations variables
- 5. 2 Informations constantes

1ère PARTIE

RELATIONS ENTRE MODULES ET CLASSES

Nous nous proposons d'étudier la gestion de la mémoire au niveau de l'unité de traitement qui constitue, au sein d'une application complète, un programme exécutable.

Une unité de traitement est un ensemble de modules et de classes.

Nous allons récapituler les liaisons dans le cadre strict de notre problème. Dans la deuxième partie, les résultats qualitatifs obtenus ici seront traduits de manière formelle.

1) Déclarations

Les identificateurs font l'objet de déclarations soit :

a) Dans le texte d'un module

La portée de tels identificateurs est réduite au module dans lequel ils sont déclarés. Les informations désignées par ces identificateurs sont utilisables pendant toute l'exécution du module.

b) Dans le texte d'une classe

L'emploi de tels identificateurs dans des modules ou dans des classes est alors régi par l'emploi de la directive "utilise".

2) Directive "Utilise"

a) Emploi de la directive dans le texte d'un module

Un module x peut comporter des directives "utilise" portant sur des

noms de classes (par exemple si y et z sont des classes, on pourra éc
dans un texte source : x utilise y, z

Ceci permet d'utiliser les identificateurs de ces classes dans le module x. identificateurs de ces classes désignent toujours la même information tant que x en cours d'exécution.

b) Emploi de la directive "utilise" dans le texte d'une classe

Une classe peut comporter des directives "utilise" portant sur le nom d'autres classes.

Supposons qu'une classe a comporte une directive "utilise" portant une classe b. Alors les identificateurs de b sont assimilés à ceux d en a, en ce sens que si un module x "utilise" la classe a, les identificateurs de b sont utilisables explicitement dans a et x au même tit que ceux de la classe a. En outre, les identificateurs de la classe b désignent toujours la même information tant que x est en cours d'exé

De manière générale, si nous considérons une suite de classes a_0, a_1....., telles que a_i "utilise a_i + 1 pour i allant de o à n-1, les identificateurs décla dans une classe a_i de cette suite sont utilisables dans tout module qui "utilise" classe a_0, et dans toute classe d'indice inférieur à i dans cette suite.

Autrement dit l'occurrence (1) dans une classe d'indice inférieur à i ou d module x, d'un identificateur déclaré dans a_i sera valide, puisque le contexte de occurrence contiendra la classe a_i.

(1) Cf. Introduction générale pour les notions d'occurrence et de contexte.

3) Appels de modules

Un module x peut appeler un module y. On peut considérer y comme un sous-programme fermé externe de x, au sens habituel.

4) Domaine de validité d'un identificateur

4. 1 Définition

L'ensemble des règles précédentes peut se résumer en définissant le domaine de validité d'un identificateur, notion déduite de celle de contexte. Le domaine de validité d'un identificateur est défini comme l'ensemble des modules ou classes dans lesquels cet identificateur peut être cité explicitement. Le domaine de validité d'un identificateur v sera noté D (v).

Soit un module x qui "utilise" une classe y dans laquelle est déclarée un identificateur v. Dire que x appartient au domaine de validité de v équivaut à dire que l'occurrence de v dans x est valide, ou que le contexte de l'occurrence de v dans x contient y. La notion de domaine de validité apparaît donc comme la réciproque de la notion de contexte. Cette dernière permet d'établir la connexion entre l'occurrence d'un identificateur dans un texte objet et sa déclaration, de manière pratique pour la compilation.

Au contraire, la notion de domaine de validité permet, partant de la déclaration d'un identificateur de définir l'ensemble des modules et des classes ayant accès à l'information que cet identificateur désigne ; ce point de vue est plus adapté à la pré-édition de liens car la durée de vie d'une information dépend évidemment de la durée des accès à cette information, et de cette durée de vie dépend en fin de compte l'allocation en mémoire centrale.

4. 2. Domaine de validité d'un identificateur en fonction de sa déclaration.

a) Le domaine de validité d'un identificateur déclaré dans un

module est ce module.

b) Le domaine de validité d'un identificateur déclaré dans une classe a comprend :

a) cette classe elle-même ;

b) tout module x (ou toute classe b) tel qu'il existe une suite v vide ou non de classes satisfaisant à

x (ou b) "utilise a_0

a_i "utilise" a_{i+1} $\quad i \in [0, n-1]$

a_n "utilise" a

4. 3. Echanges d'informations entre les modules

Un identificateur doit toujours désigner la même information pendant l'exécution d'un module appartenant à son domaine de validité.

Deux modules x et y appartenant au domaine de validité d'un même identificateur pourront donc accéder à la même information par l'intermédiaire de cet identificateur, si leurs durées d'exécutions ne sont pas disjointes.

Cette condition est remplie si et seulement s'il existe une suite vide ou non $x_0, x_1, \ldots\ldots, x_n$ de modules telle que

x appelle x_0

x_i appelle x_{i+1} $\quad i \in [0, n-1]$

x_n appelle y

Puisque les identificateurs déclarés dans une même classe ont le même domaine de validité, on peut donc dire que les modules se communiquent des informations par l'intermédiaire de classes

Par exemple, deux modules x et y, tels que x appelle y, "utilisent" la même classe a, pourront correspondre par l'intermédiaire de n'importe quel identificateur de a. Puisque deux modules ne peuvent correspondre par l'intermédiaire de leurs variables locales, les classes apparaissent donc essentiellement comme les moyens de communication.

5) Informations devant figurer simultanément en mémoire centrale à un moment donné de l'exécution

5. 1 Informations variables

Indépendamment de toute méthode de gestion de mémoire, un certain ensemble d'informations doit être présent en mémoire centrale à un moment donné de l'éxécution. Comme nous venons de voir que la présence d'une information désignée par un identificateur dépendait du fait qu'un module de son domaine de validité était ou non en cours d'exécution, nous pouvons en déduire cet ensemble.

Considérons un identificateur V et l'ensemble M_V des modules appartenant à son domaine de validité D (v).

v désigne une information qui doit être représentée en mémoire pendant l'exécution de tout module de M_V. Il est possible, lors de l'exécution, que cette représentation disparaisse dès que cesse l'activité de tout module de M_0, et réapparaisse à la suite de la réactivation d'un module de M_V. Le premier module de M_V qui est réactivé doit alors procéder d'une réinitialisation de l'information (par définition du langage) puisque toute représentation de l'information avait cessé d'exister. Il suffit donc à ce moment de disposer à nouveau l'espace mémoire pour la représentation de v, sans se soucier des valeurs que v désignait précédemment.

(Ce phénomène est tout à fait analogue aux entrées et sorties de blocs en Algol 60. Si un bloc B comporte une déclaration d'une variable V, V doit avoir une représentation en mémoire centrale pendant l'exécution d'instructions du bloc. Lorsqu'on sort du bloc, la représentation de V cesse d'exister. Lors d'une nouvelle entrée dans B, la variable V devra être réinitialisée avant tout emploi).

L'espace nécessaire à la ré-implantation de V, dans une méthode de gestion dynamique, devra être déterminé à cet instant. Dans une méthode de gestion statique il devra simplement se trouver disponible, c'est-à-dire non occupé par la représentation d'une autre variable.

De plus, la réapparition d'une représentation d'une variable n'implique absolument pas la restauration d'une ancienne valeur depuis une mémoire secondaire, puisque la première instruction référençant cette variable, doit la réinitialiser.

Considérons maintenant l'ensemble X des modules en cours d'exécution à un moment donné. Toute information désignée par un identificateur v tel que D (v) $\cap$ X soit différent de l'ensemble vide, doit avoir une représentation en mémoire centrale à cet instant. En effet, il existe alors un module en cours d'exécution appartenant au domaine de validité de v.

Supposons l'ensemble X invariant pendant un certain temps .

Pendant ce temps, nous devons faire co-habiter en mémoire centrale les représentations de toutes les informations désignées par un identificateur v ayant la propriété précédente. Plus tard, si l'ensemble X s'enrichit, il nous suffira de disposer d'espace mémoire pour l'implantation des représentations des nouvelles informations apparues. En respectant la règle de co-habitation précédente, nous garantissons la présence effective en mémoire centrale des informations, à tout moment où un module risque d'y faire accès. Nous sommes d'autre part assurés, comme il a été vu précédemment, de leur réinitialisation par les nouveaux modules dont X s'est enrichi.

Il est donc possible, en respectant cette règle, d'éviter systématiquement l'emploi d'une mémoire secondaire.

Par contre, en l'enfreignant, nous sommes amenés à chasser de la mémoire centrale des informations dont la durée de vie n'est pas achevée (c'est-à-dire des informations auxquelles les accès pourront encore être faits). Il faudrait alors les sauvegarder sur une mémoire secondaire, afin de les restaurer en mémoire centrale au prochain accès qui serait fait à ces informations.

5. 2 Informations constantes

A l'exécution, les textes objets des modules et des classes peuvent être implantés consécutivement en mémoire centrale, sans qu'on se soucie de minimiser leur encombrement.

Pour les unités de traitement importantes, il se peut qu'une telle solution ne soit pas suffisante. Il est clair que l'usage d'une mémoire secondaire ne peut plus être évité.

Le problème qui se pose alors est celui des pertes de temps dans les échanges entre mémoire centrale et mémoire secondaire. L'importance des pertes de temps est évidemment liée à la richesse du contenu de la mémoire centrale : plus ce contenu sera important moins les appels à la mémoire secondaire seront fréquents. La mémoire centrale doit être utilisée au mieux.

Il est indispensable de définir une règle de co-habitation des textes objets en mémoire, de même que nous venons de définir une règle de co-habitation pour les variables. Cependant, si la règle de co-habitation que nous avons trouvée pour les variables s'imposait simplement à l'esprit, il n'en est pas de même pour les textes objets : le critère, pour les variables, était d'éviter l'emploi d'une mémoire secondaire. Pour les textes objets, puisque la mémoire secondaire devient nécessaire, l'élaboration d'une règle de co-habitation devient beaucoup moins nette. Le critère est cette fois de réduire les pertes de temps dues aux échanges entre les mémoires tout en réduisant l'encombrement de la mémoire centrale. Pour satisfaire ce critère, deux attitudes peuvent être adoptées.

La première consiste à réaliser des mesures pendant l'exécution d'une unité de traitement, puis d'en tirer les informations utiles guidant le choix d'une structure de recouvrement pour une exécution ultérieure. Cette méthode ne peut être appliquée que pour des unités de traitement dont l'écriture est définitivement figée, et, qui, de plus, ont un comportement particulièrement constant à l'exécution.

La seconde attitude consiste à choisir à priori une structure de recouvrement qu'on pense raisonnable, suivant une méthode systématique. Une telle démarche donnera évidemment des résultats moins bien adaptés à chaque cas particulier qu'une méthode de mesures. Par contre, elle sera sûrement beaucoup moins coûteuse.

De plus, on pourra disposer immédiatement, dès la première exécution, d'une st de recouvrement qui serait de toute façon indispensable en tant que première é au cas où on désirerait appliquer une méthode de mesures.

Si on considère le déroulement d'une exécution, il est clair que le au sein des textes objets seront d'autant moins aléatoires que l'unité de trait sera fortement structurés.

Dans le cas d'une unité de traitement CIVA, ces accès dépendront étr tement des possibilités d'appels entre modules, ainsi que de l'emploi de la dir "utilise". Les possibilités d'appels sont parfaitement connues dès la compilati (en effet, nous écartons la récursivité dans les appels). De même l'emploi de l directive "utilise" est parfaitement délimité.

Une unité de traitement CIVA est fortement structurée, et sa structu bien connue avant le début de l'éxécution. Cette structuration oriente notre ch vers des méthodes de recouvrement s'appuyant sur des critères à priori. Grâce connaissance complète de la structure de l'unité de traitement, on peut en effe espérer trouver des critères ayant une efficacité satisfaisante.

D'un autre point de vue, si nous renonçons à des méthodes de mesures l'implantation des textes objets, il n'en serait pas moins souhaitable que les algorithmes déterminant une structure de recouvrement à partir de critères à pri soient compatibles avec une méthode de mesures pouvant faire l'objet d'une réali ultérieure. Pour cela nous nous efforcerons d'exprimer d'une manière simple les contraintes de co-habitation des textes objets fournies par l'application des cr à priori. Dès lors, des mesures pourront fournir des contraintes s'exprimant sou même forme, et les algorithmes d'allocation seront réutilisables. Une manière si d'exprimer ces contraintes est de les représenter par un graphe de disjonction. la méthode que nous utiliserons.

Nous n'énoncerons pas pour le moment les critères de recouvrement pou textes objets. Nous verrons qu'ils découlent naturellement de l'étude de la co-h tation des représentations des variables, à cause de la forte structuration des unit traitement.

Pour les unités de traitement dont l'encombrement n'est pas excessif, souhaitable, pour des raisons de rapidité à l'exécution, de ne pas pousser le recouvrement au niveau des textes objets. Cette situation représente un cas courant pour des machines ayant une configuration assez importante.

Nous considérons donc comme primordiale l'étude du recouvrement sur les variables.

Le recouvrement sur les textes objets sera étudié ensuite de manière corollaire.

2ème PARTIE

CHOIX D'UNE METHODE DE GESTION DE MEMOIRE POUR LES VARIABLES

1 - FORMALISATION DE LA NOTION DE DOMAINE DE VALIDITE

- 1. 1 - GRAPHE (M, A) DES APPELS DE MODULES
- 1. 2 - GRAPHE DES DIRECTIVES "UTILISE"
- 1. 3 - DOMAINE DE VALIDITE

2 - CO-EXISTENCE EN MEMOIRE CENTRALE DES REPRESENTATIONS DE VARIABLES

3 - ILLUSTRATION PAR UN EXEMPLE DES NOTIONS PRECEDENTES

4 - EVOLUTION DE L'ETAT DE LA MEMOIRE CENTRALE PENDANT L EXECUTION

5 - CONSEQUENCES D'UNE GESTION DYNAMIQUE

6 - CONSEQUENCES D'UNE GESTION STATIQUE

- 6. 1 - CAS OU (M, A) EST UNE ARBORESCENCE
- 6. 2 - CAS OU (M, A) N'EST PAS UNE ARBORESCENCE
  - 6. 2. 1 - Essai d'adaptation de la méthode précédente
  - 6. 2. 2 - Unicité de la représentation d'une variable.

7 - COMPARAISON DES CONSEQUENCES DE GESTIONS STATIQUE OU DYNAMIQUE

2ème PARTIE

CHOIX D'UNE METHODE DE GESTION DE MEMOIRE POUR LES VARIABLES

Nous nous proposons de formaliser les résultats obtenus dans la première partie, puis d'étudier les conséquences de l'adoption d'une méthode de gestion statique ou dynamique de mémoire. Nous nous intéressons uniquement aux variables dont la taille est fixe pendant l'exécution et connue à la compilation, car pour elles seules une gestion statique est envisageable. Les résultats des cinq premiers paragraphes s'appliquent néanmoins à tous les types de variable-.

- FORMALISATION DE LA NOTION DE DOMAINE DE VALIDITE

Nous avons vu que la nécessité de la présence en mémoire centrale de la représentation d'une variable, à un moment donné de l'exécution était fonction du domaine de validité de l'identificateur de la variable.

La notion de domaine de validité jouant un rôle essentiel pour la gestion de la mémoire, nous nous proposons de la formaliser.

1. 1 - GRAPHE DES APPELS DE MODULES

Soit M l'ensemble des modules mis en jeu dans une unité de traitement.

Sur M on peut définir une relation binaire A :

x et y étant deux modules de M, nous dirons que x A y si et seulement si le texte source de x comporte un ordre d'appel, conditionnel ou non, au module y.

Nous obtenons ainsi un graphe (M, A) représentant les appels possibles entre modules à l'exécution.

Par définition du langage, le graphe (M, A) doit être sans circuit.

Cette condition nous garantit la non récursivité dans les appels de modules.

De plus, le graphe (M, A) doit avoir un point d'entrée unique appelé directeur. C'est par ce module que débute toute exécution.

1. 2 Graphe des directives "utilise"

Soit C l'ensemble des classes mises en jeu dans une unité de traitemer

Posons $E = M \cup C$

Sur E, on peut définir une relation binaire U :

Soient a et b deux éléments de E, nous dirons que $a U b$ si et seulemen texte source de a comporte une directive "utilise" portant sur b.

Nous obtenons ainsi un graphe (E, U)

Par définition du langage, le graphe (E, U) est sans circuit.

D'autre part, une directive "utilise" ne pouvant porter que sur une cl il s'ensuit que l'ensemble des points d'entrée de (E, U) contient M. De plus, classe d'une unité de traitement doit en principe faire l'objet d'une directiv "utilise" dans une autre classe ou dans un module. L'ensemble des points d'ent (E, U) est donc exactement M.

1. 3 Domaine de validité

Notons U^* la fermeture transitive de U au sens large :

$$U^* = \bigcup_{n=0} U^n$$

a) Nous avons vu que si un identificateur v est déclaré dans un modul alors D (v) = x.

b) D'autre part, si v est déclaré dans une classe x , D(v) comprend la classe et toutes celles qui sont reliées à la classe par une suite de directive "utilise". Nous pouvons donc écrire que :

$$a \in D(v) \Longleftrightarrow a U^* x \qquad a \in (U^*)^{-1}(x)$$

L'ensemble des deux conditions a et b peut se résumer en une règle unique : le domaine de validité d'un identificateur v déclaré dans une classe ou un module x, est $(U^*)^{-1}(x)$.

$(U^*)^{-1}(x)$ désigne, suivant une notation classique, le sous-ensemble de E tel que :

$$a \in (U^*)^{-1}(x) \Longleftrightarrow a U^* x$$

2 - CO-EXISTENCE EN MEMOIRE DES REPRESENTATIONS DE VARIABLES

Cas a

Si un module est en cours d'exécution, l'ensemble des variables auxquelles il a accès, doit avoir une représentation en mémoire centrale.

Soit donc x un module en cours d'exécution. Il a accès aux variables v telles que $x \in D(v)$. Si v est déclaré dans une classe ou module a, :
$x \in (U^*)^{-1} a$ (par définition du domaine de validité)

Par conséquent les variables qui doivent avoir une représentation en mémoire pendant l'exécution de x sont celles déclarées dans un élément de $U^*(x)$.

Cas b

Si deux modules x et y sont simultanément en cours d'exécution, les variables de $U^*(x)$ et $U^*(y)$ doivent avoir une représentation en mémoire.

Deux modules peuvent être simultanément en cours d'exécution que s'ils sont sur un même chemin du graphe des appels (M, A).

Notons $A = \overset{+\infty}{\underset{m=1}{\cup}} A^m$. (fermeture transitive stricte).
x et y sont tels que $x \hat{A} y$ ou $y \hat{A} x$

En résumé, les représentations de deux variables v_1 et v_2 doivent co exister.

Dans le cas a : si v_1 et v_2 sont déclarées dans un même élément de U^*

Dans le cas b : si v_1 (resp v_2) est déclaré dans $U^*(x)$ et si v_2 (res est déclaré dans $U^*(y)$.

L'ensemble de ces deux cas peut se réunir en une seule règle, en not

$$A^* = \overset{+\infty}{\underset{n=0}{\cup}} A^n$$

Règle : Les représentations de deux variables v_1 et v_2, déc respectivement dans des classes ou modules U_1 et U_2 doivent coexister en mémoire centrale lorsqu'il exis deux modules en cours d'exécution (1), x et y, non forcément distincts (si x = y) on est dans le cas tels que

(I) 1) $x A^* y$ ou $y A^* x$

2) $u_1 \in U^*(x)$ et $u_2 \in U^*(y)$

(1) La clause "en cours d'exécution"est nécessaire pour une méthode de gestion dynamique (puisque c'est une condition pour que les représentations de u_1 et existent). Au contraire elle devient superflue pour une méthode statique, les localisations des représentations étant fixes pendant l'exécution.

La deuxième condition est équivalente à x D (v_1) et y D (v_2) moyennant les hypothèses sur les déclarations de v_1 et v_2.

3 - ILLUSTRATION PAR UN EXEMPLE DES NOTIONS PRECEDENTES

Nous nous proposons d'illustrer par un exemple l'allure des graphes (M, A) et (E, U), la notion de domaines de validité, et la règle de coexistence des représentations de variables.

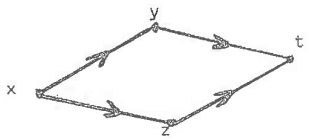
Soit $M = \{x, y, z, t\}$ et $C = \{a\}$

Supposons que

- Le module x appelle y et z, déclare un identificateur v_1
- y t , v_2
- z t , v_3
- t n'appelle aucun module, ne déclare aucun identificateur

Supposons de plus que y "utilise" a, et que z "utilise" a, et que la classe a déclare un identificateur v_4.

a) Le graphe (M, A) est alors :



x est le module directeur.

On constate sur cet exemple simple que (M, A) est sans circuit.

b) Le graphe (E, U) est :

c) Etudions les domaines de validité de v_1, v_2, v_3

v_1 est déclaré dans x

Nous savons que $D(v_1) = (U^*)^{-1}(x)$

Donc $D(v_1) = \{x\}$ nous retrouvons bien que v_1 est local au module x

De même $D(v_2) = (U^*)^{-1}(y) = \{y\}$

et $D(v_3) = \{z\}$

Enfin v_4 est déclaré dans a, donc :

$$D(v_4) = (U^*)^{-1}(a) = \{y, z, a\}$$

Nous retrouvons effectivement que v_4 est utilisable dans la classe a (puisqu'il y est déclaré), et dnas les modules y et z (puisque y "utilise" a et "utilise" a.

d) Etudions la règle de coexistence des représentations de v_1, v_2, v_3, v_4

Pour cela examinons la coexistence pour chaque couple d'identificate

- v_1 avec v_2 : $D(v_1) = \{x\}$, $D(v_2) = \{y\}$

Comme x A^* y, la coexistence est nécessaire.

- v_1 avec v_3 : $D(v_1) = \{x\}$, $D(v_3) = \{y\}$

Comme x A^* z la coexistence est nécessaire.

- v_1 avec v_4 : $D(v_1) = \{x\}$, $D(v_4) = \{y, z, a\}$

Comme x A^* y, la coexistence est nécessaire.

- v_2 avec v_3 : $D(v_2) = \{y\}$, $D(v_3) = \{z\}$

y et z n'étant pas liés par A^*, la coexistence n'est pas nécessaire.

- v_2 avec v_4 : $D(v_2) = \{y\}$, $D(v_4) = \{y, z, a\}$

Comme y A^* y, la coexistence est nécessaire.

- v_3 avec v_4 : $D(v_3) = \{z\}$, $D(v_4) = \{y, z, a\}$

Comme z A^* z, la coexistence est nécessaire.

Nous voyons que le seul couple d'identificateurs dont la coexistence n'est pas nécessaire est (v_2, v_3).

4 - EVOLUTION DE L'ETAT DE LA MEMOIRE CENTRALE PENDANT L'EXECUTION

La règle (I) de coexistence des représentations de variables a été établie indépendemment de tout choix de méthode de gestion de mémoire. Elle ne fait que traduire les contraintes à respecter dans toute méthode de gestion de mémoire.

Nous pouvons en nous appuyant sur cette règle, exprimer formellement quel est l'ensemble d'informations qui doivent être présentes à un moment donné de l'exécution. Nous pouvons aussi étudier les variations de cet ensemble au fil de l'exécution.

C'est précisément l'allure de ces variations qui va guider notre choix méthode de gestion de mémoire.

Soit X l'ensemble des modules en cours d'exécution à un moment T donné l'exécution.

X est constitué des modules qui sont en un même chemin du graphe (M, A) L'origine de ce chemin est le module directeur (c'est par lui qu'a débuté l'exécution). L'extrémité de ce chemin est le module dont les instructions exécutées par l'unité centrale au moment T.

Posons $X = \{x_0, x_1, \ldots, x_n\}$

L'ensemble V des variables qui doivent avoir une représentation en mém centrale au moment T, est l'ensemble des variables déclarées dans $\bigcup_{i=0}^{n} U^*(x_i)$ Ceci n'est que la traduction formelle du résultat obtenu au § 5.1 de la p 1.

Dans la suite de l'exécution, l'ensemble X peut se transformer

a) en un ensemble X' contenant X, dans le cas où le module x_n appelle le module x_{n+1} : $X' = \{x_0, x_1, \ldots, x_n, x_{n+1}\}$

L'ensemble V devient alors l'ensemble V' des variables déclarées dans $\bigcup_{i=0}^{n+1} U^*(x_i)$. Il s'ensuit que $V \subseteq V'$

b) en un ensemble X" contenu dans X dans le cas où l'exécution de x_n termine : $X'' = \{x_0, x_1, \ldots, x_{n-1}\}$

L'ensemble V devient alors l'ensemble V" des variables déclarées dans $\bigcup_{i=0}^{n-1} U^*(x_i)$. Il s'ensuit que $V \supseteq V''$

On voit que lorsqu'on pénètre dans un module, l'ensemble V s'agrandit que lorsqu'on termine un module l'ensemble V s'appauvrit.

Nous suivons donc une règle de blocs analogues à la règle des blocs d'Algol 60. Il est clair qu'une gestion dynamique en pile de la mémoire serait possible, les allocations étant faites à l'entrée dans les modules, et les libérations à la fin des modules.

5 - CONSEQUENCES D'UNE GESTION DYNAMIQUE

Si l'on fait un parallèle avec Algol 60, le problème de l'allocation est compliqué par le fait que le graphe (M , A) n'est pas nécessairement une arborescence. A l'entrée, dans un module x , les représentations de variables de $\hat{U}(x)$ peuvent déjà exister ou non : tout dépend du cheminement qui a été suivi dans le graphe (M , A) pour arriver à x . La seule certitude est qu'il faut implanter les variables locales à x .

De même, lorsque l'éxécution de x se termine, il faut libérer la place occupée par certaines variables de $\hat{U}(x)$, ou ne pas la libérer, selon que leur domaine de validité ne contient pas, ou contient un des modules qui ont précédé x . Seule la libération de la place occupée par les variables locales de x est certaine.

Donc si à l'entrée et à la sortie de x , on n'a aucun renseignement sur la suite des modules ayant précédé x , il est impossible de procéder aux allocations ou libérations. C'est-à-dire qu'au niveau du texte objet de x , il faut se donner la possibilité de tester l'opportunité d'une allocation ou d'une libération.

On peut donner une solution simple à ce problème : à chaque classe, on peut associer un compteur dont l'état indiquera que les variables déclarées dans la classe ont été allouées ou non.

Supposons que ce compteur ait été initialisé à 0 au début de l'éxécution.

A l'entrée d'un module x on réalisera l'allocation pour les variables locales à x (pour elles, on est sûr que l'allocation est systématique) et pour les variables des classes de $\hat{U}(x)$ dont le compteur vaut 0 , et on incrémentera de 1 tous les compteurs de $\hat{U}(x)$.

A la sortie du module x, on décrémentera de 1 tous les compteurs de $\hat{U}(x)$ et on réalisera la libération pour les éléments de $\hat{U}(x)$ dont le comp est tombé à 0, ainsi que pour les variables déclarées dans x. Il faudra en outre tenir à jour le pointeur du sommet de la pile.

Une méthode de gestion dynamique n'est pas particulièrement freiné le fait que le graphe des appels n'est pas une arborescence. En effet, cett difficulté peut se résoudre au prix de l'entretien d'un nombre restreint de compteurs (un par classe).

Nous proscrirons cependant une telle méthode pour les variables de taille fixe, pour les raisons de rapidité à l'exécution déjà évoquées.

D'autre part, puisque nous nous bornons à gérer la mémoire pour le variables de taille fixe, connue à la compilation, une gestion dynamique n'e pas à priori nécessaire. L'étude des conséquences d'une gestion statique se dès lors souhaitable.

6 - CONSEQUENCES D'UNE GESTION STATIQUE

6.1 Cas où (M, A) est une arborescence

Nous venons de voir que nous suivions une règle de blocs. Co nous ne nous intéressons qu'aux variables de taille fixe, rien s'oppose à procéder par allocation statique. Pour cela, nous pouvons nous inspirer d'une méthode parfois utilisée dans les compilateurs Algol 60 pour gérer les variables de taille fixe m locales à une procédure (la récursivité empêche de procéder de même façon pour les variables locales à des procédures) . Les emplacements de représentations des variables peuvent être défi relativement à une origine. Pour déterminer les adresses relati on initialisera un compteur d'emplacement à zéro, puis on pourra fur et à mesure de la compilation, incrémenter (lorsqu'on rentr un bloc) ou décrémenter (lorsqu'on sort d'un bloc), ce compteur d'emplacement, d'une quantité égale aux nombre de mots mémoire nécessaires pour implanter les variables de taille fixe du bloc. Lorsqu'on rentre dans un bloc, il suffit, avant d'incrémenter le

compteur d'emplacement d'implanter les variables de taille fixe déclarées dans le bloc dans une zone contigüe dont l'origine est égale à la valeur courante du compteur d'emplacement. Ainsi, on définit l'adresse relative de chaque variable comme la somme de l'adresse relative de cette zone et l'adresse relative de la variable dans cette zone.

Dans le cas où le graphe (M, A) est une arborescence, cette méthode est facilement adaptable à notre problème : il suffit d'explorer (M, A) à l'aide d'une pile. Les phénomènes d'entrée dans un bloc ou de sortie d'un bloc, pour Algol 60, correspondent respectivement à l'apparition d'un module au sommet de la pile, et à la disparition d'un module de la pile.

D'autre part, le variables à implanter lors de la montée d'un module x au sommet de la pile sont celles dont le domaine de validité contient x, et ne contient aucun des autres modules actuellement dans la pile.

