INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 725

LES FONCTIONS DE HACHAGE DIFFÉRENTIELLES : APPLICATION À LA GÉNÉRATION DE GRAPHES D'ÉTATS

BERNARD COUSIN

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 38 38 32

Les fonctions de hachage différentielles : application à la génération de graphes d'états

Bernard Cousin*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Publication interne n ° 725 — Mai 1993 — 18 pages

Résumé : Les fonctions de hachage différentielles possèdent un procédé de calcul différentiel qui permet d'optimiser le temps d'éxécution des fonctions de hachage vis-à-vis du procédé usuel. Nous définissons la propriété d'être différentielle pour une fonction de hachage, puis nous montrons que toutes les fonctions de hachage ne possédent pas cette propriété, enfin nous proposons une caractérisation de l'ensemble des fonctions qui la possède. Nous nous attachons ensuite à montrer le gain en performance induit par l'utilisation d'algorithmes différentiels de cinq fonctions de hachage trouvées dans la littérature. Les gains observés étant proportionnels à la taille de la clef, ils peuvent être extrêmement importants. Nous utilisons alors une fonction de hachage différentielle pour la génération de graphe d'états issus de modèles décrits au moyen de réseaux de Petri.

Mots-clé : fonctions de hachage, techniques de vérification, étude de performances, graphe d'accessibilité, réseau de Petri

(Abstract: pto)

à paraitre au CFIP'93 de Montréal du 7 au 9 septembre 93

*IRISA - Email address: bcousin@irisa.fr

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

INRIA
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Differential Hashing Functions : Application to Reachability Graph Generation

Abstract: Differential hashing functions have differential computation process which enables hashing function processing time to be optimized. Formal definition of the differential property for hashing functions is proposed, and we show that not all hashing functions have this property, then we propose a characterization of the differential hashing function set. Next, we show the performance acceleration produced by the differential algorithms applied to five hashing functions found in common applications. The observed accelerations can be significant because they are proportional to key length. Last we study the performances of differential hashing functions for the reachability graph exploration of distributed systems specified by a Petri net. This application demonstrates the advantages and the limits of our differential technique.
Key-words: hashing functions, validation technique, performance study, reachability graph, Petri net

1. Introduction

De nombreux outils de validation sont basés sur l'établissement de graphes d'états qui permettent de décrire tout ou partie de l'évolution de systèmes répartis modélisés formellement. De très nombreux graphes ont été développés : arbres de couverture [Karp 69], graphes réduits ou minimaux [Itoh 83, Zhao 86, Finkel 91], graphes symboliques [Haddad 86, Chiola 90], arbres de couverture de hauts niveaux ou colorés [Huber 85, Lindquist 90], et même des graphes temporisés ou stochastiques [Dutheillet 89], pour ne citer que ceux-là. L'explosion combinatoire du nombre d'états due au parallélisme des systèmes étudiés engendre des recherches constantes : notamment pour réduire au maximum le temps de construction de ces graphes et pour améliorer la pertinence et la densité de ces graphes [Dimitrijevic 89]. Dans ce domaine, on peut discerner trois grands axes de recherche : le premier tend à construire des graphes permettant de répondre à une certaine classe de propriétés [Marsan 88, Cacciari 92], le second se contente d'une exploration partielle du graphe en utilisant des techniques de simulation ou de parcours aléatoires ou dirigés [West 86, Groz 87], le troisième développe des techniques de compression des données et du temps de traitement [Holzmann 88, Algayres 91]. Bien évidemment ces trois axes de recherche peuvent se combiner utilement. Le travail que nous décrivons dans cet article appartient au troisième axe de recherche : il propose une amélioration des outils de génération de graphes. Cependant notre travail ne s'applique pas uniquement à ce type d'outil, son champ d'application ne demande qu'à être étendu, la génération de graphes d'états pour la validation des systèmes répartis a été prise comme exemple, et nous aurons l'occasion de montrer l'intérêt et les limitations de notre technique pour ce type d'outil lors du quatrième chapitre.

Les méthodes de hachage sont des méthodes efficaces de recherche d'un élément parmi une grande collection de données. En effet, ces méthodes ont la particularité de permettre les opérations de recherche, d'adjonction et de suppression en un nombre de comparaisons qui est en moyenne constant, c'est à dire qui ne dépend pas du nombre d'éléments dans la collection. Chaque élément de la collection est identifié de manière unique par sa clef. Les méthodes de hachage utilisent une *fonction de hachage* et une *fonction d'accès et de résolution des conflits*. Le domaine de définition des clefs étant très grand, la fonction de hachage est chargée d'associer une valeur à chaque clef, appelée valeur de hachage ou adresse. Le domaine de définition des valeurs de hachage est considérablement plus réduit que le domaine de définition des clefs. La valeur de hachage est alors utilisée comme une adresse pour accéder directement à l'emplacement où doit être stocké l'élément associé à la clef. Si cette zone de stockage est structurée comme un tableau, appelé table de hachage, l'adresse est alors interprétée comme un indice dans cette table. La fonction d'accès et de résolution est chargée de résoudre les collisions inhérentes à la réduction du domaine de définition : en effet plusieurs clefs peuvent partager la même valeur de hachage.

Dans cet article, nous présentons les *fonctions de hachage différentielles*. Ces fonctions de hachage possèdent un *procédé de calcul différentiel* de la clef de hachage qui peut remplacer le procédé habituel, et qui en optimise le temps d'exécution. L'optimisation obtenue est basée sur la constatation que toute clef peut être structurée comme une suite de composantes. Une composante peut être déterminée par des caractéristiques physiques (un certain nombre de bits, un mot, etc) ou sémantiques (un caractère alphanumérique, un

compteur, etc). Si on dispose déjà d'une clef et de sa valeur de hachage et que l'on veuille calculer la valeur de hachage d'une nouvelle clef qui ne diffère de la première que d'un nombre restreint de composantes, il peut être plus performant de déduire la nouvelle valeur de hachage de la première valeur que d'appliquer la fonction de hachage à toutes les composantes de la nouvelle clef.

