Rétines artificielles et Opérateurs de Rétine

-Nouveaux procédés optroniques-

Etude menée par la

Société de Calcul Mathématique, S.A.

en application du Marché C.T.A. no 97.04.095

Département "Géographie-Imagerie-Perception"

Centre Technique d'Arcueil

Direction des Centres d'Expertise et d'Essais

Délégation Générale pour l'Armement

Contrat financé par le Fonds d'Innovation Défense de la D.G.A.

28 janvier 1999

Résumé Opérationnel et Conclusions

I. Introduction

L'idée des rétines artificielles est née ces dernières années, à cause du besoin d'amélioration des systèmes classiques de traitement de l'image, qui sont relativement lents lorsqu'il s'agit d'*extraire des informations* à partir des images recueillies.

Dans les systèmes classiques de vision, l'image capturée par un composant spécifique (imageur) est convertie sous forme numérique avant d'être transportée et traitée dans un microprocesseur. Al'inverse, les rétines artificielles réunissent capture et traitement (ou au moins une partie du traitement) de l'image dans un même système. Pour cela, elles sont donc des imageurs comportant une unité de traitement au sein de chaque point élémentaire de l'image (pixel). Ces unités de traitement, qu'elles soient analogiques ou numériques, peuvent communiquer entre elles (de proche en proche seulement), mais elles reçoivent simultanément les mêmes commandes. La rétine peut donc être assimilée à un calculateur travaillant en mode SIMD (Single Instruction Multiple Data). Le fait que chaque processeur puisse communiquer avec ses voisins est une caractéristique très importante des rétines : si un processeur est ébloui, il peut ainsi dire à ses voisins d'abaisser leur seuil de sensibilité.

S'appuyant sur la technologie CMOS, les rétines artificielles numériques programmables (dont celle du CTA) ont maintenant une existence réelle.

Avec leur développement sont apparus des problèmes de nature algorithmique. Pour des raisons d'encombrement et de coût, le nombre de places mémoire associé à chaque unité de traitement est limité : ceci oblige à repenser le déroulement de la plupart des algorithmes de morpho-mathématique utilisés habituellement pour le traitement de l'image.

II. Objet de l'étude et définitions

Dans la présente étude, nous avons traité le problème suivant : étant donnée une place mémoire limitée (de 4 à 7 ou 8 plans mémoire), quelle est la meilleure façon d'implanter les algorithmes de calcul de fonctions booléennes classiques (fonctions seuil, fonctions symétriques) ? Nous avons testé les méthodes existantes qui, dans la plupart des cas, se sont révélées insuffisantes, et nous en avons conçu de nouvelles, beaucoup plus efficaces. On dispose ainsi, maintenant, d'informations quantitatives précises : telle méthode, si l'on dispose de tant de plans mémoire, requiert tant d'opérations élémentaires pour le calcul de telle fonction.

On définit, sur une rétine numérique, une image par une matrice de 0 et de 1. A chaque unité de traitement, appelée aussi *processeur élémentaire* (PE), est associé un nombre réduit (inférieur à 10) de places mémoire (il s'agit de plans mémoire). Les opérations élémentaires (o.e.) permettent de modifier les valeurs contenues soit dans deux plans successifs d'un même PE, soit sur un seul plan. Mais elles opèrent sur l'ensemble de l'image de façon identique et simultanée.

Le traitement de l'image rétinienne est alors la détermination à partir d'une ou plusieurs valeurs de l'image de la nouvelle valeur de chaque pixel. Mathématiquement, ce traitement peut se représenter par une fonction booléenne de n variables. Plus généralement, on définit une fonction booléenne associée à un traitement par :

- un curseur, figure géométrique de n pixels représentant les n variables ;

- la désignation d'un pixel du curseur, appelé la mire, où est appliqué le résultat de la fonction booléenne ;

- une table de vérité, qui est une règle définissant à chaque "coloriage" du curseur la valeur (0 ou 1) de la fonction.

Par exemple, pour décider s'il est ébloui, un processeur fera ceci : il interroge ses voisins (neuf en tout, y compris lui-même). Si parmi ces neuf il y en a six qui sont à 1, il décide "je suis ébloui", et communique cette information. Il faut donc être capable de calculer, et de calculer rapidement, la fonction (notée $T_{9,6}$), qui vaut 1 si parmi 9 pixels 6 au moins sont à 1 (naturellement, ce seuil de 6 est arbitraire : on peut mettre 5, ou 7, etc).

Le traitement de l'image revient à appliquer en chaque pixel de l'image la fonction booléenne. Définir ce traitement pour une rétine programmable nécessite de décomposer les fonctions booléennes en une suite d'opérations élémentaires applicables de façon simultanée.

III. Comparaison des différentes méthodes

La décomposition en opérations élémentaires peut se faire de multiples manières, en fonction de la place mémoire dont on dispose.

On peut distinguer deux catégories de méthodes :

- celles utilisant seulement la place mémoire associée à un PE ;

- celles regroupant la place mémoire de plusieurs PE et donc calculant l'image en plusieurs passages ("clustérisation").

Théoriquement, la clustérisation peut associer autant de PE qu'il est nécessaire ; en réalité l'association peut aller jusqu'à 4 PE sur la rétine du CTA.

Dans la première catégorie, la Forme Normale Disjonctive (FND) est la méthode la plus connue : on liste tous les cas possibles. Elle ne nécessite qu'un minimum de mémoire (2 plans en plus de l'image), mais son coût opératoire est exponentiel en n, nombre de variables, et cela devient vite rédhibitoire (plus de 100.000 o.e. pour un curseur de 16 variables).

Pour obtenir des coûts opératoires bien inférieurs, il est possible, pour les fonctions seuil seulement, de calculer et de comparer la somme des variables du curseur avec la valeur du seuil (méthode CC). Le coût opératoire est alors faible (de l'ordre de 100 o.e.), mais le nombre de plans mémoire requis augmente comme le logarithme du nombre de variables (6 pour $T_{16,8}$).

Nous avons développé une nouvelle méthode : la méthode d'Addition des Variables par Blocs (AVB) qui restreint le coût opératoire en n'utilisant que peu de mémoire : pour $T_{16,8}$, 2025 o.e. suffisent, lorsqu'on dispose de trois plans mémoire (elle est donc 500 fois plus efficace que la F.N.D.). Aucune autre méthode n'était connue pour tirer parti de trois plans mémoire.

Cependant, ce coût est encore trop grand lorsque les curseurs sont plus importants : 38.000 o.e. pour 25 variables.

Nous avons alors développé d'autres méthodes faisant toutes partie de la deuxième catégorie : mise en commun de mémoire à titre temporaire. Elles utilisent toutes le calcul de la somme des variables. Différentes par la façon d'exécuter ce calcul (notation classique et redondante, calcul de la partie variable de la somme,...), ces méthodes nécessitent le même nombre de places mémoire (3) quelles que soient les fonctions à décomposer. Le coût opératoire pour les plus performantes (CAPVr, CAPVc, CSr, CSc) est pratiquement aussi bon que pour la méthode CC (écriture directe de la somme), avec un besoin en mémoire très inférieur. Par exemple, il faut 300 o.e. pour $T_{25,12}$ contre 180 pour la méthode CC. On peut dire que ces méthodes, créées de toutes pièces, combinent les avantages de la FND (peu de mémoire nécessaire) avec les avantages de la méthode CC (peu d'opérations nécessaires). Elles sont cependant conceptuellement beaucoup plus complexes.

IV. Conclusions

Si l'on ne se soucie pas de la place mémoire, la méthode la moins coûteuse en opérations élémentaires pour les fonctions seuil est la méthode de calcul direct de la somme (CC).

Si l'on recherche un coût minimum en mémoire, la méthode FND est la plus performante, mais le nombre d'opérations est prohibitif.

Si l'on doit, comme il est naturel, tenir compte des deux contraintes, les méthodes de clustérisation perfectionnées que nous avons introduites sont de très loin les plus performantes. Il nous suffit de 438 opérations élémentaires, avec 3 plans mémoire seulement, pour calculerr $T_{36,18}$. Auparavant, il en fallait 163 352 435 399, par la Forme Normale Disjonctive !

Plan de l'étude

1. Présentation générale

Nous précisons le cadre de travail : qu'est-ce qu'une rétine et quelles sont les opérations effectuées dans ses circuits ? Nous faisons la liste des " opérations élémentaires" (recopie de plans-mémoire, opérations "et", "ou", etc) qui peuvent être effectuées de manière massivement parallèle entre deux plans mémoire. Nous modélisons les "Opérateurs de Rétine", de la façon suivante : un O.R. est défini par la donnée d'un curseur : forme géométrique du plan $\mathbb{Z}^2$ comportant n cases, d'une mire (choix particulier d'une case du curseur), et d'une fonction booléenne de n variables.

On constate que les représentations habituelles des fonctions booléennes (Forme Normale Disjonctive et *Ring Sum Expansion*) sont très mal adaptées au calcul dans la rétine. Ceci n'a rien d'étonnant : elles ont été créées dans un tout autre but, et ne se soucient pas du nombre d'opérations à effectuer (qui est en général exponentiel par rapport au nombre des variables). En revanche, la FND est efficace lorsque la place mémoire disponible est restreinte, car elle se contente de trois plans mémoires.

2. Complexité théorique

On détermine la complexité des Opérateurs de Rétine en général : combien d'opérations élémentaires faut-il pour les représenter ?

Nous montrons que, pour presque tous les O.R., la complexité est nécessairement exponentielle, et ce quelle que soit la place mémoire dont on dispose.

Un résultat de Lupanov-Muller donne une complexité de $2^n/n$, pour un opérateur dont le curseur est de taille n, à condition que la place mémoire soit très importante. Si ce n'est pas le cas, il ne reste que la F.N.D., qui nécessite $n \cdot 2^n$ opérations, mais ne requiert que trois plans mémoire.

La conclusion de ce travail est extrêmement claire : dans le cadre de l'étude que nous menons, il est tout à fait vain de vouloir implanter des opérateurs de rétine quelconques, et il est vain de vouloir utiliser des techniques générales, celles-ci étant beaucoup trop coûteuses, en nombre d'opérations ou en place mémoire.

Il faudra donc, dans la suite, choisir avec soin les opérateurs que l'on va considérer (essentiellement des fonctions symétriques ou des fonctions-seuil) et développer pour ces opérateurs des techniques spécifiques.

3. Méthode d'Addition des Variables par Blocs (A.V.B.)

Si, outre le plan où l'image est stockée, on ne dispose que de deux plans mémoire, la Forme Normale Disjonctive est la seule méthode possible (avec ses variantes, Ring Sum Expansion, etc) pour représenter une fonction booléenne quelconque. Elle est très coûteuse en nombre d'opérations : le nombre d'opérations nécessaires croît exponentiellement avec le nombre de variables.

A l'inverse, si l'on dispose d'une place mémoire suffisante, on écrit la somme des variables en base 2, et ceci permet de représenter une fonction seuil avec un très petit nombre d'opérations.

Entre ces cas extrêmes, rien n'était connu. Comment tirer parti, par exemple, de la présence de trois cases mémoire (outre celle de l'image), soit une case de plus que ne requiert la FND, mais bien moins que demandé par les méthodes de représentation triviales ?

Nous résolvons ce problème, et donnons une représentation des fonctions seuil utilisant trois cases mémoire. Cette décomposition représente, en nombre d'opérations, une économie considérable par rapport à la FND. Par exemple, dans le cas de 16 variables, la FND requiert plus de 100 000 opérations élémentaires ; notre méthode seulement 2 024. Dans le cas de 25 variables, la FND nécessite plus de 62 millions d'opérations, notre méthode seulement 38 000.

Puis, nous étendons cette nouvelle représentation aux fonctions symétriques.

4. Optimisation de la méthode A.V.B.

Qu'apporte à une rétine la configuration où l'on dispose de quatre plans mémoire (outre celui où l'image est stockée) ?

La réponse est très claire :

– Si l'on souhaite seulement traiter des fonctions seuil, cette configuration n'apporte pratiquement rien. Le gain en nombre d'opérations, entre la configuration à 3 plans et la configuration à 4, est seulement de 1/2, au mieux, tandis que la complexité de la programmation est accrue.

– Si l'on souhaite traiter des fonctions symétriques, le quatrième plan mémoire permet une optimisation des découpages par blocs : si l'on ne dispose que de trois plans, on est obligé de se restreindre à des découpages en blocs de 2 ou 3 variables. Le gain peut atteindre un facteur 1/100.

En résumé, le quatrième plan est inutile si l'on se contente de fonctions simples (fonctions seuil) ; il devient utile si l'on traite des fonctions plus complexes.

5. Calcul de la somme en binaire

Nous montrons qu'une fonction symétrique booléenne de n variables s'implante de façon optimale, dans une rétine, au moyen de $[\log_2 n] + 3$ plans mémoire ; l'évaluation requiert $2n + \frac{1}{2}(\log_2 n)^2$ opérations.

On peut se contenter de trois plans mémoire en procédant par "clustérisation", c'est à dire en regroupant entre elles, temporairement, les mémoires de plusieurs processeurs. Le nombre d'opérations nécessaires est alors $n(\log_2 n)^2$, lorsque le curseur comporte n cases.

6. Méthode Calcul et Comparaison (C.C.)

Nous nous plaçons dans un cadre où le nombre de plans mémoire est suffisant pour calculer et stocker la somme des variables des fonctions symétriques et seuils que nous envisageons d'implanter sur la rétine.

Nous présentons ensuite la décomposition de ces fonctions à partir de la somme des variables et explicitons les différentes opérations nécessaires à cette décomposition.

Nous concluons sur le coût en opérations et en plans mémoire.

7. Notation redondante

La notation binaire a pour inconvénient d'obliger à la propagation de retenues. La notation redondante (sur 3 bits au lieu de 2) permet de pallier cet inconvénient.

Nous donnons la description des méthodes permettant d'obtenir le calcul de la somme par notation classique (Csc), redondante (Csr), par le calcul de la partie variable en notation classique (CAPVc) ou redondante (CAPVr). Implantation des algorithmes et calcul des coûts respectifs.

8. Comparaison des diverses méthodes

On reprend l'ensemble des méthodes présentées, et on compare leur coût et la place mémoire qu'elles nécessitent.

9. Algorithme majoritaire

Dans le cas où le curseur est carré, nous présentons une méthode nouvelle où l'on calcule d'abord les sommes partielles suivant les colonnes. Calcul du coût en opérations élémentaires et en place mémoire.

10. Méthode par arbre de décomposition

Pour les fonctions seuil, nous avons introduit une nouvelle méthode, qui tire parti des propriétés géométriques du curseur en décomposant la valeur du seuil. Elle requiert un arbre de décomposition. Implantation de l'algorithme et calcul du coût en opérations élémentaires et en place mémoire.

Chapitre I

Présentation Générale

I. Introduction : les rétines artificielles et le traitement ultra-rapide de l'image

Une rétine artificielle est un circuit plan destiné à recueillir et traiter des images. Mais au lieu d'une caméra classique dont l'image, divisée en pixels, est traitée globalement par un circuit unique, on dispose, dans une rétine, d'une matrice de photocapteurs (appelés transducteurs), chacun ayant son circuit de traitement et sa mémoire propre. L'ensemble du circuit intégré est réalisé grâce à la technologie CMOS, qui permet un coût de revient très bas.

Par rapport à un système traditionnel de traitement de l'image, constitué d'une caméra reliée à un ordinateur, la rétine possède les avantages suivants :

– **1)** chaque capteur est associé à son propre circuit de traitement. La proximité des deux fait que chaque traitement élémentaire est très rapide, et de surcroît tous les circuits fonctionnent simultanément, de manière massivement parallèle.

– **2)** les circuits de traitement sont programmables et peuvent réaliser diverses opérations, depuis le filtrage jusqu'à la reconnaissance des formes. Ceci donne naissance à des possibilités entièrement nouvelles, que ne peuvent pas posséder les systèmes classiques de traitement de l'image. En particulier, le comportement de chaque photocapteur peut être influencé par les informations recueillies par les photocapteurs voisins. On peut ainsi, par exemple, régler un seuil de sensibilité pour un photocapteur, en fonction des luminosités mesurées par ses voisins, et ceci permet d'éviter l'aveuglement.