Vérifions que l'implantation ainsi obtenue est conforme à la règle (I) (cf. 2ème partie §2) qui définit les conditions dans lesquelles la coexistence de représentations de variables est impérative :

Pour cela considérons deux variables v_1 et v_2 respectivement déclarées dans des classes ou modules u_1 et u_2 et supposons que les hypothèses de (I) soient vérifiées :

il existe deux modules x et y non forcément distincts tels que :

$$(I) \quad \begin{cases} x\ A^*\ y \text{ ou } y\ A^*\ x \\ u_1 \in U^*(x) \text{ et } u_2 \in U^*(y) \end{cases}$$

Soit x' le "premier" module du chemin d'origine le module directeur et d'extrémité x tel que $x'\ U^*\ u_1$. (C'est-à-dire qu'il n'existe pas de module x" tel que $x''\ \hat{A}\ x'$ et $x''\ U^*\ u_1$).

De même soit y' le "premier" module du chemin d'origine le module directeur et d'extrémité y, tel que $y'\ U^*\ u_2$.

Avec cette définition de x' et y', nous pouvons écrire :

$x'\ A^*\ x$ et $y'\ A^*\ y$.

L'allocation de v_1 est réalisée lorsque x' monte au sommet de la pil
celle de v_2 lorsque c'est y'.

Comme (M, A) est une arborescence, si la première hy
thèse de la condition (I) est vérifiée, on peut conclure que x', y',
sont sur un même chemin de (M, A). Notamment nous savons que $x'\ A^*$
$y'\ A^*\ x'$.

- Si x' = y', nous réalisons des allocations distinc
  pour v_1 et v_2 lorsque x' apparaît au sommet de la

- Si x' est différent de y', x' et y' figureront sim
  tanément dans la pile à un moment donné, (puisque
  y' sont sur un même chemin).

Supposons que x', par exemple, soit apparu le premier
sommet de la pile. Alors, le compteur d'emplacement n'a fait que croî
jusqu'à l'apparition de y' au sommet de la pile, de telle manière qu
deux modules qui apparaissent successivement au sommet de la pile,
l'allocation se passe toujours sans recouvrement.

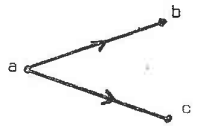
Nous sommes donc assurés que les représentations de v_1
v_2 sont toujours distinctes.

D'autre part, il est clair qu'avec cette méthode d'al
cation, nous ne faisons cohabiter en mémoire centrale que les couples
représentations de variables pour lesquelles la condition (I) est réa
En effet, nous n'allouons, lors de l'apparition d'un module au sommet
pile, que les seules variables accessibles par ce module, et qui n'on
encore été implantées.

La méthode assure donc un encombrement minimal de la
mémoire.

EXEMPLE :

Soit $M = \{a, b, c\}$ et $C = \{x, y\}$
Soit le graphe (M, A) suivant :



Supposons que a "utilise" x

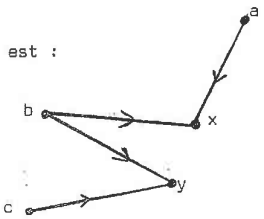
b "utilise" x et y

c "utilise" y

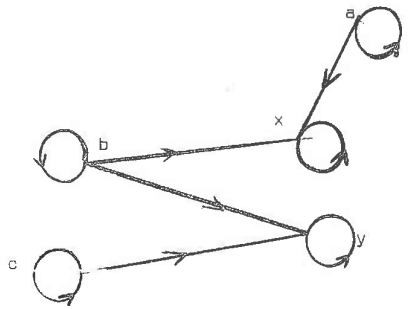
Le problème est d'implanter les variables déclarées dans a, b, c, x, y

Supposons que l'implantation des variables déclarées dans a, b, x, y nécessite respectivement 2 mots mémoire, et celles déclarées dans C, 3 mots mémoire.

Le graphe (E, U) est :



Pour définir les domaines de validité, bâtissons (E, U^*) :



D'après la définition des domaines de validité à l'aide de U^*, les domaines de validité des identificateurs déclarés dans a, b, c, x, y sont respectivement {a}, {b}, {c}, {a, b, x}, {b, c, y}

- Explorons (M, A) par une pile

  Soit Ad le compteur d'emplacement, initialement nul, et P la pile, initialement vide

1° Etat de la pile

P = {a}

On implante les variables dont le domaine de validité contient a (i celles de a et x) à l'adresse Ad, puis on incrémente Ad de 4

Valeur courante de Ad : 4

2° Etat de la pile

P = {a, b}

Les variables dont le domaine de validité contient b sont celles d b, x, y. Comme le domaine de validité des variables de x contient a, on n'impla que les variables de b et y. Puis on incrémente Ad de 4

Valeur courante de Ad : 8

3° Etat de la pile

P = {a}

On décrémente Ad de 4

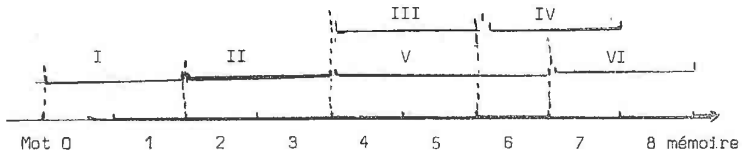
Valeur courante de Ad = 4

4° Etat de la pile

P = {a, c}

Les variables dont le domaine de validité contient C sont celles de Le domaine de validité de ces variables ne contient pas a : elles sont toutes à implanter en Ad.

L'exploration de (M, A) est terminée. Illustrons le résultat.



Lorsque le module a sera en cours d'exécution les variables de a seront implantées en I, celles de x en II.

Lorsque a et b seront actifs, les variables de b seront implantées en III et celles de y en IV.

Lorsque a et c seront actifs, les variables de c seront implantées en V et celle de y en VI.

On peut remarquer :

1) que les variables de x utilisées par les modules a et b d'un même chemin de (M, A) sont implantées à un seul endroit lorsque a et b sont actifs, ce qui garantit le passage d'information entre a et b par l'intermédiaire des variables de x.

2) que par contre, les variables de y suivant qu'elles sont manipulées par b ou c sont implantées en des endroits différents (IV et VI) ce qui empêche la communication entre b et c par l'intermédiaire des variables de y. Ceci est normal, car deux modules non sur un même chemin de (M, A) ne peuvent se communiquer d'information par l'intermédiaire d'une classe utilisée en commun.

3) que lorsque a et b sont actifs, les segments I, II, V, VI, sont manipulés par a et b. Il existe donc un moment de l'exécution où tous les mots réservés aux variables sont utilisés. L'implantation obtenue est donc optimale (son encombrement ne peut pas être réduit).

6. 2 - CAS OU (M,A) N'EST PAS UNE ARBORESCENCE

6. 2. 1 - Essai d'adaptation de la méthode précédente

Dans le cas où (M, A) est une arborescence, l'idée essentielle est de réaliser les allocations tout en suivant les chemins. Plus précisément, lorsqu'on s'intéresse à un a, toutes les variables accessibles par les prédécesseurs dans (M, A) ont déjà été implantées.

Dans le cas où (M, A) n'est pas une arborescence, c idée ne peut être généralement transposée. Soit en effet, module a, admettant deux prédécesseurs : b et c dans (M, A c'est-à-dire que b $\hat{A}$ a et c $\hat{A}$ a.

Supposons que b "utilise" une classe u_1 et que c "ut une classe u_2. A la seule vue des prédécesseurs de a, on n aucune raison d'allouer des emplacements de mémoire distinc aux variables déclarées dans u_1 d'une part et dans u_2 d'aut part, si les modules b et c ne sont pas liés par A . En et les hypothèses de la condition (I) ne sont pas remplies, et aucune raison ne s'oppose à la mise en recouvrement des var de u_1 et u_2.

Or, il se peut, précisément, que le module a, ou mêm un quelconque sucesseur commun de b et c, "utilise" à la fo les classes u_1 et u_2, ce qui exige cette fois que les variabl de u_1 et u_2 ne soient pas en recouvrement.

On s'aperçoit donc qu'un examen partiel de (M, A) ne suffire à prendre une décision de recouvrement ou non entr variables. La seule ressource est d'appliquer la condition et d'étudier le graphe (M, A) dans son ensemble.

Etudions cependant, un cas remarquable où une étude locale de (M, A) est suffisante : supposons que l'ensemble classes soit vide. Pour cela, examinons d'abord les résulta

obtenus lorsque (M, A) est une arborescence dans ce cas particulier, avec la méthode montrée au § 6. 1.

Soit $\{x_0, x_1, \ldots\ldots, x_{p-1}\}$ l'état de la pile à un instant donné.

A cet instant les variables déclarées dans :

$$\bigcup_{i=0}^{p-1} U^{*}(x_i)$$ sont déjà implantées.

Si l'ensemble des classes est vide :

$$\bigcup_{i=0}^{p-1} U^{*}(x_i) = \{x_0, x_1, \ldots\ldots, x_{p-1}\}$$

Lorsqu'un module x_p apparaît au sommet de la pile, on cherchera à implanter ses variables locales. Le compteur d'emplacement aura alors pour valeur $\sum_{i=0}^{p-1} \ell(x_i)$ si l'on appelle $\ell(x_i)$ le nombre de mots nécessaires à l'implantation des variables locales de x_i.

- Appelons longueur par les sommets d'un chemin d'un graphe, la somme des longueurs des sommets constituant ce chemin.
  Chaque sommet x_i de (M, A) étant muni de la longueur $\ell(x_i)$, nous voyons que les variables locales de (x_p) seront implantées à partir d'une adresse mémoire égale à la longueur par les sommets du chemin d'origine le module directeur et d'extrémité x_p, diminuée de $\ell(x_p)$.

  Décidons, ppour chaque module x, d'implanter ses variables locales à l'adresse $L(x) - \ell(x)$.
  Pour que l'implantation obtenue soit correcte, il faut vérifier la condition (I) :

  Considérons deux modules u_1 et u_2. Les seuls modules x et y tels que $xU^{*}u_1$ et $yU^{*}u_2$ sont respectivement u_1 et u_2 eux-mêmes. Pour que la condition (I) s'applique, il faut donc que $u_1 A u_2$ ou $u_2 A^{*} u_1$.

Distinguons deux cas :

- 1) $u_1 = u_2$: les variables locales de module sont implantées contigu
  donc sans recouvrement.

- 2) $u_1 \neq u_2$: u_1 et u_2 sont alors sur un même chemin de (M, A)

Supposons par exemple que $u_1 \hat{A} u_2$. Soit $\{u_1, z_0, z_1, \ldots, z_n, u_2\}$ un pl
long chemin par les sommets de u_1 à u_2.

Les variables de u_1 sont implantées à des adresses mémoire allant de :
$L(u_1) - \ell(u_1)$ à $L(u_1) - 1$

Celle de u_2 de $L(u_2) - \ell(u_2)$ à $L(u_2) - 1$

Or $L(u_2) = L(u_1) + \sum_{i=0}^{n} \ell(z_i) + \ell(u_2)$

On en déduit :

$$L(u_2) - \ell(u_2) = L(u_1) + \sum_{i=0}^{n} \ell(z_i) > L(u_1) - 1$$

Toutes les variables de u_2 sont donc implantées à des adres
supérieures à celles de u_1.

L'implantation obtenue satisfait donc la condition (I).

D'autre part, les variables locales des modules d'un même ch
devant coexister, on ne peut réduire davantage l'encombrement des variables.

La longueur de la zone réservée aux variables est égale à la
longueur du plus grand chemin par les sommets de (M, A).

6. 2. 2. Unicité de la représentation d'une variable

Outre la faillite de la méthode d'allocation type Algol 60, le fait que (M, A) ne soit pas une arborescence amène également une nouvelle contrainte. En effet, dans les conditions d'adoption d'une gestion statique nous avons spécifié que les variables seraient atteintes par adressage direct à l'exécution. Cette obligation a une influence capitale précisément lorsque (M, A) n'est pas une arborescence.

Soient en effet deux modules a et b, non sur un même chemin de (M, A), et une classe x telle que :

a U x et b U x.

Comme a et b ne sont pas sur un même chemin, ces modules n'ont pas à s'échanger d'informations par l'intermédiaire de la classe x. On peut donc, en principe avoir, en mémoire, deux représentations des variables déclarées dans x x : une représentation lorsque a est en cours d'exécution, et une représentation lorsque c'est b.

Cette disposition est d'ailleurs illustrée dans l'exemple qui a été précédemment traité. (Cas où (M, A) était une arborescence).

Cependant, on peut toujours supposer dans un cas général, qu'il existe un module c qui "utilise" x et tel que :

a A c et b A c. (Une telle supposition ne peut être faite lorsque (M, A) est une arborescence).

Le module c peut échanger des informations avec b ou a par l'intermédiaire des variables de x. Il est vrai que, a et b n'étant jamais actifs simultanément, le module c n'accédera jamais qu'à l'une ou l'autre des représentations des variables de x à la fois. Un tel cas n'est donc pas vraiment incompatible avec une gestion statique. En revanche, si l'on désire que les instructions de c accèdent aux variables de x par adressage direct, aussi bien dans l'une que dans l'autre représentation, faut admettre qu'on possède deux versions du texte objet de c dans lesquelles les parties adresses des instructions accédant aux variables de x sont relatives à l'une ou à l'autre représentation.

Une telle solution est évidemment impraticable.

Pour ne conserver qu'une version unique du texte objet d nous sommes obligés d'admettre que la représentation d'une variable est unique. A priori, il faut donc affirmer que toute variable a une représentation unique en mémoire.

Cette unicité a une portée considérable, car elle fait pa tout espoir d'une méthode d'allocation attachée aux chemins du graphe appels, optique qui, comme nous l'avons vu précédemment, était déjà problématique. D'autre part, l'unicité a une influence capitale sur l'encombrement maximal de la mémoire.

En effet soient C_1, C_2,....., C_n les chemins maximaux de (Nous appelons ici chemin maximal un chemin qui n'est contenu dans a autre).

Soit L_i le nombre de mots de mémoire nécessaires pour imp toutes les variables désignées par des identificateurs dont le domain validité contient au moins un module de C_i. Il était naturel de pens l'encombrement maximal de la mémoire était de $\max_{i \in [1, n]} L_i$ mots, encombr maximal qu'on obtiendrait dans une gestion dynamique de mémoire la pl économique possible.

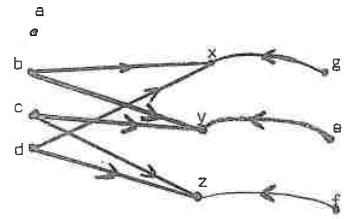
L'exemple suivant montre qu'il n'en est rien, et qu'au co l'unicité des représentations peut conduire à une perte complète des possibilités de recouvrement.

Soit M = {a, b, c, d, e, f, g} et C = {x, y, z}

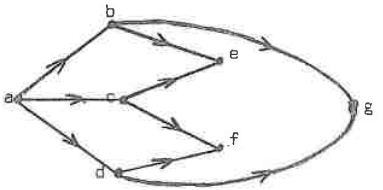
Soit :
b "utilise" x et y
c "utilise" y et z
d ----- x et z
e ----- y

f -------- z
g -------- x

D'où le graphe (E, U)



Soit (M, A) le graphe des appels :



b et c "utilisent" y et ne sont pas sur un même chemin de(M, A).

Les implantations des variables de y vis à vis de b et c pourraient être distinctes, mais le module e "utilise" y et impose l'unicité de la représentation des variables de y.

On trouve de même que les représentations des variables de y et z sont uniques.

Or, comme b "utilise" x et y, les représentations des variables de x et y doivent coexister. Il en est de même avec les classes y et z, que c "utilise", et avec les classes z et x que d "utilise".

Soit v_x, v_y, v_z des variables déclarées respectivement dans les classes x, y, z.

Les représentations de deux parmi ces trois variables doivent toujo coexister en mémoire centrale.

La seule slolution est de faire coexister les représentations respec de v_x, v_y, v_z.

Les chemins maximaux de (M, A) et les classes que les modules de ces chemins "utilisent" sont :

{a, b, g} : les modules de ce chemin "utilisant" x et y
{a, b, e} : .... x et y
{a, c ,e} : .... y et z
{a, c, f} : .... y et z
{a, d, f} : .... x et z
{a, d, g} : .... y et z

Dans un chemin maximal, deux classes seulement sont "utilisées" à la Une gestion dynamique en pile conduirait donc à faire coexister les représents des variables de deux classes au plus en mémoire centrale.

Une gestion statique exige que les représentations des variables des classes coexistent.

7 - COMPARAISON DES CONSEQUENCES DES GESTIONS STATIQUE ET DYNAMIQUE

L'étude du problème de gestion de la mémoire nous a amenés à créer outils de base qui sont les graphes des appels et des directives "utilise".

Ces deux outils permettent de définir une règle précise (condition coexistence nécesseire en mémoire centrale des représentations de deux variab

Moyennant cette règle, il nous a été possible d'aborder l'étude des implications de l'adoption de gestions statique ou dynamique de mémoire.

Nous avons vu qu'une gestion dynamique en pile était possible, puis informations suivant une règle de blocs.

Nous abandonnerons la gestion dynamique, par souci de rapidité à l'ex

L'étude d'une solution de gestion dynamique apporte néanmoins deux e

gnements, et méritait, à ce titre, d'être envisagée avant rejet :

D'abord, la gestion dynamique envisagée est valable, non seulement pour les objets dont la taille est fixe pendant l'exécution, connue dès la compilation (objets auxquels nous bornerons ultérieurement notre étude), mais aussi pour les objets dont la taille est fixe pendant l'exécution, mais inconnue à la compilation : ce sont les tableaux type Algol 60, auxquels correspondent les files à taille maximale en CIVA. L'étude d'une solution dynamique permet donc de prévoir la gestion de mémoire pour les files à taille maximale.

D'autre part, l'étude d'une gestion dynamique montre quel est l'encombrement maximal de la mémoire centrale auquel il faut s'attendre. Ceci servira de point de référence pour l'estimation de l'encombrement de la mémoire dans le cas d'une gestion statique.

Dès l'abord de l'étude d'une solution de gestion statique, la notion de règle de blocs conduit à chercher un algorithme d'allocation lié aux chemins du graphe des appels. Nous avons vu que l'impossibilité d'appliquer cette idée était dûe du fait que le graphe des appels n'est pas, en général, une arborescence. Cependant, un procédé d'allocation lié aux chemins du graphe des appels peut être mis en oeuvre si les seules variables à implanter sont celles locales aux modules. De ce procédé, nous pouvons tirer une idée plus générale, pour laquelle nous donnons l'énoncé (II :

- Etant donné un graphe sans circuit, à chaque sommet duquel est associé un lot de variables, on connaît un procédé permettant d'implanter les variables avec un encombrement minimal, de telle manière que si deux sommets s_1 et s_2 sont sur un même chemin du graphe, les variables associées à s_1 et s_2 ne sont pas en recouvrement.

Enfin, dans l'étude des conséquences d'une gestion statique, nous avons vu que l'accès aux informations par adressage direct, condition à laquelle nous tenons par dessus tout par souci de rapidité à l'exécution, impliquait l'unicité des représentations des variables dans le cas général.

En conclusion, nous pouvons dire que la mise en oeuvre d'une gestion dynamique semble la plus facile, car une gestion de la mémoire en pile présente peu de problèmes. En revanche, l'adoption d'une gestion statique soulève le problème de l'unicité des représentations des variables. La condition (I) se présente comme une contrainte de disjonction des intervalles de mémoire contenant deux informations. Passer des contraintes exprimées par cette condition à une allocation effective demande une étude particulière que nous développerons plus loin.

D'autre part, une gestion dynamique résoud le problème des informations de taille inconnue à la compilation fixe à l'exécution ce qui n'est pas le cas pour une gestion statique. Malgré les avantages et les facilités offerts par une gestion dynamique, nous adopterons la solution d'une gestion statique, pour les gains de temps qu'elle procurera lors des exécutions de chaînes de traitement répétitives.

3ème PARTIE

PROBLEME DU RECOUVREMENT POUR LES TEXTES OBJETS

1 - ELIMINATION DES INFORMATIONS VARIABLES DES TEXTES OBJETS

2 - ETUDE DES SOLLICITATIONS DU RESIDENT GERANT LES ECHANGES ET DU CONTENU MINIMAL DE LA MEMOIRE CENTRALE

3 - CHOIX D'UN CRITERE DE RECOUVREMENT

4 - ANALOGIE AVEC LES VARIABLES

3ème PARTIE

PROBLEME DU RECOUVREMENT POUR LES TEXTES OBJETS

L'ensemble des informations constantes est constitué des textes objets engendrés par le compilateur, pour tous les modules et classes d'une unité de traitement. De même que nous avons cherché une règle de co-habitation des représentations de variables, nous cherchons à définir dans quelles circonstances la présence simultanée de textes objets en mémoire centrale est souhaitable.

La notion de domaine de validité d'un identificateur, précédemment introduite est valable aussi bien pour des identificateurs de variables que pour des identificateurs de constantes. Ainsi, pouvons nous dire immédiatement qu'un module x a accès aux constantes désignées par des identificateurs déclarés dans $U^*(x)$. Nous devons nous attendre à ce que la relation "utilise" joue, dans le cas des constantes un rôle très analogue à celui qu'elle jouait pour les variables, puisqu'elle régit les conditions d'accès à une information. De même, les successions d'appels de modules (cf. graphe des appels de modules) jouaient un grand rôle dans le recouvrement des variables. Elles joueront également un rôle important dans le recouvrement des constantes % en effet, si un module x appelle un module y, la présence du texte objet de y en mémoire centrale devient indispensable, de même que doivent alors apparaître des représentations des variables locales à y.

Cependant, le critère essentiel qui a permis d'élaborer la règle de co-habitation pour les variables était d'éviter l'emploi d'une mémoire secondaire en garantissant la présence effective en mémoire centrale de la représentation d'une information pendant toute sa durée de vie. Dans le cas des constantes, l'emploi d'une mémoire secondaire est de toute façon indispensable. Ceci entraîne que nous n'avons plus à garantir la présence d'une constante en mémoire centrale tant qu'on risque de lui faire référence, comme nous le faisons pour les variables : il sera toujours temps d'aller chercher cette constante sur la mémoire secondaire.

Par conséquent, si l'organisation de l'information (condition d'accès à information, enchaînement des appels de modules) est identique pour les variab et les constantes, les problèmes de recouvrement, pour les unes ou les autres, à priori différents.

De plus, pour mettre en oeuvre un recouvrement des constantes, il est n saire d'adjoindre à l'unité de traitement un programme résidant en mémoire cen capable d'assurer les chargements de la mémoire centrale depuis la mémoire sec aux moments opportuns.

En définitive, la détermination d'un recouvrement pour les constantes dépendra de trois facteurs :

- l'organisation de l'information (c'était le seul facteur int venant pour les variables) ;
- la nécessité de mettre en oeuvre un programme résident géra chargements ;
- le souci de minimiser la fréquence des chargements.

Ces deux derniers facteurs sont importants. On ne peut sous peine de pe de temps considérables, solliciter à tout instant le programme résident pour j de l'opportunité d'un chargement (même si en définitive, ce chargement n'est p effectué) ou procéder effectivement à des chargements incessants.

Si les textes objets des modules et des classes ne comportent que des i mations constantes, nous pouvons limiter les échanges entre mémoire principale mémoire secondaire à un échange à sens unique de mémoire secondaire à mémoire centrale. Pour cela, nous allons voir comment il est possible d'éliminer toute information variable des textes objets.

1 - ELIMINATION DES INFORMATIONS VARIABLES DES TEXTES OBJETS

Puisque nous avons décidé d'implanter les variables en une zone de var extérieures aux textes objets, les textes objets qui nous restent à implanter essentiellement constitués d'informations constantes au cours de l'exécution.

Les seules informations variables qui pourraient subsister dans les textes objets sont les variables de travail nécessaires à l'exécution des programmes. Ces variables sont créées par le compilateur. Elles sont de deux provenances :

- d'une part, les variables engendrées sors de la compilation des modules et d'autre part, les variables engendrées lors de la compilation des procédures déclarées dans les classes.

Il est clair que les variables de travail d'un module ont une durée de vie qui est le temps d'exécution de ce module . Elles peuvent donc être considérées au même titre que des variables déclarées dans le module. C'est-à-dire que le mécanisme d'allocation employé pour les variables locales du module peut s'appliquer également aux variables de travail crées par le compilateur.

D'autre part, un module ne peut faire référence à une procédure d'une classe que s'il "utilise" cette classe. La durée de vie des variables de travail créées pour les procédures d'une classe est donc identique à celle des variables normalement déclarées dans cette classe.

Les variables de travail des procédures d'une classe peuvent être traitées comme les variables déclarées dans la classe.

Donc, moyennant la prise en charge des variables de travail dans l'étude de la gestion de mémoire relative aux variables, nous éliminons des textes objets toute information variable. Nous évitons ainsi une cause très importante d'échanges entre mémoires principale et secondaire : nous n'aurons jamais à sauvegarder la partie de la mémoire centrale contenant les textes objets.

2 - ETUDE DES SOLLICITATIONS DU RESIDENT GERANT LES ECHANGES ET DU CONTENU MINIMUM DE LA MEMOIRE CENTRALE

Les modules et les classes se présentent naturellement comme unités de découpage des textes objets. Le texte objet d'un module ou d'une classe sera donc toujours dans son intégralité, absent de la mémoire centrale, ou présent en mémoire centrale.

Le choix de critères de recouvrement dépend non seulement de la contrainte

imposée par la taille de la mémoire, qui doit être compatible avec la taill
textes objets qu'elle doit contenir d'un moment donné, mais aussi de l'évo
du contenu de la mémoire pendant l'exécution.

Plaçons-nous à un moment de l'exécution où une instruction
module x a le contrôle de l'unité centrale. Cette instruction peut faire ré
à n'importe quelle constante des classes de $\hat{U}(x)$. Supposons qu'elle fasse
rence à une constante déclarée dans une classe y de $\hat{U}(x)$. Il faut, pour que
l'instruction en cours se déroule normalement; que le texte objet de la cla
y soit présent en mémoire centrale. Donc, si l'on n'a pas décidé à priori q
pendant l'exécution du module x les textes objets des classes de $\hat{U}(x)$ devai
présents en mémoire centrale, il faut que toute instruction du module x fai
référence à une constante déclarée dans une classe de $\hat{U}(x)$ provoque la soll
citation du programme résident gérant les échanges. Ce programme doit alors
naître la classe à laquelle l'instruction de x fait référence. Pratiquement
donc obligé, au niveau du texte objet de x, de prévoir une séquence d'appel
résident avant toute référence d'une constante non locale à x. Outre le coû
prohibitif d'une telle solution, on voit qu'il est nécessaire de prévoir une
compilation "spéciale" de x lorsqu'on veut faire du recouvrement : il faut e
effet générer les séquences d'appel, inutiles dans l'hypothèse du non recouv
De telles contraintes sont tout à fait inadmissibles. On ne peut donc que fa
co-habiter en mémoire centrale tous les textes objets de $U^*(x)$, lorsque x e
actif.

3 - CHOIX D'UN CRITERE DE RECOUVREMENT

Nous venons de voir que lorsqu'un module x a le contrôle de
centrale, tous les textes objets de $U^*(x)$ doivent être présents en mémoire c
Avec cette hypothèse, les points de sollicitation du résident chargé des éch
sont les appels de module à un autre, et les fins d'exécution de module.

Or, en fin d'exécution d'un module, le retour s'effectue dans
module appelant. Il serait donc dommage, alors que nous étions sûrs de son ré
emploi, d'avoir chassé le module appelant de la mémoire centrale. Si nous dé
lorsqu'un module est activé, de conserver en mémoire centrale, le texte du m
qui l'a appelé, nous supprimons à la fois une cause importante d'accès à la m
secondaire, en même temps, toute raison de faire appel au résident responsab

chargements à la fin de l'exécution d'un module. Remarquons cependant que cette manière de procéder n'est pas indispensable. Il s'agit seulement là d'une décision qui semble raisonnable (1).

Or, un module appelant est lié au module appelé par la relation A.

Puisque nous décidons de faire co-habiter en mémoire centrale deux modules x et y tels que x A y, nous déciderons, partransitivité,de faire co-habiter deux modules x et y tels que : $x \hat{A} y$.

Par conséquent, si $\{x_1, x_2, \ldots, x_n\}$ est un chemin du graphe (M, A) d'origine le module directeur, lorsque le module x_n sera activé, la mémoire centrale contiendra les textes objets des modules $x_1, x_2, \ldots, x_n$.

Elle contiendra aussi les textes objets :
$\hat{U}(x_1), \hat{U}(x_2), \ldots, \hat{U}(x_n)$.

Cette constation permet de définir une condition pour que deux textes objets soient nécessairement implantés en des emplacements disjoints en mémoire centrale.

En effet, soient u_1 et u_2 deux classes ou modules. Soit x un module tel que : $x U^* u_1$. Soit y un module tel que : $y U^* u_2$.

a) Si x est identique à y, les textes objets u_1 et u_2 devront

(1) Cette règle du "cheminement arrière" est appliquée, lorsqu'on utilise une structure de recouvrement sous contrôle des moniteurs SIRIS 7 ou BPM du CII 10070.

co-habiter, puisque nous décidons de faire co-habiter tous l
textes objets de $\upsilon^*(x)$

b) Si x est différent de y, et si x et y sont sur un même che
du graphe (M, A), nous ferons également co-habiter les text
objets de u_1 et u_2.