Au procédé de calcul différentiel on associe une famille de *fonctions mono-différentielles de calcul*. Chaque fonction permet de calculer la valeur de hachage d'une clef connaissant la valeur de hachage d'une autre clef qui ne *diffère* de la première clef que de la valeur d'un seul composant donné. Par composition, on prouve aisément que les fonctions mono-différentielles permettent de calculer la valeur de hachage d'une clef à partir de la valeur de hachage d'une autre clef différant de la première de plus d'une composante.

Une première question est soulevée par notre proposition : Est-il possible d'associer un procédé de calcul différentiel à toute fonction de hachage ? Au cours du deuxième chapitre, après une définition formelle des fonctions de hachage différentielles, nous prouverons que la réponse à cette question est négative. Cependant nous présenterons une caractérisation de l'ensemble des fonctions de hachage auxquelles il est possible d'associer un procédé de calcul différentiel et nous verrons par la suite que cet ensemble recouvre la majorité des fonctions de hachage habituellement utilisées.

Le chapitre suivant (le troisième) sera l'occasion pour nous de répondre à une question supplémentaire : Le procédé de calcul différentiel est-il plus performant que le procédé usuel pour toutes les fonctions de hachages différentielles ? Nous verrons que ce n'est pas toujours le cas et nous préciserons dans quelles conditions. Toutefois cette réponse n'est pas définitive car l'algorithme utilisé lors de l'implantation, tant pour le procédé différentiel que pour le procédé usuel, a une influence prépondérante sur leurs performances. Nous verrons qu'un implantation naïve produit déjà une très bonne accélération de traitement, et qu'une implantation compacte autorisant un codage très efficace, réduit le temps de calcul considérablement.

Le quatrième chapitre sera pour nous l'occasion de mesurer les performances des fonctions de hachage différentielles pour la génération de graphes d'états utilisée par un outil de validation basé sur la description formelle de systèmes répartis au moyen de réseau de Petri. Cet exemple nous permettra de mettre en évidence les avantages et les limites de notre technique.

2. Les fonctions de hachage différentielles

2.1 Définitions

Une fonction de hachage peut être décrite comme une application h à N variables d'un produit d'ensemble $\prod_{k=1}^{N} A_k$ vers un ensemble B. C'est à dire :

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \exists\ h(<x_1,...,x_i,...,x_N>) \in B.$$

Nous appellerons $h(<x_1,\ldots,x_i,\ldots,x_N>)$ la valeur de hachage de la clef $<x_1,...,x_i,...,x_N>$.

Définition 1: fonction de calcul mono-différentiel

Une fonction de hachage h possède une fonction de calcul mono-différentiel de la valeur de hachage pour sa $i^{ème}$ composante si et seulement si il existe une fonction F_i de $B \times A_i \times A_i$ vers B tel que : $\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k$, $\forall y_i \in A_i$,

$$F_i(h(<x_1,...,y_i,...,x_N>), y_i, x_i) = h(<x_1,...,x_i,...,x_N>). \quad [1]$$

La fonction de calcul mono-différentiel (F_i) permet de calculer la valeur de hachage issue d'une clef ($<x_1,...,x_i,...,x_N>$: appelée clef_fils), à partir de la connaissance de la valeur de hachage issue d'une clef ($<x_1,...,y_i,...,x_N>$: appelée clef_père) ayant en commun avec la clef précédente toutes ses composantes sauf une (i : appelée composante à modifier). En ayant la connaissance de la valeur de la composante à modifier de la clef_père (y_i), de la valeur que doit avoir cette composante dans la clef_fils (x_i), et la valeur de hachage associée à la clef_père ($h(<x_1,...,y_i,...,x_N>)$), la fonction de calcul différentielle associée à la composante à modifier permet de calculer la valeur de hachage associée à la clef_fils ($h(<x_1,...,x_i,...,x_N>)$). On remarque que la relation père/fils est ici symétrique, puisque la même fonction F_i permet d'obtenir la valeur de hachage associée à la clef_père connaissant celle du fils.

Définition 2 : fonction de hachage différentielle

Si pour chaque composante associée à la fonction de hachage étudiée il existe une fonction de calcul mono-différentiel, alors l'ensemble des fonctions de calcul mono-différentiel constitue une famille complète de fonctions de calcul différentiel : $\forall i \in [1,N]$, $\exists$ F_i de $B \times A_i \times A_i$ vers B tel que $\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k$, $\forall y_i \in A_i$,

$$F_i(h(<x_1,...,y_i,...,x_N>), y_i, x_i) = h(<x_1,...,x_i,...,x_N>).$$

Si une fonction de hachage possède une famille complète de fonctions de calcul différentiel, on dira qu'elle est différentielle.

On remarque que chacune des fonctions de calcul mono-différenciel F_i n'a pas un ensemble de valeurs égal à l'ensemble de valeurs défini pour la fonction de hachage h. Toutefois l'ensemble de valeurs d'une fonction de hachage différentielle h est égal à l'union des ensembles de valeurs des fonctions de calcul différentiel F_i associées à chacune de ses composantes i. On rappelle que les fonctions de calcul différentiel sont des fonctions : il existe des triplets $<b, y_i, x_i> \in B \times A_i \times A_i$ pour lesquels il peut ne pas exister d'image par F_i, alors que l'on rappelle que la fonction de hachage différentielle associée est une application.

2.2 Caractérisation

On veut caractériser les fonctions de hachage qui sont différentielles, c'est à dire qui admettent une famille complète de fonctions de calcul différentiel de la valeur de hachage.

Toutes les fonctions de hachage n'admettent pas de fonctions de calcul différentiel. Par exemple la fonction de hachage "$\otimes$" construite sur un produit d'ensembles binaires et définie par l'opération binaire "et logique" n'est pas différentielle. En effet, la fonction de calcul différentiel associée au premier composant ne peut pas être déterminée : $F_1(0,0,1)$ peut être

égale soit à $F_1(\otimes<0,1>,0,1)=\otimes(<1,1>)=1$, soit à $F_1(\otimes<0,0>,0,1)=\otimes(<1,0>)=0$, ce qui est en contradiction avec la définition d'une fonction qui assure l'unicité de l'image.