– **3)** Le poids total, l'encombrement et le coût sont très réduits. Une application intéressante pourrait être la discrimination des leurres pour un missile sol-air à très courte portée : il est nécessaire de réagir très rapidement et l'aveuglement par des leurres très brillants est un problème majeur dans ce cadre.

Plus généralement, une solution de type rétine est la seule envisageable lorsque les contraintes de temps (temps d'exécution ou de réaction) sont fortes : au dessous du dixième de seconde et jusqu'à la dizaine de microsecondes.

II. Description des rétines artificielles

Une rétine consiste essentiellement en :

– des photocapteurs, arrangés en matrice rectangulaire ou carrée (256×256 est un maximum pour les technologies actuelles). Chaque photocapteur, à chaque instant, n'a que deux états : excité ou au repos, ce qui se traduit par 1 ou 0.

– Une mémoire pour chaque photocapteur. Cette mémoire, pour des raisons d'encombrement et de coût, n'est que de quelques bits, de 3 à 5 en pratique (5 bits dans la dernière réalisation).

– Des circuits de calcul, agissant de manière identique sur tous les pixels.

On est donc en présence d'une technologie massivement parallèle de type SIMD : « Single Instruction Multiple Data » : on fait partout le même calcul, mais sur des données différentes d'un photocapteur à l'autre.

III. Représentation mathématique d'une rétine

La représentation mathématique d'une rétine s'obtient par la définition des notions d'images, d'opérations élémentaires dans le circuit de traitement ; elle dépend du nombre de bits mémoire affectés par pixel.

La mémoire disponible pour chaque pixel est de quelques bits (de 3 à 5, comme dit plus haut) et on représente cette mémoire par des « plans mémoires », parallèles au plan de l'image. Il est à noter que l'image est toujours contenue dans le premier plan mémoire et est prise en compte dans le décompte des plans mémoires. Donc, si l'on dispose de 5 plans mémoire en tout, cela veut dire qu'il en reste 4 pour les calculs.

Une image, sur une rétine, est l'ensemble des indications fournies par les photocapteurs. Chaque pixel d'une image, en noir et blanc, peut être représenté par une cellule du plan discret $\mathbb{Z}^2$ et cette cellule contient soit la valeur 1 si le pixel est noir, soit la valeur 0 si le pixel est blanc. En considérant pour simplifier le plan comme infini, une image est alors un sous-ensemble du plan discret $\mathbb{Z}^2$, constitué des cellules de valeur 1. Cette image a pour seule propriété d'être finie (il n'y a qu'un nombre fini de pixels qui sont à 1, parce que dans la pratique il n'y a qu'un nombre fini de photorécepteurs).

Mais, à part cela, l'image est un sous-ensemble absolument quelconque du plan $\mathbb{Z}^2$, et n'a aucune régularité particulière : elle n'est ni convexe, ni connexe et donc peut être faite de plusieurs morceaux.

Dans le circuit de traitement, on dispose des opérations suivantes, appelées opérations élémentaires (O.E.). Ces opérations élémentaires sont réalisées entre deux plans mémoires successifs et ne nécessitent pour leur réalisation qu'un seul pas d'horloge. Elles opèrent sur l'ensemble de l'image de façon identique et simultanée et peuvent seulement combiner les cellules de mêmes coordonnées dans chaque plan.

Sur la rétine de dernière génération, les opérations élémentaires suivantes sont disponibles :

a) Recopier un plan sur un autre. Le contenu d'un plan est recopié à l'identique sur le plan suivant.

b) Prendre le complémentaire de l'image. Ceci veut dire que, l'image étant sur le plan 1, elle est recopiée en négatif sur le plan 2 : chaque pixel à 1 devient 0 et inversement. Cette opération est effectuée de manière massivement parallèle et coûte globalement un pas de temps, *quelle que soit la taille de l'image.* On désigne cette opération par « NON ». Le contraire du pixel x sera noté $\bar{x}$.

c) Effectuer des translations, dans les quatre directions N, S, E, O. L'image étant contenue dans un plan peut être globalement translatée, à l'intérieur de ce même plan, dans l'une des quatre directions. Chaque translation élémentaire compte pour un pas de temps : si on translate de 12 pixels vers l'Est et de 3 pixels vers le Sud, cela fait 15 pas de temps au total. Là encore, cette translation est massivement parallèle, sur la totalité de l'image.

d) Réaliser la fonction « ET » entre deux plans. Deux images étant respectivement sur les plans 1 et 2, l'opération « ET » agit sur des pixels de mêmes coordonnées ; elle signifie que le pixel résultat est à 1 si les deux pixels de départ étaient à 1. Le résultat est inscrit sur le plan 2, à la place de la seconde image. Chaque « ET » compte pour un pas de temps. L'opération « ET » est notée $\wedge$.

e) Réaliser la fonction « OU » entre deux plans. Deux images étant respectivement sur les plans 1 et 2, l'opération « OU » agit sur des pixels de mêmes coordonnées ; elle signifie que le pixel résultat est à 1 si l'un des deux pixels de départ était à 1. Le résultat est inscrit sur le plan 2, à la place de la seconde image. Chaque « OU » compte pour un pas de temps. L'opération « OU » est notée $\vee$.

f) Réaliser la fonction « OU EXCLUSIF » entre deux plans. Le résultat de l'opération « OU EXCLUSIF » de deux plans est une image dont la valeur des cellules est 1 si et seulement si une et une seule des deux cellules initiales était à 1. L'opération « OU EXCLUSIF » est notée $\oplus$. Le résultat de l'opération est

inscrit sur le second plan, à la place de la seconde image. Il est à remarquer que cette O.E. est plus coûteuse en temps d'horloge que les précédentes : elle nécessite de l'ordre de 10 pas de temps dans la rétine actuelle.

IV. Le traitement réalisé dans une rétine

Le *traitement* qu'on cherche à réaliser consiste en ceci : à partir des indications fournies par un nombre fixe n de capteurs, prendre une décision, c'est à dire retourner 0 ou 1. Mathématiquement, ce traitement est donc représenté par une *fonction booléenne* de n variables $f(x_1, \ldots, x_n)$, donnée au moyen d'une *représentation spatiale*. On cherche à transformer une image en une autre, en principe plus simple (ne garder que les contours, ou le squelette, etc), en respectant évidemment le principe d'invariance par translation : si on part de deux images I_1 et I_2 indentiques à une translation près, les résultats I'_1 et I'_2 seront identiques à la même translation près.

Voici un exemple de tel traitement : prenons un cadre noir consistant en 3 pixels accolés et décidons de la règle de transformation suivante : appliquons ce cadre en tout point d'une image ; si le cadre coïncide avec l'image (c'est à dire si l'image comporte 3 pixels noirs), le pixel central est mis à 1 ; dans tous les autres cas il est mis à 0. Voici ce qu'on obtient à partir d'une croix 7×7, parce qu'il n'y a que 5 endroits où le curseur coïncide avec l'image : les 5 pixels du centre de la branche horizontale.

Plus généralement, nous disposons :

– d'un curseur, figure géométrique constituée de n pixels (ces pixels ne sont pas nécessairement voisins en général, contrairement à l'exemple ci-dessus),

– de la désignation d'un pixel particulier, appelé la mire, parmi les n pixels constituant le curseur,

– d'une table de vérité, constituée de la façon suivante : il y a 2^n coloriages possibles du curseur (y compris tout blanc et tout noir). Pour chacun d'eux, nous choisissons 0 ou 1. Si 0 est choisi, cela veut dire que si le coloriage est rencontré, la mire sera mise à blanc, et si 1 est choisi, la mire sera mise à noir.

Notons $B(\mathbb{Z}^2)$ (Booléens de $\mathbb{Z}^2$) l'ensemble des images possibles : c'est donc l'ensemble des sous-ensembles de $\{0,1\}^{\mathbb{Z}^2}$ qui n'ont qu'un nombre fini de « 1 ». Le curseur, la mire, et la table de vérité déterminent entièrement un opérateur de $B(\mathbb{Z}^2)$ dans lui-même. Un opérateur de ce type s'appellera « opérateur de Rétine » (OR).

Voici un exemple :

– curseur

– mire

– table de vérité

Voici d'autres exemples d'O.R. :

• la règle « la cellule $\mathrm{P}_{i,j}$ devient noire si $\mathrm{P}_{i+2,j+3}$ l'était » définit un O.R., qui translate de deux cellules vers l'Ouest et de trois cellules vers le Sud. Ici, le curseur est une seule cellule $\mathrm{P}_{i+2,j+3}$ et la mire $\mathrm{P}_{i,j}$.

• l'érosion garde une cellule à 1 si elle-même et ses quatres voisines le sont et définit un O.R. dont le curseur est donné par cinq cellules en forme de croix et la mire est la cellule centrale. On peut décrire l' O.R. « érosion », en fonction des O.E., par la formule suivante :

$$\mathrm{f}\,(\mathrm{P}_{i,j}) \;=\; \mathrm{P}_{i+1,j} \wedge \mathrm{P}_{i-1,j} \wedge \mathrm{P}_{i,j+1} \wedge \mathrm{P}_{i,j-1} \wedge \mathrm{P}_{i,j}$$

• un algorithme majoritaire sur un voisinage est défini par : le pixel $\mathrm{P}_{i,j}$ est mis à 1 (qu'il soit à 1 ou à 0) si, à l'intérieur du carré de côté trois qui l'entoure, au moins cinq pixels sont noirs parmi les neuf.

Comme on l'a vu, un O.R. est défini par une fonction booléenne à n variables $f(x_1, x_2, \ldots, x_n)$. On cherche à le représenter au moyen des seules opérations élémentaires et en outre à le faire le mieux possible, c'est à dire en faisant le moins d'opérations possible, compte-tenu du nombre de plans mémoires disponibles.

Chapitre II

Complexité théorique

I. Introduction

Dans le précédent rapport, nous avons défini les opérateurs de rétine (O.R.) et présenté les difficultés inhérentes à la décomposition de ces O.R. en opérations élémentaires (O.E.) définies sur la rétine. Nous allons établir maintenant des résultats généraux sur la complexité des O.R.

II. Mesure de complexité d'un O.R.

La complexité d'un O.R. est définie par le nombre minimal d'opérations élémentaires nécessaires à sa réalisation, et ceci sans contrainte de places mémoires.

Ce calcul de complexité se fait donc en supposant avoir assez de plans mémoires pour effectuer n'importe quelle enchaînement d'O.E.

Malgré tout, nous allons voir que *la plupart* des opérateurs de rétine de taille n (c.-à-d. dont le curseur est de taille n) ont une mesure de complexité exponentielle en n.

Plus précisément, on peut énoncer :

Proposition 1. – *Pour $n \geq 3$, parmi les 2^{2^n} opérateurs de rétine utilisant un curseur de taille n, au plus $2^{2^{n-1}}$ peuvent être exprimés à l'aide de moins $2^n/2n$ opérations élémentaires.*

Il est à remarquer que la proportion $2^{2^{n-1}}$ est infime par rapport au nombre total 2^{2^n} ; de plus, elle tend très vite vers 0 lorsque $n \to +\infty$.

Démonstration de la Proposition. – Pour établir ce résultat, on procède par dénombrements successifs : d'abord les arbres binaires, puis les négations et enfin les translations.

1. Dénombrement des arbres binaires généraux

Un arbre binaire est un graphe dont les noeuds ont deux entrées et une sortie. Un noeud est simplement une porte binaire "gate", constituée de l'opérateur ET ou de l'opérateur OU. On représente communément les entrées vers le haut et les sorties vers le bas. L'arbre se termine nécessairement par une sortie unique.

Pour un arbre à k portes binaires, il y a $2k$ branches ou arêtes (chaque noeud a deux entrées). Nous ne comptons pas l'arête de sortie.

Le nombre $C(k)$ d'arbres binaires à k portes satisfait la relation de récurrence :

$$C(k) \;=\; \sum_{l=0}^{k-1} C(k-l-1)C(l), \qquad C(0) \;=\; 1.$$

On établit cette récurrence en retirant le noeud terminal de l'arbre, ce qui partage l'arbre binaire initial de longueur k en une paire d'arbres binaires de longueurs respectives $(k-l-1)$ et l, avec l variant de 0 à $k-1$.

Le nombre $C(k)$ est le "nombre de Catalan" :

$$C(k) \;=\; \binom{2k}{k}\frac{1}{k+1}. \tag{1}$$

Il est facile d'obtenir un encadrement de $C(k)$, pour $k = 1, 2, \ldots$:

$$3^k \ \leq \ C(k) \ \leq \ \frac{4^k}{k}. \tag{2}$$

Plus précisément, l'application de la formule de Stirling donne :

$$C(k) \ \sim \ \frac{1}{\sqrt{\pi}} \ \frac{4^k}{k^{3/2}}. \tag{3}$$

2. Prise en compte des deux opérations élémentaires

Chaque noeud de l'arbre représente une opération élémentaire ET ou OU. Pour chaque arbre, il est possible de choisir de 2^k façons les ET parmi les k symboles, les autres étant des OU. Le nombre d'arbres à k noeuds binaires, où chaque noeud est constitué d'un ET ou d'un OU est donc :

$$2^k C(k). \tag{4}$$

3. Prise en compte des négations

Soit p le nombre total de négations qui vont apparaître. Il peut y avoir au plus une négation par branche (parce que deux négations se compensent). Chaque arbre comporte $2k$ branches, il y a donc $\binom{2k}{p}$ choix possibles pour les p branches "négatives", avec $p \leq 2k$.

Le nombre d'arbres avec k sommets et p négations est donc :

$$\binom{2k}{p} 2^k C(k). \tag{5}$$

4. Prise en compte des translations

Finalement, on introduit les translations, au nombre de t. Elles commutent avec les NON. Il y a $2k$ branches et on peut mettre autant de translations qu'on veut par branche. Il y a donc $(2k)^t$ répartitions possibles. Au total, le nombre d'arbres binaires, faits avec k symboles ET et OU, comportant p négations et t translations est :

$$N(k,p,t) \ \leq \ (2k)^t \binom{2k}{p} 2^k C(k). \tag{6}$$

Si maintenant nous fixons un nombre total m d'opérations élémentaires, décomposées en k OU ou ET, p négations et t translations, avec bien sûr $k + p + t = m$, le nombre total d'opérateurs distincts que nous pouvons écrire avec ces opérations est au plus :

$$N_m \ \leq \ \sum_{k+p+t=m} (2k)^t \binom{2k}{p} 2^k C(k) \tag{7}$$

ou encore :

$$N_m \ \leq \ \sum_{k+p+t=m} (2k)^t \binom{2k}{p} 2^k \binom{2k}{k} \frac{1}{k+1}. \tag{8}$$

Ceci s'écrit :

$$N_m \ \leq \ \sum_{k=0}^{m} (2k)^{m-k} \frac{2^k}{k+1} \binom{2k}{k} \sum_{p=0}^{2k} (2k)^{-p} \binom{2k}{p}. \tag{9}$$

Or :

$$\sum_{p=0}^{2k} (2k)^{-p} \binom{2k}{p} \ = \ (1 + \frac{1}{2k})^{2k} \sim e,$$

et en reportant dans (9), nous trouvons :

$$\begin{aligned}
N_m &\leq e\sum_{k=0}^{m}(2k)^{m-k}\frac{2^k}{k+1}\binom{2k}{k} \\
&= e2^m\sum_{k=0}^{m}\frac{k^{m-k}}{k+1}\binom{2k}{k} \\
&\leq e2^m\sum_{k=1}^{m}k^{m-k-1}\binom{2k}{k} \\
&\leq e2^m\sum_{k=1}^{m}m^{m-k-1}\binom{2m}{k} \\
&\leq e2^m m^{m-1}\sum_{k=1}^{m}m^{-k}\binom{2m}{k} \\
&\leq e2^m m^{m-1}(1+\frac{1}{m})^{2m}
\end{aligned}$$

et finalement,

$$N_m \leq e^3 2^m m^{m-1}. \tag{10}$$

Pour $m = \alpha 2^n/n$ avec $\alpha < 1$ à choisir, on obtient :

$$\log N_m \leq 3 + \frac{\alpha 2^n}{n}\log 2 + (\frac{\alpha 2^n}{n} - 1)\log\frac{\alpha 2^n}{n}$$

et donc :

$$\begin{aligned}
\log N_m - \log 2^{2^n} &\leq 3 + \frac{\alpha 2^n}{n}\log 2 + \frac{\alpha 2^n}{n}\log\frac{\alpha 2^n}{n} - \log\frac{\alpha 2^n}{n} - 2^n\log 2 \\
&= 2^n\left(-\log 2 + \frac{\alpha}{n}\log 2 + \frac{\alpha}{n}\log\alpha + \alpha\log 2 - \frac{\alpha\log n}{n}\right) - \log\frac{\alpha 2^n}{n} + 3 \\
&= 2^n\left((\alpha-1)\log 2 + \frac{\alpha}{n}(\log 2 + \log\alpha) - \frac{\alpha\log n}{n}\right) - \log\frac{\alpha 2^n}{n} + 3
\end{aligned}$$

Pour tout $\alpha < 1$ fixé, ceci est bien sûr équivalent à $2^n(\alpha-1)\log 2$ lorsque $n \to +\infty$.