Dans l'hypothèse a aussi bien que dans l'hypothèse b, nous pou
écrire que : $x\ A^*y$ ou $y\ A^*x$

Nous pouvons donc énoncer la condition pour que les textes obj
de u_1 et u_2 co-habitent sous la forme :

> Il existe deux modules x et y, non forcément distincts, ta
> que :
> 1) $x\ A^*y$ ou $y\ A^*x$
> 2) $x\ \upsilon^*u_1$ et $y\ \upsilon^*u_2$

4 - ANALOGIE AVEC LES VARIABLES

Nous trouvons une condition de co-habitation remarquablement s
à la règle de co-habitation que nous avons trouvée pour les variables, bi
les démarches que nous avons suivies soient différentes : pour les variab
nous avons toujours été guidé par la nécessité, alors que pour les const
nous n'avons fait que prendre des décisions semblant raisonnables. Quoiqu
en soit, cette analogie découle de la structuration extrêmement forte d'u
unité de traitement.

Nous avons maintenant énoncé les contraintes se présentant dan
recouvrements des variables et des constantes sous une forme unique. Nous l
ront donc, à partir de ce jeu unique de contraintes, donner un moyen uni
d'obtenir une implantation effective des variables et des textes objets.

Le seul problème spécifique au cas de constante restera la mise en
oeuvre du programme résident assurant les chargements.

4ème PARTIE

MECANISMES DE LA COMPILATION ET DE L'EDITION DE LIENS

1 - ESPACES DE VARIABLES

2 - REPRESENTATION CONTIGUE DES ESPACES DE VARIABLES EN MEMOIRE CENTRALE

3 - PERTE DUE AU REGROUPEMENT EN SEGMENTS

4 - REFERENCES NON SATISFAITES APPARAISSANT DANS UNE CLASSE OU UN MODULE

5 - ROLE DU PRE-EDITEUR DE LIENS VIS A VIS DES REFERENCES NON SATISFAITES

6 - REMARQUE SUR L'ORDONNANCEMENT DE LA COMPILATION

7 - FICHIER DESCRIPTIF DES CLASSES ET DES MODULES

8 - ROLE PARTICULIER DU PRE-EDITEUR DE LIENS DANS LE PROBLEME DU RECOUVREMENT DES TEXTES OBJETS

4ème PARTIE

MECANISMES DE LA COMPILATION ET DE L'EDITION DE LIENS

1 - ESPACES DE VARIABLES

Nous n'avons fait aucune hypothèse sur le groupement des représentations en mémoire centrale des variables d'une même classe ou d'un même module, bien qu'il puisse paraître logique de les implanter de manière contigüe. Nous verrons que si un tel regroupement allège certains traitements, il amène en contrepartie une nouvelle contrainte dans l'allocation, pouvant se traduire par une augmentation de la taille de mémoire nécessaire à l'implantation des variables.

Cependant, la règle de cohabitation des représentations de deux variables en mémoire centrale est identique pour des couples de variables (v_1, v_2) et (v'_1, v'_2) pourvu que v_1 et v'_1 d'une part, v_2 et v'_2 d'autre part soient déclarées dans la même classe ou le même module. Nous sommes donc amenés à faire un regroupement logique des représentations de variables (mais non pour le moment un regroupement spatial). Nous appellerons espace des variables d'un module ou d'une classe x l'ensemble des variables déclarées dans x (ou créés à la compilation de x).

Si l'on convient que les représentations en mémoire centrale de deux variables d'un même espace ont des localisations différentes, la loi de cohabitation établie pour les variables peut se transcrire pour les espaces de variables de la manière suivante :

Les espaces de variables de deux classes ou modules u_1 et u_2 sont disjoints s'il existe deux modules x et y, non forcément distincts, tels que :

1) $x \; A^* \; y$ ou $y \; A^* \; x$
2) $x \; U^* \; u_1$ et $y \; U^* \; u_2$

Cette forme est identique à la loi de cohabitation pour les constantes, sans laisser préjuger de l'organisation d'un espace de variables en mémoire cent

Nous nous proposons maintenant d'étudier les avantages et les inconvén qu'il y a à organiser les espaces de variables de manière contigüe.

2 - REPRESENTATION CONTIGUE DES ESPACES DE VARIABLES EN MEMOIRE CENTRALE

Il est important de constater que l'espace des variables d'une classe o d'un module est à priori quelconque, c'est-à-dire que rien n'empêche que les repr sentations des variables d'un même espace soient éparpillées en mémoire centrale.

Cependant, un avantage important de la programmation modulaire est de d ner la possibilité de compilations séparées des différents éléments. Le pré-édite de liens ne peut intervenir que lorsque tous les modules et classes d'une unité de traitement sont compilés. C'est alors qu'il choisit la localisation des variables et doit créer les définitions correspondant aux références non satisfaites dans le texte objet. L'éditeur de liens doit ensuite établir la concordance entre référen et définitions, et compléter les textes objets. Avec un tel mécanisme, si les var bles d'un même module ou d'une même classe sont éparpillées en mémoire, le texte objet du module x de la classe devra comporter autant de références qu'il y a de variables. Le pré-éditeur de liens devra donner autant de définitions qu'il y a de variables dans l'unité de traitement. Enfin, l'éditeur de liens devra assurer la concordance de cette multitude de définitions et de références.

Un tel système est évidemment trop lourd. Il aurait de plus un autre inc vénient : en effet, dans le cas de mises au point particulièrement délicates, la programmeur peut être amené à demander l'impression d'une image binaire du conten la mémoire. L'examen de cette image, de toute façon fastidieuse, deviendra particulièrement pénible si les représentations des variables d'un même module ou d'une meme classe sont éparpillées.

Pour ces raisons, il semble naturel de rendre contigüe les espaces de variables de chaque classe ou module. C'est-à-dire que nous créons autant de segments de variables qu'il y a de classes ou modules comportant des variables.

De cette façon, pendant la compilation d'une classe ou d'un module, le compilateur peut attribuer à chaque variable une adresse relative dans le segment des variables. Faire la pré-édition de liens consistera alors à définir les origines de tous les segments. L'éditeur de liens, en fonction des adresses de début de segment, n'aura plus à effectuer que la transformation des adresses relatives en adresses absolues. Le nombre des références externes à traiter n'est plus excessif.

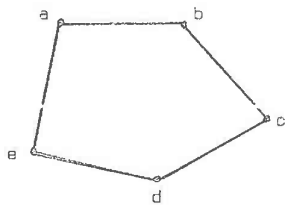
3 - PERTE DUE AU REGROUPEMENT EN SEGMENTS

Nous venons de constater qu'il était à la fois logique et intéressant de regrouper les variables en segments.

Il faut cependant remarquer que ceci constitue une nouvelle contrainte pour l'allocation de la mémoire.

Pour nous en persuader, examinons l'exemple suivant :

Soient 5 espaces de variables à implanter. Appelons les a, b, c, d, e. Supposons que l'implantation de chacun des ces espaces de variables nécessite deux mots de mémoire. Enfin, supposons que les contraintes de non recouvrement soient représentées par le graphe non orienté suivant :



Deux sommets joints par une arête correspondent à deux espaces de v nécessairement disjoints en mémoire centrale.

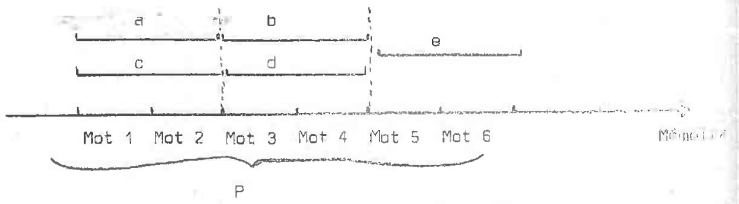
Notre but est d'implanter ces cinq espaces de variables sous forme d segments contigus, dans une partie P contigüe de la mémoire centrale petite que possible. Montrons que, pour l'exemple ci-dessus, il fau moins 6 mots pour implanter les segments.

DEMONSTRATION : pour faire la démonstration, prouvons qu'une implantation sur partie P de 5 mots est impossible.

Implanter 5 segments, c'est choisir leurs 5 origines respectives. Les origines des segments ne peuvent être choisies que parmi 4 adresses : un segm peut avoir son origine égale à l'adresse du mot de P ayant l'adresse la plus é car le deuxième mot de ce segment ne serait plus contenu dans P. Comme il y a ments à placer, il y a donc nécessairement 2 segments ayant la même origine.

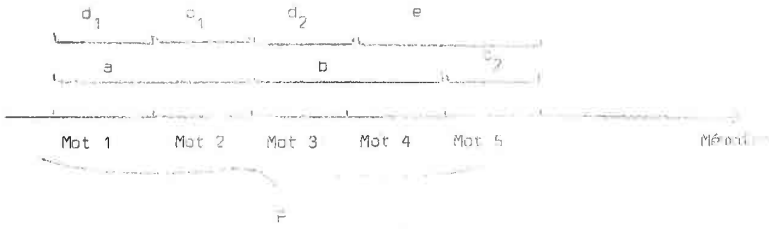
On constate facilement en examinant le graphe orienté que trois segm ne peuvent pas se recouvrir à la fois, (en effet, il n'existe pas trois segmen adjacents deux à deux). Les 2 segments qui ont nécessairement la même origine sitent 2 mots de P pour être implantés. Les 3 autres doivent être implantés n saisement ailleurs, donc dans les 3 mots restants de P. Or, il existe nécessai deux segments parmi eux qui ne peuvent être en recouvrement. Deux tels segmen nécessiteront 4 mots pour être implantés. Comme il en reste 3 disponibles dan l'allocation est impossible.

Donnons une solution avec une partie P de 6 mots :



- Supposons maintenant que l'on n'ait pas cherché à regrouper les variables en segments contigus, et que c et d puissent se scinder respectivement en deux variables dont la représentation nécessite mot. Soient c_1 et c_2 les variables constituant c, d_1 et d_2 les variables constituant d.

Alors, nous pouvons exhiber une allocation des variables dans une partie P de 5 mots :



On constate que cette implantation respecte les contraintes exprimées p le graphe non orienté traduisant le non recouvrement des espaces de variables, et que d'autre part, les représentations d'un même espace de variables cohabitent mémoire centrale (c'est-à-dire que c_1 est disjoint de c_2, d_1 de d_2).

Cet exemple montre bien toute l'importance du regroupement des espaces variables en segments contigus. Cette contrainte nous fait ici perdre un mot une partie de 5 mots, soit 20 % de perte de place.

4 - REFERENCES NON SATISFAITES APPARAISSANT DANS UNE CLASSE OU UN MODULE

Soit x une classe ou un module.

x a accès aux variables de $U^*(x)$. Pour chaque élément de $U^*(x)$, nous définissons un segment de variables adressable par des instructions de x. Les origines de ces segments apparaissent donc dans le texte objet de x comme des réf non satisfaites.

D'autre part, si x est un module, il peut appeler n'importe quel module de A (x). Les compilations des modules étant séparées, les points d'entrée dans les modules de A (x) doivent donc être aussi des références non satisfaites dans le texte objet de x.

Enfin, que x soit une classe ouun module, il peut accéder aux informations constantes des classes de $\hat{U}$ (x) : les textes objets de $\hat{U}$ (x) seront donc repérés dans x par le biais de références non satisfaites.

A toutes ces références non satisfaites doivent correspondre des définitions pour que l'édition de liens soit possible. De même qu'un segment de variables est connu par la définition de son origine, de même nous repérerons chaque texte objet par une définition unique.

L'éditeur de liens que nous utilisons et celui du moniteur SIRIS 7 implanté sur le CII 10070. Cet éditeur traite les références et les définitions comme des chaînes de caractères apparaissant dans les textes objets à éditer.

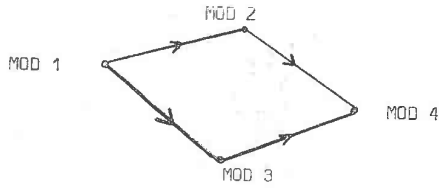
Chaque module et chaque classe sont désignés par un nom figurant dans le texte source. Ce nom servira à repérer ce module ou cette classe.

Nous conviendrons de repérer le segment de variables d'un module ou d'une classe par le nom de ce module ou de cette classe, précédé du caractère %.

Nous dirons que la chaîne de caractères ainsi obtenue est le % nom du module (ou de la classe).

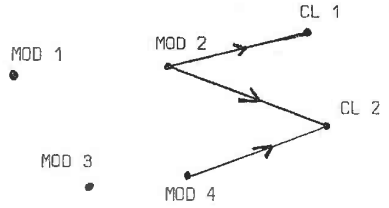
Voyons sur un exemple quelles seront les références et définitions apparaissant dans les textes objets des classes et modules.

Soit le graphe des appels (M, A) suivant :



où MOD 1, MOD 2, MOD 3, MOD 4 sont les noms des modules de M.

Soit le graphe (E, U) des directives "utilise".



où CL 1 et CL 2 sont les noms des classes de C.

a) Chaque module x doit comporter comme références

- les noms des modules de A (x) (pour que les appels de modules soient possibles)
- les noms des classes de $\hat{U}$ (x) (pour que l'accès aux informations constantes des classes de $\hat{U}$ (x) soit possible)
- les % noms des éléments de U^* (x) qui comportent effectivement des variables (pour que l'accès aux variables des classes de $\hat{U}$ (x) et l'accès aux variables déclarées dans x soient possibles)

Appliquons cette règle pour les différents modules :

- Pour MOD 1 : A (MOD 1) = {MOD 2, MOD 3}
  $\hat{U}$ (MOD 1) = ∅

Les références seront donc : MOD 2, MOD 3, % MOD 1

- Pour MOD 2 : A (MOD 2) = {MOD 4}
  $\hat{U}$ (MOD 2) = {CL1, CL 2}

Les références seront donc : MOD 4, CL 1, CL 2, % CL 1, % CL 2, % MOD 2

- Pour MOD 3 : A (MOD 3) = MOD 4
$\hat{U}$ (MOD 3) = Ø

Les références seront donc : MOD 4, Ø MOD 3

- Pour MOD 4 : A (MOD 4) = Ø
$\hat{U}$ (MOD 4) = CL 2

Les références seront donc : CL 2, Ø CL 2, Ø MOD 4

b) De même chaque classe c doit comporter comme références :

- les noms des classes de $\hat{U}_{(x)}$
- Les Ø noms des classes de $U^{*}_{(x)}$

Nous aurons donc :

- Pour CL 1 : $\hat{U}_{(CL\ 1)} = Ø$

La seule référence sera : Ø CL 1

- Pour CL 2 : $\hat{U}_{(CL\ 2)} = Ø$

La seule référence sera : Ø CL 2

c) Chaque texte objet doit se faire connaître des autres textes objets qui s'y r
rent par une définition. Chaque module et chaque classe doivent donc déclare
nom comme définition externe.

Ainsi, dans l'exemple précédent, le module MOD 1 se réfère à MOD 2 par l'intermédiaire de la référence non satisfaite MOD 2. Le module MOD 2 doit do
faire connaître de MOD 1 en déclarant dans son texte objet que la chaîne de ca
tères MOD 2 doit être considérée comme une définition. Nous dirons qu'il s'ag
d'une définition externe. Lorsque l'éditeur de liens aura fixé l'implantatio
module MOD 2, cette définition sera connue et les références à MOD 2 pourron
satisfaites.

5 - ROLE DU PRE-EDITEUR DE LIENS VIS A VIS DES REFERENCES NON SATISFAITES

Les classes et modules se réfèrent aux segments de variables par le biais de références. Ces segments doivent donc comporter leur Ø nom comme définition externe. Deux techniques peuvent être adoptées : ou bien, à la compilation d'une classe ou d'un module, le compilateur crée un texte objet qui est le segment des variables correspondant et y déclare le Ø nom de la classe ou du module qu'il est en train de compiler, comme définition externe ; ou bien ce rôle n'est confié qu'en tout dernier ressort au pré-éditeur de liens.

Examinons de plus près la seconde hypothèse. Connaissant les tailles des différents segments, et moyennant un algorithme permettant de les disposer les uns par rapport aux autres de manière optimale, le pré-éditeur de liens peut déduire le nombre N de mots nécessaires à l'implantation des segments dans un espace contigu de mémoire.

Il est alors en mésure de générer un texte objet unique dans lequel N mots sont réservés de manière contigüe, et comportant les définitions des Ø noms désignant les origines des segments : il suffit que les Ø noms soient définis avec une adresse relative dans ce texte objet. L'éditeur de liens, lorsqu'il implantera ce texte objet de manière absolue, pourra calculer de manière absolue les Ø noms. Ainsi, nous réussissons à définir un recouvrement des segments sans faire apparaître la notion de recouvrement pendant l'édition de liens.

En effet, considérons une unité de traitement pour laquelle nous ne faisons pas de recouvrement sur les textes objets des modules et classes.

L'ensemble des textes objets à traiter à l'édition de liens est constitué des textes objets des modules et des classes, et du texte objet engendré par le pré-éditeur de liens. L'éditeur de liens n'a plus qu'à implanter consécutivement les textes objets en mémoire, sans avoir à traiter de structure de recouvrement.

Au contraire, avec la première hypothèse, l'éditeur de liens ne pourrait plus implanter les textes objets bout à bout, mais aurait à éditer les différents segments de variables en respectant une structure de recouvrement déterminée par le pré-éditeur de liens. Nous verrons ultérieurement, à propos de recouvrement des textes objets, qu'un tel type d'éditions de liens n'est possible avec l'éditeur de liens du moniteur SIRIS 7 que dans des cas particuliers, peu adaptés à notre pro-

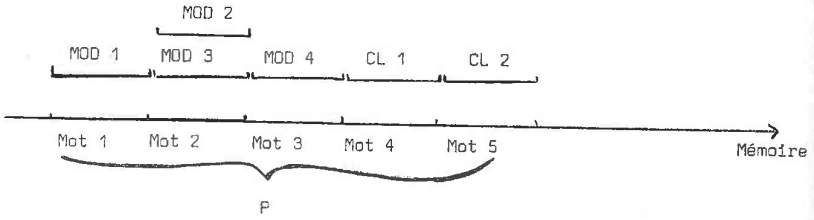
blème.

La seconde solution, consistant à regrouper toutes les variables en un même texte objet engendré par le pré-éditeur de liens, est donc incontestablement la meilleure.

Pour illustrer cette solution, reprenons l'exemple d'unité de traitement donné au § 4.

Supposons que chaque module et classe décalrent des variables dont l'implantation nécessite un mot de mémoire.

Supposons que le pré-éditeur de liens ait déterminé une implantation dans une partie P de la mémoire, comportant 5 mots, et que cette implantation ait l'allure suivante :



Le pré-éditeur de liens générera alors un texte objet reservant 5 mots où les symboles % MOD 1, % MOD 2, % MOD 3, % MOD 4, % CL 1, % CL 2 feront l'objet de définitions externes relativement au début du texte.

L'équivalent en langage METASYMBOL de ce texte objet serait :

```
DEF  % MOD 1, % MOD 2, % MOD3, % MOD 4, % CL 1, % CL 2 (déclaration
     des % noms en définition externe)
```



```
% MOD 1      EQU   %
% MOD 2      EQU   % + 1
% MOD 3      EQU   % + 1
% MOD 4      EQU   % + 2
% CL 1       EQU   % + 3
% CL 2       EQU   % + 4
              RES     5
```

} directives définissant les % noms relativement au compteur d'emplacement (noté % en Métasymbol).

} Les % noms sont donc définis ici comme des adresses relatives.

} Réservation des 5 mots nécessaires pour P.

6 - REMARQUE SUR L'ORDONNANCEMENT DE LA COMPILATION

On peut remarquer que le graphe (E, U) prée pour l'étude du pré-éditeur de liens, permet de déterminer l'ordre de compilation des classes et modules d'une unité de traitement.

Soient en effet deux éléments de E, x et y, tels que : $x \hat{U} y$.

Puisque les identificateurs déclarés dans y sont utilisables explicitement dans x, on ne peut compiler x avant d'avoir compilé y : en effet, lorsque le compilateur rencontre dans x un identificateur non déclaré dans x, il doit pouvoir déterminer dans quelle classe de $\hat{U}(x)$ cet identificateur a été déclaré.

Lorsqu'on connaît le graphe (E, U) d'une unité de traitement, il est facile d'en déduire un ordonnancement de la compilation : on commencera par compiler les points de sortie de (E, U), puis leurs prédecesseurs immédiats et ainsi de suite jusqu'à épuisement de E.

On notera que, lorsque toutes les classes sont compilées l'ordre de compilation des modules est arbitraire, puisque les identificateurs déclarés dans un module ne peuvent être utilisés dans un autre. Cette indépendance des modules les uns vis à vis des autres se traduit dans le graphe (E, U) par le fait que les modules, tous points d'entrée de (E, U) ne peuvent pas être liés par la relation $\hat{U}$.

7 - FICHIER DESCRIPTIF DES CLASSES ET DES MODULES

A toute unité de traitement est associé un fichier implanté sur la mémoire secondaire, appelé fichier descriptif de l'unité de traitement. Ce fich est crée par le compilateur : pour chaque module x, il indique la liste des modu que x appelle. Pour chaque module ou classe, il indique :

- la liste des classe que x "utilise"
- la taille en mots du texte objet de x
- le nombre de mots nécessaires à l'implantation des variables de x
- la liste des sous-programmes de la bibliothèque CIVA appelés par x.

A l'aide de ces renseignements, il est possible au pré-éditeur de li de bâtir les graphes (M, A) et (E, U). D'autre part, la connaissance des tailles des différents constituants de l'unité de traitement lui permet de réaliser les allocations et de déterminer la taille de mémoire nécessaire à l'exécution de l'u té de traitement.

8 - ROLE PARTICULIER DU PRE-EDITEUR DE LIENS DANS LE PROBLEME DU RECOUVREMENT DES TEXTES OBJETS

Nous avons étudié les relations entre compilateur, pré-éditeur de lie et éditeur de liens en ce qui concerne les variables. Si l'on veut mettre en plac une structure de recouvrement sur les textes objets, il est clair que cela ne doi affecter que le pré-éditeur de liens, qui choisira cette structure, et l'éditeur liens, qui la mettra en oeuvre. En effet, il n'est pas souhaitable de prévoir des compilations différentes suivant que l'on utilise ou non le recouvrement. Si on l'utilise, outre la détermination de la structure de recouvrement, travail facil par la ressemblance des conditions de cohabitation pour les segments de variables pour les textes objets, le pré-éditeur de liens devra fournir des efforts supplé- mentaires d'autant plus importants que les résultats de la compilation ne seront directement adaptés. Il faudra notamment que le pré-éditeur de liens introduise u programme gérant les échanges avec la mémoire secondaire. L'insertion de ce progr me dans l'ensemble des textes objets de l'unité, des traitement risque d'être

délicate.

C'est pourquoi, nous étudierons d'abord le problème de la détermination des recouvrements, partie commune avec l'implantation des variables, puis le problème de sa mise en oeuvre, partie spécifique au traitement des textes objets.

5ème PARTIE

---

1 - EXPRESSION DES POSSIBILITES DE RECOUVREMENT DES SEGMENTS DE VARIABLES A L'AIDE D'UN GRAPHE NON ORIENTE

1. 1. - Définition du graphe
1. 2. - Détermination de la matrice associée
1. 3. - Procédé pratique de détermination de la matrice associée
1. 4. - Exemple
1. 5. - Suppression des boucles

2 - DEFINITION D'UNE REPARTITION

3 - PROBLEME TYPE DES SESSIONS D'EXAMENS

4 - PREMIERE EBAUCHE POUR LA DETERMINATION DE REPARTITIONS

5 - GRAPHE REPRESENTATIF D'UNE FAMILLE D'INTERVALLES . GRAPHE DE COMPARABILITE

6 - GRAPHE ASSOCIE A UNE REPARTITION

7 - STRUCTURATION DE L'ENSEMBLE DES REPARTITIONS EN CLASSES D'EQUIVALENCE

8 - LIAISON AVEC LA METHODE DE COLORATION

8. 1. - Cohérence avec la méthode de coloration
8. 2. - Illustration sur quelques exemples
8. 3. - Influence des cliques de (E, R) sur les résultats d'une allocation statique
8. 4. - Cas où (E, R) est un graphe de comparabilité

9 - CAS OU LES LONGUEURS DES SEGMENTS SONT INEGALES

9. 1. - Comparaison avec la méthode du § 4
9. 2. - Défauts de la méthode
9. 2. 1. - Choix d'un ordre arbitraire sur les couleurs
9. 2. 2. - Exemple où la meilleure solution ne correspond pas à une γ-coloration
9. 2. 3. - Evaluation de l'erreur commise

10 - RESUME ET CONCLUSION

5ème PARTIE

SOLUTION DU PROBLEME DE L'IMPLANTATION DES VARIABLES

1 - EXPRESSION DES POSSIBILITES DE RECOUVREMENT DES SEGMENTS DE VARIABLES A L'AIDE D'UN GRAPHE NON ORIENTE

1. 1. - Définition du graphe

E étant l'ensemble des classes et des modules, nous avons vu que pour deux éléments distincts u_1 et u_2 de E, les segments de variables respectifs devaient être disjoints s'il exsitait deux modules x et y, non forcément distincts, tels que :

(I) $x\ A^*\ y$ ou $y\ A^*\ x$

$u_1 \in U^*(x)$ et $u_2 \in U^*(y)$

Cette condition s'exprime à l'aide d'un graphe non orienté (E, R) :

Deux éléments u_1 et u_2 seront joints par une arête si et seulement si la condition (I) est réalisée. Nous écrirons alors indifféremment :

$$u_1\ R\ u_2 \text{ ou } u_2\ R\ u_1$$

1. 2. - Détermination de la matrice associée

Le graphe (E, R) est une manière simple d'exprimer les contraintes de non recouvrement.

Le problème est de le bâtir, connaissant les relations A et U.

Il est peu pratique de considérer le cas de tout couple (u_1, u_2) pour examiner si la condition (I) s'applique : la recherche d'un couple (x, y) de modules satisfaisant la condition risque d'être particuliièrement longue.

Un autre moyen de procéder serait, pour tout couple (x, y) de modules liés par A^*, de joindre par une arête tout élément u_1 de $U^*(x)$ et tout élément u_2 de $U^*(y)$

Cette manière de procéder, déjà plus rapide que la précédente, n'est pas satisfaisante. Elle est redondante, l'existence d'une arête risquant d'être établie plusieurs fois.

Nous allons voir qu'il est intéressant d'exprimer les relations mises en jeu, (c'est-à-dire A^*, U^*, R) à l'aide de matrices booléennes, appelées matrices associées aux graphes.

Soit m le nombre de modules, c le nombre de classes :

$$|M| = m \quad , \quad |C| = c \quad |E| = |M \cup C| = m+c$$

Posons $M = \{x_1, x_2, \ldots, x_m\}$ et $C = \{x_{m+1}, x_{m+2}, \ldots, x_{m+c}\}$

La relation A^* peut se représenter par une (m,m) matrice booléenne [A] telle que :

$$\begin{cases} [A](i,j) = 1 \Longleftrightarrow x_i \; A^* \; x_j \\ [A](i,j) = 0 \Longleftrightarrow \text{non } x_i \; A^* \; x_j \end{cases}$$

De même la relation U^* peut se représenter par une (m+c, m+c) matrice booléenne et la relation R par une (m+c, m+c) matrice [R] symétrique.

Avec ces notations, la condition (I) devient :

Deux éléments x_i et x_j de E sont tels que [R](i, j) = 1 s'il existe un couple (x_k, x_ℓ) (k et ℓ étant inférieurs à m pour que x_k et x_ℓ soient des modules), avec k non forcément distinct de ℓ, tel que :

$$\begin{cases} [A](k,\ell) = 1 \text{ ou } [A](\ell, k) = 1 \\ [U](k, i) = 1 \text{ et } [U](\ell, j) = 1 \end{cases}$$

En notant + et . respectivement les opérations ou et et entre booléens nous pouvons donc écrire que :

$$[R](i, j) = \sum_{\substack{k \in [1, m] \\ \ell \in [1, m]}} \left\{ \left([A](k,\ell) + [A](\ell, k) \right) . [U](k, i) . [U](\ell, j) \right\}$$

En effet, si la condition (I) est réalisée au moins un terme de cette somme vaudra 1, et [R] (i, j) vaudra 1. Si la condition (I) n'est jamais réalisée tous les termes de la somme vaudront 0, et [R] (i, j) vaudra 0.

Notons $[\bar{A}]$ la matrice $[A] + [A]^t$ (c'est-à-dire que nous appliquons l'opérateur + entre un élément de A et un élément de mêmes indices de sa transposée). La matrice $[\bar{A}]$ est symétrique : $[\bar{A}](\ell,k) = \bar{A}(k,\ell)$

L'expression précédente s'écrit

$$[R](i,j) = \sum_{\substack{k \in [1,m] \\ \ell \in [1,m]}} [\bar{A}](\ell,k) . [U](k,i) . [U](\ell,j)$$

ou encore :

$$[R](i,j) = \sum_{\ell \in [1,m]} \left\{ [U](\ell,j) \sum_{k \in [1,m]} \{ [\bar{A}](\ell,k) [U](k,i) \} \right\}$$

Dans la somme la plus à droite, on reconnait l'élément d'indices (ℓ,i) du produit booléen de la matrice $[\bar{A}]$ par la matrice constituée des m premières lignes de [U]. Notons $[\tilde{U}]$ cette matrice et $([\bar{A}] . [\tilde{U}])(\ell,i)$ cet élément.

L'expression devient :

$$[R](i,j) = \sum_{\ell \in [1,m]} [U](\ell,j) . ([\bar{A}] . [\tilde{U}])(\ell,i)$$

ou encore en utilisant la transposée de $[\tilde{U}]$ et en remarquant que pour $\ell \leq m$: $[U](\ell,j) = [\tilde{U}]^t(j,\ell)$

$$[R](i,j) = \sum_{\ell \in [1,m]} [\tilde{U}]^t(j,\ell) . ([\bar{A}] . [\tilde{U}])(\ell,i).$$

On reconnait cette fois l'élément d'indices (j,i) du produit booléen de $[\tilde{U}]^t$ par $[\bar{A}] . [\tilde{U}]$

Donc $[R](i,j) = ([\tilde{U}]^t . [\bar{A}] . [\tilde{U}])(j,i)$

La matrice $[\bar{A}]$ est symétrique. Elle est post multipliée par la matrice $[\tilde{U}]$ et prémultipliée par sa transposée. Le résultat est donc une matrice symétrique. Nous ne pouvions avoir qu'un tel résultat puisque [R], d'après la définition que nous en avons donnée est symétrique.