Propriété :

Les fonctions de hachage, qui admettent une fonction de calcul différentiel pour leur $i^{ème}$ composant, sont caractérisées par la propriété suivante :

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k, \qquad [2]$$

$$h(<x_1,...,x_i,...,x_N>)=h(<y_1,...,x_i,...,y_N>) \Leftrightarrow$$
$$h(<x_1,...,y_i,...,x_N>)=h(<y_1,...,y_i,...,y_N>).$$

Nous allons prouver que la propriété [2] est une condition nécessaire et suffisante pour que la fonction de hachage admette une famille complète de fonctions de calcul mono-différentiel.

Prouvons d'abord que c'est une condition nécessaire. Soit h une fonction de hachage possédant une fonction de calcul différentiel pour sa $i^{ème}$ composante, alors d'après la définition [1] on a la propriété suivante : la fonction de calcul différentiel F_i de BxA_ixA_i vers B est telle que : $\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \forall y_i \in A_i,$

$$h(<x_1,...,x_i,...,x_N>)=F_i(h(<x_1,...,y_i,...,x_N>),\ y_i,\ x_i)$$

Donc $\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k$, d'après la définition [1] on peut écrire que : $h(<x_1,...,x_i,...,x_N>) = F_i(h(<x_1,...,y_i,...,x_N>),\ y_i,\ x_i)$.

Si on suppose les prémices de la condition [2] :

$$h(<x_1,...,y_i,...,x_N>) = h(<y_1,...,y_i,...,y_N>)$$

alors on en déduit que $h(<x_1,...,x_i,...,x_N>) = F_i(h(<y_1,...,y_i,...,y_N>),\ y_i,\ x_i)$.

On peut écrire aussi d'après la définition [1] que $\forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k,$

$$\forall x_i \in A_i,\ h(<y_1,...,x_i,...,y_N>) = F_i(h(<y_1,...,y_i,...,y_N>),\ y_i,\ x_i) .$$

On en déduit immédiatement la conclusion de la condition [2] :

$$h(<x_1,...,x_i,...,x_N>) = h(<y_1,...,x_i,...,y_N>).\ (cqfd)$$

Ensuite nous prouvons que c'est une condition suffisante. Soit une fonction de hachage h ayant la propriété suivante :

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} A_k,\ \forall <y_1,...,y_i,...,y_N> \in \prod_{k=1}^{N} A_k,$$
$$h(<x_1,...,x_i,...,x_N>)=h(<y_1,...,x_i,...,y_N>) \Leftrightarrow$$
$$h(<x_1,...,y_i,...,x_N>)=h(<y_1,...,y_i,...,y_N>).$$

Soit la famille de relations F_i de $BxAixAi$ vers B définie par :

$F_i(x_i, y_i, b) = \{h(<x_1,...,y_i,...,x_N>)$ tel que $h(<x_1,...,x_i,...,x_N>)=b\}$. Nous allons prouver que F_i est une fonction.

Supposons qu'il existe au moins deux éléments z et w en relation par F_i avec le triplet (x_i, y_i, b).
Par définition de la relation F_i, ces deux éléments peuvent s'écrire :

$z=h(<z_1,...,y_i,...,z_N>)$ et $w=h(<w_1,...,y_i,...,w_N>)$.

De même, par définition de F_i on sait que :

$h(<z_1,...,x_i,...,z_N>)=h(<w_1,...,x_i,...,w_N>)=b$.

Une alternative se présente :
. Soit les deux éléments z et w sont issus d'une même clef ($\forall i \in \{1,...,i-1,i+1,...N\}$ $z_i=w_i$) alors puisque la fonction de hachage h est une application, on en déduit que la relation F_i associe un élément unique $h(<z_1,...,y_i,...,z_N>)=h(<w_1,...,y_i,...,w_N>)$ au triplet (x_i, y_i, b).
. Soit ils sont issus de deux clefs différentes alors grâce à la propriété [2] qui est applicable puisque $h(<z_1,...,x_i,...,z_N>) = h(<w_1,...,x_i,...,w_N>) = b$, on prouve que $h(<z_1,...,y_i,...,z_N>) = h(<w_1,...,y_i,...,w_N>)$, et donc que z=w. Dans ces conditions, la relation F_i associe au plus un élément z=w au triplet (x_i, y_i, b).
La relation F_i est donc bien une fonction. (cqfd)

Cette caractérisation bien que fondamentale est assez difficile à prouver de manière générale. En effet, la description des algorithmes usuels des fonctions de hachage ne permet pas de déterminer aisément si ces fonctions possèdent la propriété [2]. Toutefois, la formulation d'un algorithme différentiel fait apparaître naturellement les fonctions de calcul différentiel nécessaires à la preuve de la propriété [2].

2.3 Exemple

Prenons le cas simple d'une fonction de hachage "h" basée sur le "ou-exclusif" de tous les composants formant une clef. On suppose que tous les composants sont de même taille et appartiennent au même ensemble B. La fonction de hachage h peut se décrire formellement :

$$\forall <x_1,...,x_i,...,x_N> \in \prod_{k=1}^{N} B,\ h(<x_1,...,x_i,...,x_N>) = x_1 \oplus \ldots \oplus x_i \oplus \ldots \oplus x_N$$

avec $\oplus$ l'opération usuelle "ou-exclusif" définie de BxB vers B.

La preuve que cette fonction de hachage h est bien différentielle peut être trouvée dans [Cousin 93]. La fonction mono-différentielle de calcul de la clef de hachage associée au $i^{ème}$ composant F_i, que nous cherchons, est définie par :

$$F_i(h(<x_1,...,y_i,...,x_N>), y_i, x_i) = h(<x_1,...,y_i,...,x_N>) \oplus y_i \oplus x_i .$$

3. Etude des performances

3.1 Présentation

Nous allons montrer dans ce chapitre l'augmentation des performances que l'on peut attendre de l'utilisation du procédé de hachage différentiel. Pour ce faire nous allons comparer les performances obtenues par des fonctions de hachage en utilisant leur algorithme usuel à celles obtenues par l'utilisation d'un algorithme différentiel.

Dans la littérature on peut trouver de nombreuses études des performances des méthodes de hachage [Froidevaux 90, Wirth 87, Knott 75, Knuth 73, Ulmann 72, Lum 71, Morris

68, etc]. Notre étude se focalise sur l'étude des performances des fonctions de calcul de la valeur de hachage (adresse) sans nous préoccuper des fonctions d'accès et de résolution des conflits. En effet notre technique s'adapte sans aucun problème à toutes les fonctions d'accès et de résolution des conflits, l'augmentation des performances constatées lors de l'utilisation d'un algorithme différentiel pour implanter les fonctions de calcul de la valeur de hachage sera donc intégralement conservée.