Pour obtenir un résultat quantitatif, on donne une valeur précise à α et pour $\alpha = 1/2$, on obtient :

$$\log N_{2^{n-1}/n} - \log 2^{2^n} \leq 2^n\left(-\frac{1}{2}\log 2 - \frac{1}{2}\frac{\log n}{n}\right) - \log\frac{2^{n-1}}{n} + 3$$

Or, dès que $n \geq 4$, l'inégalité suivante est vérifiée :

$$\frac{-2^{n-1}}{n}\log n - \log\frac{2^{n-1}}{n} + 3 \leq 0, \tag{11}$$

et on obtient ainsi la formule, pour $n \geq 4$:

$$\frac{N_{2^{n-1}/n}}{2^{2^n}} \leq 2^{-2^{n-1}}$$

ou encore

$$N_{2^{n-1}/n} \leq 2^{2^{n-1}}. \tag{12}$$

On voit donc que le nombre maximum d'arbres binaires qui utilisent $2^{n-1}/n$ opérations élémentaires est au plus égal à $2^{2^{n-1}}$. Cette même majoration vaut donc pour le nombre total d'opérateurs de rétine utilisant $2^{n-1}/n$ opérations élémentaires.

Comme le nombre total d'O.R. correspondant à un curseur de taille n est 2^{2^n}, le nombre d'O.R. utilisant plus de $2^{n-1}/n$ opérations élémentaires est $2^{2^n} - 2^{2^{n-1}} = 2^{2^n}(1 - \frac{1}{2^{2^{n-1}}})$, c'est-à-dire presque tous.

Pour $n = 3$, il faut revenir à la formule (8) et faire un calcul direct. On a $2^n/2n < 2$, et $N_2 \leq 16$ (évaluation grâce à la formule (8)). L'évaluation (12) reste donc encore valable et termine la démonstration du lemme.

Après avoir établi une borne inférieure, en termes de nombre d'opérations nécessaires, pour la majorité des O.R., nous allons établir une borne supérieure, valable pour tous. Elle est donnée par le théorème de Lupanov (voir [1]) ; ce résultat est aussi attribué à Muller (voir [2]). L'énoncé de Lupanov nous a été communiqué par Antoine Manzanera.

III. Bornes supérieures optimales valables pour l'ensemble des O.R.

Théorème (Lupanov). – *Tout opérateur de rétine de taille de curseur n a une complexité en $\frac{2^n}{n}(1+o(1))$.*

Ceci signifie que, pour tout $\varepsilon > 0$, il existe un $n_0 \geq 1$ tel que, pour tout $n \geq n_0$, tout opérateur de rétine de taille n peut s'écrire avec au plus $(1+\varepsilon)\frac{2^n}{n}$ opérations.

On se contentera de démontrer ici un résultat sensiblement moins fort : la complexité est en $\frac{2^n}{n}(4+o(1))$. Ce choix est justifié par deux raisons :

- la démonstration est plus simple et plus facile à comprendre,
- de toute façon, une complexité exponentielle est trop élevée pour une implantation dans la rétine.

Tout O.R. de taille n peut se représenter par une fonction booléenne f de n variables. Au premier niveau, cette fonction peut se décomposer de la façon suivante :

$$f(x_1, x_2, \ldots, x_n) \;=\; (f(x_1, x_2, \ldots, x_{n-1}, 0) \;\wedge\; \overline{x}_n) \;\vee\; (f(x_1, x_2, \ldots, x_{n-1}, 1) \;\wedge\; x_n)$$

et récursivement, il est possible d'obtenir une décomposition, au niveau $(n-k)$, en fonctions booléennes de k variables $(1 \leq k \leq n)$. Au niveau de récursivité $(n-k)$, il est nécessaire de calculer 2^{n-k} fonctions à k variables qui interviennent dans la décomposition de f. Si on suppose que l'on calcule toutes les fonctions booléennes à k variables, la complexité de tout O.R. à n variables est inférieur à :

$$N \;\leq\; N_1(n-k) + N_2(k),$$

où :

* $N_1(n-k)$ est le nombre d'opérations élémentaires nécessaires pour assembler les 2^{n-k} fonctions booléennes à k variables. Ceci peut être représenté par un arbre binaire dont on élague les branches supérieures au niveau $(n-k)$.

* $N_2(k)$ est le nombre d'opérations pour calculer 2^{2^k} fonctions à k variables.

Il est facile d'obtenir $N_1(n-k)$ par récurence. On a les relations :

$$N_1(n-k) \;=\; 3 + 2N_1(n-k-1)$$

et

$$N_1(1) \;=\; 3,$$

d'où

$$N_1(n-k) \;=\; 3\,(2^{n-k} - 1).$$

Pour calculer les fonctions booléennes à k variables, on calcule toutes les fonctions à $(k-1)$ variables, puis toutes les expressions de la forme :

$$f_i(x_1, x_2, \ldots, x_{k-1}) \;\wedge\; x_k \;, \tag{12}$$

et de la forme :

$$f_i(x_1, x_2, \ldots, x_{k-1}) \ \wedge \ \overline{x}_k \ . \tag{13}$$

Enfin, on peut exprimer n'importe quelle fonction booléenne à k variables par une F.N.D., soit :

$$f(x_1, x_2, \ldots, x_k) \ = \ (f_i(x_1, x_2, \ldots, x_{k-1}) \ \wedge \ x_k) \ \vee \ (f_j(x_1, x_2, \ldots, x_{k-1}) \ \wedge \ \overline{x}_k) \, .$$

La complexité $N_2(k)$ est donc la somme de $N_2(k-1)$ opérations élémentaires nécessaires à l'obtention des fonctions à $(k-1)$ variables, de $2 \times 2^{2^{k-1}}$ opérations nécessaires aux calculs de l'ensemble des expressions (12) et (13), de $\left(2^{2^{k-1}}\right)^2 \ = \ 2^{2^k}$ opérations $\vee$ nécessaires à l'obtention de la F.N.D. de n'importe quelle fonction booléenne à k variables, d'où :

$$N_2(k) \ = \ N_2(k-1) + 2 \times 2^{2^{k-1}} + 2^{2^k}.$$

Comme $2 \times 2^{2^{k-1}}$ et $N_2(k-1) \ = \ (k-2)2^{2^{k-1}}$ sont en $o(2^{2^k})$, on a :

$$N_2(k) \ = \ 2^{2^k} \, (1 + o(1)),$$

et

$$N(k) \ = \ 3 \, (2^{n-k} - 1) + 2^{2^k} (1 + o(1)).$$

Pour $k = \log_2(n - \log_2(n))$, on obtient :

$$\begin{aligned} N_{\log_2(n-\log_2(n))} \ &= \ 3 \, \frac{2^n}{(n - \log_2(n))} + \frac{2^n}{n}(1 + o(1)) \\ &= \ \frac{2^n}{n}(4 + o(1)). \end{aligned}$$

Finalement, comme annoncé, la complexité de tout O.R. est au plus en $\frac{2^n}{n}(4 + o(1))$. Ceci achève la démonstration.

Notons que cette décomposition "économique" en opérations nécessite en revanche une importante place mémoire, car il faut connaître et enregistrer l'ensemble des fonctions booléennes à k variables.

IV. Conclusion

Ces décompositions nous ont permis d'obtenir des valeurs générales sur la complexité des O.R. Elles montrent de façon claire qu'il va falloir faire un choix pour les O.R. que l'on va implanter, parce que l'implantation d'un O.R. quelconque est hors de portée, compte-tenu de la faible place mémoire dont on dispose et du temps de calcul court que l'on requiert. Pour réaliser ce choix, on va passer en revue les O.R. les plus utiles pour les traitements d'images dans la rétine, et pour ceux-là on mettra en place une implantation optimale. Ce sera l'objet des prochains rapports.

Référence

[1] A. Manzanera, *Mesures de complexité sur la rétine*, Rapport interne, avril 1998.

[2] R. B. Boppana and M. Sipser : The complexity of finite functions. *Handbook of theoretical computer science*, Ed. Jan Van Leeuwen, Elsevier and M.I.T. Press, vol. A, 1990.

V. Les différentes représentations et leurs problématiques

En se basant sur certaines représentations des fonctions booléennes, on peut se contenter d'un nombre restreint de plans mémoires. Il y a deux représentations très classiques :

La forme normale disjonctive

Pour obtenir la forme normale disjonctive, notée F.N.D., on décrit la fonction f de la façon suivante :

$$f(x_1, x_2, \ldots, x_n) \;=\; [\, f(x_1, x_2, \ldots, x_{n-1}, 1) \wedge x_n \,] \;\vee\; [\, f(x_1, x_2, \ldots, x_{n-1}, 0) \wedge \bar{x}_n \,]$$

d'où récursivement :

$$f(x_1, \ldots, x_n) \;=\; \sum_{i_1 < \cdots < i_n} y_{i_1} \cdots y_{i_n}$$

où y_i est soit x_i, soit $\bar{x}_i$ (opposé de x_i). Dans cette formule, le signe $\sum$ désigne le «ou »et les produits désignent les «et »(cette notation, très commune, sera constamment utilisée dans la suite).

Ainsi, dans l'exemple précédent, si x_1, x_2, x_3 désignent les 3 pixels du curseur,

$$f(x_1, x_2, x_3) \;=\; x_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3.$$

Cette représentation peut être utilisée pour la décomposition de l'Opérateur de Rétine en Opérations Elémentaires, de la façon suivante :

– sur le premier plan, on dispose l'image,

– en chaque point de l'image, on dispose le curseur, et on note ce que valent $x_1 x_2 x_3$. Au moyen d'un non et de deux translations, on calcule sur un deuxième plan $x_1 x_2 \bar{x}_3$ et on marque 1 à chaque mire où cette quantité vaut 1. On inscrit le résultat sur le plan 3. On passe au monôme $x_1 \bar{x}_2 x_3$ que l'on calcule sur le plan 2, et on inscrit le «OU »$(x_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3)$ sur le plan 3, et ainsi de suite avec les deux autres monômes : $\bar{x}_1 x_2 x_3$ et $x_1 x_2 x_3$. On voit ainsi que trois plans mémoires suffiront, mais que :
– il y a n translations par monôme,
– il y a 2^n monômes,
soit un nombre minimal d'opérations de $n \times 2^n$, sans compter les négations.

Considérons par exemple la «fonction seuil »définie par :

$$f(x_1, x_2, \ldots, x_n) \;=\; 1 \qquad \text{si} \quad \sum_{i=1}^{n} x_i \geq \frac{n}{2}\,,$$

où n est le nombre de cellules du curseur. Pour cette fonction, les 2^n monômes possibles apparaissent presque tous. Cette fonction «simple » est parmi les pires imaginables : sa représentation sous forme normale disjonctive requiert un nombre d'opérations qui croît exponentiellement avec n.

B. La «Ring Sum Expansion »

Dans la FND, les monômes sont évidemment distinctes, mais chaque variable apparaît dans un très grand nombre de monômes. Il peut être tentant de distinguer les termes où x_n apparaît et ceux où il n'apparaît pas (et ainsi de suite avec les autres variables). C'est l'objet de la «Ring Sum Expansion », notée R.S.E.

Pour l'obtenir, on décompose f de la façon suivante :

$$f(x_1, \ldots, x_n) \;=\; f(x_1, \ldots, x_{n-1}, 0) \oplus (x_n \cdot Q(x_1, \ldots, x_{n-1}))\,, \tag{1}$$

où Q est défini par les formules :

– si $f(x_1, \ldots, x_{n-1}, 0) = 0$ et $f(x_1, \ldots, x_{n-1}, 1) = 0$, $Q(x_1, \ldots, x_{n-1}) = 0$.

– si $f(x_1, \ldots, x_{n-1}, 0) = 0$ et $f(x_1, \ldots, x_{n-1}, 1) = 1$, $Q(x_1, \ldots, x_{n-1}) = 1$.

– si $f(x_1, \ldots, x_{n-1}, 0) = 1$ et $f(x_1, \ldots, x_{n-1}, 1) = 0$, $Q(x_1, \ldots, x_{n-1}) = 1$.

– si $f(x_1, \ldots, x_{n-1}, 0) = 1$ et $f(x_1, \ldots, x_{n-1}, 1) = 1$, $Q(x_1, \ldots, x_{n-1}) = 0$.

On peut noter ceci par une formule abrégée :

$$Q(x_1, \ldots, x_{n-1}) \;=\; f(x_1, \ldots, x_{n-1}, 0) \oplus f(x_1, \ldots, x_{n-1}, 1).$$

Par exemple, la fonction $x_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1x_2\bar{x}_3 + x_1x_2x_3$ s'écrit $(x_3\bar{x}_2) \oplus x_1$.

En itérant la formule (1), on obtient une représentation de f sous forme de monômes :

$$f(x_1, \ldots, x_n) \;=\; \oplus_{i_1 < \cdots < i_k} x_{i_1} \cdots x_{i_k} \;.$$

La différence avec la FND est que les variables ne sont jamais remplacées par leur négation. Le «OU», par contre, est remplacé par un ou exclusif.

Cette représentation est en général moins intéressante que la FND : le nombre de monômes est le même, et le ou «exclusif» est plus long à calculer.

Dans certains cas, on peut prendre en compte la «dimension spatiale» des O.R. Considérons par exemple l'O.R. d'érosion, dont la fonction booléenne est :

$$f((P_{i,j})) \;=\; P_{i,j} \wedge P_{i+1,j} \wedge P_{i+1,j+1} \wedge P_{i,j+1} \;.$$

On peut l'écrire comme la composition de deux O.R. O_0 et O_1 définis par

$$\begin{aligned} O_0(P_{i,j}) &= P_{i,j} \wedge P_{i+1,j} \\ O_1(P_{i,j}) &= P_{i+1,j+1} \wedge P_{i,j+1} \;. \end{aligned}$$

et le calcul sera plus rapide en utilisant cette décomposition. Mais elle n'a pas de sens pour une fonction booléenne en général, ni même pour chaque O.R.

VI. Conclusion

Nous avons présenté les opérateurs de Rétine et montré en quoi les représentations usuelles des fonctions booléennes étaient inadaptées pour une implantation efficace de ces opérateurs. Dans la suite de notre travail, nous allons :

– donner des bornes inférieures pour le nombre minimal d'opérations requises en général,

– montrer que pour certains types de fonctions (fonctions symétriques en particulier) des implantations efficaces peuvent être réalisées.

Chapitre IV

Optimisation de la méthode A.V.B.

I. Généralités

Dans le précédent rapport, nous avons présenté une nouvelle méthode de décomposition des fonctions seuil et des fonctions symétriques. Cette nouvelle méthode, appelée méthode d'Arrangements des Variables par Blocs (AVB) nécessite, dans le cas général, trois cases mémoire pour être applicable pour les fonctions seuil et peut être étendue aux fonctions symétriques si l'on se limite aux découpages par blocs de 2 (voir rapport 3) ou de 3 (voir annexe B du présent rapport). Nous avons aussi montré que cette méthode était toujours moins coûteuse en nombre d'o.e. que la FND pour les fonctions seuil.