En profitant des symétries, nous pouvons donc écrire :

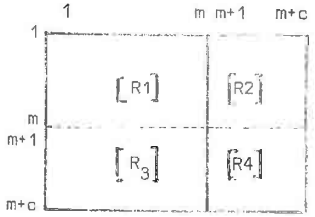
$$\boxed{[R] = [\tilde{U}]^t . [\bar{A}] . [\tilde{U}]}$$

1. 3. - Procédé pratique de détermination de la matrice associée [R]

A cette définition formelle de [R] nous pouvons associer un proc calcul pratique, tenant compte de la symétrie de [R] et de l'allure de

En effet, nous nous sommes arrangés pour que les indices de 1 à n désignent les modules et les indices supérieurs les classes.

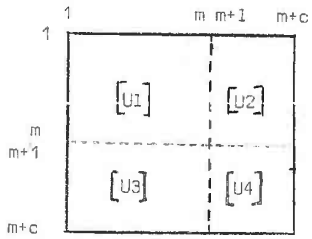
La numérotation des classes et des modules fait qu'il est naturel diviser [R] en quatre sous-matrices [R1] , [R2] , [R3] , R4 de la ma re suivante :



Par suite des symétries : $[R3] = [R2]^t$.

D'autre part la sous matrice [R1] ne met en jeu que des modules. On sai que les segments de variables relatifs à des modules ne doivent pas se recouvrir dans le seul cas où ces modules sont liés par A^*. Il faut don s'attendre à ce que $[R1] = [\bar{A}]$.

Dans ce cas nous n'aurons plus à calculer que [R2] et [R4] . De même subdivisons [U] en 4 sous matrices :



[Ŭ] est constitué des deux sous matrices [U1] et [U2] .

Nous écrirons : $[\check{U}] = [[U1] , [U2]]$

La sous matrice [U1] est relative aux modules. Or nous savons qu'un module ne peut "utiliser" que des classes. Puisque [U1] représente la restriction de U^* à l'ensemble des modules, [U1] est donc égale à la (m,m) matrice identité que nous nommerons [I] .

Donc : $[\check{U}] = [[I] , [U2]]$

Evaluons $[\check{U}]^t . [\bar{A}] . [\check{U}]$

$$[\bar{A}] . [\check{U}] = [[\bar{A}][I] , [U2]] = [[\bar{A}] , [\bar{A}] . [U2]]$$

$$[\check{U}]^t [\bar{A}] . [\check{U}] = \begin{bmatrix} [I] \\ [U2]^t \end{bmatrix} \left[[\bar{A}] , [\bar{A}] . [U2] \right]$$

$$\begin{bmatrix} [\bar{A}] & , & [\bar{A}] . [U2] \\ [U2]^t . [\bar{A}] & , & [U2]^t . [\bar{A}] . [U2] \end{bmatrix}$$

Nous en déduisons

a) $[R1] = [\bar{A}]$ comme nous nous y attendions

b) $[R2] = [\bar{A}] . [U2]$

c) $[R3] = [U2]^t . [\bar{A}]$ (on retrouve effectivement que $R3 = R2^t$)

d) $[R4] = [U2]^t . [\bar{A}] . [U2] = [U2]^t . [R2]$

Pour calculer R, nous commencerons donc par selectionner la sous-matrice [U2] dans [U] . Puis nous calculerons [R2] entièrement. Enfin, il suffira de déterminer une moitié de [R4] (par exemple la diagonale supérieure) en multipliant [R2] par $[U2]^t$.

Cette facilité de calcul nous conduira à représenter les graphes sous forme de matrices associées. Les opérations logiques, qui sont en général parmi les plus brèves sur les calculateurs doivent permettre une détermination de [R] extrêment rapide, même pour des dimensions imposantes.

1. 4. - Exemple

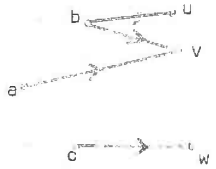
Traitons un exemple simple.

Soit $M = \{a,b,c\}$ $C = \{u,v,w\}$

Soit le graphe (M,A) :

D'où (M,A^*) :

Le graphe (E,U) :



D'où (E,U^*) :

1.4.1 Détermination de [R] par application directe de la règle de coexistance

Une manière de procéder pour construire (E,R) est pour tout couple de modules (x,y) non forcément distincts, liés par A^*, de joindre par une arête tout élément de $U^*(x)$ et tout élément de $U^*(y)$.

Cet algorithme est résumé dans le tableau suivant qui a autant de lignes que de couples de modules possibles :

module x	module y	liés par A^*?	$U^*(x)$	$U^*(y)$	arêtes obtenues
a	a	oui	a,v	a,v	a-a, a-v, v-v
a	b	oui	a,v	b,u,v	a-b,a-u,a-v,v-b,v-u,v-v
a	c	oui	a,v	c,w	a-c,a-w,v-c,v-w
b	b	oui	b,u,v	b,u,v	b-u,b-v,u-v,b-b,u-u,v-v
b	c	non	b,u,v	c,w	
c	c	oui	c,w	c,w	w-w,c-w,c-c

D'où la matrice [R] (nous n'écrirons que la triangulaire supérieure)

[R] =	a	b	c	u	v	w
a	1	1	1	1	1	1
b		1	0	1	1	0
c			1	0	1	1
u				1	1	0
v					1	1
w						1

Les couples de segments qui peuvent être mis en recouvrement sont donc :

(b,w), (b,c), (u,w), (c,u).

1.4.2 Méthode matricielle

Résolvons maintenant le problème par les procédés matriciels

$$[A] = \begin{array}{c} \\ a \\ b \\ c \end{array}\!\!\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \quad [\bar{A}] = [A] + [A^t] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

En comparant avec la valeur de [R] déjà trouvée, on retrouve que R1 = A

$$[U2] = \begin{array}{c} \\ a \\ b \\ c \end{array}\!\!\begin{array}{c} \begin{array}{ccc} u & v & w \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

(par définition : [U2] (i,j) = 1 ⇐ le module i U^* la classe j)

D'où $[R2] = [\tilde{A}] . [U2]$

$$= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

En comparant avec la valeur de $[R]$ déjà trouvée, on constate qu'on trouve le même résultat pour $[R2]$

Enfin $[R4] = [U2]^t [R2]$. On ne détermine que la triangulaire supérieure.

$$[R4] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ X & 1 & 1 \\ X & X & 1 \end{bmatrix}$$

On retrouve la valeur correcte de $[R4]$.

1. 5. - Suppression des boucles

Les procédés employés font apparaître des boucles dans le graphe (E,R). Ces boucles n'étant pas utiles dans la suite de notre étude, nous déciderons de les supprimer. Nous appellerons simplement (E,R) le graphe obtenu par le procédé matriciel et amputé de ses boucles.

2 - DEFINITION D'UNE REPARTITION

L'ensemble des contraintes de non recouvrement est exprimé par le graphe (E,R). D'autre part tout point x_i est affecté d'une longueur ℓ_i qui est la taille en mots du segment de variables. Nous cherchons à déterminer l'origine des $|E|$ segments de variables de manière à respecter les contraintes de non recouvrement. Soit $|E| = n$. Appelons $O(x_i)$ l'origine du segment de variables correspondant à x_i.

Si deux points x_i et x_j de E sont réunis par une arête, les segments correspondants ne sont pas en recouvrement. C'est-à-dire que l'une des deux inégalités suivantes est respectée :

$$\text{III} \quad \begin{cases} O(x_i) + \ell(x_i) \leqslant O(x_j) \\ O(x_j) + \ell(x_j) \leqslant O(x_i) \end{cases}$$

Définition : Soit $E = \{x_1, x_2, \ldots, x_n\}$

Chaque sommet x_i de E étant muni d'une longueur entière $\ell(x_i)$, nous appellerons répartition pour (E,R,ℓ) un ensemble de n valeurs entières $O(x_1) \ldots, O(x_n)$ telles que si x_i et x_j sont joints par une arête, une des deux inégalités précédentes est satisfaite.

Si nous considérons une unité de traitement, toute répartition pour (E,R,ℓ) donne une implantation des variables satisfaisant les contraintes. L'unité de traitement est donc viable. Par exemple, si nous posons $O(x_1) = 0$, puis $O(x_{i+1}) = O(x_i) + \ell(x_i)$ pour i allant de 1 à n-1, nous implantons les segments bout à bout et l'une des deux inégalités (III) est toujours respectée pour tout couple de points de E (donc pour tout couple de points liés par une arête).

Il est intéressant de définir la longueur d'une répartition. Soit V une répartition pour (E,R,ℓ).

L'adresse de mémoire la plus haute d'un mot alloué aux variables est : $\max_{i \in [1,n]} \{O(x_i) + \ell(x_i)\} - 1$, l'adresse la plus basse : $\min_{j \in [1,n]} O(x_j)$

Toutes les variables étant réservées dans un espace de mémoire contigu, le nombre total de mots réservés aux variables est :

$$\max_{i \in [1,n]} \{O(x_i) + \ell(x_i)\} - \min_{j \in [1,n]} O(x_j)$$

Cette quantité dépend de V. Nous l'appellerons longueur de la répartition V et la noterons L (V).

En résumé, le problème se pose dans les termes suivants :
Etant donné un graphe (E,R) et une longueur ℓ définie sur E, à valeurs entières strictement positives, on cherche une répartition pour (E,R,ℓ) de longueur aussi petite que possible.

3 - PROBLEME TYPE DES SESSIONS D'EXAMENS

Un problème bien connu dit problème "des sessions d'examens", se pose à peu près dans les mêmes termes que celui que nous avons à résoudre une session d'examen se décompose en n épreuves de même durée (prenons par exemple, une durée t). Certaines épreuves peuvent se dérouler simultanément, d'autres ne le peuvent pas. Le problème consiste, connaissant les possibilités de simultanéité entre les épreuves, d'organiser la session d'examen en un laps de temps minimum. Ce problème est tout à fait semblable au nôtre

prenons pour E l'ensemble des épreuves. Deux épreuves seront liées par R si elles ne peuvent avoir lieu simultanément. Pour tout x de E nous poserons $\ell(x) = t$. Le problème consiste bien à chercher une répartition pour (E,R,ℓ) de longueur minimum.

Ce problème admet une solution classique : considérons une coloration de (E,R) à l'aide d'un nombre minimum p de couleurs (on dit que à une coloration (E,R) si à chaque point de E est affectée une couleur telle manière que deux sommets adjacents n'aient jamais la même couleur. Le nombre minimum de couleurs nécessaires pour colorer le graphe est appelé nombre chromatique).

Soient $C_1, C_2, \ldots, C_p$ les couleurs utilisées. Soient $E_1, E_2, \ldots$ les sous ensembles de E peints respectivement en $C_1, C_2, \ldots, C_p$. Les E_i forment une partition de E. Deux points d'un même sous ensemble E_i ne jamais adjacents. Il est donc possible de dérouler en simultanéité inte le les épreuves d'un même E_i. Nous déciderons par exemple que les épreu de E_i se dérouleront dans l'intervalle de temps $]t(i-1), ti[$. Des épr ves de deux sous ensembles E_i et E_j différents seront non simultanées. réalise ainsi la session d'examen en un temps tp et en respectant les c traintes. Inversement, montrons qu'une session plus courte est irréali ble[1] : considérons une session S_1 de durée inférieure à tp et plaçons son début à l'instant 0. Pour toute épreuve x de E, appelons k_x l'entier tel que le début de x soit dans l'intervalle $]t(k_x-1), tk_x[$. Pour tous x de E, déplaçons leur début à l'instant $t(k_x-1)$. Nous obtenons ainsi session S_2 de durée inférieure ou égale à la précédente, dont la durée au plus $t(p-1)$. Vérifions qu'elle respecte les contraintes si S_1 les r pecte : si deux épreuves x et y ne peuvent être simultanées, elles sont telles dans la session S_1 que k_x et k_y sont différents. (En effet si $k_x = k_y$, leurs débuts sont espacés d'une durée inférieure à t ; comme la durée d'une épreuve est t il existerait un intervalle de temps non nul elles seraient simultanées).

Donc dans la session S_2, x et y sont disjointes. S_2 respecte les c traintes. Si on colore avec une couleur c_i les points de E correspondant aux épreuves se déroulant dans S_2 dans l'intervalle de temps $]t(i-1), t$

(1) Ceci est immédiat si l'on suppose a priori, comme il est fait souve que les épreuves sont soit totalement disjointes, soit totalement e recouvrement. Cette hypothèse n'est cependant pas indispensable.

on réalise une coloration de (E,R) à l'aide de (p-1) couleurs au plus, ce qui contredit le fait que (E,R) est p chromatique. Une session de durée inférieure à tp ne peut donc exister.

4 - PREMIERE EBAUCHE POUR LA DETERMINATION DE REPARTITIONS

Dans le cas où les longueurs affectées aux points de E sont égales, nous connaissons une solution au problème de la détermination d'une répartition. Dans le cas général où les longueurs sont variables, cette solution ne peut s'appliquer.

Elle permet cependant de réaliser une approximation par la méthode suivante : soit une coloration de (E,R) en p couleurs et $E_1, E_2, \ldots, E_p$ les sous ensembles de E de points de même couleur.

Pour j allant de 1 à p, posons : $L_j = \max_{x \in Ej} \{\ell(x)\}$.

Pour tout x de E, posons, si x appartient à Ej :

$$\begin{cases} 0^-(x) = \sum_{k=1}^{j-1} L_k & \text{si } j > 2 \\ 0(x) = 0 & \text{si } j = 1. \end{cases}$$

Il est clair que l'on obtient ainsi une répartition V pour (E,R,ℓ), de longueur

$$L(V) = \sum_{i=1}^{p} L_i .$$

Remarquons que, connaissant le nombre chromatique de (E,R) sans connaître une coloration, on peut majorer L(V) : $L(V) \leq p. \max_{x \in E} \{\ell(x)\}$

Cette méthode est cependant abusive. En effet, elle revient à implanter systématiquement de manière disjointe des segments correspondant à des points de E de couleurs différentes. Or deux points de couleurs différentes ne sont pas forcément adjacents. Prenons les points de plus grandes longueurs de E_i et E_{i+1} ; nous avons implanté les segments correspondants bout à bout ; il se peut que ces segments puissent en fait se recouvrir. La répartition obtenue est donc susceptible d'être contractée. Pour cette raison, nous sommes amenés à chercher une meilleure solution que la précédente, qui est trop grossière.

La théorie des graphes met en évidence les propriétés de certains graphes, appelés graphes représentatifs d'une famille d'intervalles et graphes de

comparabilité. Nous nous proposons d'abord de constater que ces graphe peuvent donner une certaine image d'une répartition, et qu'il est pos d'en déduire d'autres graphes, que nous appelerons graphes associés à répartitions, tels que la longueur de leur plus grand chemin est lié longueur des répartitions. Nous verrons ensuite qu'il est possible de la synthèse de la notion de graphe associé et de la méthode de colora pour en déduire une solution plus fine au problème de la détermination répartitions de longueur minimale.

5 - GRAPHE REPRESENTATIF D'UNE FAMILLE D'INTERVALLES GRAPHE DE COMPARABILI

Le défaut qui avait été constaté dans la méthode précédente é de ne pas profiter à fond des possibilités de recouvrement. Ce paragra a pour but de donner une interprétation précise de cette perte.

En théorie des graphes existe la notion de graphe représentatif d famille d'intervalles : soit I un ensemble d'intervalles fermés de R, peut construire un graphe non orienté dont les sommets sont les points et où deux sommets sont joints par une arête si et seulement si les de intervalles correspondants ne sont pas disjoints. Le graphe obtenu est représentatif de I.

Considérons une répartition V pour (E,R,ℓ) et la famille d'interv $\{[0(x_1), 0(x_1) + \ell(x_1)], \ldots, [0(x_n), 0(x_n) + \ell(x_n)]\}$. Il est né de bâtir le graphe représentatif de cette famille : il exprime l'usage l'on a fait des possibilités de recouvrement. Puisqu'à chaque x_i de E c respond un intervalle, nous désignerons par E l'ensemble des sommets de graphe. Les graphes représentatifs d'intervalles ont des propriétés pa culières. Notamment leur graphe complémentaire[1] est un graphe de com bilité. Complémentons le graphe représentatif de notre famille d'inter Nous obtenons un graphe (E,R'). Si deux points de E sont adjacents dan ils sont adjacents dans (E,R') mais la réciproque est fausse : (E,R) es graphe partiel de (E,R').

Le principale propriété d'un graphe de comparabilité est qu'on peu orienter ses arêtes de manière à en faire le graphe d'une relation d'or

(1) (X,Γ_1) est dit complémentaire de (X,Γ_2) si pour tout couple (x,y) points de X : $x \Gamma_1 y \iff$ non $x \Gamma_2 y$

Deux sommets x et y de (E,R') adjacents correspondent à des intervalles disjoints. L'une des deux inégalités suivantes est donc vraie :

$$\begin{cases} (a) & 0(x) \geq 0(y) + \ell(y) \\ (b) & 0(y) \geq 0(x) + \ell(x) \end{cases}$$

Pour définir une relation d'ordre T il suffit d'orienter l'arête x y de x vers y si, par exemple l'inégalité (b) est vérifiée. Il est clair que l'on vérifie ainsi l'antisymétrie. La transitivité s'établit facilement : soient trois points x, y, z tels que x T y et y T z.

Alors : $0(z) \geq 0(y) + \ell(y)$ et $0(y) \geq 0(x) + \ell(x)$. Comme $\ell(y)$ est positif : $0(z) \geq 0(x) + \ell(x)$. Cette inégalité montre que les intervalles correspondant à z et x sont disjoints ; et que ces deux points sont donc adjacents dans (E,R'). Comme nous orientons l'arête z - x de x vers z, nous vérifions bien que : x T z.

Le graphe (E,T) obtenu donne une idée précise de la répartition : non seulement il donne l'organisation des recouvrements entre segments, mais encore il reflète l'ordre existant entre les différents segments. En effet x T y signifie que le segment y est situé à des adresses mémoire plus hautes que x, Nous allons voir de plus que la longueur du plus grand chemin de ce graphe est liée à la longueur de la répartition V à partir de laquelle il a été bâti.

Proposition 1 : Soit L_o la longueur du plus grand chemin par les sommets de (E,T). Alors : $L(V) \geq L_o$.

Démonstration : Soit $\{x_1, x_2, \ldots, x_q\}$ la longueur du plus grand chemin par les sommets de (E,T)

Par définition de T, nous avons : $0(x_{i+1}) \geq 0(x_1) + \ell(x_1)$ pour $i \in [1, q-1]$.

D'où
$$\begin{cases} 0(x_2) \geq 0(x_1) + \ell(x_1) \\ 0(x_3) \geq 0(x_2) + \ell(x_2) \\ \text{----------} \\ 0(x_q) \geq 0(x_{q-1}) + \ell(x_{q-1}) \end{cases}$$

En sommant membre à membre, il vient :

$$0(x_q) \geq 0(x_1) + \sum_{i=1}^{q-1} \ell(x_i)$$

Puis après avoir ajouté $\ell(x_q)$ à chaque membre :

$$O(x_q) + \ell(x_q) \geq O(x_1) + \sum_{i=1}^{q} \ell(x_i)$$

Or L_o est précisement égal à $\sum_{i=1}^{q} \ell(x_i)$

D'où $O(x_q) + \ell(x_q) - O(x_1) \geq L_o$

Enfin, en majorant $O(x_q) + \ell(x_q)$ par $\max_{x_i \in E} \left[O(x_i) + \ell(x_i) \right]$ et en minorant $O(x_1)$ par $\min_{x_j \in E} \left[O(x_j) \right]$, on obtient :

$$L(V) = \max_{x_i \in E} \left[O(x_i) + \ell(x_i) \right] - \min_{x_j \in E} \left[O(x_j) \right] \geq L_o$$

La quantité L_o se présente donc comme un minorant de L(V). Considérons maintenant le graphe partiel de (E,T) obtenu en retirant de (E,T) les arcs x - y tels qu'il existe une arête x - y dans (E,R') et pas d'arête x-y dans (E,R). On obtient ainsi un graphe orienté (E,S) où deux points seront joints par un arc si et seulement s'ils sont joints par une arête dans (E,R') (nous avons vu en effet que (E,R) était un graphe partiel de (E,R')).

(E,S) étant un graphe partiel de (E,T) la longueur L_1 de son plus grand chemin est au plus égale à la longueur L_o du plus grand chemin de (E,T).

L_1 est donc aussi un minorant de L(V).

Si nous nous rappelons l'énoncé II (cf partie 2 §7), nous pouvons espérer implanter les segments sur L_1 mots de mémoire, en déduisant de (E,S) une nouvelle répartition pouvant être de longueur inférieure à L_o, en effet le graphe (E,S) traduit bien toutes les contraintes de non recouvrement ; d'autre part, en tant que graphe partiel d'un graphe transitif, il est sans circuit. Nous obtiendrions ainsi un résultat meilleur que ne le laissait envisager la proposition 1.

6 - GRAPHE ASSOCIE A UNE REPARTITION

Le graphe (E,S) peut être obtenu, à partir de la répartition que l'on s'est donnée de manière plus directe que celle décrite précédemment : soit V une répartition pour (E,R,ℓ) ; il suffit de définir la relation S entre deux points x et y de E de la manière suivante :

$$x \; S \; y \Longleftrightarrow x \; R \; y \quad \underline{\text{et}} \quad O(y) \geq O(x) + \ell(x)$$

Le graphe (E,S) ainsi obtenu est dit graphe associé à la répartition V.

D'après l'inégalité $L_1 \leq L_o$ obtenu plus haut, on peut énoncer :

Proposition 2 : Soit V une répartition pour (E,R,ℓ) et (E,S) son graphe associé. Soit L_1 la longueur du plus grand chemin par les sommets de (E,S). Alors $L(V) \geq L_1$.

Nous espérons atteindre l'égalité $L(V) = L_1$. Nous allons en tirer une propriété caractéristique des répartitions pour (E,R,ℓ) de longueur minimale :

Proposition 3 : Toute répartition V pour (E,R,ℓ) est de longueur minimale si et seulement si sa longueur est égale à la longueur L_1 du plus grand chemin par les sommets de son graphe associé.

Démonstration :

a) Soit $L(V) = L_1$. D'après la proposition 2, on ne peut pas trouver de répartition de longueur inférieure à celle de V. V est donc de longueur minimale.

b) Inversement, montrons que si V est de longueur minimale, elle satisfait à $L(V) = L_1$.

Pour cela, donnons nous une répartition V' telle que $L(V') \geq L_1$ et montrons que V' n'est pas de longueur minimale :

Soit (E,S) le graphe associé à V'. Pour tout point x de E, notons m (x) la longueur du plus grand chemin par les sommets de (E,S), d'extrémité x (sommet x inclus). Posons $A(x) = m(x) - \ell(x)$.

Soient x et y deux sommets adjacents dans (E,R). D'après la définition de (E,S), ils sont joints par un arc : ou bien x S y, ou bien y S x. Le sommet x (respectivement y) allonge le plus grand chemin d'extrémité y (resp x) d'une longueur ℓ(x) (resp. ℓ(y)).

Nous pouvons donc écrire :

$$\begin{cases} \text{ou} & m(y) \geq m(x) + \ell(y) \\ \text{ou} & m(x) \geq m(y) + \ell(x) \end{cases}$$

soit encore, en utilisant les définitions de A (x) et A (y) :

$$\begin{cases} \text{ou} & A(y) \geq A(x) + \ell(x) \\ \text{ou} & A(x) \geq A(y) + \ell(y) \end{cases}$$

La fonction A satisfait donc à la définition d'une répartition pour (E,R,ℓ). Nous obtenons ainsi une nouvelle répartition V". Evaluons sa longueur :

$$L(V'') = \max_{x \in E} \left[A(x) + \ell(x)\right] - \min_{y \in E} \left[A(y)\right]$$

Or $\max_{x \in E} \left[A(x) + \ell(x)\right] = \max_{x \in E} \left[m(x)\right] = L_1$, puisque L_1 est la longueur du plus grand chemin de (E,S).

D'autre part $\min_{y \in E} \left[A(y)\right] = \min_{y \in E} \left[m(y) - \ell(y)\right] = 0$, ce minimum étant atteint pour les points d'entrée de (E,S) pour lesquels $m(y) = \ell(y)$. Nous vérifions donc $L(V'') = L_1$ et avons ainsi exhibé une répartition de longueur inférieure à celle de V', et précisément de longueur L_1.

7 - STRUCTURATION DE L'ENSEMBLE DES REPARTITIONS EN CLASSES D'EQUIVALENCE

La notion de graphe associé permet de partitionner l'ensemble des répartitions pour (E,R,ℓ) en classes d'équivalence : deux répartitions seront dites équivalentes si elles ont même graphe associé. Il est clair que si le nombre de répartitions possibles n'est pas fini, il existe par contre autant de classes équivalences qu'il y a d'orientations possibles de (E,R) de manière à en faire un graphe sans circuit et que ce nombre d'orientations est fini.

La partie b de la démonstration de la proposition 3 est particulièrement intéressante : d'une part elle permet, partant d'une répartition quelconque d'une classe d'équivalence, de construire une répartition de cette classe de longueur minimale. Intuitivement, ce procédé correspond au "tassage" d'une répartition : les segments sont distribués dans un certain ordre (c'est l'ordre défini sur le graphe associé), et on cherche, sans bouleverser cet ordre (c'est-à-dire qu'un segment x situé à droite d'un segment y qu'il ne peut recouvrir restera toujours à droite de y) à tasser au maximum la distribution des segments sur elle-même. D'autre part cette partie de démonstration permet, étant donné un graphe associé, autrement dit une orientation de (E,R), de trouver un des éléments les plus intéressants pour notre problème : une répartition de longueur minimale de la classe d'équivalence. On peut donc énoncer :

Proposition 4 : De tout graphe sans circuit (E,S) déduit de (E,R) en orientant ses arêtes, on peut déduire une répartition dont la longueur est la longueur du plus grand chemin par les sommets de (E,S).

Notre problème se pose donc sous un nouvel aspect : trouver une orientation de (E,R) optimale. Il faut, parmi toutes les orientations possibles de (E,R), trouver celles qui conduisent au plus grand chemin par les sommets aussi petit possible. Nous pourrons ensuite en déduire facilement une répartition de longueur minimale.

8 - LIAISON AVEC LA METHODE DE LA COLORATION

8. 1. Cohérence avec la méthode de coloration

Nous nous sommes momentanément éloignés de la solution par coloration de (E,R). Il serait intéressant de reconnecter la méthode des graphes associés avec la méthode de coloration. En raisonnant sur les graphes sans circuit déduits par orientation de (E,R) nous devons trouver le même résultat que celui donné par la coloration.

Pour cela donnons nous un graphe (E,S) sans circuit qui est une "meilleure" orientation de (E,R) :
si nous supposons que tous les points de E sont affectés d'une même longueur ℓ_o, cette orientation de (E,R) sera telle qu'il n'existe pas de graphe (E,S) donné par une autre orientation dont le plus grand chemin comporte moins de sommets que le plus grand chemin de (E,S).

Supposons que le plus grand chemin du graphe (E,S) comporte q sommets. D'après la proposition 4, nous pouvons bâtir une répartition de longueur $q.\ell_o$. Si p est le nombre chromatique de (E,R) la méthode de coloration donne pour résultat $p\ell_o$. Pour vérifier la cohérence des deux méthodes, montrons que q = p.

Montrons d'abord que $q \neq p$ est impossible.
Pour cela définissons pour (E,S) une fonction ordinale : pour tout x de E nous poserons $f(x) = k$, k étant le nombre de sommets du plus grand chemin d'extrémité x (x inclus). k peut prendre une valeur entière de 1 à q. Appelons E_i le sous ensemble de E constitué des points x tels que : $f(x) = i$. Il y a q sous ensembles E_i : $E = \bigcup_{i=1}^{q} E_i$

Il est clair que s'il existe un arc entre deux sommets x et y ces deux sommets ne peuvent appartenir au même sous ensemble E_i (les plus grands chemins d'extrémités x et y ne peuvent avoir la même longueur).

D'autre part un sommet ne peut appartenir à deux sous ensembles E_i et E_j différents. Nous avons donc réalisé une partition de E en q sous ensembles intérieurement stables[1]. Comme toute paire de sommets jointe par un arc dans (E,S) est jointe par une arête dans (E,R), l'ensemble des sommets de (E,R) est décomposé aussi en q sous ensembles intérieurement stables. (E,R) est donc q chromatique. On ne peut donc supposer que $q < p$.

(1) C'est-à-dire que deux points d'un sous ensemble ne sont jamais joints par un arc.

D'autre part il est facile de vérifier que l'orientation de (E,R) peut donner un graphe sans circuit à p sommets. Considérons une répartition obtenue par la méthode de coloration : dans cette méthode, on colore (E,R) puis on définit une relation d'ordre total sur les couleurs cette relation consiste à décider dans le problème des sessions d'examen que les épreuves correspondant aux sommets peints en une couleur auront lieu avant les épreuves correspondant aux sommets peints en une autre couleur. L'ordre défini sur les couleurs est tout à fait arbitraire. Suivons cette démarche pour orienter (E,R) : commençons par définir une coloration de (E,R) puis choisissons arbitrairement un ordre sur les couleurs. Orientons ensuite toute arête x - y de (E,R) de x vers y si la couleur de x est "plus grande" que celle de y. Il est clair que le graphe obtenu ne peut comporter un circuit. D'autre part une couleur ne peut pas se répéter sur les sommets d'un même chemin. Tout chemin comporte donc au plus p sommets. Nous obtenons donc ainsi, par orientation de (E,R), un graphe sans circuit dont tout chemin comporte au plus p sommets. La longueur du plus grand chemin par les sommets de ce graphe est $p.\ell_0$. D'après la proposition 4, nous savons construire une répartition pour (E,R,ℓ) de longueur $p.\ell_0$ qui nous donne le même résultat qu'avec la méthode directe de coloration.