Nous avons choisi 5 fonctions de hachage parmi celles trouvées dans la littérature, cet échantillon ne couvre pas toutes les fonctions de hachage existantes, toutefois ces 5 fonctions couvrent un spectre de fonctions très fréquemment utilisées. Nous ne cachons que la fréquence d'utilisation n'est pas le seul critère qui soit intervenu dans le choix de ces cinq fonctions, la simplicité de conception des algorithmes usuels de ces fonctions de hachage, gage de bonnes performances, est aussi intervenue comme critère. En effet, un algorithme simple et de code court est souvent plus rapide qu'un algorithme complexe et long. Actuellement des études supplémentaires sont entreprises pour rechercher l'algorithme différentiel de nouvelles fonctions de hachage, et ainsi augmenter le spectre de notre étude.

Nos fonctions de hachage utilisent les opérations de base suivantes : multiplication/division, modulo, addition/soustraction, pliage ("folding"), extraction de bits particuliers [Knott 75]. D'autres opérations peuvent être proposées : élévation à la puissance/racine, changement de base, calcul polynomiaux, etc [Lum 71]. Ces autres opérations ont souvent l'inconvénient d'être difficile à calculer, c'est pourquoi nous avons choisi de ne pas les utiliser dans un premier temps. Pour répartir équitablement sur l'ensemble des valeurs de hachage possibles, les opérations de base peuvent faire intervenir des coefficients multiplicatifs qui sont généralement choisis parmi les nombres premiers. Les cinq fonctions de hachage proposées utilisent généralement une combinaison de plusieurs opérations de base.

La première fonction (n°1) utilise une technique de pliage basée sur l'utilisation de l'opération logique "ou exclusif". La deuxième fonction (n°2) étudiée effectue la somme de tous les composants. La troisième fonction (n°3) utilise une technique d'extraction de quelques groupes de bits parmi la suite des bits formant la clef. La quatrième fonction (n°4) est une somme pondérée des composants par des coefficients premiers. La dernière fonction (n°5) désignée dans la littérature sous le nom "hpjw" combine des opérations logiques de rotation et d'addition.

Nous allons voir que nous sommes parvenus à trouver les algorithmes différentiels pour chacune de ces cinq fonctions, toutefois si des accélérations remarquables sont obtenues pour quatre d'entre-elles, l'algorithme différentiel de la quatrième est aussi peu performant que l'algorithme usuel, qui est malheureusement aussi le moins rapide des cinq! Cette conclusion n'est pas définitive car des implantations judicieuses peuvent accélérer considérablement la vitesse de calcul des fonctions de hachage différentielles.

Ces fonctions ont été codées au moyen du langage C. Leur code est présenté en annexe : l'annexe A pour les algorithmes usuels, l'annexe B pour les algorithmes différentiels. Pour simplifier ces tests et sans perte de généralité, nous avons identifié les composants des clefs à des octets. Si les tests ont été effectuées sur ordinateur Sun Sparc ayant 16 Moctets de mémoire centrale, nous sommes persuadés que des résultats similaires auraient été obtenus sur d'autres machines et pour d'autres configurations. Nous avons rassemblé les mesures

effectuées lors de ces tests sous forme de figures : la première figure rassemble les résultats des cinq fonctions de hachage avec leur algorithme usuel [cf Figure 1], la seconde les résultats obtenus avec l'algorithme différentiel en supposant que le composant d'indice médian est à modifier [cf Figure 2]. Nous verrons par la suite que le temps d'exécution des algorithmes différentiels associés à certaines fonctions de hachage peut être dépendant de la position du composant à modifier : nous avons effectué un choix moyen avec la position médiane. L'unité de la durée d'exécution de ces algorithmes est le $1/60.10^{-6}$ seconde.

3.2 Discussion préalable

Avant de comparer les deux types d'algorithmes (usuels et différentiels), j'aimerais d'abord établir certain faits vis-à-vis des différentes méthodes employées par chacune des 5 fonctions de hachage. Tout d'abord on constate que toutes les fonctions ont une durée d'exécution croissante en fonction de la taille de la clef sauf la fonction numéro 3. En effet, cette fonction utilise une technique d'extraction de bits situés en position fixe : la durée du calcul de la clef de hachage est alors constante. On constate aussi que cette fonction a la plus courte durée d'exécution. Nous tenons à souligner (et des tests le prouvent [Deudon 92]) que la répartition des adresses obtenues après hachage de la clef est, sous hypothèses générales, très inégale. En effet, il est recommandé de faire intervenir dans le hachage tous les composants constituant une clef, s'ils sont significatifs [Froidevaux 90] : recommandation que ne suit évidemment pas la technique d'extraction.

Si les autres fonctions ont un comportement similaire (croissance de la durée d'exécution en fonction de la taille de la clef) le rapport de leurs durées d'exécution pour une même longueur de clef varie de 1/2 pour les petites clefs à plus de 1/20 pour les plus grandes clefs. Ainsi plus une fonction de hachage est rapide plus elle a tendance à être rapide pour les grandes clefs. Cela justifie à posteriori notre choix de 5 fonctions de hachage ayant un codage simple et par conséquent rapide.

Nous voulons rappeler que les temps relevés ici ne sont pas représentatifs des performances globales des méthodes de hachage, en effet les méthodes de hachage sont formées d'un couple "fonction de hachage/fonction d'accès et de résolutions des conflits", et nous soulignons qu'une fonction de hachage inadaptées peut générer un très grand nombre de collisions dégradant de manière considérable les performances globales de la méthode. Toutefois, il ne faudrait pas en conclure que les résultats obtenus sur les temps d'exécution des fonctions de hachage sont incorrects. En effet, le temps d'exécution des fonctions de résolutions des collisions étant, en général, indépendant de la taille des clefs, il tend à devenir négligeable pour des grandes clefs vis-à-vis du temps d'exécution des fonctions de hachage [Deudon 92].