Dans le cadre de quatre cases mémoire, nous présentons une version optimisée de la méthode AVB pour les fonctions seuil. Nous donnons une description du principe, la gestion des cases mémoires, le coût du calcul en o.e. et des comparaisons numériques avec la méthode de la FND.

II. Optimisation de la méthode AVB pour les fonctions seuil

Pour les fonctions seuil, la méthode AVB ne nécessite que trois cases mémoire et est plus performante que la FND. L'ajout d'une case mémoire supplémentaire ne permet pas de modifier fondamentalement la déroulement de cette méthode. Cependant, il est possible d'optimiser le coût du calcul en stockant sur la case mémoire supplémentaire des résultats intermédiaires, qui seront nécessaires ultérieurement et ainsi d'économiser ces calculs redondants.

Déjà, dans le précédent rapport (voir annexe C), nous avions présenté une optimisation de la méthode AVB qui reposait sur la possibilité de stocker un résultat intermédiaire dans la case C, quand elle est disponible.

Maintenant, nous disposons systématiquement d'une case mémoire supplémentaire pour stocker les calculs intermédiaires et nous pouvons envisager différentes méthodes d'optimisation. La différence entre ces méthodes est seulement liée à la nature du résultat stocké sur cette case. Nous présentons une méthode qui optimise le calcul des comparaisons entre la somme partielle des variables d'un bloc et un seuil partiel, en limitant le nombre d'opérations nécessaires à son obtention. Une seconde approche est présentée succinctement à la fin.

II.1 Optimisation des comparaisons

a) Description de la méthode

La méthode AVB optimisée suit le même principe que la méthode AVB. Pour un découpage donné par blocs de k variables, on énumère tous les m-uplets $(p_1, \ldots, p_m)$ de $S_b(p, k, m)$. Ensuite, pour chaque m-uplet $(p_1, \ldots, p_m)$ de la première énumération, on compare successivement, dans chaque bloc, la somme des variables du bloc avec le seuil partiel p_j.

Cependant, dans la nouvelle méthode, le calcul des comparaisons entre la somme des variables du bloc et le seuil partiel est optimisé.

Dans la méthode AVB, pour la plupart des valeurs de seuil, cette comparaison est obtenue grâce au lemme 3 (voir rapport $n°3$ § II.1.c) dont on rappelle ici brièvement l'idée. Pour connaître le résultat de la comparaison entre la somme partielle S_j des variables du bloc et le seuil p_j, il suffit de vérifier que tous les ensembles E à $k - p_j + r$ éléments, pris parmi les indices des k variables, vérifient tous :

$$\sum_{i \in E} x_i \ \geq \ r$$

pour que la condition

$$S_j \;=\; \sum_{i=1}^{k} x_i \;\geq\; p_j$$

soit vérifiée.

Dans la suite de ce paragraphe, l'indice j est omis et sans perte de généralité, nous raisonnons sur le premier bloc, noté B_1.

En choisissant $r = 1$, tous les ensembles E sont des sous-ensembles à $k - p_1 + 1$ éléments de $\{1, \ldots, k\}$ et on note $i_1, \ldots, i_{k-q+1}$ les indices qui le composent. La condition $\sum_{i \in E} x_i \geq 1$ s'écrit donc :

$$x_{i_1} \vee \cdots \vee x_{i_k - q+1} \tag{1}$$

et peut se calculer sur une seule case mémoire. La condition "Pour tout E, $\sum_{E} x_i \geq 1$" s'écrit finalement :

$$\bigwedge_{i_1 < \cdots < i_{k-q+1}} x_{i_1} \vee \cdots \vee x_{i_k - q+1}$$

et dans la méthode AVB, le coût d'une comparaison est de $N_{compar}(\text{AVB}) \;=\; p_1 \binom{k}{p_1} - 1$ opérations élémentaires.

On optimise le calcul de cette comparaison. Le parcours sur tous les ensembles E est inévitable. Cependant, on peut réduire le nombre d'opérations en conservant en mémoire une partie du calcul commun à plusieurs ensembles E.

En lieu et place du calcul systématique, pour chaque E, de l'expression (1), on calcule et on stocke la moitié de cette expression et on utilise cette information pour tous les E comportant la même première moitié dans la séquence d'indices.

On obtient donc la méthode suivante. En posant $u = k - p_1 + 1$ et $v = [u/2]$, pour les indices $i_1 < i_2 < \cdots < i_v$ fixés, on calcule $x_{i_1} \vee \cdots \vee x_{i_v}$, que l'on stocke sur la quatrième case mémoire, notée C'.

On calcule ensuite, pour tous les E comportant cette même suite d'indices et v autres indices vérifiant $i_v < i_{v+1} < i_{v+2} < \cdots < i_{2v}$, l'expression suivante :

$$C' \;\vee\; x_{v+1} \vee \cdots \vee x_{2v} \tag{2}$$

Pour chaque E, cette expression correspond à (1). Mais elle est moins coûteuse en o.e., car l'estimation de la première moitié de l'expression n'est effectuée qu'une seule fois.

b) Calcul du coût de la nouvelle méthode

Le calcul du nombre d'o.e. de cette méthode repose sur l'obtention d'une part du coût de calcul de plusieurs ensembles E comportant un nombre d'indices communs et d'autre part sur l'énumération, pour i_v fixé, de toutes les suites vérifiant $i_1 < i_2 < \cdots < i_v$.

Pour chaque ensemble E ayant les v premiers indices en commun, il faut v OU pour calculer l'expression (2), plus un ET entre deux ensembles E (sauf pour le dernier). Pour obtenir le nombre d'o.e., il faut ensuite estimer, pour i_v fixé, le nombre d'ensembles E ayant les mêmes v premiers indices : il est équivalent au choix de v autres indices vérifiant $i_v < i_{v+1} < i_{v+2} < \cdots < i_{2v}$, parmi les $k - i_v$ restants. Il y a $\binom{k-i_v}{v}$ ensembles E pour tout i_v allant de v à $k - v$.

On obtient, pour i_v fixé, le nombre d'opérations élémentaires suivant : $v\binom{k-i_v}{v}$ OU et $\binom{k-i_v}{v}-1$ ET, auquel il faut ajouter l'estimation des v premières variables, soit $v-1$ OU.

Au total, $v-2+(v+1)\binom{k-i_v}{v}$ o.e. sont nécessaires pour obtenir toutes les expressions de tous les ensembles E ayant v indices communs $i_1 < i_2 < \cdots < i_v$.

Mais cette opération est répétée pour toute valeur de i_v allant de v à $k-v$ et le nombre d'o.e. est égal à :

$$\sum_{i_v=v}^{k-v} v-2+(v+1)\binom{k-i_v}{v}$$

pour chaque suite de $v-1$ premiers indices tels que : $i_1 < i_2 < \cdots < i_v$.

Ensuite, pour une valeur fixée de i_v, le choix des $v-1$ premiers indices $i_1 < i_2 < \cdots < i_v$ peut se faire lui aussi de plusieurs façons : il y a $\binom{i_v-1}{v-1}$ choix possibles.

Il faut prendre en compte entre chaque choix possible, une opération ET, sauf pour le dernier. Finalement, le coût de chaque comparaison est égal à :

$$\begin{aligned}
N_{compar}(\text{AVB}_{opt}) &= \sum_{i_1<\cdots<i_v} \left(\sum_{i_v=v}^{k-v} \left(v-2+(v+1)\binom{k-i_v}{v} \right) + 1 \right) - 1 \\
&= \sum_{i_v=v}^{k-v} \left(\sum_{i_1<\cdots<i_v} \left(v-1+(v+1)\binom{k-i_v}{v} \right) \right) - 1 \\
&= \sum_{i_v=v}^{k-v} \binom{i_v-1}{v-1} \left(v-1+(v+1)\binom{k-i_v}{v} \right) - 1 \qquad (3)
\end{aligned}$$

Pour calculer le gain en nombre d'o.e. entre les méthodes AVB et AVB optimisée, on peut estimer le gain effectué sur chaque comparaison en comparant $N_{compar}(\text{AVB}_{opt})$ avec son équivalent pour la méthode AVB, à savoir $N_{compar}(\text{AVB})$. Le gain est alors défini par le rapport suivant :

$$\begin{aligned}
G &= N_{compar}(\text{AVB}_{opt})/N_{compar}(\text{AVB}) \\
&= \frac{1}{p_1\binom{k}{p_1}} \sum_{i=\frac{k-p_1+1}{2}}^{\frac{k+p_1+1}{2}} \binom{i-1}{\frac{k+p_1-1}{2}} \left(\frac{k-p_1-1}{2} + \frac{k-p_1+3}{2}\binom{k-i}{\frac{k-p_1+1}{2}} \right)
\end{aligned}$$

où k est le nombre de variables dans chaque bloc et p_1 est le seuil compris entre 0 et k. Plus le rapport est petit, plus le gain est important.

Pour les valeurs de k allant de 5 à 100 et des seuils p_1 allant de 2 à $k-1$ (les cas $p_1 = 0, 1$ et k étant traités de façon différente), on estime numériquement les valeurs $N_{compar}(\text{AVB}_{opt})$ et $N_{compar}(\text{AVB})$ et le gain G :

k	5	6	9	12	100
p_1	2	3	4	5	49
$N_{compar}(\text{AVB}_{opt})$	18	51	376	2685	$2 * 10^{30}$
$N_{compar}(\text{AVB})$	20	60	504	3960	$4 * 10^{30}$
Gain G	0.9	0.85	0.74	0.67	0.52

Dans tous les cas, l'optimisation proposée permet d'obtenir un gain. Mais ce gain est d'autant plus important que le nombre k de variables est grand. Asymptotiquement, pour k tendant vers l'infini, le gain tend vers $1/2$.

Connaissant le coût d'une comparaison entre les variables d'un bloc et un seuil partiel, il est possible d'estimer le nombre d'o.e. nécessaire pour une fonction seuil $T_{n,p}$.

On note maintenant $N_{compar}(\text{AVB}_{opt}) = C(p_j)-1$ où p_j est le seuil partiel dont dépend la comparaison et on considère, sans perte de généralité, les seuls cas où tous les blocs sont égaux à k.

On en déduit que le nombre d'o.e. que requiert le test du m-uplet $(p_1, p_2, \cdots, p_m)$ est :

$$N_{op}(p_1, \ldots, p_m) = C(p_1) + \cdots + C(p_m) - 1$$

Le nombre d'opérations requises lors de l'énumération de toutes les suites $(p_1, \ldots, p_m)$ est donc :

$$N_{op} \leq \sum_{(p_1,\ldots,p_m)} \left(\sum_j C(p_j) - 1 \right).$$

Pour obtenir le nombre total d'opérations, il est nécessaire d'ajouter à ce nombre N_{op} les opérations OU nécessaires entre deux m-uplets $(p_1, \ldots, p_m)$ de l'énumération, soit au total $s(p,k,m)$ opérations, d'où :

$$\begin{aligned} N_O &\leq N_{op} + s(p,k,m) \\ &\leq \sum_{(p_1,\ldots,p_m)} \sum_j C(p_j). \end{aligned}$$

A l'intérieur de la première somme, les m-uplets $(p_1, \ldots, p_m)$ sont astreints à $p_j \leq k$ et $\sum p_j = p$, c'est-à-dire $(p_1, \ldots, p_m) \in S(p,k,m)$ et le nombre total d'opérations est majoré par :

$$N_O \leq \sum_{j=1}^{m} \sum_{(p_1,\ldots,p_m) \in S(p,k,m)} C(p_j).$$

La somme sur j est constituée de m termes égaux. On a donc :

$$N_O \leq m \sum_{(p_1,\ldots,p_m) \in S_b(p,k,m)} C(p_m)$$

Le nombre de suites $(p_1, p_2, \ldots, p_m) \in S(p,k,m)$ se détermine de la façon suivante. Le seuil p_m ne peut prendre que les valeurs $0, 1, \ldots, k$. Si $p_1 + \cdots + p_m = p$, on a $p_1 + \cdots + p_{m-1} = p - p_m$, et donc $(p_1, p_2, \ldots, p_{m-1}) \in S(p - p_m, k, m-1)$ et on obtient finalement :

$$m \sum_{(p_1,\ldots,p_m) \in S(p,k,m)} C(p_m) = m \sum_{p_m=0}^{\min(k,p)} s(p - p_m, k, m-1)\, C(p_m).$$

Proposition 1. – *Lorsqu'on dispose de quatre cases mémoire, le nombre d'opérations nécessaires à l'implantation de la fonction seuil $T_{n,p}$, obtenu avec la méthode AVB optimisée, lors d'un découpage en blocs de k variables, est au plus égal à :*

$$N_O \leq m \sum_{p_m=0}^{\min(k,p)} s(p - p_m, k, m-1)\, C(p_m)$$

où $n = km$ et $C(p)$ est égal à :

$$C(p) = \sum_{i_v=v}^{k-v} \binom{i_v - 1}{v - 1} \left(v - 1 + (v+1) \binom{k - i_v}{v} \right).$$

c) Estimations numériques

Nous allons maintenant donner des estimations numériques du nombre d'opérations élémentaires nécessaires à l'implantation de la "pire" fonction seuil, la fonction $T_{n,[n/2]}$, avec la nouvelle méthode AVB optimisée. Ces estimations sont obtenues grâce à la formule de la proposition 1 et elles sont comparées avec celles de la FND et de AVB.

On note N_{FND} le nombre d'opérations de la Forme Normale Disjonctive, et $N_T(k)$ le nombre d'opérations donné par la méthode AVB, lorsque les n variables sont divisées en blocs de k et $N_O(k)$ le nombre d'opérations donné par la méthode AVB optimisée. Pour le nombre n de variables variant de 9 à 36, on obtient les estimations suivantes :

(n,p)	(9,4)	(16,8)	(25,12)	(36,18)
k	5	6	9	9
N_{FND}	503	102 959	62.10^{10}	163.10^{9}
N_{AVB}	107	2024	38 084	672 628
$N_{compar}(\text{AVB}_{opt})$	98	1722	28 190	497 762
Gain G AVB/AVB_{opt}	0.9	0.85	0.74	0.67

Le gain obtenu par l'optimisation de la méthode AVB s'avère être très faible pour les fonctions seuil ayant un petit nombre de variables. Ce gain ne justifie pas l'ajout d'une quatrième case mémoire.

Remarque

Au lieu d'utiliser la quatrième case mémoire pour économiser des calculs à l'intérieur de chaque ensemble E, on peut l'utiliser –le raisonnement est identique– pour stocker des résultats partiels relatifs à l'énumération $(p_1,\ldots,p_m)$. On peut décider par exemple de fixer $(p_1,\ldots,p_{m/2})$ (et les résultats de $S_1 \geq p_1, \ldots, S_{m/2} \geq p_{m/2}$ sont stockés) et de faire varier $(p_{m/2+1},\ldots,p_m)$. On recalcule à chaque fois $S_{m/2+1} \geq p_{m/2+1}, \ldots, S_m \geq p_m$. Le principe est exactement le même et le gain est inférieur à 1/2 : il est voisin de 1/2 lorsque $m \to \infty$. La méthode présentée plus haut est donc préférable lorsque k est grand et m petit, celle-ci lorsque k est petit et m grand.

Annexe A

Sur la représentation de certaines fonctions booléennes sous forme de chaînes

Soient $x_1,\ldots,x_n$ des variables booléennes. Une *chaîne* de longueur 1 est simplement la donnée de l'une quelconque des variables, disons x_{n_1}.

Supposons définie une chaîne de longueur $k-1$, notée C_{k-1}. Alors une chaîne de longueur k est :

$$C_k \;=\; (C_{k-1})T_k x_{n_k}$$

où T_k est l'un quelconque des trois opérateurs $\vee$, $\wedge$, $\oplus$, et x_{n_k} l'une quelconque des variables.

En d'autres termes, on passe de la longueur $k-1$ à la longueur k en combinant avec l'une quelconque des variables ; les répétitions sont admises.

Exemple :

$$((x_1 \vee x_2) \wedge x_3) \oplus x_1 \tag{1}$$

est une chaîne de longueur 4 constituée avec trois variables.

