

L'AUTEUR



Jean-Paul DELAHAYE
est professeur à l'Université
de Lille et chercheur
au Laboratoire d'informatique
fondamentale de Lille (LIFL).

qui peut être un ensemble de nombres, ou de lettres ou de codes numériques chargés de représenter informatiquement des objets réels. On se donne aussi un nombre entier n et une application F de S^n dans S^n , c'est-à-dire une transformation qui à tout n -uplet d'éléments de S en associe un autre. Dans l'exemple de la permutation circulaire, l'application F de S^3 dans S^3 est celle qui, à tout triplet de valeurs (A, B, C) , associe (B, C, A) .

Calcul *in situ* général

Quand on considère des variables booléennes, cela signifie que chaque élément de S est 0 ou 1. En plus du cas booléen, nous envisagerons deux autres cas finis. Si nous calculons avec des nombres entiers écrits en binaire avec k chiffres alors les éléments de S sont $0, 1, 2, \dots, 2^k - 1$; si nous calculons modulo p (nous remplaçons chaque nombre par son reste lors de la division par p), les éléments de S sont $0, 1, \dots, p - 1$.

Dans le cas général, un calcul *in situ* pour une application F est une suite d'affectations analogues à celles que nous avons utilisées

dans les exemples. Ces affectations sont envisagées une à une et chacune attribue une valeur nouvelle à l'une des variables dont l'ancienne valeur est donc perdue... c'est tout le problème ! La valeur nouvelle ne dépend que des valeurs disponibles à cet instant du calcul et nous nous imposons, dans le calcul *in situ*, de n'utiliser aucune variable supplémentaire.

Pour la permutation circulaire de n variables proposée dans l'encadré 1, nous considérons l'opération :

$$F: (A_1, A_2, \dots, A_n) \rightarrow (A_2, \dots, A_n, A_1).$$

La méthode proposée montre que $n + 1$ affectations suffisent pour réaliser cette permutation circulaire, sans que nous n'utilisions aucune autre variable que celles données, et sans que nous ayons à stocker dans les variables des nombres plus grands que $2^k - 1$ dans le cas des nombres en binaire, ou que $p - 1$ dans le cas du calcul modulo p . L'encadré 2 explique que lorsqu'on se donne des variables où l'on peut mémoriser des entiers sans limitation de taille (ce qui est irréaliste en informatique), alors toute affectation F de S^n dans S^n se calcule *in situ* en au plus $n + 1$ affectations.

Cette valeur de $n + 1$, valable pour les permutations circulaires de n variables et pour les transformations F quelconques quand nous utilisons des variables non bornées, n'est pas générale. Pour la transformation $F(A_1, A_2, A_3, A_4) = (A_2, A_1, A_4, A_3)$, on ne sait pas faire mieux que six affectations pour un calcul *in situ* avec des variables bornées.

Les programmes *in situ*, surtout s'ils sont courts, devraient aider à concevoir des puces électroniques moins gourmandes en mémoire, ce qui augmentera la rapidité du calcul. En effet, calculer «localement» est souvent plus efficace pour un processeur que de faire des requêtes aux mémoires externes qu'on doit associer à l'utilisation de variables auxiliaires.

Cette économie de mémoire entraîne un allongement de la taille des programmes. C'est là un nouvel exemple d'une règle générale rencontrée en maintes occasions en informatique : pour économiser de la mémoire, il faut parfois complexifier le calcul. Les algorithmes de compression

2. Calcul *in situ* dans le cas infini

Au départ, nous avons un n -uplet : $(A_1, A_2, \dots, A_n)$. Les variables $A_1, A_2, \dots, A_n$, sont choisies dans un ensemble S non borné, par exemple l'ensemble des entiers.

Une application F attribue de nouvelles valeurs aux variables $A_1, A_2, \dots, A_n$ qui deviennent $B_1, B_2, \dots, B_n$.

Utilisons des variables où nous pouvons mémoriser des entiers sans limitation de taille et voyons comment il a été démontré, dans ce cas particulier, que toute application F peut se calculer *in situ* en au plus $n + 1$ affectations. Pour que le calcul soit *in situ*, nous devons tenir compte du fait que lorsque nous mettons une valeur dans A_i , celle-ci écrasera l'ancienne valeur dès lors indisponible. Pas question donc d'écrire un programme du type :

$$A_1 := E_1(A_1, A_2, \dots, A_n), A_2 := E_2(A_1, A_2, \dots, A_n), \dots, A_n := E_n(A_1, A_2, \dots, A_n).$$

L'idée du calcul est de remplacer A_1 par $A' = 2^{A_1} 3^{A_2} \dots p_n^{A_n}$, où la suite p_i désigne la suite des nombres premiers $(2, 3, 5, 7, \dots)$. Cette étape réunit dans A_1 toute l'information contenue dans les variables $A_1, A_2, \dots, A_n$. Pour récupérer A_i à partir de A' , on calculera le nombre de fois que A' est divisible par p_i . Le fait de placer B_2 dans la variable A_2 ne fait pas perdre l'information A_2 , car celle-ci reste présente, mêlée aux autres valeurs dans A' . On peut alors calculer tous les nouveaux A_i en récupérant toutes les valeurs des anciens A_i dans A' , puis en utilisant l'application adéquate. À l'étape $n + 1$, on calcule A_1 en fonction de A' .

Comme nous avons regroupé et codé (avec la décomposition en facteurs premiers) toutes les valeurs de $A_1, A_2, \dots, A_n$ dans A' , nous n'avons pas besoin de mémoire supplémentaire et nous gardons à portée de main durant le calcul toutes les valeurs initiales des variables mises «au chaud». Bien sûr, cette astuce ne fonctionne pas en informatique où chaque variable ne peut prendre qu'un nombre fini de valeurs.