En fait, nous avons testé de nombreuses variantes d'algorithmes pour plusieurs fonctions de hachage, notamment les fonctions n°1 et n°3 : manipulation par mots (groupe de 4 octets) et non plus par composants (1 octet), modification du traitement des débordements, modification des opérations arithmétiques en opérations logiques, etc. Les comportements décrits précédemment sont conservés même si des optimisations locales ont été observées.

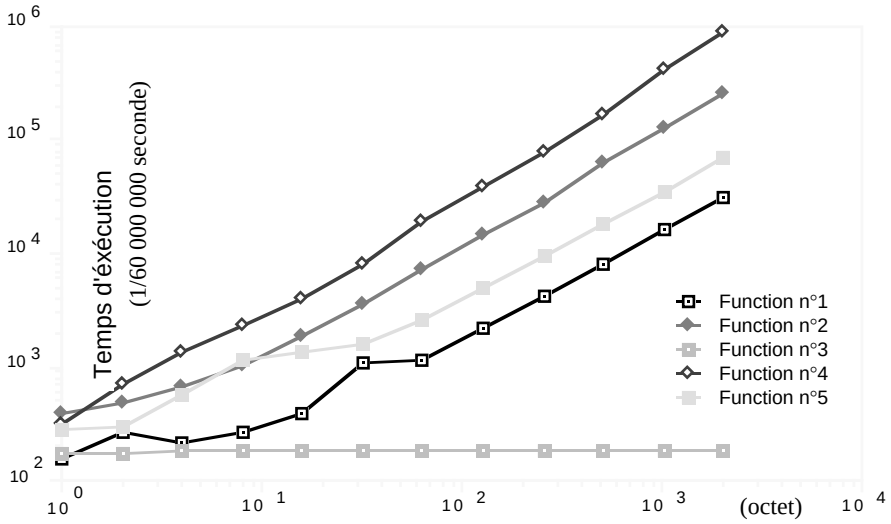

Figure 1- Procédé usuel

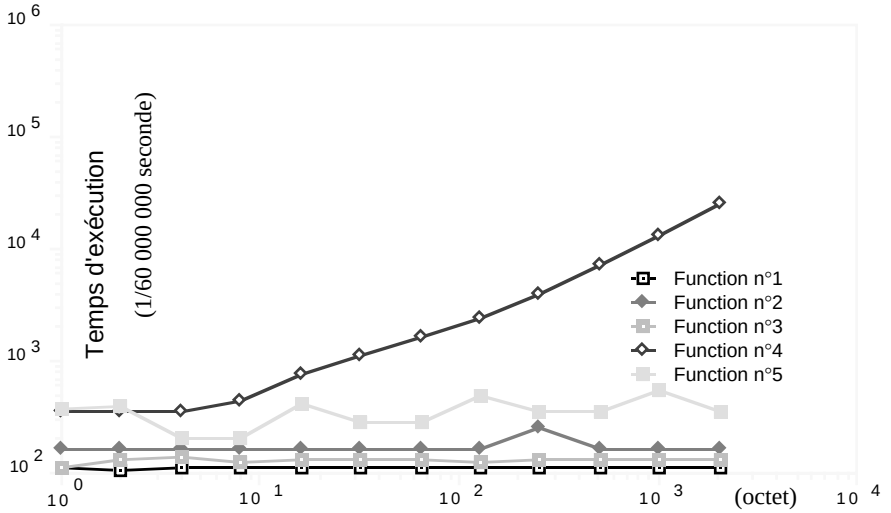

Figure 2- Procédé différentiel

3.3 Les résultats

Si maintenant nous comparons les résultats obtenus par les algorithmes différentiels avec ceux obtenus par les algorithmes usuels, on constate qu'en général le temps d'exécution des algorithmes différentiels est, d'une part plus court que le temps d'exécution de l'algorithme usuel, d'autre part constant vis-à-vis de la taille de la clef. Toutefois ces deux

affirmations sont mises en défaut pour l'algorithme différentiel de la fonction de hachage n°4. En effet l'algorithme différentiel mis en oeuvre pour cette fonction nécessite une élévation à la puissance du produit du composant et du coefficient associé tel que l'exposant est égal à la place du composant à modifier. Or nous savons que l'élévation à la puissance est une opération couteuse, et que son coût est fonction de l'exposant donc de la place du composant dans la clef.

Cette influence de la place du composant à modifier peut être étudiée sur toutes les fonctions à travers leur algorithme différentiel. On y constate que cette influence est faible pour l'ensemble des fonctions, hormis la fonction n°4 dont on vient de parler au paragraphe précédent. Pour ces premières, bien que les durées absolues soient faibles, on peut observer des variations qui semblent erratiques (notamment pour la fonction n°5). En fait elles sont engendrées par la coïncidence ou non de l'indice du composant modifier avec certaine constante utilisée par la fonction de calcul différentiel.

Nous rappelons que l'ensemble des résultats précédents sont obtenus par un algorithme basé sur un procédé mono-différentiel. Dans le cas de différences multiples (cas où la clef_fils diffère de la clef_père de plus d'un composant), on peut déduire la durée de calcul d'une valeur de hachage en multipliant la valeur trouvée lors des tests précédents par le nombre de différence. La durée est alors proportionnelle au nombre de composants différents entre la clef_fils et la clef_père. Le ratio entre la durée obtenue pour les algorithmes usuels et celle obtenue pour les algorithmes différentiels permet de décider du nombre de composants différents qu'il faut atteindre pour que l'algorithme différentiel soit moins performant que l'algorithme usuel. Les valeurs relevées lors de nos tests montrent que pour des clefs d'une centaine d'octets l'algorithme différentiel est plus rapide que l'algorithme usuel dès que le nombre de composant à modifier est inférieur : pour la fonction n°1 à 15, pour la fonction n°2 à 66, pour la fonction n°3 on autorise au plus une composante, pour la fonction n°4 à 12, et pour la fonction 5 à 8. Pour l'application proposée dans le chapitre suivant comme mise en oeuvre de cette technique de hachage différentiel ce taux de différence est respecté (de 5 à 10% avec des modèles de taille moyenne). Ceci expliquera les bons résultats obtenus.