Dans la suite, nous omettrons les parenthèses :

$$C_k \;=\; x_{n_1} T_1 x_{n_2} T_2 \cdots T_{k-1} x_{n_k}$$

signifie

$$C_k \;=\; (\cdots (x_{n_1} T_1 x_{n_2}) T_2 \cdots)\, T_{k-1} x_{n_k}.$$

L'intérêt des chaînes, dans le contexte rétine, est qu'elles peuvent être calculées avec une seule case mémoire. Par exemple, pour (1), on calcule successivement :

$$C \leftarrow x_1, \quad C \leftarrow C \vee x_2, \quad C \leftarrow C \wedge x_3, \quad C \leftarrow C \oplus x_1.$$

Il est donc utile de savoir quelles sont les fonctions booléennes qui peuvent être représentées par des chaînes, et en particulier pour les fonctions symétriques. Parmi celles-ci, les fonctions "somme" jouent un rôle essentiel, car toute fonction symétrique est combinaison de fonctions somme.

Définisons $S_k(x_1, \ldots, x_n)$ par :

$$\begin{aligned} S_k(x_1, \ldots, x_n) \;&=\; 1 \quad \text{si} \quad \sum_1^n x_i = k \\ &=\; 0 \quad \text{sinon,} \end{aligned}$$

c'est la $k$ème fonction somme.

Nous allons considérer successivement les valeurs de n.

a) Pour $n = 2$

Les seules fonctions somme sont S_1 et S_2, qui s'écrivent respectivement $x_1 \oplus x_2$ et $x_1 \wedge x_2$: toutes deux se représentent donc trivialement par des chaînes.

b) Pour $n = 3$

Il est évident que $S_3 = x_1 \wedge x_2 \wedge x_3$. Regardons maintenant S_2.

Ecrivons la table de vérité :

x_1	x_2	x_3	S_2	f_1	f_2	f_3
0	0	0	0	0	0	0
1	0	0	0	0	0	1
0	1	0	0	0	1	1
0	0	1	0	1	1	1
1	1	0	1	1	0	1
1	0	1	1	0	0	1
0	1	1	1	0	1	1
1	1	1	0	1	0	1

Ecrivons $S_2 = f_1 T x_3$, f_1 et T à déterminer. Le choix de x_3 est arbitraire : de toute façon S_2 est symétrique. Voyons d'abord quel doit être l'opérateur T.

– Si $T = \wedge$, on aurait $S_2 = 0$ lorsque $x_3 = 0$, ce qui n'est pas toujours le cas,
– Si $T = \vee$, on aurait $S_2 = 1$ lorsque $x_3 = 1$, ce qui n'est pas toujours le cas.

Donc $T = \oplus$, et $S_2 = f_1 \oplus x_3$. On obtient $S_2 \oplus x_3 = f_1 \oplus x_3 \oplus x_3 = f_1$, et donc $f_1 = S_2 \oplus x_3$ (cf table de vérité).

Ensuite, on obtient de même $f_2 = f_1 \oplus x_2$, $f_3 = f_2 \oplus x_1$, et la table de vérité de f_1 montre que $f_1 = x_1 \vee x_2 \vee x_3$. On a donc obtenu :

$$S_2 \;=\; x_1 \vee x_2 \vee x_3 \oplus x_1 \oplus x_2 \oplus x_3.$$

Pour S_1, procédant de même, on obtient comme l'a remarqué Damien Mercier :

$$S_1 \;=\; x_1 \wedge x_2 \wedge x_3 \oplus x_1 \oplus x_2 \oplus x_3.$$

c) Pour $n = 4$

Nous allons montrer que S_3 ne peut pas se mettre sous la forme d'une chaîne. Le même argument que précédemment montre que le premier opérateur ne peut être que $\oplus$, de même pour les second, troisième, quatrième.

Ecrivons la table de vérité :

x_1	x_2	x_3	x_4	S_3	f_1	f_2	f_3	f_4
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	1	1
0	0	1	0	0	0	1	1	1
0	0	0	1	0	1	1	1	1
1	1	0	0	0	0	0	1	0
1	0	1	0	0	0	1	1	0
1	0	0	1	0	1	1	1	0
0	1	1	0	0	0	1	0	0
0	1	0	1	0	1	1	0	0
0	0	1	1	0	1	0	0	0
1	1	1	0	1	1	0	1	0
1	1	0	1	1	0	0	1	0
1	0	1	1	1	0	1	1	0
0	1	1	1	1	0	1	0	0
1	1	1	1	0	1	0	0	0

où $S_3 = f_1 \oplus x_4$, $f_1 = f_2 \oplus x_3$, $f_2 = f_3 \oplus x_2$, $f_3 = f_4 \oplus x_1$.

On constate sur la table que $f_4 = S_1$, qui vaut 1 si et seulement si $\sum_1^4 x_i = 1$. Mais alors l'opérateur suivant :

– ne peut être $\vee$ (même raison que précédemment),

– ne peut être $\wedge$ (même raison que précédemment),

– ne peut être $\oplus$, car il a déjà servi avec toutes les variables : $x_1 \oplus x_1 = 0$, et ceci nous ramènerait plus haut dans la liste.

La fonction S_3 ne peut donc s'écrire comme une chaîne.

Annexe B

Enumération de l'ensemble $S(p,k,m)$

Le programme informatique ci-joint réalise cette énumération ; les entiers p, k et m sont entrés par l'utilisateur.

Le programme est écrit en C. Il fonctionne de la manière suivante :

Principe de l'algorithme.

A tout m uplet vérifiant les contraintes (à savoir $p_j \leq k$ et $\sum p_j = p$), on associe le nombre

$$N(p_1,\ldots,p_m) \;=\; p_1 + p_2(k+1) + \ldots + p_m(k+1)^{m-1}.$$

Les m uplets vont être rangés par ordre croissant du nombre associé.

- Un premier vecteur (m-uplet) est créé, dont les composantes sont les plus à gauche possible, c'est à dire pour lequel le nombre N est le plus petit possible. Ce vecteur est obtenu en prenant $p_j = k$, $j = 1,\ldots,a$ et $p_{a+1} = b$, où a et b sont respectivement le quotient et le reste de la division euclidienne de p par k.

Ensuite, le vecteur est remplacé par le vecteur immédiatement suivant. C'est l'objet de la procédure "Next", qui fonctionne en trois étapes :

- la première coordonnée non nulle du vecteur est détectée (procédure "premier"),

- la première coordonnée suivant celle-ci, et non égale à k, est détectée (toutes les coordonnées égales à k sont ignorées, car elles ne peuvent plus être augmentées). Cette coordonnée s'appelle "second" dans la procédure.

- la coordonnée "second" est augmentée d'une unité, et la coordonnée "premier" est diminuée d'une unité.

- l'ensemble des coordonnées situées strictement à gauche de "second" est renvoyé le plus à gauche possible, en respectant la somme totale (qui ne doit pas varier) : c'est la procédure "flushback".

Illustrons ceci sur un exemple : $p = 3$, $k = 2$, $m = 4$.

On commence par $(2,1,0,0)$.

Supposons qu'à un moment de l'énumération nous en soyons arrivés à $(0,1,2,0)$.
Alors "premier"=1 (les coordonnées commencent à 0) et "second"=3.
On fait successivement $(0,1,2,1)$ (accroissement de "second"), $(0,0,2,1)$ (décrément de "premier"), et $(2,0,0,1)$ (flushback de la partie $(0,0,2)$). On obtient bien ainsi le suivant de $(0,1,2,0)$, qui est $(2,0,0,1)$.

- La procédure s'arrête lorsque le vecteur est en position extrême à droite : les dernières cases sont à k (dans notre exemple, ce sera $(0,0,1,2)$).

La procédure est extrêmement rapide. A titre d'exemple, pour $p = 7$, $k = 4$, $m = 15$, l'énumération comporte 114 479 lignes ; le fichier fait un peu plus de 5 Mo, et l'ensemble de la procédure prend un peu moins de 5 secondes sur un PC **NEC Direction SP** à 233 MHz.

Chapitre V

Calcul de la somme en notation binaire

I. Rappels sur les fonctions symétriques booléennes

Une fonction symétrique booléenne (en abrégé FSB) est une fonction $f(x_1, \ldots, x_n)$, à valeurs 0 ou 1, dont la valeur ne dépend pas de l'ordre des variables. En d'autres termes, pour toute permutation σ de $\{1, \ldots, n\}$:

$$f(x_{\sigma(1)}, \ldots, x_{\sigma(n)}\ \}\ =\ f(x_1, \ldots, x_n) \tag{1}$$

Comme les variables x_i sont elles-mêmes booléennes, on obtient immédiatement le résultat suivant : une fonction booléenne est symétrique si et seulement si elle ne dépend que du *nombre* de variables qui sont à 1 (et non pas du choix particulier de ces variables). Nous avons déjà rencontré des exemples de fonctions symétriques avec les "fonctions-seuil" $T_{n,p}(x_1, \ldots, x_n)$ définies par :

$$\begin{aligned} T_{n,p}(x_1, \ldots, x_n)\ &=\ 1 \quad \text{si } \sum_1^n x_i \geq\ p \\ &=\ 0 \qquad \text{sinon} \end{aligned} \tag{2}$$

Posons $S(x_1, \ldots, x_n) = \sum_1^n x_i$, c'est la somme des variables. Evidemment, $0 \leq S \leq n$.

II. Représentation binaire de la somme $S(x_1, \ldots, x_n)$

Posons $\nu = [\log_2 n] + 1$. On peut écrire $S(x_1, \ldots, x_n)$ en base 2, sous la forme :

$$S\ =\ \varepsilon_0 + \varepsilon_1.2 + \varepsilon_2.2^2 + \cdots + \varepsilon_{\nu-1}2^{\nu-1}, \tag{3}$$

où $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$ sont elles-mêmes des fonctions booléennes symétriques, dépendant de $x_1, \ldots, x_n$.

Par exemple, si $n = 4,\ \nu = 3$,

$$S\ =\ \varepsilon_0 + \varepsilon_1.2 + \varepsilon_2.2^2,$$

et $\varepsilon_0(x_1, x_2, x_3, x_4)$ prend les valeurs suivantes :

– si $\sum_1^4 x_i = 0,\ \varepsilon_0 = 0$
– si $\sum_1^4 x_i = 1,\ \varepsilon_0 = 1$
– si $\sum_1^4 x_i = 2,\ \varepsilon_0 = 0$
– si $\sum_1^4 x_i = 3,\ \varepsilon_0 = 1$
– si $\sum_1^4 x_i = 4,\ \varepsilon_0 = 0$

et de même pour $\varepsilon_1,\ \varepsilon_2$.

Dans le calcul d'une FSB quelconque $f(x_1, \ldots, x_n)$, les $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$ vont jouer un rôle particulier : celui de *variables intermédiaires*. A partir de $x_1, \ldots, x_n$ on calcule $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$ et on les stocke, et on s'aperçoit que f ne dépend plus que de $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$. On peut dire ceci sous forme condensée : **une FSB est en fait une fonction de ν variables seulement**. On décompose donc le calcul de $f(x_1, \ldots, x_n)$ en deux étapes :

– évaluation de $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$,
– évaluation de f à partir de $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$.

Nous allons maintenant voir comment calculer $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$.

Lemme 1. – *Pour tous* $x_1, \ldots, x_n$,

$$\varepsilon_0(x_1, \ldots, x_n) \;=\; x_1 \oplus \cdots \oplus x_n \ .$$

Démonstration. Il suffit de montrer que $x_1 \oplus \cdots \oplus x_n$ vaut 0 si et seulement si $S(x_1, \ldots, x_n)$ est pair : dans la décomposition binaire de S, ε_0 vaut 0 si et seulement si S est pair.

Ceci va être fait par récurence sur n. Pour $n = 1$, il n'y a rien à démontrer. Supposons le résultat vrai pour n ; posons

$$A_n \;=\; x_1 \oplus \cdots \oplus x_n \ .$$

Alors :

$$\begin{aligned} A_n \oplus x_{n+1} \;=\; 0 \;&\Leftrightarrow\; \{A_n = 0 \text{ et } x_{n+1} = 0\} \text{ ou } \{A_n = 1 \text{ et } x_{n+1} = 1\} \\ &\Leftrightarrow\; \{\sum_1^n x_i \text{ pair et } x_{n+1} = 0\} \text{ ou } \{\sum_1^n x_i \text{ impair et } x_{n+1} = 1\} \\ &\Leftrightarrow\; \sum_1^{n+1} x_i \text{ pair} \ , \end{aligned}$$

ce qui prouve le lemme.

La détermination d'une formule explicite pour les autres variables n'est pas aussi simple. Elle est possible, grâce à la Forme Normale Disjonctive, en énumérant tous les cas possibles. Par exemple, pour ε_1, on a $\varepsilon_1 = 1$ si et seulement si $S \equiv 0$ modulo 4.

En pratique, il vaut mieux procéder récursivement en utilisant des compteurs. Nous allons montrer comment construire un compteur à ν places permettant l'écriture binaire de S.

Notons $b_0, \ldots, b_{\nu-1}$ les cases du compteur. Initialement, tous les b_i sont à 0.

On balaye successivement $x_1, \ldots, x_n$.

– Si tous les x_i sont à 0, terminé : tous les b_i restent à 0.

– Sinon, on rencontre un premier x_i à 1, noté x_{i_1}. On met b_0 à 1 et on continue à balayer les x_i, $i > i_1$. Si on atteint x_n sans avoir rencontré d'autre 1, terminé : b_0 est à 1 et les autres b_j à 0. Sinon, on rencontre x_{i_2} , $i_2 > i_1$, qui est à 1. On remet les b_0 à 0 et on met b_1 à 1.

On continue le balayage des x_i, $i > i_2$ de la même manière. Soit une situation quelconque des compteurs (les croix désignent des compteurs à 1). Soit j_1 l'indice du premier compteur à 0 en commençant par la gauche $(0 \leq j_1 \leq n)$.

Si un nouveau 1 est rencontré, tous les compteurs avec j_1 sont remis à 0 et b_{j_1} est mis à 1. Dans notre exemple, $j_1 = 2$, b_0 et b_1 seront remis à 0 et b_2 sera mis à 1.

Ceci se formalise de la façon suivante : quand un nouvel $x_i = 1$ est rencontré :

b_0 est remplacé par $b_0 \oplus 1$,
b_1 est remplacé par $(1 \wedge b_0) \oplus b_1$,
b_2 est remplacé par $(1 \wedge b_0 \wedge b_1) \oplus b_2$,
etc.

Observons au passage que, pour tout k, l'écriture de $\sum_1^k x_i$ nécessite seulement $\varphi(k) = [\log_2 k] + 1$ compteurs : seuls $b_0, \ldots, b_{\varphi(k)-1}$ sont utilisés. Les autres sont nécessairement à 0.

Nous allons maintenant déterminer le nombre d'opérations nécessaires pour implanter ces compteurs.

Lemme 2. – *L'écriture binaire* $S = \sum_1^n x_i$ *se réalise en :*

$$2n + \frac{1}{2}([\log_2 n] + 1)([\log_2 n] + 2)$$

opérations au plus.

Démonstration. Ce nombre est évidemment maximal si toutes les variables valent 1 : une variable qui vaut 0 traverse tous les compteurs sans les affecter. Par "opération", on entend ici tout calcul élémentaire $x \wedge y$, $x \vee y$, $x \oplus y$.

On suppose donc que tous les x_i sont à 1. Commençons par lire x_1. Le compteur b_0 est mis à 1 (calcul de $b_0 \oplus 1$), ce qui requiert une opération.

On lit x_2. Le compteur b_0, étant à 1, est mis à 0 (1 opération) et le compteur b_1, étant à 0, est mis à 1 (1 opération).

On lit x_3. Le compteur b_0 est remis à 1, les autres sont inchangés.
Plus généralement, le nombre d'opérations est calculé comme suit :

– Pour x_1, x_3, x_5 ,..., (tous les x_i d'indice impair), la lecture se fait en une seule opération (b_0, qui était à 0, est mis à 1),

– le nombre de x_i d'indice impair est $n - [n/2]$.

– Pour x_2, x_6, x_{10} ,..., (ceux dont l'indice est divisible par 2, pas par 4), la lecture se fait en deux opérations, portant sur b_0 et b_1. Le nombre de ces x_i est $[n/2] - [n/2^2]$.