Puisque nous avons vu précédemment qu'il n'était pas possible d'obtenir une répartition de longueur inférieure, nous constatons que la méthode de coloration est cohérente avec la méthode consistant à orienter (E,R). Nous pouvons exprimer cette cohérence en énonçant :

Proposition 5 : Si (E,R) est p-chromatique, les graphes sans circuit déduits de (E,R) par orientation de ses arêtes, comportent un chemin passant par p sommets au moins. De plus il existe des orientations de (E,R) donnant des graphes sans circuit, telles que tout chemin d'un graphe ainsi obtenu comporte au plus p sommets.

8. 2. - Illustrations sur quelques exemples :

8.2.1 : (E,R) est sans arête.

Deux sommets ne sont jamais adjacents. (E,R) est donc 1-chromatique. Dans ce cas dégénéré, il est évident que toute "orientation" des arêtes de (E,R) conduit à un graphe sans circuit dont tout chemin comporte un seul sommet.

La détermination d'une répartition à l'aide de la proposition 4 donne le même résultat que la méthode de coloration : tous les segments sont sans recouvrement.

8.2.2 (E,R) est complet[1]

(E,R) étant complet, il est |E| chromatique.

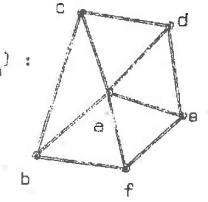
Un graphe sans circuit obtenu par orientation de (E,R) sera complet. D'après la proposition 5, il comportera un chemin passant par ses |E| sommets. Nous obtenons ainsi une répartition telle qu'aucun segment ne sera en recouvrement avec un autre : les segments seront implantés consécutivement en mémoire.

Nous pouvons remarquer que nous retrouvons le théorème de Redeï : Tout graphe orienté complet comporte un chemin hamiltonien. Nous retrouvons donc ce théorème dans le cas des graphes sans circuit.

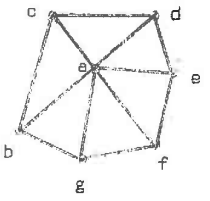
8.2.3 Comparons les deux exemples suivants :

a) E_a = (a,b,c,d,e,f) b) E_b = (a,b,c,d,e,f,g)

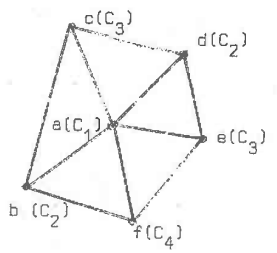
(E_a,R_a) :

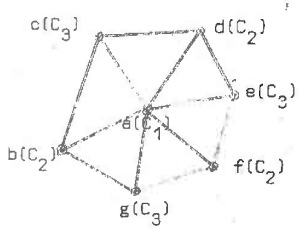


(E_b,R_b) :



On constate facilement que si la coloration de (E_b, R_b) avec 3 couleurs est possible, celle des (E_a, R_a) ne l'est pas. Considérons des colorations réalisées avec des couleurs notées C_1, C_2, C_3 pour (E_b, R_b) et une couleur C_4 supplémentaire pour (E_a, R_a) :



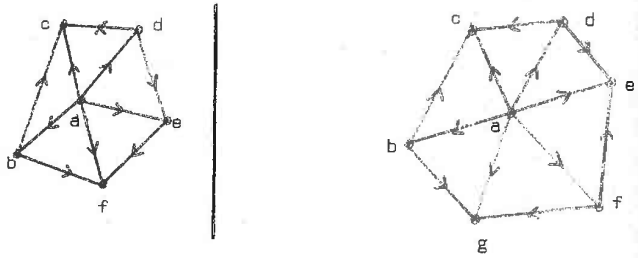


(1) Nous dirons qu'un graphe est complet si deux sommets quelconques sont toujours joints par une arête (s'il s'agit d'un graphe orienté) ou par un arc (s'il s'agit d'un graphe non orienté).

Choisissons un ordre arbitraire des couleurs :

$$C_1 > C_2 > C_3 > C_4$$

Nous en déduisons les orientations de (E_a, R_a) et (E_b, R_b) :



Dans ces deux graphes, les chemins maximaux sont :

{a, b, c}	{a, b, g}
{a, b, f}	{a, b, c}
{a, d, c}	{a, d, c}
{a, d, e, f}	{a, d, e}
	{a, f, e}
	{a, f, g}

On constate que ces chemins comportent au plus 4 sommets pour le graphe déduit de (E,R_a) (en effet le nombre chromatique de (E_a, R_a) est 4) et 3 sommets pour le graphe déduit de (E_b, R_b) (le nombre chromatique de (E_b, R_b) est 3).

Soit ℓ_o la longueur $\ell(x)$ de tout sommet x de E_a ou E_b.

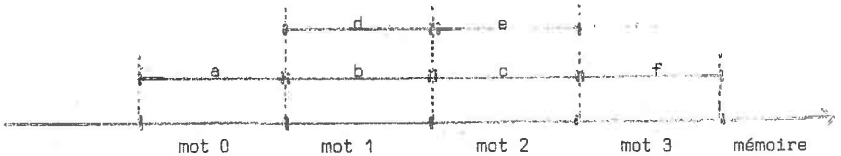
Pour déterminer des répartitions pour (E_a, R_a, ℓ) et (E_b, R_b, ℓ) nous allons appliquer la technique de la partie b de la démonstration de la proposition 3 : nous prendrons pour origine d'un segment la longueur du plus grand chemin par les sommets d'extrémité le sommet représentant ce segment.

Ces longueurs, pour les différents sommets de E_a et E_b, sont :

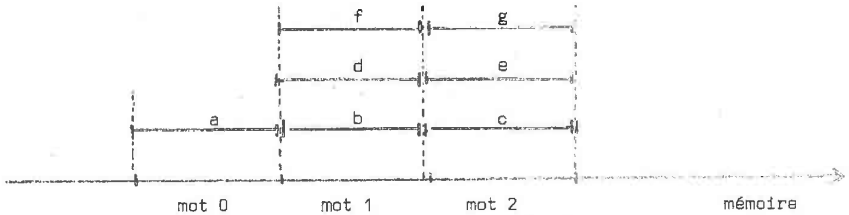
a : 0	a : 0
b : ℓ_o	b : ℓ_o
c : $2\ell_o$	c : $2\ell_o$
d : ℓ_o	d : ℓ_o
e : $2\ell_o$	e : $2\ell_o$
f : $3\ell_o$	g : $2\ell_o$

Prenons pour simplifier, $\ell_o = 1$. Les répartitions ainsi déterminées, correspondent aux implantations suivantes en mémoire centrale :

- Pour l'exemple a :



- Pour l'exemple b :



On constate facilement, pour les deux exemples, que l'implantation obtenue est effectivement compatible avec les contraintes de non recouvrement exprimées par (E_a, R_a) et (E_b, R_b).

8. 3. - Influence des cliques de (E,R) sur les résultats d'une allocation statique

Si l'on établit une comparaison entre les deux exemples, on voit que l'exemple a conduit à un encombrement de la mémoire plus important que l'exemple b, bien que $|E_b|$ soit supérieur à $|E_a|$ et que les graphes traduisant le non recouvrement aient la même allure. Voyons à quoi cette différence est due, en même temps, réexaminons un exemple vu dans la 2ième partie.

Nous avons déjà vu (cf 2ième partie), en comparant les conséquences des gestions statiques et dynamiques, qu'une gestion dynamique pouvait conduire à un encombrement plus important de la mémoire centrale qu'une gestion statique. L'exemple étudié mettait en jeu trois classes. Alors qu'une gestion dynamique exigeait la présence en mémoire des variables de deux parmi ces trois classes, une gestion statique exigeait la présence des variables des trois classes car les variables de deux des trois classes ne pouvaient se recouvrir. Autrement dit, ces trois classes formaient un sous graphe complet de (E,R)[1]. Après orientation de (E,R), la présence de cette clique

(1) Un sous graphe complet s'appelle aussi une clique.

se traduit, d'après le théorème de Redei, par l'existence d'un chemin hamiltonien constitué des trois classes dont les variables, par conséquent, seront implantées sans recouvrement en mémoire d'après l'algorithme que nous avons choisi.

De même, une gestion statique conduisant à des résultats dissemblables entre l'exemple a et l'exemple b, nous pouvons nous demander si une gestion dynamique donnerait le même type d'anomalie.

Si une méthode de gestion dynamique conduit à faire cohabiter à un instant donné n segments en mémoire centrale, alors deux quelconques parmi ces n segments ne peuvent se recouvrir. Ces n segments sont donc deux à deux liés par la relation R. Ils forment donc une clique de (E,R).

Soit N le nombre maximum de segments simultanément présents en mémoire avec une gestion dynamique.

Notons $\mathcal{C}$ un sous ensemble de sommets de E formant une clique.

Posons $M = \max_{\mathcal{C} \subseteq E} |\mathcal{C}|$

Nous avons $N \leqslant M$

Dans les exemples a et b, nous avons M = 3.

Au contraire une méthode de gestion statique fait cohabiter au plus γ segments, γ étant le nombre chromatique de (E,R).

Il est clair que γ est supérieur ou égal à M : en effet il faut M couleurs pour colorer les sommets d'une clique de M points.

Dans les exemples a et b, nous avons respectivement : $\gamma = 4$ et $\gamma =$

Pour qu'une gestion statique donne des résultats aussi bons qu'une gestion dynamique il faut donc

condition 1) que N = M (ce qui n'était pas le cas dans l'exemple de la partie 2).

condition 2) que $\gamma = M$. Le graphe (E,R) est dit γ - parfait.

(E_a, R_a) n'est pas γ - parfait, (E_b, R_b) est γ - parfait, ce qui explique la dissemblance des résultats obtenus.

8. 4. - Cas où (E,R) est graphe de comparabilité.

Nous avons vu au § 5 que si (E,R) est un graphe de comparabilité, il est possible d'orienter ses arêtes de manière à obtenir une relation d'ordre sur les sommets. Cette relation est donc transitive. Soit (E,T) le graphe obtenu.

Tous les sommets d'un même chemin feront par conséquent partie d'une même clique. Plaçons nous toujours dans le cas où tous les sommets sont affectés d'une même longueur ℓ_o.

Si le plus grand chemin de (E,T) comporte q sommets, ces sommets forment une clique d'après la remarque ci-dessus et, d'après la proposition 5, il est possible de construire une répartition de longueur $\gamma \times \ell_o$, γ étant le nombre chromatique de (E,R). La proposition 4 indique que l'on peut construire une répartition de longueur égale à la longueur du plus grand chemin par les sommets de (E,T) soit $q \times \ell_o$. Nous obtenons que γ est égal à q, et donc que (E,R) est γ - parfait. Nous retrouvons ainsi une propriété connue des graphes de comparabilité : ils sont γ - parfaits.

Ainsi, dans le cas où (E,R) est un graphe de comparabilité, la condition 2 du § 8.3 est remplie.

9 - CAS OU LES LONGUEURS DES SEGMENTS SONT INEGALES

Jusqu'à présent nous n'avons donné qu'une ébauche pour la détermination d'une répartition (cf. § 4). Par contre, dans le cas où les longueurs des segments sont égales, le problème est tout à fait résolu.

Notre méthode, qui consiste, parmi toutes les orientations possibles de (E,R), à trouver celles qui donnent un plus grand chemin par les sommets aussi petit que possible, s'applique particulièrement bien si les longueurs des sommets sont égales : elle équivaut à chercher une orientation de (E,R) telle que le chemin qui comporte le plus de sommets, en comporte un minimum. C'est de ce critère que nous sommes partis pour orienter (E,R) après coloration.

Dans le cas où les longueurs sont inégales, on ne peut plus affirmer qu'un chemin sera d'autant moins long qu'il comportera moins de sommets. Les "critères" minimum de sommets" et "minimum de longueur" ne coïncident plus.

Nous ne connaissons pas, dans ce cas, de méthode donnant une solution optimale. Il semble d'ailleurs délicat d'élaborer une méthode raisonnablement envisageable, car elle devrait être assez puissante pour résoudre dans un cas particulier (cas des longueurs égales) le problème de la coloration d'un graphe.

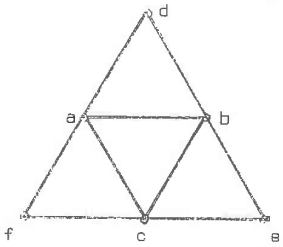
Cependant si les critères "minimum de sommets" et "minimum de longueur" ne coïncident plus, ils ne sont pourtant pas étrangers l'un à l'autre. Nous décidérons de continuer à les confondre. Cette approximation sera d'autant plus valable que les longueurs des sommets seront voisines. C'est-à-dire, que pour déterminer une "bonne" orientation de (E,R), nous ferons comme si tous les sommets étaient de même longueur. De l'orientation de (E,R) obtenue, nous savons ensuite déduire une répartition.

9. 1. - Comparaison avec la méthode du § 4

Rappelons que cette méthode consiste à colorer (E,R) et à placer en recouvrement les segments correspondant à des sommets de même couleur. L'inconvénient était que des segments correspondant à des sommets de couleurs différentes étaient implantés systématiquement de manière disjointe, alors que le fait que deux sommets ont des couleurs différentes n'implique pas qu'ils soient adjacents. Les répartitions obtenues étaient donc susceptibles d'être tassées.

Etudions l'importance de ce tassage sur un exemple.

Soit E = {a, b, c, d, e, f} et le graphe (E,R) suivant :



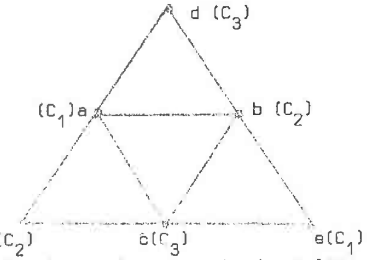
Soient, pour les points de E, les longueurs suivantes :

$\ell(a) = 1$
$\ell(b) = 1$
$\ell(c) = 1$
$\ell(d) = 7$
$\ell(e) = 6$
$\ell(f) = 5$

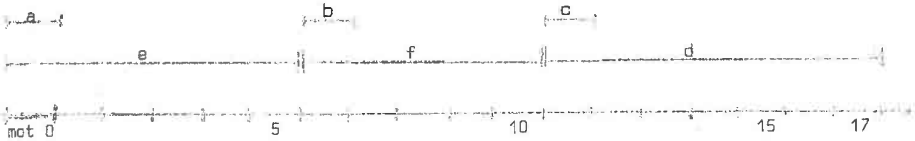
Déterminons une coloration de (E,R). On voit facilement que ce graphe est 3 - chromatique.

Soient C_1, C_2, C_3 les couleurs affectées aux sommets.

Considérons la coloration suivante :



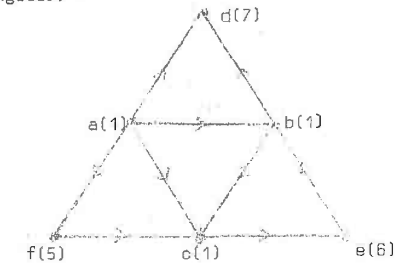
Si on décide d'implanter les segments de couleur C_1, puis C_2 et C_3, la méthode du § 4 donne l'implantation suivante :



Il faut au total, 18 mots de mémoire. On remarque que les segments d et f, correspondent à des sommets de couleurs respectives C_3 et C_2 qui ne sont pas adjacents.

Procédons maintenant par orientation de (E,R) en conservant le même ordre arbitraire sur les couleurs.

Nous obtenons le graphe suivant (à côté de chaque sommet est indiquée sa longueur) :



Les chemins maximaux de ce graphe sont :

{a, b, d} , {e, b, d} , {a, b, c} , {a, f, c} , {e, b, c}

Leurs longueurs sont respectivement :

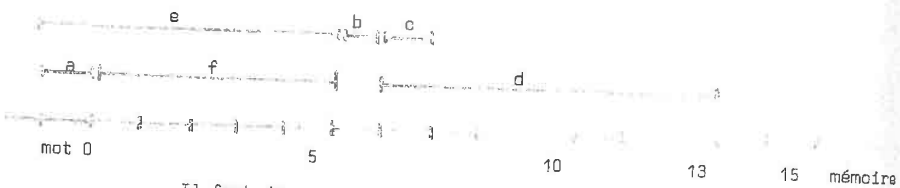
1+1+7 = 9, 6+1+7 = 14, 1+1+1 = 3, 1+5+1 = 7, 6+1+1 = 8

Le chemin le plus long est de longueur 14. Il faudra donc 14 mots pour loger l'ensemble des segments.

Les chemins maximaux de longueur maximale, d'extrémité donnée sont (les longueurs sont données sans ajouter la longueur du dernier sommet. On obtient ainsi une répartition) :

extrémité	a : {a}	de longueur	0
"	b : {e,b}	"	6
"	c : {e,b,c}	"	7
"	d : {e,b,d}	"	7
"	e : {e}	"	0
"	f : {a,f}	"	1

D'où l'implantation suivante :



Il faut donc, comme prévu, 14 mots, au lieu de 18 comme précédemment.

9. 2. - Défauts de la méthode

9.2.1 Choix d'un ordre arbitraire sur les couleurs

La méthode précédente exige de choisir un ordre arbitraire sur les couleurs. Ceci n'avait aucune importance dans le cas des segments de longueurs égales. Dans le cas contraire, il n'en n'est plus de même.

En effet soient q points de E auxquels sont affectées des couleurs C_1, C_2,..., C_q successives : c'est-à-dire que deux couleurs C_i et C_{i+1} sont telles qu'il n'existe pas de couleur supérieure à C_i, inférieure à C_{i+1}.

Si ces q points forment une chaîne dans (E,R)[1], alors, après orientation de (E,R) ils forment un chemin dans le graphe obtenu. Si nous définissons un ordre différent sur les couleurs, les arêtes de la chaîne, après orientation, ne donneront plus nécessairement des arcs formant un chemin.

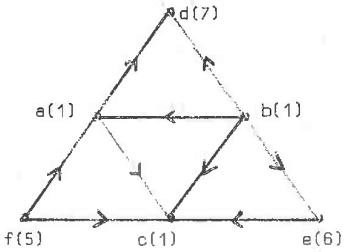
(1) On appelle chaîne une suite de sommets $x_1, x_2, \ldots, x_p$ telle qu'il existe une arête joignant x_i à x_{i+1} pour i = 1,p-1

Par conséquent, lorsqu'on change l'ordre des couleurs, en général le chemin le plus long change également ainsi que la longueur de la répartition.

Pour illuster cette variation, reprenons l'exemple du § 9.1 que nous avons traité avec l'ordre C_1, C_2, C_3 sur les couleurs.

Traitons cet exemple avec l'ordre C_2, C_1, C_3.

Le graphe (E,R) après orientation, devient cette fois.



Les chemins maximaux sont :

{f, a, d} , {f, a, c} , {b, a, d} , {b, a, c} , {b, c, e}

Leurs longueurs sont respectivement :

13, 7, 9, 3, 8

Le chemin le plus long est de longueur 13. L'allocation est possible avec 13 mots.

Avec l'ordre C_1, C_2, C_3 il fallait 14 mots.

Il nous faudrait, pour établir le meilleur ordre à définir sur les couleurs, examiner les $\gamma!$ ordres possibles des couleurs (γ étant le nombre chromatique de (E,R)), et déterminer, pour chacun d'eux, la taille du plus grand chemin du graphe obtenu.

On peut remarquer qu'un ordre C_1, C_2..., C_γ sur les couleurs donne le même résultat que l'ordre inverse C_γ , $C_{\gamma-1}$,..., C_1. En effet, si une suite de γ points de couleurs C_1, C_2, ...,C_γ forment une chaîne dans (E,R), cette suite inversée formera également une chaîne de sommets de couleurs C_γ , $C_{\gamma-1}$, ..., C_1.

Il suffirait donc d'examiner $\frac{1}{2}\gamma!$ permutations, et encore faudrait-il le faire pour toutes les γ-colorations possibles de (E,R).

Nous verrons ultérieurement qu'avec des graphes (E,R) de 256 sommets, comportant environ 60 % du nombre maximal d'arêtes (ce nombre maximal est atteint si (E,R) est complet ; il vaut $\frac{|E|\,(|E|-1)}{2}$) il faut s'attendre à un nombre chromatique de l'ordre de 30, ce qui rend impensable l'examen des permutations possibles.

Nous nous en tiendrons donc à un ordre arbitraire. D'ailleurs, l'examen de toutes les permutations ne conduirait pas forcément à une solution optimale. Il n'est pas certain que la meilleure orientation de (E,R) se déduise d'une χ-coloration.

9.2.2 Exemple où la meilleure solution ne correspond pas à une χ-coloration

Examinons à nouveau l'exemple précédent .

On constate facilement qu'il n'existe (aux permutations de couleurs prés) qu'une seule 3-coloration de (E,R) qui est celle indiquée : en effet, on est obligé d'attribuer 3 couleurs différentes C_1, C_2, C_3 aux sommets a, b, c car ils forment une clique de 3 sommets.

Dés lors, on n'a plus le choix pour colorer d, e, f.

Les différentes permutations possibles des couleurs sont :

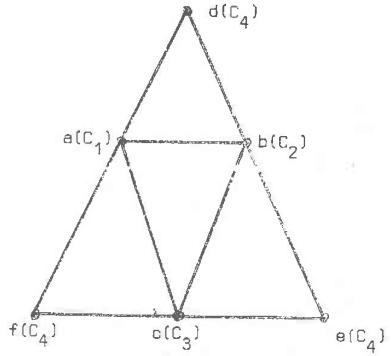
1) C_1, C_2, C_3 (ou inversement C_3, C_2, C_1) pour laquelle on obtiendra le chemin $\{e,b,d\}$ (inv. $\{d,b,e\}$) de longueur 14.

2) C_1, C_3, C_2 (inv. C_2, C_3, C_1) pour laquelle on obtiendra le chemin $\{e,c,f\}$ (inv. $\{f,c,e\}$) de longueur 12.

3) C_2, C_1, C_3 (inv. C_3, C_1, C_2) pour laquelle on obtiendra le chemin $\{f,a,d\}$ (inv. $\{d,a,f\}$) de longueur 13.

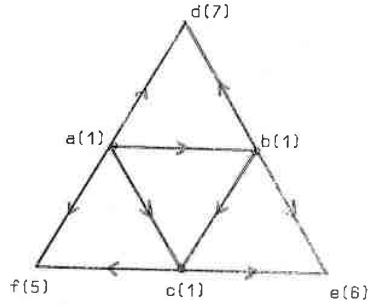
Pour toute orientation possible de (E,R) correspondant à une 3-coloration, on aboutira donc à une implantation sur au moins 12 mots de mémoire (le cas 2 étant le plus favorable).

Définissons maintenant sur (E,R) une 4 coloration de la manière suivante :



Prenons, comme ordre des couleurs C_1, C_2, C_3, C_4.

Nous obtenons le graphe suivant (où les longueurs des sommets sont indiquées)



Les chemins maximaux sont :

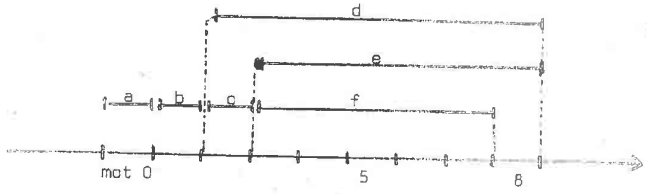
{a, b, d} ,{a, b, c, e} ,{a, b, c, f}

Leurs longueurs respectives sont :

9, 9, 8

L'implantation est donc possible en 9 mots.

Donnons-en l'allure :



On peut remarquer que la longueur de répartition ci-dessus est minimale : en effet {a, b, d} est le plus long chemin trouvé. Les sommets de ce chemin forment une clique et correspondent donc à des segments disjoints qui sont implantés aux mieux contiguement. A cause de cette clique, toute répartition sera donc de longueur 9 au moins.

Nous avons donc ici un exemple où une 4 - coloration conduit à un résultat optimum, alors que le graphe (E,R) est 3 chromatique.

9. 2. 3 Evaluation de l'erreur commise

Quelle que soit la méthode envisagée, nous pouvons trouver une minoration de la solution optimale. Nous pouvons aussi prévoir le résultat que donnera notre méthode dans le cas le plus défavorable.

Nous savons qu'à toute répartition V il est possible d'adjoindre son graphe associé qui est déduit de (E,R) par orientation des arêtes. Si (E,R) est γ-chromatique, le graphe associé à V comportera un chemin de γ sommets. Les sommets de ce chemin forment une chaîne dans (E,R). Inversement, toute chaîne de γ sommets ne devient pas un chemin après orientation des arêtes ce que nous savons, c'est qu'au moins une chaîne c_o de γ sommets devient un chemin. Soit ℓ_{c_o} la longueur de la chaîne c_o.

Toute répartition étant de longueur supérieure ou égale à la longueur du plus grand chemin de son graphe associé, nous pouvons donc écrire que :

$L(V) \geq \ell_{c_o}$.

Soit Ch l'ensemble des chaînes de γ sommets de (E,R).

c_o peut être minoré de la manière suivante :

Posons $\ell_1 = \min_{c \in Ch} (\ell_c)$. Alors $L(V) \geq \ell_{c_o} \geq \ell_1$

Si seules les chaînes de γ sommets de longueur minimum deviennent des chemins dans le graphe associé à V, l'égalité entre L (V) et ℓ_1 pourra être atteinte d'après la proposition 3 (cf § 6).

Toute répartition est donc de longueur supérieure ou égale à la longueur de la plus petite chaîne de γ points de (E,R)

Considérons maintenant la plus grande clique de (E,R) (c'est-à-dire celle dont la somme des longueurs des sommets est maximale). Nous savons qu'après orientation de (E,R) cette clique donnera un chemin hamiltonien entre ses sommets (théorème de Redeï). Ce chemin hamiltonien se retrouvera dans le graphe associé de toute répartition. Toute répartition sera donc de longueur au moins égale à la somme ℓ_2 des longueurs des sommets de la plus grande clique.

En définitive, nous avons trouvés deux minorants ℓ_1 et ℓ_2 pour la longueur d'une répartition. Il est donc impossible, quelle que soit la méthode adoptée, de descendre en dessous de la valeur $\max(\ell_1, \ell_2)$[1].

Dans le cas le plus défavorable, la méthode que nous avons adoptée conduira à une répartition telle que la plus grande chaîne de γ sommets (E,R) (c'est-à-dire celle pour laquelle la somme des longueurs est maximale) devient un chemin dans le graphe associé.

Appelons ℓ_3 la longueur de la plus grande chaîne de (E,R).

(1) Remarquons qu'il n'est pas possible d'atteindre cette valeur quelle que soit la méthode.

La méthode que nous employons conduira à une répartition de longueur au plus égale à ℓ_3.

Nous pouvons construire une répartition V de longueur L (V) telle que :

$$\max(\ell_1, \ell_2) < L(V) < \ell_3.$$

L'optimum est compris également entre ces bornes.

L'écart de la longueur de la répartition par rapport à l'optimum vaut donc au plus :

$$\ell_3 - \max(\ell_1, \ell_2).$$

Exemples :

1) Soit un graphe (E,R) γ - chromatique dont tous les sommets sont de longueur 1.

La plus petite chaîne de γ sommets est de longueur $\ell_1 = \gamma$.

La plus grande clique comporte au plus γ sommets (sinon (E,R) ne serait pas γ-chromatique). Donc $\ell_2 \leq \gamma$.

$$\max(\ell_1, \ell_2) = \gamma.$$

Enfin la plus grande chaîne de γ sommets est de longueur γ.

L'écart par rapport à l'optimum est nul.

Remarquons que si (E,R) est γ-parfait nous avons $\ell_1 = \ell_2 = \ell_3 = \gamma$

2) Prenons comme graphe (E,R) l'exemple du § 9.1.

Ce graphe est 3-chromatique.

La plus petite chaîne de 3 sommets est {a,b,c} , de longueur $\ell_1 = 3$

La clique la plus grande est {a,b,d}. Donc $\ell_2 = 9$.

$$\max(\ell_1, \ell_2) = 9$$

La plus grande chaîne de longueur est {a,b,e} , de longueur $\ell_3 = 14$.

Nous pouvons donc trouver des répartitions de longueur 14 au plus.

En envisageant toutes les solutions données par une 3 coloration, nous avions trouvé effectivement des répartitions de longueur 12, 13, 14. De plus, à partir d'une 4-coloration nous avions trouvé une répartition de longueur effectivement égale à 9.

9. 3. - Contraction d'une répartition

Afin de chiffrer les gains d'espace mémoire, nous pouvons définir la contraction C (V) d'une répartition V pour (E,R,ℓ) comme le rapport :

$$C(V) = \frac{L(V)}{\sum_{x \in E} \ell(x)} .$$

C (V) est le rapport de la taille de mémoire que nous utilisons effectivement et de taille de mémoire qu'il faudrait pour implanter les variables de manière contigue.

Par exemple, si les longueurs de tous les segments sont égales à ℓ_0 nous savons déterminer pour un graphe (E,R) χ-chromatique une répartition V de longueur L (V) = $\chi \cdot \ell_0$.