Remarquons que certaines fonctions (notamment la fonction n°1) propose un procédé de calcul différentiel multiple dont la durée d'exécution de l'algorithme est proche de celle de l'algorithme associé au procédé mono-différentiel. C'est cette fonction de hachage munie de ce procédé de calcul de la valeur de hachage que nous utiliserons par la suite.

4. Application à la génération de graphes d'états

Le codage des états nécessite une taille de stockage importante. En effet, cette taille est fonction du parallélisme et de la précision de la modélisation du système étudié. Dans la littérature on peut trouver les chiffres suivants : 1916 octets pour la modélisation du "P-channel protocol" [Doldi 92], typiquement une centaine d'octets pour Holzmann [Holzmann 91], notre propre étude tend à montrer que pour des modèles d'applications ou de protocoles réels il est fréquent d'obtenir des états de taille conséquente : plusieurs centaines d'octets.

Certains logiciels de génération de graphes d'états préfèrent utiliser une technique de recherche où la structure de stockage des données est un arbre binaire (et équilibré) ou encore une structure hiérarchique spécifique [Chiola 87, Crubillé 88]. Ces techniques ne sont pas

les plus fréquemment utilisées car elles présentent, soit une trop grande spécificité (c'est à dire qu'elles ne sont adaptées qu'à une certaines classe de propriétés), soit un surcôut tant en espace (pointeur d'indirection) qu'en temps de d'accès (parcours des indirections), vis-à-vis de la technique de recherche par hachage.

Nous avons effectué l'étude des performances à l'aide de l'outil Bouster [Campergue 92] sur un jeu de plusieurs modèles décrit au moyen de réseaux de Pétri ayant un nombre de places variant de 2 à 50000, un nombre de transitions de 1 à 50000, un nombre d'arcs de 1000 à 100000 et générant plusieurs dizaines de milliers d'états. Cette étude montre que les fonctions de calcul de la clef de hachage ("f_hashing()"), de recherche et d'insertion d'un nouveau noeud ("put_in_table()") et de comparaison de chaîne de caractères ("bcmp()") sont les trois fonctions les plus intensément utilisées : 15 à 30 % du temps de calcul du processeur en mode utilisateur pour chacune d'entre-elles en fonction des différents modèles et dans un ordre variable. L'ordre et la valeur du taux d'utilisation de ces trois fonctions sont variables car leur performance dépend du nombre de collisions, qui lui même dépend à la fois de la fonction de hachage choisie et de la taille de la table de hachage. Toutes les autres fonctions sans exception utilisent chacune moins de 10% du temps du processeur et pour la grande majorité d'entre-elles moins d'un millième.

Ces chiffres montrent à la fois l'importance et les limites des gains que l'on peut espérer avec notre méthode. En effet, le processeur passe en moyenne près de 40% de son temps dans le code des fonctions chargées d'effectuer le calcul de la clef de hachage et d'appliquer l'algorithme de recherche et d'insertion d'un nouveau noeud. Une fonction de hachage rapide, judicieuse et équilibrée devrait être capable de réduire de manière conséquente ce temps en minimisant d'une part la fréquence des collisions et d'autre part le temps de calcul de la clef. Toutefois l'accélération des performances induites par ce type de technique ne peut pas réduire de manière magique la complexité inhérente au modèle étudié, notamment le nombre gigantesque de noeuds qu'il est parfois nécessaire de générer. Les deux autres axes de recherche évoqués lors de l'introduction (densification des informations portées par le graphe et/ou développement partiel de ce graphe) sont alors utilement combinés à notre technique, et cette possibilité fera l'objet de notre prochaine étude.

Ces conclusions ne doivent pas nous cacher un phénomène important déjà relevé par de nombreux chercheurs avant nous [Jard 91]: l'influence du mécanisme de mémoire virtuelle sur le temps total d'exécution. En effet, un ralentissement notable est constaté dès que les données de l'application ne peuvent plus être conservées en totalité en mémoire centrale. Toutefois la technique d'accès direct, qu'offre la méthode par hachage lorsqu'un faible taux de collision est conservé, favorise cette méthode vis-à-vis des autres méthodes proposées dans la littérature car elle minimise le nombre d'entrées/sorties entre la mémoire secondaire et la mémoire centrale.

La fonction "bcmp()" de comparaison de deux chaînes de caractères provient aussi de la bibliothèque standard du système Unix. Elle appartient à l'ensemble des trois fonctions les plus intensément utilisées. Nous nous proposons pour poursuivre l'optimisation, premièrement d'étudier l'implantation de nouveaux algorithmes de comparaison [Knuth 77, Boyer 77], deuxièmement d'utiliser la technique de compression du graphe proposée par Holzmann [Holzmann 88] qui permet de se passer de l'utilisation d'une telle fonction.

5. Conclusion

On constate aux vues des résultats de nos tests de performance que la technique différentielle de hachage est d'autant plus performante que la taille de la clef est importante. En effet, en général, les fonctions de hachage utilisent la totalité de la clef (ce procédé est recommandé pour avoir une répartition la plus équilibrée possible de la valeur de hachage et donc moins de collisions) : la complexité de l'algorithme usuel est donc proportionnelle à la taille de la clef. Les hypothèses de travail pour notre application_test (très large graphe, très grand nombre d'états) ont pour corollaire une clef de taille appréciable et cette affirmation est corroborée par les exemples trouvés dans la littérature. Les algorithmes différentiels sont eux d'une complexité proportionnelle aux nombres de composants à modifier entre la clef précédente (clef_père) et la clef à calculer (clef_fils). L'application_test choisie génère un graphe qui possède naturellement des états successifs ayant des clefs qui présentent, dans leur immense généralité, très peu de composants différents (moins de 10%). Ce fait est en parfaite adéquation avec le comportement des systèmes modélisés : dans un système réparti tous les sous-systèmes dont il est constitué ne changent pas simultanément et à tout moment.