– Plus généralement, pour les x_i dont l'indice est divisible par 2^j mais non par 2^{j+1}, la lecture se fait en $j+1$ opérations, portant sur les compteurs $b_0, \ldots, b_j$. Le nombre de ces x_i est

$$[n/2^j] - [n/2^{j+1}].$$

Le nombre total d'opérations est donc

$$N_{op} \;=\; \sum_{j=0}^{[\log_2 n]} (j+1)\left([n/2^j] - [n/2^{j+1}]\right) \tag{4}$$

Or, si $n/2^{j+1} = a_j + b_j$, a_j entier et $b_j < 1$, $n/2^j = 2a_j + 2b_j$, et

$$[n/2^j] - [n/2^{j+1}] \;=\; 2a_j + 2b_j - a_j \;=\; a_j + 2b_j \;=\; b_j + a_j + b_j,$$

et donc

$$N_{op} \;=\; \sum_{j=0}^{[\log_2 n]} (j+1)\left(n/2^{j+1} - [n/2^{j+1}]\right) + \sum_{j=0}^{[\log_2 n]} (j+1)n/2^{j+1}.$$

Commençons par évaluer la seconde somme notée N''. Avec $\nu = [\log_2 n] + 1$, elle vaut :

$$N'' \;=\; \sum_{j=1}^{\nu} jn/2^j \;=\; n\sum_{1}^{\nu} j/2^j.$$

Or

$$\sum_{j=0}^{\infty} \frac{j}{x^{j+1}} \;=\; \frac{1}{(x-1)^2},$$

et, avec $x = 2$

$$\sum_{j=0}^{\infty} j/2^{j+1} \;=\; 1, \quad \sum_{j=0}^{\infty} j/2^{j} = 2,$$

et

$$N'' \;\leq 2n.$$

Pour N', on utilise simplement le fait que $a - [a] \leq 1$, et donc

$$N' \;\leq \sum_{j=0}^{[\log_2 n]} j+1 \;=\; \sum_{j=1}^{\nu} j \;=\; \frac{\nu(\nu+1)}{2}.$$

Finalement on obtient l'estimation :

$$N_{op} \;\leq\; 2n + \frac{1}{2}([\log_2 n]+1)([\log_2 n]+2) \tag{5}$$

comme annoncé.

III. Calcul d'une FSB lorsque la mémoire est suffisante

Voyons maintenant combien de cases mémoire sont nécessaires pour implanter ces opérations.

Pour déterminer $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$, il faut ν cases mémoire pour les compteurs, plus une pour stocker les retenues (propagation d'un compteur à l'autre). Il faut $2n + \frac{1}{2}\nu(\nu+1)$ opérations.

Une fois que les variables $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$, sont évaluées, reste à calculer f. La façon de le faire dépend de la forme explicite de f, mais *au pire*, cela peut être réalisé grâce à la FND portant sur $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$. Il faut pour cela 2 cases mémoire supplémentaires et $\nu.2^{\nu}$ opérations.

Au total, nous voyons que f, fonction symétrique booléenne quelconque, peut être calculée avec $\nu + 3$ cases mémoire et un nombre d'opérations au plus égal à :

$$N \;=\; 2n + \frac{1}{2}\nu(\nu+1) + 2\nu n, \tag{6}$$

c'est à dire $0(2n \log_2 n)$. Remarquons que c'est la partie FND qui requiert le plus d'opérations : sans elle, il suffirait de $2n$ opérations. On peut bien sûr avoir recours à la méthode AVB (arrangement de variables par blocs), introduite dans nos rapports précédents : elle requiert moins d'opérations, mais une case mémoire de plus.

Cette implantation est infiniment plus satisfaisante que celle prévue par la théorie pour une fonction booléenne quelconque, pour laquelle le nombre d'opérations est en général *exponentiel* en n.

Exemple numérique : la fonction seuil $T_{n,p}$, avec $n = 25$ et $p = 12$ requiert 721 opérations et 7 cases mémoire. Ceci est à comparer aux 38000 opérations, dans le cas de trois cases mémoire (notre rapport no 3). Dans le cas présent, les opérations se décomposent comme suit :

– calcul des variables intermédiaires $\varepsilon_0, \ldots, \varepsilon_{\nu-1}$,

– evaluation de la fonction T

Ceci requiert évidemment une place mémoire importante, qui n'est pas disponible sur la rétine. Nous allons maintenant voir comment y remédier.

IV. Calcul d'une FSB lorsque la mémoire est insuffisante : "clusterisation"

L'idée est simple : lorsque le nombre de plans-mémoire est insuffisant ($< \nu + 3$), on regroupe ensemble les "colonnes-mémoire" associées à plusieurs pixels de l'image. Cette opération de regroupement de la mémoire s'appelle "clusterisation". Naturellement, les pixels ainsi regroupés ne peuvent plus travailler indépendamment : la clusterisation se fait aux dépens du parallélisme. Il faut donc choisir avec attention le "facteur de clusterisation", pour augmenter la mémoire sans trop nuire au parallélisme. Ici, nous associerons les pixels par blocs de ν ($\nu = [\log_2 n] + 1$), deux blocs distincts travaillent en parallèle.

Au lieu d'avoir n opérations en parallèle, nous avons donc n/ν opérations en parallèle : la perte n'est pas trop grande.

Nous allons distinguer, dans ce qui suit, le cas de la dimension 1 et celui de la dimension 2. Nous verrons que, dans les deux cas, trois plans mémoire suffisent.

A. Curseur linéaire de taille n

1. Cas général : fonction booléenne symétrique quelconque

Dans ce premier cas, notre curseur est une simple règle avec n cases. Soit $\nu = [\log_2 n] + 1)$ comme, précédemment.

Le curseur étant de taille n, n variables sont lues à chaque instant. On note S la somme $\sum_1^n x_i$.

Pour une position donnée du curseur (la mire étant considérée à l'extrémité gauche), on va utiliser ν colonnes de mémoire. Comme on a 3 plans mémoire, cela fait 3ν cases mémoire disponibles. L'image étant notée C_0, les trois lignes mémoire sont notées C_1, C_2, C_3. La figure ci-dessous illustre le cas où $n = 25$, $\nu = 5$.

La dernière rangée, C_3, va être utilisée pour stocker les résultats. Ainsi, $C_{3,1} = 1$ signifiera

$$f(x_1, \ldots, x_n) = 1$$

$C_{3,2} = 1$ signifiera $f(x_2, \ldots, x_{n+1}) = 1$, etc (décalage du curseur d'un cran).

Les cases $C_{3,1}, \ldots, C_{3,\nu}$ vont être calculées *successivement* (nous dirons comment dans un instant), mais les cases $C_{3,1}$, $C_{3,\nu+1}$, $C_{3,\nu+2}, \ldots$, sont calculées *simultanément*, de même que $C_{3,2}$, $C_{3,\nu+2}$, $C_{3,\nu+3}, \ldots$, etc. Le calcul est *séquentiel* à l'intérieur d'un bloc, parallèle d'un bloc à l'autre.

La première rangée, C_1, est utilisée, à l'intérieur de chaque bloc, pour stocker la somme S en binaire. Précisément, les ν cases $C_{1,1}, \ldots, C_{1,\nu}$ servent à coder la somme $\sum_1^n x_i$: il y a exactemment la place qu'il faut pour cela. La case $C_{2,1}$ est d'abord utilisée pour stocker les retenues intermédiaires. Lorsque les cases $C_{1,1}, \ldots, C_{1,\nu}$ sont remplies, la case $C_{2,1}$ est libérée.

De même, $C_{1,\nu+1}, \ldots, C_{1,2\nu}$ code $\sum_{\nu+1}^{n+\nu} x_i$, etc.

Le codage de S se fait en N_{op} opérations, défini en (5) du paragraphe précédent.

Ceci fait, l'estimation de f à partir de S se fait par une FND qui nécessite deux cases mémoire : une pour le calcul des monômes $\varepsilon_{i_1} \wedge \varepsilon_{i_2} \wedge, \ldots$, et une pour le résultat ("ou" des monômes précédents). Le résultat, nous l'avons dit, est mis en $C_{3,1}$. La case de calcul intermédiaire est $C_{2,1}$, qui sert maintenant à cela.

Une fois ceci fait, le curseur est décalé d'un cran : il lit $x_2, \ldots, x_{n+1}$. La somme S est laissée au même endroit, mais elle est diminuée de x_1 et augmentée de x_{n+1}. Ceci est fait de la manière suivante :

– Si $x_1 = x_{n+1}$, S n'est pas modifié.

– Si $x_1 = 0$ et $x_{n+2} = 1$, S est incrémenté d'une unité (on cherche le premier compteur à 0, on le met à 1 et tous les précédents sont mis à 0). Ceci demande ν opérations au plus (puisqu'il y a ν compteurs).

– Si $x_1 = 1$ et $x_{n+2} = 0$, S est diminué d'une unité (on cherche le premier compteur à 1, on le met à 0 et on met tous les précédents à 1). Ceci demande aussi ν opérations au plus.

On a alors, sur les cases $C_{1,1}, \ldots, C_{1,\nu}$, la somme $S(x_2, \ldots, x_{n+1})$.

On calcule $f(x_2, \ldots, x_{n+1})$ comme expliqué précédemment, grâce à la case $C_{2,2}$, et le résultat est mis en $C_{3,2}$. On décale encore le curseur d'un cran : $x_3, \ldots, x_{n+2}$, etc, jusqu'à $x_{\nu-1}, \ldots, x_{\nu+n}$. Rappelons que le paquet $x_\nu, \ldots, x_{\nu+n-1}$, quant à lui, à été traité simultanément.

Nous allons calculer combien d'opérations ont été effectuées séquentiellement :

– pour calculer S : $N_{op} \leq 2n + \frac{1}{2}(\nu+1)(\nu+2)$.

– pour calculer $f(x_1, \ldots, x_n)$: $\nu 2^\nu$.

– pour calculer $S(x_2, \ldots, x_n) = \nu$.

– pour calculer $f(x_2, \ldots, x_n) = \nu.2^\nu$,
etc.

Il y a $\nu - 1$ décalages, chacun réclamant ν opérations pour recalculer S et $\nu.2^\nu$ opérations pour la FND. Au total, nous avons donc

$$2n + \frac{1}{2}(\nu+1)(\nu+2) + \nu(\nu + \nu 2^\nu) \;=\; 2n + \frac{1}{2}(\nu+1)(\nu+2) + \nu^2(2^\nu + 1)$$

opérations, et ceci est de l'ordre de $2n(\log_2 n)^2$. On a perdu un facteur $\log_2 n$ par rapport au cas où la mémoire était suffisante, et maintenant on n'utilise plus que trois plans mémoire.

Remarque. Il y a dans cette étape une perte d'efficacité du fait qu'un seul processeur travaille (il utilise la mémoire de tous les autres). On pourrait songer à paralléliser davantage : découper S en tronçons calculés simultanément, puis regroupés par paquets ; calculs simultanés de $S(x_2, \ldots, x_{n+1})$, de $S(x_3, \ldots, x_{n+2})$, etc. Mais tout ceci n'est possible que si l'on dispose de davantage de plans mémoire. Avec 3, on est astreint à attendre que le processeur P ait fini le calcul de $f(x_1, \ldots, x_n)$ pour transformer $S(x_1, \ldots, x_n)$ en $S(x_2, \ldots, x_{n+1})$ et attaquer le calcul de $f(x_2, \ldots, x_{n+1})$. Si par exemple on disposait de 3 cases mémoires supplémentaires, on pourrait calculer les S trois par trois, regagnant ainsi en parallélisme.

2. Cas d'une fonction seuil

Le cas d'une fonction seuil est notablement plus simple que le cas général : on n'a pas besoin de FND. On code l'entier p sur la seconde rangée et on met $C_{3,1}$ à 1 si $S(x_1, \ldots, x_n) \geq p$. La comparaison de deux entiers écrits en binaire se fait bit à bit (en commençant par la droite), et requiert au plus ν opérations.

Pour une fonction seuil, le nombre d'opérations effectuées séquentiellement est donc :

– pour calculer S : $N_{op} \leq 2n + \frac{1}{2}(\nu+1)(\nu+2)$
– pour comparer S à p : ν
– pour calculer $S(x_2, \ldots, x_{n+1})$: ν
– pour comparer $S(x_2, \ldots, x_{n+1})$ à p : ν, etc.
Au total $2n + \frac{1}{2}(\nu+1)(\nu+2) + 2\nu(\nu+1)$ opérations, soit $0(n)$.

B. Cas du curseur bi-dimensionnel

Le curseur est maintenant un carré $n \times n$ dans un plan. Le principe est très voisin de celui vu précédemment, et trois plans mémoire suffisent encore.

Notant, comme précédemment $\varphi(k) = [\log_2 k] + 1$, on réserve, en haut et à gauche du curseur, un carré C de côté $m = [\sqrt{\varphi(n^2)}] + 1$, si bien que $m^2 \geq \varphi(n^2)$. On peut donc stocker sur la première couche C_1, au-dessous du carré C, la somme $S = \sum_{i,j=1}^{n} x_{ij}$.

Soit f une fonction symétrique booléenne à calculer. Elle dépend en fait des $\nu' = \varphi(n^2)$ variables utilisées pour la représentation de S en binaire. On la calcule par une FND, demandant 2 cases mémoire supplémentaires et $\nu'.2^{\nu'}$ opérations. Le résultat est stocké sur la première case en haut à gauche de C_3, notée $C_3(1,1)$.

Le calcul intermédiaire est fait sur $C_2(1,1)$, qui a d'abord servi aux retenues lors de l'écriture sur la couche C_1.

Une fois que $f((x_{ij})_{i,j=1,\ldots,n})$ a été calculé, on décale le curseur d'un cran vers la droite. La somme S est remplacée par

$$\sum_{i=1}^{n} \sum_{j=2}^{n+1} x_{i,j}$$

qui vaut

$$S - \sum_{i=1}^{n} x_{i,1} + \sum_{i=1}^{n} x_{i,n+1} \ .$$

On voit que $2n$ variables sont lues lors du calcul de ce nouvel S, qui est stocké à la place du précédent, sur la couche C_1 au-dessous de C. Chaque variable requiert ν' opérations. Ensuite on calcule $f((x_{ij})_{\substack{i=1,\ldots,n \\ j=2,\ldots,n+1}})$. On décale d'un cran vers la droite.

Une fois qu'on a réalisé $m-1$ décalages à droite, c'est à dire lorsque le curseur est parvenu à la limite du carré C initial, on décale d'un cran vers le bas, et on repart à gauche. Voici la description des déplacements.

Il y a en tout $m^2 - 1$ déplacements, chacun d'une case (la case en haut à gauche du curseur se déplace en occupant successivement toutes les cases du carré C, au nombre de m^2).

Chaque situation nouvelle exige :

– le calcul de S, soit $2n$ opérations,
– une FND, soit $\nu' \ 2^{\nu'}$ opérations.

Comme au départ nous avions, pour le premier calcul de S, un nombre d'opérations au plus égal à $2n^2 + \frac{1}{2}(\nu'+1)(\nu'+2)$, cela nous donne au total :

$$N_{op} \ = \ 2n^2 + \frac{1}{2}(\nu'+1)(\nu'+2) + (m^2-1)(2n\nu' + \nu'.2^{\nu'})$$

avec $m = [\sqrt{[\log_2 n^2] + 1}] + 1$, $\nu' = [\log_2 n^2] + 1$. On obtient l'estimation :

$$N_{op} \sim 4n^2(\log_2 n)^2 + 2n^2 + 8n(\log_2 n)^2 + 2(\log_2 n)^2$$

qui est identique à celle du cas mono-dimensionnel : si N est le nombre de cases du curseur, $N = n^2$, $N_{op} \sim N(\log_2 N)^2$.

Exemple numérique

Reprenons le cas de la fonction seuil $T_{n,p}$, avec $n = 25$ et $p = 12$. Le curseur est un carré 5×5. Par clusterisation, on n'utilise que trois plans mémoire et le nombre total d'opérations est 3600.

Chapitre VI

Méthode Calcul et Comparaison

I. Introduction : Techniques générales

Dans les précédents rapports, nous avons présenté des décompositions des fonctions seuils et symétriques adaptées à un nombre restreint de plans mémoires. Leurs performances respectives (en nombre d'o.e.) étaient meilleures que celles de la FND.