D'autre part $\sum_{x \in E} \ell(x) = |E| \cdot \ell_0$

D'où $C(V) = \frac{\chi}{|E|}$

Si nous reprenons l'exemple du § 9.1 nous avons déterminé des répartitions allant de L (V_1) = 14 à L (V_1) = 9.

$\sum_{x \in E} \ell(x) = 21$

Les contractions de V_1 et V_2 sont : $C(V_1) = \frac{14}{21} = 66\ \%$

$C(V_2) = \frac{9}{21} = 43\ \%$ (contraction minimale puisque V_2 est une répartition de longueur minimale).

10 - RESUME ET CONCLUSION

Pour résoudre le problème de l'implantation des variables, nous sommes amenés à effectuer les opérations suivantes.

1) Calcul de la fermeture transitive de (M,A). Soit $[A]$ la matrice d'incidence de (M,A^*).
2) Calcul de la fermeture transitive de (E,U). Soit $[U]$ la matrice d'incidence de (E,U^*)
3) Calcul de la matrice $[\bar{A}] = [A] + [A]^t$
4) Calcul de la matrice $[R]$ d'incidence de $[E,R]$ par la formule.

$$[R] = [\hat{U}]^t \cdot [\bar{A}] \cdot [\hat{U}]$$

$[\hat{U}]$ étant obtenue à partir de $[U]$ en ne conservant que les lignes correspondant aux modules

5) Coloration de (E,R)
6) Orientation de (E,R) dans le sens des couleurs croissantes (pour fixer les idées) après avoir choisi un ordre total arbitraire sur les couleurs.
7) Définition de l'origine de chaque segment comme la longueur du plus grand chemin dans (E,R) orienté, d'extrémité le sommet correspondant au segment.

Pour mener à bien ces différentes opérations, nous devons disposer essentiellement d'un algorithme de calcul de fermeture transitive (ce qui présente peu de difficulté, mais demande un choix parmi les méthodes existantes) et d'un algorithme de coloration. Ce dernier point risque d'être assez délicat, car les méthodes de coloration sont généralement très coûteuses. Nous exposerons ultérieurement les méthodes de calcul de fermeture transitive et de coloration que nous avons retenues.

Comme nous avons vu que les contraintes de recouvrement étaient identiques pour les constantes et les variables, nous constatons que les points 1 à 4 constituent un traitement commun aux deux cas.

Dans la méthode de détermination d'une répartition à partir du graphe (E,R) nous avons fait une approximation (nous avons dit que minimiser la longueur d'un chemin revenait à minimiser le nombre des sommets par lesquels il passe) dont nous avons mis les défauts en évidence. Cependant, cette approximation qui néglige l'importance de la longueur des segments permet de rendre commun la presque totalité des traitements relatifs aux constantes et aux variables. En effet pour implanter les textes objets, nous procéderons aussi par coloration et orientation de (E,R). Ce traitement (points 5 et 6) sera commun aux cas des variables et des constantes et ne sera fait qu'une seule fois.

Ce n'est qu'au point 7 que les traitements divergent, puisque c'est à ce moment qu'interviennent les longueurs des segments et que, pour un module ou une classe, les longueurs des segments de variables et de constantes sont évidemment différentes. Cependant, nous pouvons parfaitement concevoir un algorithme unique acceptant comme paramètre la fonction longueur définie sur E : suivant la manière dont nous fixerons ce paramètre, l'algorithme déterminera une répartition pour les variables ou pour les textes objets.

L'imperfection de notre méthode se trouve donc contrebalancée par, le fait que les traitements sont accélérés (points 1 à 6 communs aux cas des variables et des textes objets) et que l'algorithme du point 7 ne doit être rédigé qu'une seule fois.

Le problème de l'implantation des textes objets se trouve déjà résolu dans une large mesure. Nous allons voir maintenant comment le résoudre entièrement, et comment mettre en œuvre un programme résident procédant aux chargements des segments de constantes depuis la mémoire secondaire.

e PARTIE

SOLUTION AU PROBLEME DES TEXTES OBJETS

1 - COMPATIBILITE AVEC LE CAS DES VARIABLES

2 - MODULATION DU RECOUVREMENT

3 - PROBLEMES SPECIFIQUES AU CAS DES TEXTES OBJETS

3. 1. - Programme assurant les chargements depuis la mémoire secondaire

3. 2. - Cas du recouvrement sur les modules seuls

3.2.1 - Entrées dans le résident

3.2.2 - Traitement effectué par le résident

3.2.3 - Génération résident par le prééditeur

3.2.4 - Edition de liens

3.2.4.1 Edition séparée des différents modules.

3.2.4.2 Réalisation de l'édition de liens

3.2.4.3 Exemple d'édition de liens

3. 3. - Cas du recouvrement sur les modules et les classes.

3.3.1 - Vérification de la présence des textes en mémoire centrale

3.3.2 - Mise à jour de la table indicatrice de la présence des textes en mémoire centrale

3.3.3 - Edition de liens

4 - CONCLUSION

6ème PAPTIE

SOLUTION AU PROBLEME DES TEXTES OBJETS

1 - COMPATIBILITE AVEC LE CAS DES VARIABLES

Nous avons admis jusqu'a présent que nous prenions en compte le même ensemble E de classes et de modules, pour l'étude des variables d'une part, et des textes objets d'autre part.

Ceci revient à admettre qu'à chaque classe ou module correspondent à la fois un segment de variables et un texte objet. Pratiquement, cette hypothèse peut se trouver infirmée de deux façons :

1) Un module ou une classe peut donner lieu à la génération d'un texte objet, sans qu'un segment de variable lui corresponde. Ce cas peut se produire lorsque classe ne déclare que des constantes, ou lorsqu'un module ne comporte aucune variable locale, explicitement déclarée, ou créée par le compilateur.
2) Une classe peut ne pas donner lieu à la génération d'un texte objet (ce qui n'est pas possible pour un module). Seules les variables qui y sont déclarées seront à prendre en compte à la préédition de liens

Nous sommes donc amenés à définir l'ensemble E_v des modules et des classes auxquels correspond effectivement un segment de variables. De même nous appellerons E_c l'ensemble des modules et des classes donnant lieu à la génération d'un texte objet.

Le graphe exprimant les contraintes de non recouvrement entre les segments de variables sera le sous graphe (E_v,R) de (E,R). Les contraintes de non recouvrement entre les textes objets seront traduites par le sous graphe (E_c,R) de (E,R).

En toute rigueur, les méthodes d'allocation que nous avons développées devraient s'appliquer à (E_v,R) pour le traitement des variables, et à (E_c,R) pour le traitement des textes objets.

Cependant, les ensembles E_v et E_c sont en général assez peu différents. Il ne sera donc pas indispensable d'effectuer pour chacun de ces deux graphes les traitements de coloration et d'orientation des arêtes. Lorsque le prééditeur de liens en arrive au traitement des textes objets, l'allocation des variables est déjà réalisée, et le graphe (E_v,R) a été coloré et orienté

Si à chaque sommet x de E_V nous affectons une longueur $\ell'(x)$, égale à la longueur du texte objet de x si x appartient à $E_c \cap E_V$, ou une longueur nulle si x n'appartient pas à $E_c \cap E_V$, nous pouvons à partir du graphe (E_V, R) orienté, déterminer une allocation pour les textes objets de $E_c \cap E_V$. Les textes objets des modules et classes de E_c, non inclus dans $E_c \cap E_V$ restent à traiter. Nous déciderons simplement de les implanter consécutivement en mémoire centrale, en dehors de la zone où sont implantés les textes objets en recouvrement.

Nous arrivons donc à implanter tous les textes objets, au prix de deux approximations :

1) Nous renonçons à tout recouvrement pour les modules et classes de $E_c \cap E_V$.
2) Des sommets de (E_V, R) sont affectés d'une longueur nulle. Ceci n'est qu'un artifice destiné à les éliminer au moment de l'allocation. Cependant à ces sommets sont affectées inutilement des couleurs : si de E_V nous éliminions les points affectés d'une longueur nulle, nous obtiendrions un sous graphe de (E_V, R) dont le nombre chromatique serait moins élevé que celui de (E_V, R).

La présence de ces sommets inutiles risque donc de conduire à des résultats moins bons que dans le cas où ils seraient éliminés.

Ces deux approximations se justifient par le fait que les ensembles E_c et E_V sont dans la pratique très peu dissemblables. Elles ont d'autre part pour mérite d'accélérer les traitements, en augmentant la compatibilité entre les cas des variables et des textes objets.

2 - MODULATION DU RECOUVREMENT

Nous avons déjà dit qu'il était peu souhaitable de profiter de toutes les possibilités de recouvrement entre les textes objets. Mieux vaut utiliser au maximum l'espace de mémoire centrale dont nous disposons.

C'est pourquoi il semble préférable, lorsque la taille de la mémoire centrale le permet, de n'effectuer de recouvrement que sur les textes objets des modules, et non plus, comme nous l'avons vu ci-dessus, sur l'ensemble des modules et des classes.

Nous savons qu'un module x à accès à toutes les constantes de $\hat{U}(x)$. Si le recouvrement porte sur l'ensemble des textes objets des modules et des classes, l'appel de x par un autre module risque d'impliquer une série de transferts depuis la mémoire secondaire. Dans le meilleur des cas, nous devons au moins vérifier la présence en mémoire centrale des textes objets des éléments de $\hat{U}(x)$.

Au contraire, si le recouvrement ne porte que sur l'ensemble des modules, seule la présence en mémoire centrale du texte objet de x est à vérifier. En cas d'absence, seul ce texte objet sera à transférer de la mémoire secondaire à la mémoire centrale. Les classes, quant à elles, seront implantées consécutivement.

La limitation du recouvrement aux seuls textes des modules apporte donc des avantages certains.

Nous allons voir que, de plus, l'implantation des textes des modules est particulièrement facile. Nous disposons en effet d'un algorithme qui permet à partir d'un graphe sans circuit (graphe (E_V, R) orienté) de déterminer une allocation des textes objets. Il suffit donc de soumettre le graphe (M,A) des appels de modules à cet algorithme. Nous obtiendrons immédiatement une implantation des textes des modules.

La taille de mémoire nécessaire à l'implantation de ces textes sera égale à la longueur par les sommets du plus grand chemin de (M,A).

Si cette taille est excessive, il nous sera possible de procéder à un recouvrement plus complet, exploitant les possibilités de recouvrement entre modules et classes.

Nous arrivons donc à moduler dans une certaine mesure les allocations en fonction de la taille de mémoire centrale disponible. Nous obtenons finalement trois possibilités qui prises dans l'ordre ci-dessous, nécessitent de moins en moins de mémoire centrale :

- recouvrement sur les variables uniquement
- recouvrement sur les variables d'une part, les textes objets des modules d'autre part.
- recouvrement sur les variables d'une part, tous les textes objets d'autre part.

3 - PROBLEMES SPECIFIQUES AU CAS DES TEXTES OBJETS

3. 1. - Programme assurant les chargements depuis la mémoire secondaire

Il est nécessaire d'adjoindre à l'unité de traitement un programme résident assurant les chargements depuis la mémoire secondaire vers la mémoire centrale.

Ce programme doit intervenir à chaque appel de module.
Nous nous proposons d'étudier une possibilité de liaison entre les modules et ce programme dans les deux cas du recouvrement sur les modules seuls, puis du recouvrement sur les modules et les classes.

3. 2. - Cas du recouvrement sur les modules seuls

3.2.1 - Entrées dans le résident

Soit $M = \{x, x_0, x_1, \ldots, x_m\}$ l'ensemble des modules, x étant le module directeur.

Dans une unité de traitement, les modules sont désignés par des noms. Il est possible, à la préédition de liens, de leur affecter un indice qui permet, comme nous le verrons plus loin, de les désigner plus commodément (nous verrons également qu'il est inutile d'affecter un indice au module directeur).

Supposons qu'un module appelle un autre module x_j. Au niveau du texte objet du module appelant, cet ordre d'appel est une instruction (nous en donnons l'équivalent en langage d'assemblage) : BAL,RL x_j.

Son effet est de provoquer un branchement à l'adresse x_j. De plus, si cette instruction est effectuée à l'adresse a, la quantité a + 1 est chargée dans le registre RL.

A l'exécution, nous ne pouvons laisser cette instructions se dérouler normalement, puisque nous ne sommes pas sûrs de la présence du texte objet de x_j en mémoire centrale. Nous devons intercepter cet ordre de branchement et de manière générale, tous les ordres de branchements résultant de l'appel d'un module à un autre.

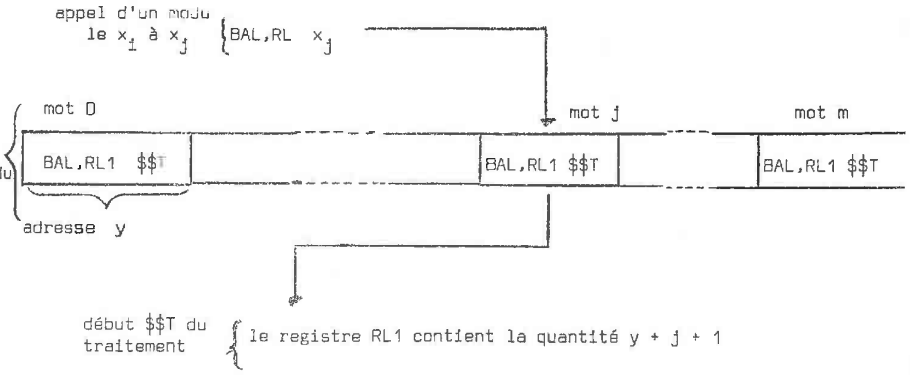
Pour cela, nous pouvons, lorsque l'éditeur de liens travaille sur un module appelant x_j, altérer la définition de x_j, de manière à ce que l'instruction BAL,RL x_j provoque un branchement, non plus au module x_j, mais dans le programme résident.

Dans le programme résident, il faudra alors identifier le module appelé par cette instruction.

Nous proposons de faire varier le point d'entrée dans le résident en fonction du module appelé. Le module directeur ne pouvant être appelé nous devons définir m+1 points d'entrée dans le résident. Après être rentré dans le résident par un de ces points, nous devons procéder à un traitement commun à tous les modules.

Nous proposons la solution suivante :
Soit $$T l'adresse de début de traitement dans le résident[1].
Nous pouvons définir les m+1 points d'entrée comme une table débutant à l'adresse $$ PER (table des Points d'Entrée dans le Résident) de m+1 mots. Chaque mot de cette table contiendra une instruction : BAL,RL1 $$T. Supposons qu'à l'édition de liens nous ayons satisfait la référence à x_j avec la quantité $$PER+j. A l'éxecution un branchement au module x_j va provoquer un branchement au mot j de la table $$PER et de là, un nouveau branchement à l'adresse $$T. Le registre RL1 contiendra alors la quantité $$PER + j + 1, grâce à laquelle le module appelé est immédiatement identifié.

Nous pouvons schématiser ce fonctionnement



L'avantage de cette manière de procéder est de ne nécessiter aucune transformation du texte objet d'un module : seules des références non satisfaites seront alterées à l'édition de liens.

3.2.2 3.2.2 Traitement effectué par le résident :

Le résident doit s'assurer de la présence en mémoire centrale du texte objet du module appelé. Pour cela il est pratique de définir une matrice booléenne de m+1 éléments. Nous la représenterons par une table de (m+1)

(1) De manière générale les adresses symboliques utilisées dans le résident commenceront par les 2 caractères $$ de manière à éviter à l'édition de liens des confusions avec des définitions externes provenant de classes ou modules.

octets débutant à l'adresse $$ PTMC (table de Présence des Textes en Mémoire Centrale). Nous conviendrons que si l'octet j contient la valeur le module x_j est présent en mémoire centrale, et qu'il est absent dans le cas contraire.

Si le texte est présent en mémoire, il ne reste plus qu'à effectuer le branchement au point d'entrée du module x_j. Pour cela nous devons obtenir une table $$PEM des Points d'Entrée des Modules : le mot j contient l'adresse du point d'entrée de x_j. Connaissant l'indice j du module appelé il sera alors facile d'atteindre son point d'entrée.

Si le texte objet est absent de la mémoire centrale, il faut le charger depuis la mémoire secondaire. Les textes objets étant des unités que nous chargeons séparément, nous pouvons faire correspondre à chaque texte objet un fichier implanté sur la mémoire secondaire, contenant la version originale de ce texte. Il est logique de regrouper l'ensemble des fichiers relatifs à une unité de traitement dans un même super fichier : nous en ferons un fichier partitionné[2] dont chaque partition contiendra le texte objet d'un module. Une partition est caractérisée par son nom et le nom du fichier auquel elle appartient[ ]. Si on veut charger le texte objet de x_j, il faut donc disposer du nom de la partition correspondante. Pour cela, on pourrait constituer une table des noms de partitions dont le $j^{è}$ élément donne le nom de la partition. Cette solution est rapide, mais offre l'inconvénient d'encombrer la mémoire centrale avec une table supplémentaire. Il paraît plus intéressant, connaissant l'indice j, d'en déduire une chaîne de caractères qui est le nom de la partition.

Considérons par exemple la partition d'indice 25. Cet indice est codé en mémoire centrale sous forme binaire. De cette forme binaire il est possible de déduire la chaîne de caractères '25' (soit F2 F5 en code EBCDIC). Cette chaîne de caractères constituera le nom de la partition. Nous procéderons ainsi pour chaque module. Notons qu'il est possible sur le 10070 de convertir la forme binaire en chaîne de caractères en EBCDIC en une seule instruction câblée[3]. La conversion est donc rapide. Plus précisément la chaîne de caractères obtenue comprend 4 caractères. Pour le module d'indice 25 nous obtiendrons en fait F0 F0 F2 F5 en code EBCDIC, soit les caractères 0025.

(1) Nous choisissons les octets car ce sont les plus petites cellules adressables sur le 10070.

(2) cf notice "Système de gestion de fichiers" documentation Siris 7/8

(3) Il s'agit de l'instruction CVS qui nécessite une table de conversion [illegible] 10070".

En procédant ainsi pour chaque module, nous évitons la présence d'une table des noms de partitions en mémoire centrale, tout en assurant une bonne rapidité.

Aprés chargement d'un texte objet dans un certain emplacement de mémoire, les textes objets qui occupaient une partie de cet emplacement sont détruits. Il faut donc mettre à jour la table indicatrice des présences en mémoire centrale : l'octet d'indice j est mis à 1 (le texte de x_j est chargé) et les octets correspondant aux textes détruits par le chargement de x_j doivent être remis à 0. Cette mise à jour n'étant entreprise que si l'on charge effectivement x_j, ce qui nécessite l'accés à la partition P_j contenant son texte objet, il est inutile de faire résider en mémoire centrale la liste des modules détruits par le chargement de x_j : il suffit de faire figurer cette liste sur mémoire secondaire dans la partition P_j.

3.2.3 Génération du résident par le prééditeur

Le texte du résident est constitué de deux parties bien distinctes.

a) un traitement consistant à vérifier la présence des textes objets en mémoire centrale et à les charger en cas d'absence. Ce traitement peut être écrit une fois pour toutes. Il admet deux paramètres : le nombre de modules mis en jeu et le nom du fichier partitionné contenant les textes objets de l'unité de traitement.

b) Des tables : - table $$ PTMC (présence des textes en mémoire centrale
- table $$ PER (points d'entrée dans le résident)
- table $$ PEM (points d'entrée dans les modules)

Ces tables ont des tailles ou un contenu qui dépend de l'unité de traitement considérée. Elles devront donc être générées spécifiquement par le prééditeur de liens, sous forme d'un texte objet les décrivant. Ce texte objet, adjoint au texte objet de la partie traitement, constituera le programme résident.

3.2.4. Edition de liens

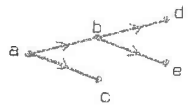
3.2.4.1 Edition séparée des différents modules

La plus grande difficulté du procédé que nous venon d'exposer réside dans l'édition de liens Elle est réalisée à l'aide de l'éditeur

de liens du moniteur Siris7.

Cet éditeur de liens permet la mise en oeuvre de recouvrement, moyennant une hypothèse restrictive : la structure de recouvrement est décrite à l'aide d'une arborescence appellée arbre de recouvrement[1]

Prenons un exemple : soit S = {a,b,c,d,e} l'ensemble des segments. Soumettons à l'éditeur de liens l'arbre de recouvrement suivant[2],[1] :



L'éditeur de liens implantera consécutivement les segments d'un même chemin de l'arborescence. Par exemple les segments {a,b,e } seront implantés de telle manière que l'origine de b soit située après l'extrémité de a, l'origine de a après l'extrémité de e.

Si le graphe (M,A) était une arborescence, il suffirait donc de le soumettre à l'éditeur de liens. Celui-ci se chargerait du calcul de l'implantation des textes objets et de la mise en place d'un algorithme de chargement des segments[1], la prééédition de liens pour les textes objets devient inutile. Il en serait d'ailleurs de même si le graphe partiel formé par les chemins maximaux de (M,A) était une arborescence : en effet, dans un tel recouvrement, il est permis à l'exécution de "sauter" d'un segment à son successeur (par exemple de a à b), mais aussi d'un segment à tout autre segment situé sur un même chemin (par exemple de a à e).

En général le graphe (M,A) ne remplit pas cette hypothèse, et on ne peut le soumettre à l'éditeur de liens. Il faut donc, comme nous l'avons vu, déterminer l'implantation grâce au prééditeur, puis la soumettre à l'éditeur de liens. Comme celui-ci ne peut traiter un ensemble de segments que dans les conditions exposées ci-dessus, nous sommes obligés de lui soumettre les textes objets un par un.

Pour l'édition d'un texte objet, les données à fournir sont l'adresse d'implantation et la définition des références non satisfaites de ce texte

(1) cf Notice CII "Editeur de liens sous Siris 7/8"

(2) Sous forme de cartes de données ou de fichier sur mémoire secondaire.

3.2.4.2 Réalisation de l'édition de liens

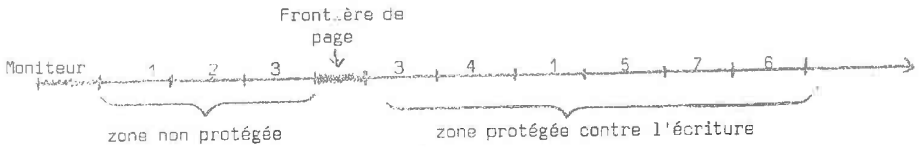
Le prééditeur de liens doit connaître l'implantation exacte de tous les constituants de l'unité de traitement pour que l'édition de liens soit possible.

Ce sont

1) les sous programmes de bibliothèque[1]
2) les segments de variables des classes et de modules
3) les tables du programme assurant les chargements
4) le texte de ce programme
5) les textes des classes
6) les segments en recouvrement (ce sont les textes des modules)
7) les interfaces avec le système d'exploitation[2]

L'implantation de ces divers éléments les uns par rapport aux autres est arbitraire, choisissons la distribution suivante en mémoire centrale (les numéros renvoient aux différents constituants ci-dessus) :

Schéma d'organisation de la mémoire centrale



La mémoire se subdivise en deux zones de protections différentes. La protection s'effectue sur le 10070 au niveau des pages (1 page = 512 mots). La seconde zone doit débuter à une frontière de page.

Nous trouvons :

1) Une zone non protégée contre l'écriture contenant les variables, les sous programmes de bibliothèque ne pouvant être protégés (c'est le cas s'ils comportent des variables) et la table $$PTMC (qui indique au chargeur la présence ou l'absence du texte d'un module en mémoire centrale).

2) Une zone protégée contre l'écriture : cette zone contient les constantes : tables $$PER (table des points d'entrée dans le chargeur) et $$PEM (table des points entrée dans les modules), le texte du chargeur, les

(1) Sous programmes de service (conversions de types, fonctions standard, etc

(2) Notamment tables DCB (tables servant aux entrées sorties).

sous programmes de bibliothèque en protection, les textes des classes implantés consécutivement, la zone dans laquelle le chargeur gérera les segments, les interfaces avec le système.

Notons que la zone contenant les modules étant en protection, il sera nécessaire que le chargeur ait un pouvoir spécial pour détruire un module en le remplaçant par un autre[3]. Afin de réserver la place pour cette zone, le prééditeur générera un module objet important une directive de réservation d'un nombre de mots égal à la taille de la mémoire nécessaire à l'implantation des modules et qu'il a précédemment déterminée.

Le prééditeur de liens connait la taille de tous les constituants. Il peut donc calculer les définitions correspondant à toutes les références non satisfaites.

Il est alors possible d'effectuer l'édition de liens de la manière suivante :

A) Edition de tous les constituants à l'exception des modules

Tous ces constituants sont implantés consécutivement. En effet nous avons vu que l'ensemble des segments de variables constitue en fait un module objet. Il est donc possible d'éditer tous les constituants en une seule fois.

Le prééditeur de liens connaît la liste des modules de bibliothèque, qui doivent être joints à l'unité de traitement. Plutôt que de laisser l'éditeur de liens chercher dans la bibliothèque CIVA les sous programmes correspondant aux références, il est plus rapide que le prééditeur fournisse à l'éditeur la liste de ces sous-programmes.

Notons qu'une partie des tables du chargeur étant en zone non protégée et l'autre en zone protégée, il est nécessaire que le prééditeur génére deux modules objets contenant 1) la table $$PTMC sans protection

2) les tables $$PER et $$PEM avec protection.

Ces deux modules objets seront fournis à l'éditeur dans la liste des textes à éditer.

B) Edition des modules

Les modules sont édités un par un. Si un module se référe à un autre module, cette référence doit être satisfaite par la définition de l'adresse d'un élément de la table $$PER. Au contraire si un module se référe à une classe ou à un sous programme de bibliothèque, cette référence doit

(3) Ce pouvoir consiste à modifier les verrous de protection des pages (cf Manuel de présentation du 10070)

être satisfaite par une définition donnant l'adresse effective de la classe ou du sous programme.

Afin de satisfaire ces différentes références, le prééditeur de liens générera une fois pour toutes un module objet comportant les définitions :

- des sous programmes de bibliothèque
- des origines des classes
- des origines des segments de variables des classes
- des points d'entrée dans le résident
- des interfaces avec le système d'exploitation.

L'édition de liens d'un module CIVA avec ce module objet suffira à satisfaire toutes les références. Le produit de cette édition de liens sera le texte exécutable du module CIVA. Ce texte constituera une partition du fichier partitionné grace auquel le chargeur pourra régénerer la mémoire centrale.

Tous les modules seront édités successivement avec ce module objet L'éditeur de liens recevra une donnée supplémentaire qui est l'origine du module à éditer : cette origine a été précédemment déterminée par le prééditeur de liens (l'ensemble des origines constitue la table $$PEM).

3.2.4.3 Exemple d'édition de liens

3.2.4.3.1 Différents constituants (les adresses sont, pour simplifier, exprimées en base 10).

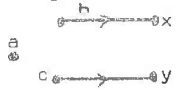
Soit l'unité de traitement suivante :

Ensemble des modules : M = {a,b,c}

Ensemble des classes : {x,y}

Graphe des appels (M,A) :

Graphe des directives "utilisé"



Supposons que 1) Chaque module et classe nécessite 50 mots pour son segment de variables et 50 mots pour son texte objet.

2) L'ensemble des sous programmes de bibliothèque appelés soit {SP1, SP2} avec

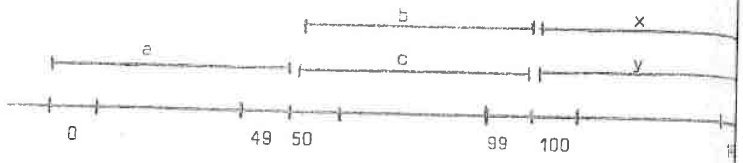
$$P1 sans protection, de longueur 25 mots

$$P2 avec protection, de longueur 25 mots

3) Le partie fixe du chargeur (la partie réalisée une fois pour toutes) nécessite 40 mots. Appelons $$CH le module objet constituant cette partie fixe du chargeur.

4) Le moniteur s'achève à l'adresse 4999 : 5000 est donc l'adresse à partir de laquelle s'implantent les programmes d'un utilisateur.

5) L'implantation des variables, relativement à l'adresse 0, soit la suivante :



Les définitions relatives des $ noms sont donc :

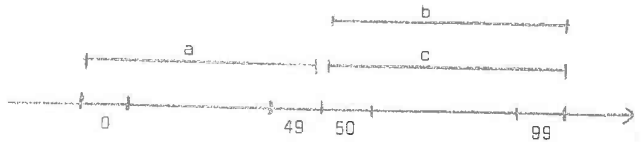
$a = 0

$b = $c = 50

$x = $y = 100

Le prééditeur de liens a généré un module objet définissant les $ noms et réservant 150 mots pour les variables. Appelons $$M ce module objet.

6) L'implantation des textes objets des modules, relativement à l'adresse 0, soit la suivante :



Faisons le décompte des constituants en zone non protégée. Ce sont :

- $P1 : taille 25 mots
- les variables : taille 150 mots
- la table $$PTMC de |M| -1 octets (1 octet par module, sauf pour le module directeur) soit 2 octets. Le prééditeur de liens génère cette table sous forme d'un module objet que nous appellerons $$TABLE1.

D'après le schéma d'organisation de la mémoire centrale que nous avons précédemment donné, ces constituants seront implantés aux adresses suivantes :

$P1 en 5000

les variables en 5025 (après $P1)

la table $$PTMC en 5025 + 150 = 5175 (après M)

Les définitions absolues des $noms sont donc :

$a = 0 + 5025 = 5025

$b = $c = 50 + 5025 = 5075

$x = $y = 100 + 5025 = 5125.

Supposons que la prochaine page située immédiatement après la zone non protégée, débute à l'adresse 6000.

Nous trouverons alors en zone protégée :

- les tables $$PEM et $$PER faisant chacune 2 mots.