Les fonctions de hachage différentielles ne sont pas une réponse universelle au problème de temps de calcul des fonctions de hachage : elles n'existent pas toujours et quand elles existent elles ne sont pas toujours plus performantes. Toutefois, notre étude nous amène à affirmer que, premièrement l'ensemble des fonctions de hachage que nous avons jusqu'à présent rencontrées peuvent être associées à une famille de fonctions de calcul différentiel, deuxièmement dans l'immense majorité des cas le temps de calcul induit par le procédé de calcul différentiel est notablement plus faible que celui du procédé de calcul usuel, troisièmement cette technique de calcul différentiel ne s'applique raisonnablement que si l'obtention des composantes modifiées entre une clef et la suivante est d'un coût faible voire nul. L'application proposée pour mettre en oeuvre les fonctions de hachage différentielles possède toutes les caractéristiques qui font que l'on obtient un gain en performance conséquent.

Remerciements

Je remercie Christian Houillon et Guillaume Deudon pour leur réalisation et leur étude des performances des fonctions de hachage différentielles sous une forme préliminaire [Deudon 92]. Mes remerciements aussi à Christian Ronse pour sa participation à la caractérisation des fonctions différentielles.

Bibliographie

[Aho 75] A.V.Aho, M.J.Corasick, "Efficient string matching : an aid to bibliographic search". Communications of the ACM vol 18 n°6, June 1975, p333-343.

[Algayres 91] B.Algayres,L.Doldi, H.Garavel, Y.Lejeune, C.Rodriguez, "Vesar : a pragmatic approach to formal specification and verification", Computer Network and ISDN Systems : special issue on tools for FDT's, vol 25 n°7, February 1991.

[Boyer 77] R.S.Boyer, J.S.Moore, "A fast string searching algorithm". Communications of the ACM vol20 n°10, 1977, p762-772.

[Campergue 92] C.Campergue, C.Nouaille, "Bouster : génération parallèle du GMA associé à un réseau de Petri", rapport de recherche, Bordeaux-France, Juin 1992.

[Cacciari 92] L.Cacciari, O.Rafiq, "On improving reduced reachability analysis", proceedings of 5th international conference on formal description techniques (FORTE'92), Lannion-France, 13-16 octobre 1992.
[Chiola 90] G.Chiola, "Symbolic reachability graph", 11h international workshop on theory and applications of Petri nets, Paris-France, 1990.
[Cousin 93] B.Cousin, "Differential hashing functions", 5th international conference on computing and information (ICCI'93), Sudbury - Canada, 27-29 May 1993.
[Crubillé 88] P.Crubillé, "Mec : un outil de clacul sur les systèmes de transitions", Colloque francophone sur l'ingéniérie des protocoles (CFIP'88), Bordeaux-France, Septembre 1988, Eyrolles, 1988.
[Deudon 92] G.Deudon, C.Houillon, "Techniques de hachage", rapport de recherche ENSERB, Bordeaux-France, Juin 1992.
[Dimitrijevic 89] D.D.Dimitrijevic, M.S. Chen, "Dynamic state explosion in quantitative protocol analysis". Proceedings of PSTV-IX, Twente-Netherland, 6-9 June 1989.
[Doldi 92] L.Doldi, P.Gauthier, "Veda-2 : power to the protocol designers", proceedings of FORTE'92, Lannion-France, 13-16 octobre 1992.
[Dutheillet 89] C.Dutheillet, S.Haddad, "Regular stochastic Petri nets", Workshop on Petri nets and its application, Bonn - Germany, 1989.
[Finkel 91] A.Finkel, C.Johnen, "Construction efficace du graphe de couverture minimal : application à l'analyse de protocole", Colloque francophone sur l'ingéniérie des protocoles (CFIP'91), Pau-France, Septembre 1991, Hermès, 1991.
[Froidevaux 90] C.Froisdevaux, M-C.Gaudel, M.Soria, "Types de données et algorithmes", McGraw-Hill, 1990.
[Groz 87] R.Groz,"Unrestricted verification of protocol properties on simulation using an observer approach". Proceedings of the PSTV-VII, Montréal - Canada, June 1986. North-holland, p255-268.
[Haddad 86] S.Haddad, J.M.Bernard, "Les réseaux réguliers, spécification et validation par le logiciel ARP", 3ème colloque de génie logiciel Afcet, Versailles- France, 1986.
[Huber 85] P.Huber, A.M.Jensen, L.O.Jepsen, K.Jensen, "Towards reachability trees for high-level Petri nets", Proceedings of advances in Petri nets (1984), Lecture notes in computer sciences n°188, p215-233, Springer verlag, 1985.
[Holzmann 88] G.J.Holzmann, "An improved protocol reachability analysis technique", Sofware, practice and experience, vol 18 n°2, February 1988, p137-161.
[Holzmann 91] G.J.Holzmann, "Design and validation of computer protocols", Prentice-Hall, 1991.
[Itoh 83] M.Itoh, H.Ichikawa, "Protocol verification algorithm using reduced reachability analysis", Transactions of the Institute of Electronic and Communication Engineers of Japan, vol E66 n°2, February 1983.
[Jard 91] C.Jard, T.Jéron, "Bounded-memory algorithms for verification on-the-fly", Workshop on computer-aided verification (CAV'91), Danemark, July 1991.
[Karp 69] R.M.Karp, R.E.Miller "Parallel program schemata", Journal of computer and system sciences, vol 3 n°4, p167-195, May 1969.
[Knuth 73] D.E.Knuth, "The art of computer programming : sorting and searching", vol 3, Addison-Wesley, 1973.
[Knuth 77] D.E.Knuth, J.H.Morris, V.R.Pratt,"Fast pattern matching in strings", SIAM Journal on Computing vol6 n°2, June 1977, p323-349.

[Knott 75] G.D.Knott, "Hashing functions", The Computer Journal, vol 18, n°3, August 1975, p265-278.

[Lindquist 90] M.Lindquist, "Parametrized reachability trees for predicate/transition nets". Proceedings of the 11th international conference on application and theory of Petri nets, Paris-France, 1990, Springer-Verlag, p22-41.

[Lum 71] V.Y.Lum, P.S.T.Yuen, M.Dodd, "Key-to-adress transform techniques : a fundamental performance study on large existing formatted files", Communications of the ACM vol14 n°4, April 1971.

[Marsan 88] M.A.Marsan, G.Chiola, "Improving efficiency of analysis of DSPN models", Lecture notes in computer sciences n°424, Advances in Petri nets, Springer-Verlag, 1989.

[Morris 68] R.Morris, "Scatter storage techniques", Communication of the ACM Vol 11 n°1, January 1968, p38-44.