Nous nous plaçons maintenant dans un cadre où le nombre de plans mémoire (obtenu par clustérisation ou disponible) est légérement supérieur au nombre nécessaire pour stocker, en binaire, la somme des variables des fonctions symétriques que nous envisageons d'implanter sur la rétine. Après l'obtention de la somme des variables, les fonctions symétriques et des fonctions seuils se déduisent alors simplement à partir de la valeur de cette somme.

Dans ce rapport, nous présentons, dans la première partie, les choix possibles pour l'implantation optimale dans une rétine, des opérations nécessaires au codage des fonctions symétriques. Il s'agit du calcul d'une somme de n variables et de la comparaison entre deux nombre. Dans la deuxième partie, nous décrivons l'obtention des fonctions symétriques à partir du calcul de la somme des variables et nous donnons son coût en place mémoire et en o.e.

II. Construction de la somme de n variables

Il est intéressant d'obtenir dès à présent une description du calcul de la somme de n variables, car la valeur d'une fonction symétrique ne dépend que de la valeur de la somme de ces variables.

Pour n variables notées $(x_1, x_2, \cdots, x_n)$, on veut déterminer la somme $\sum_{i=1}^{n} x_i$ en utilisant, dans une rétine, le moins possible de plans mémoires et avec le moins possible d'opérations élémentaires. De plus, il faut nécessairement que l'algorithme décrivant l'obtention de la somme soit programmable sur une rétine.

Nous décrivons maintenant les résultats obtenus : le nombre de plans nécessaires, le nombre d'opérations élémentaires (o.e.), l'algorithme composé d'o.e. disponibles sur la rétine, qui nous semblent les plus performants.

II.1 Plans mémoire nécessaires

On détermine *a priori* le nombre minimum de plans mémoire nécessaires au stockage du résultat : c.-à-d. au stockage de la somme. A ce nombre, il faut ensuite ajouter le ou les plans mémoire utilisés pour le calcul effectif de cette somme.

On rappelle qu'il faut exactement k plans mémoire pour stocker, en binaire, tous les nombres compris entre 2^{k-1} et $2^k - 1$. En effet, on découpe le nombre à stocker en une somme de puissance croissante de 2 et on stocke sur le ième plan $(0 \leq i \leq k-1)$ la valeur (0 ou 1) correspondante, dans la décomposition, à la puissance 2^i. De cette relation, on en déduit le lemme suivant :

Lemme 1. – *Il est nécessaire et suffisant d'avoir* $k = \lceil \log_2(n+1) \rceil = [\log_2(n)] + 1$ *plans mémoire pour stocker, en binaire, la somme de* n *variables.*

Démonstration du Lemme 1. – La somme de n variables atteint au plus la valeur n. On détermine k vérifiant l'encadrement suivant :

$$2^{k-1} \leq n \leq 2^k - 1$$

Il est donc nécessaire et suffisant d'avoir k plans mémoire pour stocker n. De plus, on en déduit la relation :

$$2^{k-1} + 1 \leq n+1 \leq 2^k$$

d'où :

$$\log_2(2^{k-1}) < \log_2(2^{k-1}+1) \leq \log_2(n+1) \leq \log_2(2^k)$$
$$\Leftrightarrow (k-1) < \log_2(n+1) \leq k$$
$$\Leftrightarrow \lceil \log_2(n+1) \rceil = k$$

De même, il est facile de montrer que :

$$(k-1) \leq \log_2(n) + 1 \leq \log_2(2^k - 1) < k$$

d'où :

$$\Leftrightarrow [\log_2(n)] + 1 = k$$

A cette estimation, il faut ajouter le ou les plans utilisés pour le calcul effectif de la somme (propagation de retenue).

II.2 Description de l'algorithme

Dans une rétine, les n variables intervenant dans la somme correspondent à des pixels de l'image et ces pixels sont *a priori* répartis dans tout le plan (généralement, ils appartiennent à un voisinage géométrique). Pour cette raison, il n'est pas possible d'accéder simultanément à plusieurs des variables. Ceci nous oblige à envisager le calcul de la somme de n variables comme le résultat final des calculs successifs des sommes partielles $\sum_{i=1}^{l} x_i$ où chaque somme partielle est obtenue par l'ajout de la variable x_l, à la somme partielle $\sum_{i=1}^{l-1} x_i$ déjà calculée.

L'algorithme décrivant ce processus est alors le suivant :

- Pour $l = 1$ à n faire

  Accès à la variable x_l

  Somme partielle (l) = Somme partielle $(l-1) + x_l$

- Fin

où Somme partielle (j) $= \sum_{i=1}^{j} x_i$ pour $1 \leq j \leq n$.

Il reste à décrire le passage de la somme partielle de rang $(l-1)$ à celle de rang l lors de l'ajout de la variable x_l et cette modification doit être indépendante de l.

Au cours du calcul, l'ajout d'une variable x_l peut modifier l'ensemble des plans mémoire par la propagation d'une retenue. Il est donc nécessaire de modifier l'ensemble des plans mémoire à chaque prise en compte d'une nouvelle variable.

On note $b_0, b_1, \cdots, b_{k-1}$ avec $k = [\log_2(n)] + 1$ les plans mémoire permettant le stockage de la somme des n variables. Dans le plan b_i $(0 \leq i \leq k-1)$, on stocke la valeur (0 ou 1) correspondante, dans la décomposition de la somme, à la puissance 2^i. Lors de l'ajout d'une variable, il est évident que le plan le plus souvent modifié est b_0 (le bit de poids faible) et le plus rarement modifié est b_{k-1} (le bit de poids fort).

Si, à un instant donné, $b_0, b_1, \cdots, b_{k-1}$ représentent la somme partielle $\sum_{i=1}^{l-1} x_i$ et x la variable x_l, le calcul de la somme $\sum_{i=1}^{l} x_i$ (représentée par $b'_0, b'_1, \cdots, b'_{k-1}$) est donné par la succession des o.e. suivantes :

$$
\begin{aligned}
b'_0 &= b_0 \oplus x \\
b'_1 &= b_1 \oplus (b_0 \wedge x) \\
b'_2 &= b_2 \oplus (b_1 \wedge (b_0 \wedge x)) \\
&\cdot \\
&\cdot \\
b'_{i-1} &= b_{i-1} \oplus (b_{i-2} \wedge (b_{i-3} \cdots \wedge (b_0 \wedge x)) \\
b'_i &= b_i \oplus (b_{i-1} \wedge (b_{i-2} \cdots \wedge (b_0 \wedge x)) \\
&\cdot \\
&\cdot \\
b'_{k-1} &= b_{k-1} \oplus (b_{k-2} \wedge (b_{k-3} \cdots \wedge (b_0 \wedge x))
\end{aligned}
$$

où $\oplus$ est le X.O.R. et $\wedge$ est le ET de deux plans mémoire.

Pour l'implantation dans une rétine, avec un minimum de plans mémoire, de la prise en compte d'une nouvelle variable, il est judicieux de modifier l'ordre de calcul de certains o.e. Dans ce cadre, on rappelle l'égalité suivante :

$$d \wedge e = \overline{(d \oplus e)} \wedge e$$

qui s'applique plus généralement à :

$$
\begin{aligned}
& b_{i-1} \wedge (b_{i-2} \wedge (b_{i-3} \cdots \wedge (b_0 \wedge x)) \\
&\quad = \overline{b_{i-1} \oplus (b_{i-2} \wedge (b_{i-3} \cdots \wedge (b_0 \wedge x))} \wedge (b_{i-2} \wedge (b_{i-3} \cdots \wedge (b_0 \wedge x)). \\
&\quad = \overline{b'_{i-1}} \wedge (b_{i-2} \wedge (b_{i-3} \cdots \wedge (b_0 \wedge x)) \\
&\qquad\qquad \forall\ 1 \leq i \leq k-1.
\end{aligned}
$$

Finalement, on peut écrire :

$$
\begin{aligned}
b'_i &= b_i \oplus (b_{i-1} \wedge (b_{i-2} \cdots \wedge (b_0 \wedge x)) \\
&= b_i \oplus [\, \overline{b'_{i-1}} \wedge (b_{i-2} \wedge (b_{i-3} \cdots \wedge (b_0 \wedge x))\,] \\
&\qquad\qquad \forall\ 1 \leq i \leq k-1.
\end{aligned}
$$

De cette égalité, on déduit que, pour obtenir la nouvelle valeur de b'_i, il suffit de connaître les valeurs de b_i et de b'_{i-1} et de l'expression $(b_{i-2} \wedge (b_{i-3} \cdots \wedge (b_0 \wedge x))$ qui vient de servir au calcul de b'_{i-1}.

La prise en compte d'un nouvelle variable dans la somme partielle peut être donc décrite, dans une rétine, par un enchaînement d'o.e. nécessitant un seul plan mémoire supplémentaire noté C. Il suffit pour cela de

modifier les plans mémoire $b_0, b_1, \cdots, b_{k-1}$ dans l'ordre d'indice croissant de la façon suivante :

Déroulement des instructions	C	$b_1, b_2, \cdots, b_{k-1}$
Modification de b_0	$C \leftarrow b_0 \wedge x$	$b_0 \leftarrow b_0 \oplus x$
Modification de b_1		$b_1 \leftarrow b_1 \oplus C$
Modification de b_2	$C \leftarrow C \wedge \overline{b_1}$	$b_2 \leftarrow b_2 \oplus C$
. .	. .	. .
Modification de b_i	$C \leftarrow C \wedge \overline{b_{i-1}}$	$b_i \leftarrow b_i \oplus C$
. .	. .	. .
Modification de b_{k-1}	$C \leftarrow C \wedge \overline{b_{k-2}}$	$b_{k-1} \leftarrow b_{k-1} \oplus C$

Cette dernière partie termine la description de l'algorithme et on peut maintenant évaluer son coût en nombre de plans mémoire et en nombre d'opérations élémentaires.

II.3 Coût

Le nombre de plans mémoires utilisés par cet algorithme est le nombre de plans nécessaires pour stocker la somme plus un pour les calculs, soit :

$$N_{plans} \;=\; k+1 \;=\; [\; log_2(n)\;] + 2.$$

Pour obtenir le nombre d'opérations élémentaires (o.e.), on détermine d'abord le nombre d'o.e. nécessaires pour l'ajout d'une variable à la somme : il y a k opérations $\oplus$ et $k-1$ opérations $\wedge$, soit $2k-1$ o.e. par variable. Pour la somme de n variables, il faut au total :

$$N_{o.e.}(n) \;=\; n(2k-1) \;=\; n(2\;[\;\log_2(n)\;]+1).$$

Cependant, on peut diminuer le coût du calcul de la somme de n variables en ne faisant, lors de l'ajout d'une variable, que les opérations de propagation de la retenue jusqu'au plan mémoire adéquat. En effet, la somme partielle des l premières variables ne peut pas dépasser l et donc l'ajout de la l-ième variable ne peut modifier qu'au plus le plan b_j avec $j \;=\; [\; log_2(l)\;]+1$. Il est donc seulement nécessaire pour passer de la somme partielle $\sum_{i=1}^{l-1} x_i$ à la somme $\sum_{i=1}^{l} x_i$ de modifier tous les plans mémoire de b_0 à b_j avec $j \;=\; [\; log_2(l)\;]+1$. Pour cet ajout, il faut donc j opérations $\oplus$ et $j-1$ opérations $\wedge$, soit $2j-1$ opérations

élémentaires (au lieu de $2k-1$ o.e.). Pour l'ensemble de la somme, le nombre minimum d'opérations est :

$$\begin{aligned} N_{min\ o.e.}(n) &= \sum_{i=1}^{n} 2j(i) - 1 \;=\; \sum_{i=1}^{n} 2([\ \log_2(i)\ ] + 1) - 1 \\ &= n \;+\; 2\sum_{i=1}^{n} [\ \log_2(i)\ ] \\ &= n \;+\; 2 \sum_{i=1}^{[\ \log_2(n)\ ]-1} 2^i\ i \\ &\qquad\qquad + \;2\big(n - 2^{[\ \log_2(n)\ ]} + 1\big)\ ([\ \log_2(n)\ ]) \\ &= n \;+\; 2\big(2^{[\ \log_2(n)\ ]}\ ([\ \log_2(n)\ ] - 2) + 2\big) \\ &\qquad\qquad + \;2\big(n - 2^{[\ \log_2(n)\ ]} + 1\big)\ ([\ \log_2(n)\ ]) \\ &= n \;+\; 2\ ([\ \log_2(n)\ ])\ (n+1)\ - 2^{[\ \log_2(n)\ ]+2} + 4 \end{aligned}$$

On présente les coûts en o.e. et en plans mémoire pour des sommes allant de 3 à 36 :

n	3	5	9	16	25	36
$N_{plans}(n)$	3	4	5	6	6	7
$N_{o.e.}(n)$	9	25	63	144	225	396
$N_{min\ o.e.}(n)$	7	17	41	92	173	282

La limitation de la propagation de la retenue entraîne une diminution de 20 à 30 % du nombre des o.e. pour le calcul de la somme. De ce fait, c'est cette méthode qui sera employée dans les paragraphes suivants.

Pour déterminer la valeur des fonctions seuils ou symétriques, il est nécessaire de comparer cette somme avec des seuils fixés. L'obtention du résultat de ces comparaisons est proposé dans les paragraphes suivants.

III Comparaison supérieure

La valeur d'une fonction seuil est définie par la comparaison de la somme des variables avec le seuil. Après la description de l'obtention de la somme, on décrit maintenant la comparaison de la somme S avec le seuil p ; plus exactement on veut savoir si la somme S est supérieure ou égale au seuil préalablement fixé p.

Pour comparer la somme S avec le seuil p, on utilise le lemme suivant :

Lemme 2. – *Soit p un seuil inférieur à $2^k - 1$ et S une somme comprise entre 2^{k-1} et $2^k - 1$. Si $S - p + 2^k \geq 2^k$, alors $S \geq p$: c'est-à-dire le seuil est atteint.*

Démonstration du Lemme 2. – La condition $S - p + 2^k \geq 2^k$ est équivalente à $S - p \geq 0$: c.-à-d. $S \geq p$.

On va maintenant présenter un algorithme utilisant cette propriété.

On suppose que la somme S a été préalablement calculée et qu'elle est maintenant représentée par sa décomposition binaire sur les plans notés $b_0, b_1, \cdots, b_{k-1}$. Connaissant le seuil p, on code en binaire le

nombre $\widetilde{p} = 2^k - p$ (représenté par les valeurs binaires $\widetilde{p_0}, \widetilde{p_1}, \cdots, \widetilde{p_{k-1}}$) et on effectue l'opération de sommation entre S et $\widetilde{p}$ grâce à l'algorithme suivant :

$$\begin{aligned}
b'_0 &= b_0 \wedge \widetilde{p_0} \\
b'_1 &= (b'_0 \wedge (b_1 \oplus \widetilde{p_1})) \vee (b_1 \wedge \widetilde{p_1}) \\
b'_2 &= (b'_1 \wedge (b_2 \oplus \widetilde{p_2})) \vee (b_2 \wedge \widetilde{p_2}) \\
&\cdot \\
b'_i &= (b'_{i-1} \wedge (b_i \oplus \widetilde{p_i})) \vee (b_i \wedge \widetilde{p_i}) \\
&\cdot \\
b'_{k-1} &= (b'_{k-2} \wedge (b_{k-1} \oplus \widetilde{p_{k-1}})) \vee (b_{k-1} \wedge \widetilde{p_{k-1}})
\end{aligned}$$

En fait, cet algorithme ne détermine pas la valeur de la somme ; mais seulement la propagation de la retenue. Les plans mémoires de $b'_0, b'_1, \cdots, b'_{k-1}$ contiennent successivement les valeurs des retenues définissant le passage d'une puissance de 2 à la suivante. La valeur du dernier plan mémoire b'_{k-1} nous donne donc le résultat de la comparaison :

- si $b'_{k-1} = 0$, le résultat de la sommation n'excède pas 2^k et le seuil n'est pas atteint ;

- si $b'_{k-1} = 1$, le résultat de la sommation dépasse 2^k et le seuil est atteint.