Les mots de la table $$PER contiennent des instructions BAL,RL1 $$T (branchement à la partie traitement du chargeur). Nous calculerons un peu plus loin le contenu de $$PEM (adresses absolues des origines des modules) et précisons l'allure exacte du texte objet produit par le prééditeur pour ces deux tables.

- le texte du chargeur (module objet $$CH) de longueur 40 mots
- le sous programme de bibliothéque $$P2 de longueur 25 mots
- les textes des classes x et y implantés consécutivement, soit 50 + 50 = 100 mots
- les interfaces avec le système. Supposons qu'ils nécessitent 100 mots
- la zone de recouvrement des modules, qui nécessite 100 mots.

Ces éléments seront implantés aux adresse suivantes, d'après le schéma d'organisation de la mémoire.

$$PEM en 6000

$$PER en 6002 (après $$PEM)

la partie fixe du chargeur en 6004 (après $$PER)

SSP en 6044 (après la partie fixe du chargeur)

Les textes des classes : x en 6069 et y en 6119

Les interfaces en 6119 + 50 = 6169 (après le texte de y)

La zone de recouvrement des modules en 6169 + 100 = 6269

Nous pouvons préciser l'implantation des divers modules :

module directeur a : origine en 6269

module b : origine en 6269 + 50 = 6319

module c : origine en 6229 + 50 = 6319

A chaque module autre que le module directeur est affecté un indice permettant au chargeur de le repérer. Supposons que b ait reçu l'indice 0 et C l'indice 1. Le mot 0 de la table $$PEM contiendra la quantité 6319 ainsi que le mot 1 (les deux modules b et c sont implantés à la même adresse 6319).

Donnons l'équivalent en langage Métasymbol, du module objet généré par le prééditeur pour les tables $$PEM et $$PER (appelons $$TABLE 3 ce module objet) :

```
            REF     $$T      <--référence au point d'entrée du chargeur
            DEF     $$PEM,$$PER  <--mise en définition externe des
                                     tables. Ces tables seront connues
                                     du chargeur par des références
                                     qui seront satisfaites par ces
                                     définitions.
$$PEM       DATA    6319     <-- définition du mot 0 de $$PEM
            DATA    6319     <-- définition du mot 1
$$PER       BAL,RL1 $$T
            BAL,RL1 $$T
```

3.2.4.3.2 Phase d'édition de liens

Le prééditeur de liens a, de plus, généré un module objet réservant 100 mots en protection correspondant à la zone de recouvrement des modules . Appelons $$RECOUV ce module objet.

Dans un premier temps on procéde à une édition de liens de tous les correspondants, soit dans l'ordre :

$ P1, M$, $$TABLE1, $$TABLE2, $$CH, SSP2, classes x et y, interfaces, $$RECOUV.

Le prééditeur de liens génére ensuite le module objet suivant, donnant toutes des définitions externes de l'unité de traitement nécessaires à l'edition des modules a, b etc :


```
            DEF     SSP1
            DEF     SSP2
            DEF     $a,$b,$c,$x,$y      } mise en définition externe
            DEF     b,c,x,y             } des symboles

SSP1        EQU     5000
SSP2        EQU     6044
$a          EQU     5025    } définition absolue des différents
$b          EQU     5075    } symboles d'après les calculs pré-
$c          EQU     5075    } cédents
$x          EQU     5125
$y          EQU     5125
b           EQU     6002    } les origines des modules
c           EQU     6003    } sont définies fictivement comme les adresses
                            } des mots 0 (pour b) et 1
                            } (pour c) de la table $$PER
x           EQU     6069
y           EQU     6119
```

L'édition de liens d'un module (b ou c) avec ce module objet permettra la résolution de toutes les adresses.

Par exemple, soit le module b. Il "utilise" la classe x et fait référence à son propre segment de variables $b. Ses références non satisfaites sont donc : x, $x, $b. Ces références seront satisfaites après édition de liens avec le module objet précédent (Il en sera de même avec le module c). De plus l'origine d'implantation de b doit être précisée à l'éditeur de liens : elle a été calculée précédemment et vaut 6319.

Notons cependant une légère anomalie pendant l'édition de liens : il y a conflit entre la définition externe d'un module (le nom du module b est en effet déclarée en définition externe dans le texte objet de ce module) et la définition externe donnée dans le module objet créé par le prééditeur de liens (le nom du module b y apparaît également, défini comme le mot 0 de la table $$PER). Il y a donc double définition contradictoire. Ceci se traduira, à l'édition de liens, par un avertissement. Cette double définition n'est pas gênante ici, puisque les textes édités ne comportent aucune expression dépendant de cette définition. Nous pouvons donc admettre la présence de cette anomalie, sans qu'aucune gêne en résulte.

Au début de l'exécution, le contrôle sera passé au chargeur. Celui ci chargera le module directeur et lui donnera le contrôle . L'exécution se déroulera ensuite comme il a été précédemment expliqué. Il aurait été possible d'éditer le module directeur en même temps que les constituants autres que les modules. Cette édition aurait nécessité la satisfaction des références du module directeur aux $ noms auxquels il a accés et aux noms des modules qu'il appelle. Les références aux noms sont satisfaites par les définitions du module objet créé par le prééditeur, permettant l'édition des modules un par un. Il aurait donc fallu consulter ce module objet . Or il comporte également des définitions des $noms. Comme le module $$MD comporte également des définitions des $noms, il y aurait eu, lors de l'édition de liens du module directeur avec les autres constituants, apparition de doubles définition des $noms. C'est pourquoi nous préférons ne pas faire d'exception pour le module directeur : nous pouvons admettre une double définition à l'édition des modules mais ne voulons pas éditer le module directeur avec une liste de doubles définitions pouvant être assez longue.

3. 3. - Cas du recouvrement sur les modules et les classes

Nous avons déjà vu la manière de déterminer une implantation des textes objets. Nous nous intéresserons ici au fonctionnement du chargeur et aux modalités de l'édition de liens.

3.3.1 Vérification de la présence des textes en mémoire centrale

Le principe de fonctionnement du chargeur décrit dans le cas du recouvrement sur les modules seuls reste le même : nous devons toujours intercepter les appels d'un module à un autre. Les tables $$PER et $$PEM restent donc inchangées. Seule la vérification de la présence de textes objets en mémoire centrale pose un nouveau problème. En effet si un module x appelle un module y, il faut s'assurer de la présence en mémoire centrale de tous les textes objets de U^x (y) : cet ensemble est constitué du module y et des classes de $\hat{U}$ (y). Au moment où y est appelé, il faut donc connaître l'ensemble $\hat{U}$ (y). Pour cela, nous pouvons envisager deux solutions :

1) Si la taille de la mémoire centrale le permet, nous pouvons pour chaque module x, faire résider en mémoire centrale la liste des classes de $\hat{U}$ (x). La manière la plus compacte de représenter cette information est d'utiliser la matrice $[U_2]$ (cf parties § 4) représentée sous forme de matrice de bits. Plus exactement, il est intéressant d'utiliser sa transposée pour des raisons que nous allons exposer immédiatement.

Rappelons que $[U_2]^t$ (i,j) = 1 signifie que la classe x_i fait partie de $\hat{U}$ (x_j). Supposons la matrice $[U_2]^t$ rangée ligne par ligne en mémoire centrale. S'il y a m modules il faudra (m : 8) + 1 octets (le symbole : désigne la division entière), pour représenter une ligne ; le reste de la division de m par 8 donne le nombre de bits occupés dans le dernier octet utilisé.

Par exemple si m = 50 :

(50 : 8) + 1 = 7

reste de 50 : 8 = 2

On se réserve 7 octets pour représenter une ligne, 2 bits sont utilisés dans le 7^{è} octet.

Nous déciderons d'implanter la matrice $[U_2]^t$ ligne par ligne, chaque ligne débutant à une frontière d'octet (c'est-à-dire que pour chaque ligne nous perdons 8 - reste (m : 8) bits, en n'implantant la ligne suivante qu'au début du prochain octet).

Ainsi il est facile de localiser l'élément $[U_2]^t$ (i,j) : cet élément se trouve dans l'octet numéro (j : 8) + 1 [1] de la ligne i ; c'est le bit numéro reste (j : 8) de cet octet. Comme (i-1) lignes précédent la ligne i, l'élément cherché est le bit n° reste (j : 8) de l'octet n° (i-1) × [(m : 8) + 1] + (j : 8) + 1 de la matrice $[U_2]^t$. Cette expression se calcule facilement. L'intérêt d'avoir utilisé $[U_2]^t$ plutôt que $[U_2]$ est le suivant : lorsqu'on cherche les classes de $\hat{U}$ (x_j) on travaille à indice de colonne constant, et la position relative des bits cherchés est constante dans les Octets contenant ces bits. L'exploration de la colonne j peut alors le faire de la manière suivante : on suppose calculée une fois pour toutes la constante (m : 8) + 1 ; pour explorer la colonne j il suffit de calculer 1 + (j : 8) et son reste (une instruction sur le 10070) ; en fonction du reste on sélectionnera un des huit masques suivants (exprimés en hexadécimal) :

X'00000001'	si reste = 1
X'00000002'	si reste = 2
X'00000004'	si reste = 3
X'00000008'	si reste = 4
X'00000010'	si reste = 5
X'00000020'	si reste = 6
X'00000040'	si reste = 7

(1) les octets d'une ligne sont numérotés de 1 à (m : 32) + 1

Pour traiter la 1ére ligne il suffit de sélectionner l'octet 1 + (j : 8) de la matrice (ceci est réalisé en une instruction LB (chargement d'octet), cette instruction charge l'octet voulu dans les 8 derniers bits d'un registre, et met le reste de ce registre à 0) et de réaliser un et logique bit par bit entre le registre où on a chargé l'octet et le masque correspondant (instruction AND du 10070). Si le résultat de cette opération est nul, c'est que le bit cherché était nul ; s'il est différent de 0, c'est que le bit cherché était à 1. Pour traiter la ligne suivante, on sélectionnera l'octet 1 + (j : 8) + 1 + (m : 8) et on répétera la même opération avec le même masque, puisque nous nous sommes arrangés pour que les déplacements des bits cherchés dans les octets qui les contiennent soient identiques.

Cette manière de représenter, pour chaque module x, l'ensemble $\hat{U}(x)$ est à la fois compacte et permet un traitement extrêment rapide, les instructions utilisées étant parmi les plus rapides du 10070.

Evaluons l'encombrement de cette matrice pour une unité de traitement comportant 100 classes et 100 modules (ce qui représente une unité de traitement déjà considérable) :

Une ligne occupe (100 : 8) + 1, soit 13 octets.

Pour la matrice complète : 100 x 13 = 1300 octets, soit 325 mots. L'encombrement est donc tout à fait raisonnable, même pour une unité de traitement importante.

L'intérêt de faire résider cette matrice en mémoire centrale est que tout accés à la mémoire secondaire est évité si les textes voulus sont effectivement présent en mémoire centrale.

Si, malgré la compacité de la représentation de la matrice $[U_Z]^t$, il n'est pas possible de la faire résider en mémoire centrale, nous proposons, la solution suivante : le texte de tout module x sur la mémoire secondaire comportera un en-tête constituant la liste des classes de $\hat{U}(x)$. Cette liste se présentera sous forme d'une suite de bits : le $n^{é}$ bit à 1 signifie que la classe d'indice n fait partie de $\hat{U}(x)$. Nous admettons que cette suite de bits, lue en une seule fois sur la mémoire secondaire, est assez petite pour tenir en mémoire centrale. Nous n'aurons à effectuer qu'un seul accés à la mémoire secondaire pour obtenir la liste des textes à charger.

3.3.2 Mise à jour de la table indicatrice de la présence des textes en mémoire centrale

Cette table, appellée $$PIMC était de $|M| - 1$ octets dans le cas du recouvrement pour les modules seuls. Elle sera maintenant de $|E_c| - 1$

(E_c désignant l'ensemble des classes et des modules de l'unité de traitement ayant effectivement donné lieu à génération d'un texte objet).

De même que dans le cas du recouvrement sur les modules, nous conviendrons que le texte objet de chaque classe et module x sur la mémoire secondaire comporte un enregistrement qui est la liste des modules et des classes dont le texte en mémoire centrale est détruit par le chargement de x. La mise à jour de la table $$PIMC ne présente donc aucune difficulté.

3.3.3 Edition de liens

Le principe de l'édition de liens qui a été donné dans le cas du recouvrement sur les modules reste le même : de la même manière que nous éditions un par un les modules, nous éditerons maintenant un par un les modules et les classes. Pour rendre possible cette édition de liens, le préditeur générera un module objet donnant la définition absolue de tous les $noms et noms de classes et des modules.

4 - CONCLUSION

Les problèmes posés par le recouvrement des textes objets soulévent peu de difficultés du fait de la compatibilité avec le recouvrement des segments de variables. Nous avons vu que cette compatibilité pouvait être atteinte au prix de quelques palliatifs, sans conséquence importante. La partie la plus délicate est la réalisation de l'édition de liens. L'éditeur de liens du moniteur Siris7 n'est pas tout à fait adapté au traitement que nous avons effectuer, ce qui implique, comme nous l'avons vu, que les éditions de liens des modules sont effectuées successivement, grâce au prédéditeur de liens qui génére un module objet intermédiaire. Ce procédé un peu lourd pourrait être évité aux prix de la réalisation d'un éditeur de liens spécifique à CIVA, qui pourrait travailler de manière plus liée avec le prédéditeur de liens. Un tel éditeur de liens serait évidemment de structure assez voisine de l'éditeur dont nous disposons. Cependant, nous n'envisagerons pas ici son étude.

7ème PARTIE

ORGANISATION GENERALE DU PREEDITEUR DE LIENS

1 - PRINCIPALES FONCTIONS A ASSURER

1. 1. - Calcul des fermetures transitives

1. 2. - Calcul de la matrice [R] associée à (E,R)

1. 3. - Coloration et orientation de (E,R)

1. 3. 1. - Méthode de Welsh et Powell

1. 3. 2. - Essais de l'algorithme de Welsh et Powell

1. 4. - Allocation

2 - ENCHAINEMENT DES DIFFERENTES FONCTIONS.

7ème PARTIE

ORGANISATION GENERALE DU PREEDITEUR DE LIENS

1 - PRINCIPALES FONCTIONS A ASSURER

1. 1. - Calcul de fermetures transitives

La première tâche que doit assurer le préditeur de liens est, partant du fichier descriptif des classes et des modules, de bâtir les graphes (M,A) et (E,U). Nous ne détaillerons pas cette phase de travail qui ne présente ni difficulté théorique, ni difficulté de programmation.

Nous avons vu (cf Partie 5 § 1.2) que la détermination du graphe (E,R) était particulièrement facile si les graphes (M,A^*) et (E,U^*) étaient représentés sous forme matricielle. Cette raison nous conduit de manière générale à représenter tous les graphes sous forme de matrices booléennes. Le nombre de modules et classes pouvant être élevés, nous sommes contraints de représenter ces matrices sous forme de matrices de bits. Une telle représentation a déjà été introduite dans la partie 6 § 3.3. Nous procéderons de même qu'il a été vu alors mais, au lieu de cadrer les débuts des lignes sur des frontières d'octets, nous les cadrerons sur des frontières de mots. Nous perdons ainsi un peu plus de place, mais sommes moins soumis à des contraintes d'espace mémoire pendant la préédition de liens que nous ne le sommes pour le chargeur. Ce cadrage sur des frontières de mots est destiné à faciliter la programmation, et donc à accélérer la vitesse d'exécution du préditeur de liens, de nombreuses instructions du 10070 étant à adressage par mot.

Le choix d'une représentation sous forme de matrice de bits peut faire craindre une manque d'efficacité, puisque les bits ne sont pas directement adressables. Cet inconvénient peut souvent être compensé par l'utilisation de méthodes bien choisies. Par exemple nous avons vu à propos du chargeur que la recherche d'un élément donné d'une matrice consistait à calculer l'adresse de l'octet dans lequel se trouvait l'élément cherché et en même temps à sélectionner un parmi huit masques. Nous pourrons ici reprendre ce procédé en utilisant 32

masques, puisque nous travaillons sur des mots.

Le fait d'avoir choisi une représentation sous forme de matrice associée pour (M,A) et (E,U) nous conduit à sélectionner, parmi toutes les méthodes de calcul de fermeture transitive existantes, celles qui sont facilitées par une telle représentation.

L'algorithme bien connu de Warshall, comme nous allons le voir, est particulièrement bien adapté aux représentations matricielles que nous avons choisies. Son principe est le suivant : Soit $[M]$ la matrice associée au graphe dont on cherche la fermeture transitive.

Pour chaque ligne d'indice i de $[M]$ on substitue à chaque ligne d'indice j la somme logique des lignes i et j, si l'élément $[M](j,i)$ est égal à 1. Nous appelons somme logique de deux lignes de n bits, la suite de n bits obtenus en effectuant un ou logique entre les bits de même rang des deux lignes.

Nous pouvons remarquer que dans cet algorithme

- l'élément $[M](j,i)$ est testé à i fixé alors que j varie : on accède donc à la matrice $[M]$ colonne par colonne. Nous avons déjà indiqué une méthode d'accès colonne par colonne pour des matrices dont les lignes étaient cadrées à des frontières d'octets. Cette méthode est aisément transposable au cas des matrices dont les lignes sont cadrées sur des frontières de mots.

- Une opération OU est réalisée fréquemment entre deux lignes. Le 10070 est doté d'une instruction permettant de réaliser une opération OU entre deux mots. Cette opération est parmi les plus rapides du 10070.
Elle nous permettra de traiter en une seule fois 32 éléments d'une ligne. Nous voyons ici que la représentation que nous avons choisie est extrêmement favorable à l'application de cet algorithme.

Nous pouvons donc grâce à l'algorithme de Warshall déterminer facilement les fermetures transitives. Cet algorithme donnant les fermetures strictes , nous devrons néanmoins forcer à 1 la diagonale principale des matrices obtenues.

1. 2. - Calcul de la matrice $[R]$ associée à (E,R)

Cette matrice est déduite de (M,A^*) et de (E,U^*) par calcul matriciel. Bien que cette matrice soit symétrique (elle représente une relation non orientée que nous assimilons à une relation réflexive), nous la calculerons dans son ensemble. En effet ne calculer que la diagonale supérieure (par exemple) amènerait de telles complications dans l'organisation de l'informa-

tion que le bénéfice serait nul. D'autre part, nous nous proposons par la suite d'orienter (E,R), ce qui se traduira par des transformations sur l'ensemble de la matrice $[R]$. (parmi deux éléments $[R](i,j)$ et $[R](j,i)$ égaux à 1, il faudra opter pour l'un d'eux). Il sera alors souhaitable de disposer de $[R]$ dans son ensemble.

Un moyen pratique de calcul de $[R]$ a été donné (cf Partie 5 § 1.3). Il nécessite la transposition de certaines sous matrices. Nous réaliserons donc un algorithme de transposition.

Nous profiterons de cet algorithme de transposition pour simplifier et accélerer les calculs de produits matriciels.

Soient en effet deux matrices booléennes A et B de n colonnes. L'élément d'indices (i,j) de leur produit est :

$$[C](i,j) = \sum_{k=1}^{n} A\ (i,k) \cdot [B](k,j), \text{ ce qui s'écrit aussi}$$

en utilisant la transposée de $[B]$: $\sum_{k=1}^{n} [A](i,k) \cdot [B]^t(j,k)$.

Pour effectuer le produit de $[A]$ et $[B]$ transposons préalablement $[B]$.

Appliquons l'instruction AND du 10070, qui réalise un ET logique bit à bit entre deux mots, entre les 32 premiers éléments de la ligne i de A et de la ligne j de $[B]^t$. Si le mot obtenu est différent de 0, il est inutile d'aller plus loin : $[C(i,j)]$ est égal à 1. Si le mot obtenu est nul, on traitera ensuite les 32 éléments suivants des lignes, et ainsi de suite, jusqu'à épuisement des éléments (si la dernière opération AND donne un résultat nul, c'est que $[C](i,j) = 0$), ou jusqu'à ce qu'une instruction AND donne un résultat non nul (alors $[C](i,j) = 1$). Cette méthode accélére considérablement les calculs de produits de matrices. La transposition qu'elle nécessite est rapide. Dans le cas de nos calculs nous pouvons, pour l'éviter au maximum procéder de la manière suivante :

Rappelons que nous avons à calculer les sous matrices de $[R]$:

$$[R_2] = [A] \cdot [U_2]$$
$$[R_3] = [U_2]^t [A]$$
$$[R_4] = [U_2]^t \cdot [A] \cdot [U_2]$$

La matrice $[A]$ est symétrique.

Calculons $[U_2]^t$ (en effectuant une transposition).

Puisque $[A]$ est symétrique, il est inutile de la transposer pour calculer $[R_3]$. Notre procédé nous donne $[R_3]$ immédiatement.

Remarquons maintenant que $[R_4]$ est le produit de $[U_2]^t$ par $[A] \cdot [U_2]$

Notre méthode exige, pour calculer ce produit, de transposer $[\bar{A}] \cdot [U_2]$, ce qui donne $[U_2]^t \cdot [\bar{A}]$ en tenant compte une nouvelle fois de la symétrie de $[\bar{A}]$, soit précisement la matrice $[R_3]$ que nous venons de calculer. $[R_4]$ se calcule donc sans aucune transposition.

Enfin $[R_2]$ s'obtient simplement comme la transposée de $[R_3]$.

En enchaînant correctement les transpositions avec notre méthode de calcul de produit, nous aboutissons à un procédé très rapide : les matrices $[R_2]$, $[R_3]$, $[R_4]$ se calculent aux prix de 2 transpositions qu'il aurait fallu faire de toute façon si les produits avaient été calculés de manière classique.

1. 3. - Coloration et orientation de (E,R)

1.3.1 Méthode de Welsh et Powell

La coloration d'un graphe est un problème délicat. Les algorithmes existants sont nombreux et admettent de nombreuses variantes. Signalons simplement la méthode de Maghout, s'appuyant sur les propriétés des équations booléennes, et les procédures par séparation et évaluation appliquant le principe du criblage. Parmi les divers algorithmes, notre attention a été retenue parune méthode heuristique, de mise en oeuvre remarquablement simple, due à Welsh et Powell. Nous pensons que, dans la mesure où cette méthode fournit de bonnes colorations (c'est-à-dire que le nombre des couleurs utilisé est voisin du nombre chromatique), il est inutile de chercher, au prix de calculs excessifs, une amélioration minime. En effet il n'est pas capital pour nous d'obtenir le minimum strict de couleurs. Ayant sélectionné cette méthode, nous nous sommes efforcés de tester ses performances. Nous verrons plus loin les résultats qu'elle apporte.

L'algorithme est le suivant :

(1) On numérote de 1 à n les sommets par degrés décroissants $d_1, d_2, \ldots, d_n$.

On pose $i = 1$.

(2) On colore le point d'indice le plus faible à l'aide d'une couleur c_i. Puis on colore avec la même couleur le point d'indice le plus petit non adjacent au premier point. Ainsi de suite, on colore en c_i le point d'indice le plus faible qui n'est adjacent à aucun des points déjà colorés. Lorsqu'un tel point n'existe plus, on utilise une nouvelle couleur (on incrémente i de 1) et on recommence le processus en (2) en traitant les points non déjà colorés.

Ce processus se termine lorsque tous les points sont colorés.

Cet algorithme permet de trouver un majorant du nombre chromatique, exprimé par le théorème suivant :

Si k est l'indice du dernier sommet tel que $k \leqslant d_k + 1$, alors le nombre chromatique est au plus égal à k.

Démonstration : Montrons que lors du déroulement de l'algorithme, tous les points sont colorés si i atteint la valeur k :

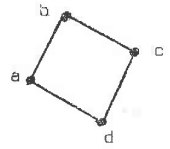
- Si i a atteint la valeur k, les points d'indices 1 à k ont été colorés, puisque, lorsqu'on change de couleur, on commence par colorer les points d'indices les plus faibles.
- Considérons maintenant un point x d'indice supérieur à k.

Supposons qu'il n'est pas coloré. S'il n'a pas été coloré pour une valeur donnée de i, c'est qu'il était adjacent à au moins un point coloré en c_i. Comme i a pris k valeurs, le degré de x est au moins égal à k. Comme les points sont classés par ordre de degré décroissant, le degré de x est inférieur ou égal à d_{k+1}. Donc $d_{k+1} \geqslant k$.

D'où $d_{k+1} + 1 \geqslant k + 1$, ce qui contredit l'hypothèse du théorème. Nous sommes donc certains que tous les points sont colorés lorsque $i = k$.

Notons que ce théorème donne une majoration du nombre chromatique ; l'algorithme peut donner un résultat meilleur que cette majoration.

Etudions par exemple le cas suivant :

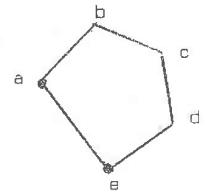


Tous les sommets sont de degré 2. k est égal à 3

L'algorithme conduit à une 2 - coloration, donc un résultat meilleur que ne le laisse prévoir le théorème.

Au contraire pour un cycle impair, l'algorithme aurait donné une 3 - coloration.

Par exemple :



Numérotons les points :

	a	b	c	d	e
numéros :	1	2	3	4	5
degrés :	2	2	2	2	2

Nous avons toujours k = 3.

Dans l'algorithme, pour i = 1, on colore en C_1 les points a et c

pour i = , on colore en C_2 les points b et d

enfin pour i = 3 on colore en C_3 le point e

Nous avons testé la méthode de Welsh et Powell et avons constaté que les résultats de l'algorithme étaient en fait bien meilleur que ne le laisse espérer la majoration.

1.3.2 Essais de l'algorithme de Welsh et Powell

Nous nous proposons de tester les résultats donnés par la méthode de Welsh et Powell sur une série de graphes, où nous ferons varier le nombre des sommets et le nombre d'arêtes.

Pour un graphe de n sommets, le nombre maximum d'arêtes pouvant exister est $\frac{n\ (n-1)}{2}$ (le graphe est alors complet).

Nous appellerons densité d'arêtes d'un graphe comportant N arêtes, la quantité

$$d = \frac{}{n\ (n-1)}$$

De même nous parlerons de la densité d'arcs d'un graphe sans circuit[1].

Nous avons étudié une série de graphes, en faisant varier le nombre de sommets. A nombre de sommets fixé nous avons comparé le nombre chromatique donné par Welsh et Powell, avec le nombre chromatique réel, en faisant varier la densité.

Le problème était de générer des graphes au hasard, de densité fixée, et de nombre chromatique connu.

Pour cela nous nous appuyons sur la proposition suivante :

le nombre chromatique d'un graphe G sans circuit transitif est égal au nombre maximal de sommets d'un chemin.

Démonstration : Cette proposition découle des propriétés des graphes de comparabilité : remplaçons les arcs par des arêtes dans le graphe transitif Nous obtenons un graphe de comparabilité G'. Un tel graphe est γ-parfait : son nombre chromatique est égal au nombre maximal de sommets d'une clique.

(1) En effet le nombre maximal d'arcs d'un graphe sans circuit de n sommets est également n (n-1)/2.

(à cause de la transitivité). Réciproquement les sommets d'une clique forment un chemin de G, en vertu du théorème de Redeï (toute clique comporte un chemin hamiltonien).

Donc les sommets de tout chemin de G constituent une clique dans G' ; Si un chemin de G comporte au plus γ sommets, une clique de G' comportera au plus γ sommets. G' étant γ parfait, on obtient la proposition annoncée.

Il est donc pratique de tester la méthode de Welsh et Powell sur de tels graphes; ce qui restreint un peu la généralité des tests. Nous ne pensons cependant pas que ces graphes facilitent ou dégradent l'algorithme, la propriété "graphe de comparabilité" n'intervenant absolument pas dans le fonctionnement de l'algorithme.

Etudions une méthode pour générer au hasard un graphe sans circuit transitif de n sommets et de densité d donnée (en fait de densité voisine de d).

Nous ne pouvons pas générer un graphe sans circuit au hasard, puis calculer sa fermeture transitive : la densité ne serait pas contrôlée.

Nous proposons donc de construire le graphe progressivement.

Soient $\{x_1, x_2, \ldots, x_n\}$ les sommets du graphe que nous cherchons à construire. Nous procéderons en n étapes.

Aux étapes 1,2,...,n nous construisons des graphes $G_1, G_2, \ldots, G_n$, transitifs de densité aussi voisine de d que possible, de sommets $\{x_1\}$, $\{x_1, x_2\}$,..., $\{x_1, x_2, \ldots, x_n\}$. Un graphe G_{k+1} sera déduit de G_k par introduction du sommet x_{k+1} et d'arcs d'origine x_{k+1} en quantité juste suffisante pour que G_{k+1} soit de densité d (ainsi G_k est un sous graphe de G_{k+1}).

Calculons le nombre d'arcs d'origine x_{k+1} de telle manière de G_{k+1} soit de densité d :

G_k est de densité d comporte donc $\frac{k\ (k-1)\ d}{2}$ arcs.

G_{k+1} doit être de densité d. Il doit donc comporter $\frac{(k+1)\ kd}{2}$ arcs.

Sans G_{k+1} il faut donc introduire $\frac{d}{2}\left[k\ (k+1) - k\ (k-1)\right] = kxd$ arcs.

Donc si à toute étape k + 1, on introduit kxd arcs d'origine x_{k+1}, on obtiendra finalement un graphe G_n de densité d.