[Peterson 57] W.W.Peterson, "Adressing for random access storage", IBM Journal on Research and Development vol 1 n°2, April 1957, p130-146.

[Ulmann 72] J.D.Ulmann, "A note on the efficiency of hashing functions", Journal of the ACM vol19 n°3, July 1972, p569-575.

[West 86] C.H.West, "Protocol validation by random state exploration", 6th international workshop on protocol specification, testing and verification (PSTV-VI), Montréal - Canada, June 1986.

[Wirth 87] N.Wirth, "Algorithmes et structures de données", Eyrolles, 1987. Traduction de : N.Wirth, "Algorithms and data structures", Prentice-Hall, 1986.

[Zhao 86] J.Zhao, G.Bochmann, "Reduced reachability analysis of communication protocols : a new approach", 6th international Workshop on Protocol Specification, Testing and Verification (PSTV-VI), Montréal - Canada, June 1986.

Annexe A

```
/*** Fichier fct_hash.c ***//*** version 1.2 - du 11/12/93 ***/
extern int n;    /* Taille du tableau de hachage */
extern char* key;       /* La clef */
extern int l;           /* Longueur de la clef (en octet)*/

int function_1() {
/* "ou_exclusif entre composants de 2 octets          */
register int i; register unsigned result = 0;
int x=0;
        for (i=1; i<l; i+=2){
                x = key[i-1]<<8;
                x ^= key[i];
                result ^= x;
        }
        if (l % 2) result ^= key[l-1];
        return (result % n);
} /* function_1() */

int function_2() {
/* somme ponderee de tous les composants  */
 register int i; register unsigned result = 0;
        for (i=1; i<=l; i++) result += i*key[i-1];
        return(result % n);
} /* function_2() */

int function_3() {
/* extraction des premier troisieme et dernier octets */
 register unsigned result = 0;
        result = key[0] ^ key[1]<<3 ^ key[l-1]<<7;
        return(result % n);
} /* function_3() */

int function_4() {
/* technique multiplicative et somme pondérée       */
/*      par des nombres premiers                    */
 register int i; register unsigned result = 0,g;
static  int  coef[19]={67,61,59,53,47,43,41,37,31,29,23,  17,13,11,9,7,5,3,1};
        for (i=0; i<l; i++){
                result = result * 5 + key[i] * coef[i % 19];
                if ((g = result & 0xF0000000) != 0) {
                        result ^= g ; result += (g>>28) ;
                }
        }
        return (result % n);
}/* function_4() */
```

```
int function_5() {
/*      fonction hpjw                   */
register int i ; register unsigned result=0,g ;
        for(i = 0 ; i< l ; i++) {
                result = (result << 4) ^ key[i] ;
                if ((g = result & 0xF0000000) != 0) {
                        result = result ^ (g>>28) ;
                        result = result ^ g ;
                }
        }
        return (result % n) ;
}/* function_5() */
```

Annexe B

```
/*** Fichier fct_hash_diff.c ***//*** version 1.3 du 15/12/92 ***/
/* calcul du nouveau resultat de la fonct. de hachage connaissant la valeur  */
/* de l'ancien et du nouveau composant et son indice = [0,n[                 */

int function_1_diff(int result_prec, int indice, char ancien_octet, char nouvel_octet){
/* ou_exclusif entre composants de 2 octets          */
register unsigned result;
        if (indice&0x00000001) {     /* impair? */
                result = result_prec ^ ancien_octet ^ nouvel_octet;
        } else { result = result_prec ^ (ancien_octet ^ nouvel_octet)<<8;
        } return(result % n);
} /* function_1_diff() */

int function_2_diff(int result_prec, int indice, char ancien_octet, char nouvel_octet) {
/* somme ponderee de tous les composants  */
register unsigned result = 0;
        result = result_prec + (nouvel_octet - ancien_octet) * (indice+1);
        return(result % n);
} /* function_2_diff() */

int function_3_diff(int result_prec, int indice, char ancien_octet, char nouvel_octet) {
/* extraction des premier troisieme et dernier octets */
register unsigned result = 0;
        switch(indice) {
        case 0 : result = result_prec ^ (nouvel_octet ^ ancien_octet);
                break;
        case 1 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<3;
                break;
        default : if (l-1==indice)
                        result = result_prec ^ (nouvel_octet ^ ancien_octet)<<7;
                else result=result_prec;
                break;
        }  return(result % n);
} /* function_3_diff() */
```

```
int function_4_diff(int result_prec, int indice, char ancien_octet, char nouvel_octet){
/* technique multiplicative et somme pondérée       */
/*      par des nombres premiers                                    */
register int result = 0; register int g;
static int coef[19]={67,61,59,53,47,43,41,37,31,29,  23,17,13,11,9,7,5,3,1};
register int x;  int i;
        x=1;
        for (i=1; i<=l-1-indice; i++) {
                x *= 5;
                f ((g = x & 0xF0000000) != 0) {
                        x ^= g ; x += (g>>28) ;
                }
        }
        x *= coef[indice%19];
        if ((g = x & 0xF0000000) != 0) {
                x ^= g ; x += (g>>28) ;
        }
        x *= nouvel_octet - ancien_octet;
        result = result_prec + x;
        if ((g = result & 0xF0000000) != 0) {
                result ^= g ; result += (g>>28) ;
        }
        return(result % n);
} /* function_4_diff() */

int function_5_diff(int result_prec, int indice, char ancien_octet, char nouvel_octet) {
/*      fonction hpjw                   */
register unsigned int result; register int x_haut, x_bas;
        switch((l-1-indice) % 7) {
        case 0 : result = result_prec ^ (nouvel_octet ^ ancien_octet); break;
        case 1 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<4; break;
        case 2 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<8; break;
        case 3 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<12; break;
        case 4 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<16; break;
        case 5 : result = result_prec ^ (nouvel_octet ^ ancien_octet)<<20; break;
        case 6 : x_bas = ((nouvel_octet ^ ancien_octet) & 0x0f)<<24;
                 x_haut = ((nouvel_octet ^ ancien_octet) & 0xf0)>>4;
                 result = result_prec ^ (x_haut ^ x_bas);
                break;
        }
        return(result % n);
}/* function_5_diff() */
```