Nous devons en plus nous arranger pour que les G_k soient transitifs : supposons que G_k soit transitif. Pour construire G_{k+1}, nous introduirons un arc au hasard d'origine x_{k+1}, puis nous relierons par un arc x_{k+1} à tous les successeurs du point situé à l'extrémité de cet arc. Si la densité du graphe obtenu est inférieure à d (ce qui équivaut à dire que les arcs d'origine x_{k+1} sont en nombre inférieur à dxk), nous introduirons un nouvel arc au hasard

d'origine x_{k+1} et nous relierons à nouveau x_{k+1} aux successeurs de l'extrémité de cet arc, et ainsi de suite, jusqu'à ce que la densité du graphe obtenue atteigne la valeur d. Nous obtiendrons finalement un graphe G_{k+1} (transitif puisque G_k était transitif) de densité un peu supérieur à d[1]. Pendant ce traitement on suppose connue la longueur des cheminsdd'origines $\{x_1, x_2, \ldots, x_k\}$. Il est immédiat d'en déduire la longueur des chemins d'origines $\{x_1, x_2, \ldots, x_{k+1}\}$ au fur et à mesure que l'on ajoute de nouveaux arcs.

Ce procédé permet donc de construire finalement un graphe G_n de sommets $\{x_1, x_2, \ldots, x_n\}$, de densité à peu près égale à d ; (cette approximation n'a aucune importance : nous cherchons simplement à tester l'algorithme de Welsh et Powell pour différents ordres de grandeurs de densités) transitif, et dont le plus grand chemin est connu immédiatement.

Nous avons utilisé cette méthode pour produire des graphes de 32, 64, 128, 256 sommets.

Pour chacune de ces valeurs, nous avons étudié des graphes de densités 20 %, 40 %, 60 %, 80 % (cinq exemples ont été traités pour chacune de ces densités).

Nous avons donc traité au total 80 graphes.

Les résultats ci-joints montrent que la majoration donnée par le théorème est excéssive, mais que par contre l'algorithme conduit à des résultats tout à fait remarquables. Le nombre de couleurs trouvé par l'algorithme de Welsh et Powell est presque toujours égal au nombre chromatique effectif.

Ces tests justifient donc le choix de la méthode de Welsh et Powell, puisque, outre son extrême rapidité, elle conduit à des résultats très satisfaisants.

(1) Les résultats obtenus montrent qu'on s'approche bien de la densité désirée. Il faut cependant arrêter l'introduction de nouveaux arcs non pas lorsque le nombre d'arcs issus de x_{k+1} est supérieur à dxk, mais plutôt lorsque la densité obtenue est supérieure à d. Ceci évite de cumuler les excés d'arcs d'une étape à l'autre.

DIMENSION DU GRAPHE : 32

% ARCS	NB C. REEL	NB C. POWELL	NB C. MAXI
20	9	9	9
23	8	8	8
19	7	7	7
20	7	7	8
19	7	7	8
40	9	9	13
41	9	9	14
39	10	10	13
40	10	10	13
40	9	9	13
59	11	12	18
60	12	12	17
60	15	15	19
60	12	12	18
60	13	13	18
79	15	15	24
79	16	16	23
79	16	16	24
80	16	16	24
79	16	16	24

DIMENSION DU GRAPHE : 64

% ARCS	NB C. REEL	NB C. POWELL	NB C. MAXI
20	9	9	16
19	13	13	15
20	9	9	15
19	9	9	14
20	11	11	14
40	15	15	26
39	12	12	25
40	13	13	26
40	12	12	26
40	13	13	26
60	18	18	35
60	17	17	35
59	16	16	35
60	15	15	36
60	20	20	36
80	25	25	47
80	23	23	47
79	26	26	47
80	24	24	46
80	25	25	48

DIMENSION DU GRAPHE : 128

% ARCS	NB C. REEL	NB C. POWELL	NB C. MAXI
20	17	17	30
20	14	14	29
20	15	15	27
20	15	15	30
20	14	14	28
40	19	19	48
40	21	21	49
40	17	17	49
39	19	19	48
40	20	20	48
60	23	23	70
60	27	27	69
60	26	26	70
60	24	24	70
60	28	28	70
80	39	39	93
80	42	42	93
80	40	40	94
80	34	35	94
80	34	35	93

DIMENSION DU GRAPHE : 256

% ARCS	NB C. REEL	NB C. POWELL	NB C. MAXI
20	18	18	56
19	17	17	54
20	17	17	58
20	19	19	55
19	18	18	57
40	22	22	96
40	28	28	93
39	25	25	94
40	25	25	98
40	22	23	96
59	33	34	139
60	39	39	139
60	32	32	137
60	34	34	139
60	37	37	139
79	55	55	187
79	53	53	186
80	58	58	185
79	53	53	186
80	53	53	185

1. 4. - Allocation

Le problème est de passer d'un graphe sans circuit $(\mathcal{E}, \Gamma)$ (ce sera le graphe (M,A) ; ou le graphe (E,R) après orientation des arêtes) à la détermination d'une répartition. Soit $\lambda(x)$ la longueur du sommet x de $\mathcal{E}$ (ce sera soit la taille du segment de variables de x, soit la taille de son texte objet).

Nous avons trouvé que l'origine O (x) à donner à un segment x est égale à la longueur par les sommets du plus grand chemin d'extrémité x (x exclus).

Nous obtenons ainsi une répartition dont la longueur est la longueur L par les sommets du plus grand chemin du graphe.

Les O (x) ainsi obtenus, sont tels que deux segments correspondant à des sommets x et y qui sont sur un même chemin de $(\mathcal{E}, \Gamma)$ soient disjoints. Si, de la même manière, nous calculons les origines O (x), non plus sur le graphe $(\mathcal{E}, \Gamma)$, mais sur $(\mathcal{E}, \Gamma^{-1})$, nous obtiendrons une répartition de même longueur, ou les contraintes de non recouvrement seront satisfaites : en effet si $\{x_1, x_2, \ldots, x_p\}$ est un chemin de $(\mathcal{E}, \Gamma)$, alors $\{x_p, x_{p-1}, \ldots, x_1\}$ est un chemin de $(\mathcal{E}, \Gamma^{-1})$; donc la longueur du plus grand chemin de $(\mathcal{E}, \Gamma)$ est égale à la longueur du plus grand chemin de $(\mathcal{E}, \Gamma^{-1})$ et d'autre part, deux points x et y sur un même chemin de $(\mathcal{E}, \Gamma)$ sont sur un même chemin de $(\mathcal{E}, \Gamma^{-1})$.

Nous allons voir qu'il est facile de trouver un processus appliqué à $(\mathcal{E}, \Gamma)$, mais donnant les quantités O (x) relatives à $(\mathcal{E}, \Gamma^{-1})$.

Pour cela on explore $(\mathcal{E}, \Gamma)$ par une pile[1].

Lorsque la pile regresse, c'est que tous les successeurs du point sortant de la pile ont été trouvés. Autrement dit, si nous avons deux points x et y tels que $x \Gamma y$, y sortira de la pile avant x.

Supposons que nous implantions les segments correspondant aux points de sortie de $(\mathcal{E}, \Gamma)$ à l'adresse 0, et un segment correspondant à un sommet x, qui n'est pas un point de sortie, à l'adresse :

$$O(x) = \max_{y \in \Gamma(x)} \left[O(y) + \lambda(y) \right]$$

Nous obtenons ainsi les quantités O (x) relatives à $(\mathcal{E}, \Gamma^{-1})$.

Notons que nous aurions pu obtenir les quantités O (x) relatives à $(\mathcal{E}, \Gamma)$ soit en explorant $(\mathcal{E}, \Gamma^{-1})$, soit en explorant $(\mathcal{E}, \Gamma)$ mais en calculant des quantités O' (x) de la manière suivante : $O'(x) = -\lambda(x) + 1$ si x est point de sortie, sinon

$$O'(x) \min_{y \in \Gamma(x)} \left[O'(y) \right] - \lambda(x), \text{ ce qui nous donnerait une}$$

(1) Cf par exemple : "Problèmes de cheminement dans les graphes" par J.C. DERNIAME et C. PAIR (Dunod).

implantation calculée sur des adresse négatives (la plus haute adresse utilisée est 0). Il suffit ensuite de translater toutes les origines d'une quantité égale à la longueur de la répartition pour obtenir les O (x) relatifs à $(\mathcal{E},\Gamma)$.

Ces opérations sont inutiles si l'on se contente des O (x) relatifs à $(\mathcal{E},\Gamma^{-1})$. Nous avons donc retenu cette solution par souci de rapidité.

2 - ENCHAINEMENT DES DIFFERENTES FONCTIONS

Nous venons d'étudier les différentes fonctions du prééditeur de liens. Elles s'organisent selon le schéma suivant :

a) si l'ensemble C des classes est vide :

il est inutile de construire (E,U) et (E_V,R). En effet le graphe (M,A) nous indique les contraintes de non recouvrement pour les segments de variables. Nous prendrons donc comme graphe $(\mathcal{E},\Gamma)$ le graphe (M,A) la longueur λ étant égale à ℓ (longueur des segments de variables) et réaliserons l'allocation des variables. Si la taille de la mémoire est suffisante, nous obtenons une unité de traitement viable. Sinon nous essaierons de déterminer un recouvrement sur les textes objets des modules (en prenant cette fois λ égale à 'ℓ', longueur des textes objets). Si la taille de mémoire est suffisante, nous obtenons une unité de traitement avec recouvrement sur les variables et les textes des modules. Sinon, il n'y a plus rien à faire : l'unité de traitement demande trop de mémoire centrale.

b) si C n'est pas vide

Nous commencerons comme dans le cas a par déterminer un recouvrement sur les variables, puis si la taille de la mémoire reste insuffisante, sur les textes objets des modules. Cependant, si nous obtenons à nouveau un résultat trop grand pour la taille de la mémoire centrale, il nous est cette fois possible d'effectuer une nouvelle tentative en cherchant un recouvrement sur les textes objets des modules et des classes (le graphe $(\mathcal{E},\Gamma)$ sera à nouveau pris égal à (E,R)). Nous obtiendrons alors une unité de traitement avec recouvrement sur les variables d'une part, et tous les textes objets d'autre part. Si la taille de la mémoire reste insuffisante, nous avons alors épuisé toutes les possibilités. L'unité de traitement demande trop de mémoire centrale.

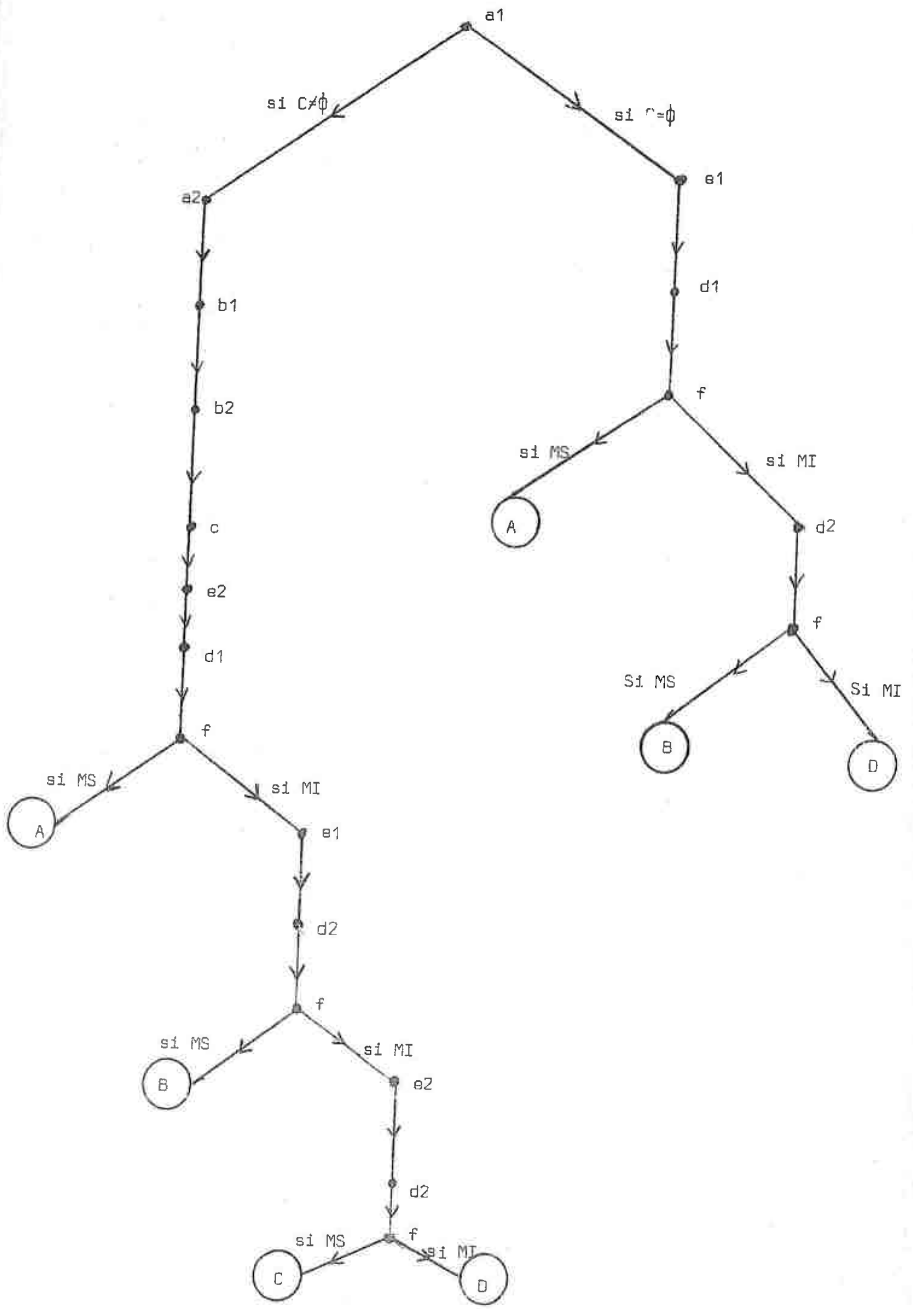
Nous pouvons décomposer ce traitement en 6 traitements élémentaires. Un traitement élémentaire pourra éventuellement assurer plusieurs tâches analogues :

- traitement a
  - tâche 1 : construction de (M,A) à partir du fichier descriptif
  - tâche 2 : construction de (E,U) à partir du fichier descriptif
- traitement b
  - tâche 1 : calcul de (M,A^*)
  - tâche 2 : calcul de (E,U^*)
- traitement c : construction de (E,R) et orientation de ses arêtes
- traitement d
  - tâche 1 : détermination d'une allocation, pour un graphe $(\mathcal{E},\Gamma)$ donné et une longueur ℓ.
  - tâche 2 : même chose pour une longueur ℓ'

  (Ce traitement trouve la matrice associée à $(\mathcal{E},\Gamma)$ dans un endroit fixe de mémoire (ce qui évite l'utilisation de l'adressage indirect)). Suivant que l'on désire traiter (M,A) ou (E,R), il faut copier les matrices associées à (M,A) ou (E,R) dans cet endroit fixe. Cette opération est notée symboliquement $(\mathcal{E},\Gamma) := (M,A)$ ou $(\mathcal{E},\Gamma) := (E,R)$ dans le traitement e.
- traitement e :
  - tâche 1 : $(\mathcal{E},\Gamma) := (M,A)$
  - tâche 2 : $(\mathcal{E},\Gamma) := (E,R)$
- traitement f : comparaison de la taille de mémoire nécessaire au lancement de l'unité de traitement, à la taille de mémoire dont on dispose.

Une tâche sera désignée par la lettre minuscule correspondant au traitement, suivie éventuellement du numéro de la tâche dans le traitement (si le traitement comporte plusieurs tâches).

Le traitement du préditeur de liens peut se schématiser par une arborescence dont un chemin maximal (il y en a 7) représente la séquence de tâches effectuées par le prééditeur pour une unité de traitement. Les sigles M I et M S signifient respectivement "Mémoire centrale Insuffisante" et "Mémoire centrale Suffisante". (figure page 132).



Le prééditeur de liens peut donc avoir 7 comportements différents. Suivant que le traitement se termine en un point marqué A, B, C, ou, D, on obtient :

A : un recouvrement uniquement sur les variables.

B : un recouvrement sur les variables d'une part, et les textes des modules d'autre part.

C : un recouvrement sur les variables d'une part, et tous les textes objets d'autre part.

D : la taille de la mémoire centrale reste insuffisante alors que toutes les possibilités de recouvrement ont été épuisées.

CONCLUSION

L'emploi d'une gestion statique de mémoire avait été envisagée dans l'hypothèse d'unités de traitement répétitives. Cette hypothèse nous paraissait primordiale du fait de la complexité de la structure d'une unité de traitement, nous ne pouvions envisager des traitements coûteux que dans la mesure où leurs résultats seraient fréquemment exploités.

Nous avons créé en définitive un produit assez rapide pour que son emploi ne soit pas limité au cadre de cette hypothèse restrictive. Nous avons vu en effet que des traitements souvent considérés comme coûteux (par exemple les déterminations de fermetures transitives) pouvaient être accélérés par une représentation, ou par une exploitation judicieuse de l'information. D'autre part le problème délicat de la coloration d'un graphe a été résolu grâce à l'algorithme très simple et rapide de Welsh et Powell. Des essais ont montré l'excellence de ses résultats.

Les travaux de mise au point et de tests des principaux algorithmes (Warshall, Welsh et Powell) montrent que ces algorithmes nécessitent un temps de l'ordre de 5 à 10 secondes pour des graphes de 256 sommets, ce qui représente des unités de traitement extrêment importantes.

Cette rapidité permet donc de lever dans une large mesure l'hypothèse restrictive de répétivité des unités de traitement, et de rendre possible l'utilisation courante du prééditeur de liens.

La rapidité a été obtenue grâce aux choix de méthodes heuristiques. Les approximations apparaissent à deux niveaux : d'une part dans la recherche d'une meilleure orientation des arêtes du graphe traduisant les contraintes de non recouvrement ; d'autre part dans la méthode de coloration de ce graphe.

Cependant la première de ces approximations, s'il est incontestable qu'elle puisse avoir desrépercussions sensibles, offre en contrepartie l'avantage d'unifier les traitements relatifs aux variables d'une part, et aux textes objets d'autre part. Alors que l'on pouvait craindre de voir doubler la durée des traitements dans les cas où il est indispensable de faire appel au recouvrement sur les textes objets, il se trouve au contraire que la détermination de ce recouvrement est pratiquement immédiate, la presque totalité du travail ayant été déjà menée à bien pour les variables. Cette [illegible] des traitements a facilité également dans une large mesure la réalisation du prééditeur de liens, ce qui ne peut qu'assurer sa fiabilité.

La seconde de ces approximations ne semble pas avoir de conséquences sur la qualité des services du prééditeur. En effet des tests assez abondants ont montré l'efficacité de la méthode de Welsh et Powell. Il paraît donc peu souhaitable de faire appel à des méthodes plus précises, mais conduisant à des traitements excessivement longs.

Les avantages (pour la recherche d'une meilleure orientation) ou le peu d'inconvénients pour l'algorithme de Welsh et Powell qu'amènent ces approximations les justifient donc dans une large mesure.

Avec cette méthode de gestion statique, l'encombrement de la mémoire centrale risque d'être plus important qu'avec une méthode de gestion dynamique. Cela est dû aux approximations que nous venons d'évoquer, aussi bien qu'à des raisons théoriques qui ont été exposées dans notre étude. Cependant, nous avons atteint l'objectif que nous nous étions fixé, qui était d'assurer un accés rapide aux informations. Nous évitons ainsi les lourdes pertes de temps à l'exécution, dues à une méthode de gestion dynamique. L'hypothèse des unités de traitement répétitives reprend ici tout son sens, car les gains de temps seront alors substantiels.

Une certaine lourdeur de mise en oeuvre du prééditeur se manifeste pourtant dans le traitement du recouvrement sur les textes objets. Cette lourdeur est due à la nécessité de se plier aux contraintes de l'éditeur de liens et du système de gestion de fichiers du moniteur Siris 7. Il est certain qu'un éditeur de liens de conception traditionnelle ne peut pas s'adapter très efficacement aux exigences d'un système de programmation modulaire. Prééditeur de liens et éditeur de liens devraient former un ensemble harmonieux si l'on voulait voir disparaître toute lourdeur de mise oeuvre. Nous sommes ainsi amenés à penser, sinon à la réécriture, du moins au remodelage du software de base, problème qui se pose chaque fois que l'on veut implanter une application aux exigences très spécifiques sur un système d'exploitation préexistant. Nous ne pensons cependant pas qu'une telle solution soit souhaitable, du moins pour une machine importante, car elle ne tient pas compte des impératifs actuels de l'exploitation d'un centre de calcul.

Malgré une relative et inévitable lourdeur de mise en oeuvre, le prééditeur de liens a été conçu avec un souci constant de rapidité. La rapidité dans l'accomplissement des tâches qu'il assure doit en faire un service pratique. La rapidité d'exécution qu'il confère aux unités de traitement qui lui sont soumises doit en faire un service efficace.

ANNEXE

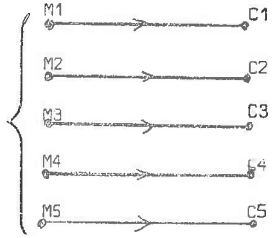
Les exemples suivants ont été soumis au prééditeur de liens, pour détermination de l'implantation des segments de variables.
Les paragraphes 1 et 2 donnent respectivement la définition des exemples et les résultats fournis par le prééditeur.

1 - DEFINITION DES EXEMPLES

Exemple 1

graphe des appels :

graphe des directives "utilisé" :

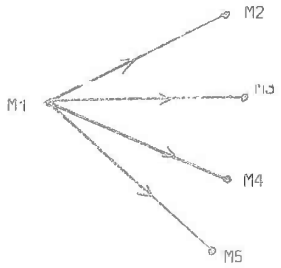


La taille des segments de variables est de 10 mots pour chaque module et 20 mots pour chaque classe.

Observations : le graphe des appels comportant un seul chemin maximal, aucun recouvrement n'est possible.

Exemple 2

graphe des appels :

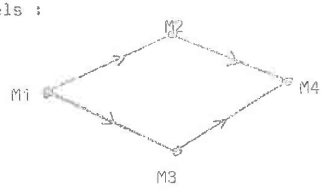


Le graphe des directives "utilise" et les tailles des segments sont les même que dans l'exemple 1.

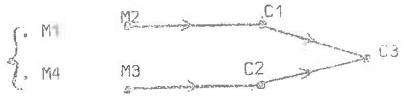
Observations : Deux modules M_i et M_j (i et $j \geq 2$) peuvent se recouvrir puisqu'ils ne sont pas sur un même chemin du graphe des appels. De même les classes C_i et C_j qu'ils "utilisent" peuvent se recouvrir. De plus C_i peut recouvrir M_j et C_j peut recouvrir M_i.

Exemple 3

graphe des appels :



graphe des directives "utilise" :



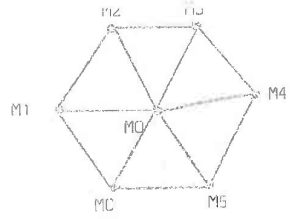
Les tailles des segments sont les mêmes que dans les exemples précédents.

Observations : les modules M2 et M3, non sur un même chemin du graphe des appels peuvent se recouvrir. Les représentations des variables de la classe C2 n'existent pas en même temps que les représentations des variables de M3 et C2. C1 peut donc recouvrir M3 et C2. De même C2 peut recouvrir M2 et C1. Par contre la classe C3, "utilisée" quelque soit le cheminement dans le graphe des appels ne saurait être en recouvrement avec aucun autre élément.

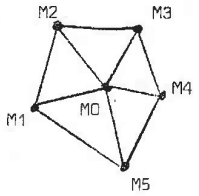
Exemples 4 et 5

Les graphes (E,R) soumis au prééditeur sont respectivement :

exemple 4 :



exemple 5 :



Les tailles des segments ont été choisies égales à 10 mots.

Observations : les résultats obtenus pour de tels graphes ont été étudiés dans la partie 5, § 8.2.3.

Exemples 6,7,8

Le graphe (E,R) soumis au prééditeur est celui défini au § 9.1, partie 5.

Observations : nous avions vu (cf § 9.2) que la méthode d'allocation était sensible aux choix d'un ordre sur les couleurs. Pour les exemples 6, 7, 8 le choix de divers ordres sur les couleurs a été imposé au prééditeur, qui en déduit les longueurs des répartitions correspondantes.

2 - RESULTATS

Le prééditeur de liens donne, en numérotation hexadécimale, l'origine des segments de variables. L'implantation commence à l'adresse 0. La contraction a été calculée. Le temps d'unité centrale nécessaire au fonctionnement du prééditeur est exprimé en minutes et centièmes de minutes. Nous remarquerons que les temps obtenus sont indiqués comme nuls (en fait inférieurs au centième de minute), ce qui illustre la rapidité du prééditeur.

```
   M1     : 0000
   M2     : 000A
   M3     : 0014
   M4     : 001E
   M5     : 0028                    1
   C1     : 0032
   C2     : 0046
   C3     : 005A
   C4     : 006E
   C5     : 0082
CONTRACTION 1.00
JOB STEP 19 TERMINATED AFTER 0000.00MIN
```

```
   M1     : 001E
   M2     : 0000
   M3     : 0000
   M4     : 0014
   M5     : 0000                    2
   C1     : 0028
   C2     : 000A
   C3     : 000A
   C4     : 0000
   C5     : 000A
CONTRACTION 0.40
JOB STEP 20 TERMINATED AFTER 0000.00MIN
```

```
   M1     : 001E
   M2     : 0014
   M3     : 0000
   M4     : 0028                    3
   C1     : 0000
   C2     : 000A
   C3     : 0032
CONTRACTION 0.70
```

```
      M0    : 0014
      M1    : 000A
      M2    : 0000
      M3    : 000A                    4
      M4    : 0000
      M5    : 000A
      M6    : 0000
 CONTRACTION 0.42
 JOB STEP 19 TERMINATED AFTER 0000.00MIN

      M0    : 001E
      M1    : 000A
      M2    : 0014
      M3    : 0000                    5
      M4    : 000A
      M5    : 0000
 CONTRACTION 0.66
 JOB STEP 20 TERMINATED AFTER 0000.00MIN

      F     : 0000
      B     : 0000                    6
      A     : 0007
      E     : 0001
      D     : 0008
      C     : 0002
 CONTRACTION 0.61
 JOB STEP 21 TERMINATED AFTER 0000.00MIN

      A     : 0000
      E     : 0000
      F     : 0001
      B     : 0006                    7
      D     : 0007
      C     : 0008
 CONTRACTION 0.57
 JOB STEP 22 TERMINATED AFTER 0000.00MIN

      F     : 0000
      B     : 0000
      D     : 0001
      C     : 0007                    8
      A     : 0008
      E     : 0008
 CONTRACTION 0.66
 JOB STEP 23 TERMINATED AFTER 0000.00MIN
```


On constate que pour l'exemple 1 le prééditeur donne une implantation consécutive des segments. La contraction est égale à 1.

Pour l'exemple 2 on constate que les couples (M_i, C_i) sont en recouvrement avec les couples (M_j, C_j) (pour i et j $\geq$ 2) (adresses 0 à 1 D)

Pour l'exemple 3, on constate, comme prévu, le recouvrement entre les couples (M2,C1) et (M3,C2) qui s'étendent de l'adresse 0 à l'adresse 1 D.

Les résultats trouvés pour les exemples 4 à 8 confirment les résultats que nous avions obtenus dans la partie 5.

BIBLIOGRAPHIE

BERGE C. : Théorie des graphes et ses applications - (Dunod 1968)

Compagnie Internationale pour l'Informatique : documentation de base relative au moniteur Siris 7/8.

DENDIEN J. : Gestion statique de mémoire dans un sytème de programmation modulaire - (Session systèmes III, congrés AFCET 72 - Grenoble, novembre 1972 - brochure 1).

DERNIAME J.C. et PAIR C. : Problèmes de cheminement dans les graphes (Dunod 68)

DUFOUR J.F. : Traitement automatique de la Gestion scolaire d'une université (Thèse de 3ème cycle soutenue à l'Université de Nancy I, décembre 1971).

GILMORE HOFFMAN : A characterisation of comparability graphs and interval graphs (Canadian Journal of Mathematics, 16, 1964).

HERZ J.C. : Quelques considérations sur les problèmes d'emploi du temps (Revue française de Recherche Opérationnelle n° 38, 1966).

KAUFMANN A : Introduction à la combinatorique en vue des applications (Dunod 1968).

MAGHOUT K : Applications de l'algèbre de Boole à la théorie des graphes (Cahiers du Centre d'Etudes de R. O. Bruxelles 1963)

ROY B. : Algèbre moderne et théorie des graphes orientées vers les sciences économiques et sociales (Dunod 1969).

ROY B. : Prise en compte des contraintes disjonctives dans les méthodes de chemin critique.

TABLE DES MATIERES

			pages
INTRODUCTION			1
1ére Partie	:	RELATIONS ENTRE MODULES ET CLASSES	7
2éme Partie	:	CHOIX D'UNE METHODE DE GESTION DE MEMOIRE POUR LES VARIABLES	17
3ème Partie	:	PROBLEME DE RECOUVREMENT POUR LES TEXTES OBJETS	42
4ème Partie	:	MECANISMES DE LA COMPILATION ET DE L'EDITION DE LIENS	50
5ème Partie	:	SOLUTION DU PROBLEME DE L'IMPLANTATION DES VARIABLES	65
6ème Partie	:	SOLUTION DU PROBLEME DES TEXTES OBJETS	99
7ème Partie	:	ORGANISATION GENERALE DU PREEDITEUR DE LIENS	119
CONCLUSION			134
ANNEXE			A1

NOM DE L'ETUDIANT : DENDIEN Jacques /

Nature de la thèse : DOCTEUR INGENIEUR

Vu, Approuvé

et permis d'imprimer

NANCY, le 27 février 1973

Le Président du Conseil de l'Université de NANCY I

J.R. HELLUY

Le Président