L'inplantation de cet algorithme se fait sur une rétine de façon un peu différente. Les valeurs binaires de $\widetilde{p}$ ne sont pas à proprement parler des variables : ce sont des données de la comparaison. Elles peuvent être prise en compte directement dans le code "pilotant" la rétine. Comme elles n'interviennent dans le calcul des retenues que par les opérations de OU EXCLUSIF, il est possible de transformer ces opérations par l'opération de complémentation que l'on définit par :

$$x/y = \begin{cases} x & \text{si } y = 0 \\ \overline{x} & \text{si } y = 1 \end{cases}$$

et qui vérifie :

$$x/y = x \oplus y.$$

En complémentant la variable b_i par $\widetilde{p_i}$, on transforme alors l'égalité :

$$b'_i = (b'_{i-1} \wedge (b_i \oplus \widetilde{p_i})) \vee (b_i \wedge \widetilde{p_i}). \tag{1}$$

Dans le cas $p_i = 1$, l'égalité (1) devient :

$$\begin{aligned}
b'_i &= (b'_{i-1} \wedge \overline{b_i}) \vee (b_i \wedge 1) \\
&= (b'_{i-1} \wedge \overline{b_i}) \vee b_i \\
&= b'_{i-1} \vee b_i.
\end{aligned}$$

Dans le cas $p_i = 0$, l'égalité (1) devient :

$$\begin{aligned}
b'_i &= (b'_{i-1} \wedge b_i) \vee (b_i \wedge 0) \\
&= (b'_{i-1} \wedge b_i) \vee 0 \\
&= b'_{i-1} \wedge b_i.
\end{aligned}$$

Le calcul de la retenue se résume donc, selon la valeur de p_i, à l'une ou l'autre de ces opérations élémentaires. Le calcul de la comparaison est alors un enchaînement (qui dépend de $\widetilde{p}$) d'opérations élémentaires effectuées dans l'ordre croissant des indices des plans mémoire. Son coût en opérations élémentaires est de $k = [\log_2(n)] + 1$o.e. et ne nécessite pas de plans mémoire supplémentaires.

IV Comparaison égalité

La valeur d'une fonction symétrique peut être définie par la comparaison de la somme de ses variables et d'un ou plusieurs seuils. Après la description, au paragraphe précédent, de la comparaison supérieure ou égale, on décrit la comparaison égalité entre la somme S et un seuil p ; plus exactement on veut savoir si la somme S est égale au seuil p préalablement fixé.

Pour cela, on suppose que la somme S est connue par sa décomposition binaire sur les plans notés

$$b_0, b_1, \cdots, b_{k-1}$$

et le seuil p par ses valeurs notées $p_0, p_1, \cdots, p_{k-1}$.

Pour avoir l'égalité entre les deux nombres, il faut et il suffit que l'ensemble de leurs valeurs binaires soit identique et donc que R défini par :

$$\begin{aligned} R \;=\; \overline{(b_0 \;\oplus\; p_0)} \wedge \overline{(b_1 \;\oplus\; p_1)} \;\wedge\; \overline{(b_2 \;\oplus\; p_2)} \;\wedge\; \cdots \\ \wedge\; \overline{(b_i \;\oplus\; p_i)} \;\wedge\; \cdots \;\wedge\; \overline{(b_{k-1} \;\oplus\; p_{k-1})} \end{aligned}$$

soit égal à 1. Le calcul de R de cette manière est trop coûteux.

Pour l'implantation sur une rétine, les valeurs binaires de p étant des données de la comparaison, on remplace les opérations $\oplus$ par la complémentation par rapport au valeurs de p_i :

$$\overline{b_i \;\oplus p_i} \;=\; \overline{b_i / p_i} \;=\; \begin{cases} \overline{b_i} & \text{si} \;\; p_i = 0 \\ b_i & \text{si} \;\; p_i = 1 \end{cases}$$

d'où :

$$R \;=\; \widetilde{b_0} \wedge \widetilde{b_1} \;\wedge\; \widetilde{b_2} \;\wedge\; \cdots \;\wedge\; \widetilde{b_{k-1}}$$

avec

$$\widetilde{b_i} \;=\; \overline{b_i / p_i}.$$

Le calcul de la comparaison est alors un enchaînement (qui dépend de $\widetilde{p}$) d'opérations élémentaires. Son coût en opérations élémentaires est de $k-1 = [\log_2(n)]$ o.e. et ne nécessite pas de plans mémoire supplémentaires.

Il est maintenant possible de décrire les fonctions seuils et symétriques à partir de ces opérations.

V Ecriture des fonctions seuils

La valeur d'une fonction seuil $T_{n,p}$ dépend uniquement de la comparaison de la somme de ses variables avec le seuil p. Sa définition est donné par :

$$T_{n,p}\,(x_1, x_2, \ldots, x_n) \;=\; \begin{cases} 1 \;\; \text{si} \;\; \displaystyle\sum_{i=1}^{n} x_i \;\geq\; p \\ 0 \;\; \text{sinon} \end{cases}$$

Pour obtenir $T_{n,p}$, on peut calculer explicitement la somme de ses variables et la comparer au seuil p. Les résultats des paragraphes précédents nous permettent d'obtenir le coût en o.e. et en plans mémoire nécessaires à l'écriture de n'importe quelle fonction seuil $T_{n,p}$ de cette manière. On le résume dans le tableau suivant :

	Calcul de la somme	Comp. supérieure	Fonction $T_{n,p}$
$N_{plans}(n)$	$[\log_2(n)] + 2$	$[\log_2(n)] + 1$	$[\log_2(n)] + 2$
$N_{o.e.}(n)$	n $+\ 2\ ([\ \log_2(n)\ ])\ (n+1)$ $-2^{[\ \log_2(n)\ ]+2}\ +4$	$[\log_2(n)] + 1$	n $+\ 2\ ([\ \log_2(n)\ ])$ $\times\ (n + \frac{3}{2})\ +5$ $-2^{[\ \log_2(n)\ ]+2}$

Le nombre total d'o.e. est la somme des o.e. de chaque étape et le nombre de plans mémoire est le maximum du nombre de plans nécessaires pour chacune des 'tapes. En effet, le nombre maximum nécessaire de plans est celui de la phase la plus coûteuse des différentes phases du calcul ; car ces phases se font séquentiellement sur les mêmes plans.

On peut remarquer que le coût en o.e. est indépendant du seuil p de la fonction et que la partie la plus dispendieuse est le calcul de la somme (la comparaison est négligeable en coût d'o.e.).

On présente les coûts en o.e. et en plans mémoire pour des fonctions seuils ayant de 3 à 36 variables et on les compare au coût de la méthode AVB :

n	3	5	9	16	25	36
$N_{plans}(n)$	3	4	5	6	6	7
$N_{o.e.}(n)$	9	20	45	97	178	288
$N_{AVB}(n)$	4	18	107	2024	38084	672 628

Le coût de cette nouvelle méthode est nettement moins élevé que les méthodes présentées précédemment. Cependant, elle est plus dispendieuse en plans mémoire.

VI Ecriture des fonctions symetriques

Par définition, la valeur d'une fonction symétrique ne dépend que de la valeur de la somme des variables. On peut donc décrire une fonction symétrique par l'ensemble des cas (portant sur la somme) où la fonction vaut 1, c'est-à-dire

$$f(x_1, x_2, \ldots, x_n) \;=\; \begin{cases} 1 \;\text{ si } \;\displaystyle\sum_{i=1}^{n} x_i \;=\; p_l \\ 0 \;\text{ sinon} \end{cases}$$

où $(p_l)_{1\leq l\leq l_0}$ est un des seuils de la fonction. Le nombre de seuils l_0 est au plus égal à $[n/2]$. Au delà, il est plus judicieux de calculer la fonction $\bar{f}$ (qui est elle aussi symétrique), mais qui comporte moins de seuils et de prendre l'opposé du résultat pour obtenir f.

Pour calculer n'importe quelle fonction symétrique, il est possible de calculer explicitement la somme de ses variables et de comparer cette somme successivement à chacun des seuils $(p_l)_{1\leq l\leq l_0}$. Il est à noter cependant que la succession des comparaisons égalité entre la somme des variables et les différents seuils p_l est toujours possible, car la comparaison égalité telle qu'elle est présentée dans le paragraphe précédent n'altère pas les valeurs des plans $b_0, b_1, \cdots, b_{k-1}$ représentant la somme.

Comme pour les fonctions symétriques, les paragraphes précédents nous permettent d'obtenir le coût en o.e. et en plans mémoire nécessaires à l'écriture de toute les fonctions symétriques.

On le résume dans le tableau suivant :

	Calcul de la somme	Comp. égalité	Fonction $T_{n,p}$
$N_{plans}(n)$	$[\log_2(n)] + 2$	$[\log_2(n)] + 1$	$[\log_2(n)] + 2$
$N_{o.e.}(n)$	n $+ 2\ ([\ \log_2(n)\ ])\ (n+1)$ $-2^{[\ \log_2(n)\ ]+2}\ + 4$	$l_0\ [\log_2(n)]$	n $+\ ([\ \log_2(n)\ ])$ $\times\ (2n + 2 + l_0)\ + 4$ $-2^{[\ \log_2(n)\ ]+2}$

On en déduit une majoration du coût en o.e. grâce aux fonctions symétriques ayant exactement $[n/2]$ seuils (par exemple, la fonction valant 1 quand la somme est paire), d'où :

$$N_{o.e.}(n) \ \leq \ \frac{5}{2}n[\ \log_2(n)] \ - \ 3n \ + \ 2[\ \log_2(n)\ ] + 4$$

On obtient donc les coûts suivants en o.e. et en plans mémoire pour des fonctions symétriques ayant de 3 à 36 variables et $[n/2]$ seuils :

Nombre de variables	3	5	9	16	25	36
$N_{plans}(n)$	3	4	5	6	6	7
$N_{o.e.}(n)$	8	21	53	116	221	372

Comme précédemment, on remarque que la majeure partie du coût provient du calcul de la somme, puisque la différence entre les coûts d'une fonction seuil $T_{n,p}$ et la "pire"des fonctions symétriques est faible et que la partie commune au calcul de ces deux fonctions est le calcul de la somme.

VII Conclusion

En décomposant les fonctions symétriques et seuils en une ou plusieurs comparaisons de la somme de leurs variables avec les seuils, leurs calculs est largement moins coûteux en e.o. que toutes les méthodes présentées jusqu'ici. Cependant, cette méthode nécessite un nombre de plans mémoire important et le faible coût du calcul est alors compensé par le coût élevé en plans mémoire.

Chapitre VII

Notation Redondante

I. Introduction

Pour la détermination des fonctions seuils et symétriques, l'obtention de la somme des variables et la comparaison de celle-ci avec les seuils est une des méthodes les plus performantes dès que la mémoire est suffisante pour stocker la somme. Dans un certain nombre de cas, la "clustérisation" nous permet de pallier au manque de mémoire ; mais augmente considérablement le coût opératoire.

Dans ce rapport, nous utilisons une nouvelle méthode de calcul de la somme dont la notation permet une "parallélisation" du calcul et donc une diminution du coût ; mais qui nécessite une clustérisation. Cette méthode repose sur la notation redondante de la somme.

II. Notation redondante

On peut représenter n'importe quel nombre binaire F comme la somme de deux nombres binaires de même taille D et E s'ils vérifient :

$$F \ = \ 2D \ + \ E \tag{1}$$

où $+$ est l'opération arithmétique de sommation.

Cette représentation qui n'est pas unique est appelé une notation redondante, car on stocke dorénavant deux nombres en lieu et place d'un seul.

II.1 Propriétés de la notation redondante

Même si ce stockage est coûteux, il permet néanmoins des manipulations plus simples lors de sommations successives. Par exemple, on peut utiliser la formule (1) pour exprimer la somme de trois nombres binaires A, B, C :

$$A \ + \ B \ + \ C \ = \ 2D \ + \ E \tag{2}.$$

Cette égalité (2) est toujours valide pour chaque case mémoire de cette représentation, d'où les égalités :

$$A_i \ + \ B_i \ + \ C_i \ = \ (2D)_i \ + \ E_i \tag{3}.$$

où i représente le i-ème case de la décomposition de ces nombres. Si à ces conditions (3), on ajoute la condition suivante : "pour chaque case, $(2D)_i$ doit représenter la retenue de la somme de $A_i \ + \ B_i \ + \ C_i$ et E_i l'unité", on obtient l'unicité de la détermination de D_i et E_i et on en déduit les formules suivantes :

$$D_i \ = \ (A_i \ \wedge \ B_i) \ \vee \ \ (A_i \ \wedge \ C_i) \ \vee \ \ (B_i \ \wedge \ C_i) \tag{4.1}$$

$$E_i \ = \ A_i \ \oplus \ B_i \ \oplus \ C_i. \tag{4.2}$$

Lorsque l'on calcule ces formules, on obtient explicitement les nombres D et E ; mais pas $2D$, encore moins $2D + E$.

Mais si on désire utiliser le résultat de $A \ + \ B \ + \ C$ (c-à-d D et E) pour faire de nouveau une opération de sommation, il n'est pas nécessaire de connaître $2D \ + \ E$. En effet, on choisit de calculer $2D$ (en décalant d'une case mémoire vers la gauche la représentation de D) et de réutiliser la formule (2) avec $2D$, E et la nouvelle valeur C', d'où :

$$2D \ + \ E \ + \ C' \ = \ 2D' \ + \ E'.$$

Les nombres D' et E', obtenus grâce aux formules (4.1) et (4.2), représentent maintenant la somme $A \ + \ B \ + \ C \ + C'$. On peut réitérer cette opération autant de fois que nécessaire pour calculer une somme de plusieurs termes.

Pour le calcul de $A + B + \sum_{i=1}^{n} C_j$, on effectue les opérations suivantes :

$$A + B + C^1 = 2D^1 + E^1;$$

puis

$$(A + B + C^1) + C^2 = 2D^2 + E^2$$

que l'on obtient en réalisant :

$$2D^1 + E^1 + C^2 = 2D^2 + E^2.$$

Ensuite de façon générale, on effectue :

$$2D^k + E^k + C^{k+1} = 2D^{k+1} + E^{k+1} \tag{5}$$

jusqu'à l'obtention de D^n et E^n. Il suffit ensuite de calculer $2D^n + E^n$ pour obtenir le résultat final.

Le passage de la représentation de la somme partielle des k premières valeurs (soit D^k et E^k définis par les valeurs D_i^k et E_i^k pour $0 \leq i \leq m-1$) à la somme partielle des $(k+1)$ premières valeurs (soit D^{k+1} et E^{k+1} définis par les valeurs D_i^{k+1} et E_i^{k+1} pour $0 \leq i \leq m$) décrit par l'égalité (5) est défini, sur chaque case mémoire de cette représentation ($0 \leq i \leq m$), par les égalités suivantes :

$$D_i^{k+1} = (E_i^k \wedge D_{i-1}^k) \vee (E_i^k \wedge C_i^k) \vee (D_{i-1}^k \wedge C_i^k) \tag{6.1}$$

$$E_i^{k+1} = E_i^k \oplus D_{i-1}^k \oplus C_i^k \tag{6.2}$$

où i représente le i-ème case de la décomposition de ces nombres et avec les conventions $D_{-1}^k = 0$ et $E_m^k = C_m^k = 0$. Pour obtenir les nouvelles valeurs D^{k+1} et E^{k+1}, il est nécessaire d'effectuer les mêmes opérations élémentaires sur l'ensemble des cases mémoire de la représentation. Cependant, entre chaque case mémoire, il n'y a pas de notion de récurence et ces opérations peuvent être effectuer dans n'importe quel ordre et de manière indépendante, contrairement à la notation classique avec une propagation de retenue. Il est donc judicieux de choisir dans la rétine une disposition qui permette d'effectuer sur l'ensemble des cases mémoire une seule fois chaque opération élémentaire ; le calcul devient en quelque sorte "parallèle". Ce calcul "parallèle" nécessite la disposition de la représentation de D et de E, dans la rétine, sur deux plans distincts. Pour stocker ces représentations, on a recours à la clustérisation.

Finalement, le coût de l'ajout d'un nombre C^k est alors de 7 opérations élémentaires ; mais le stockage est lui coûteux.

II.2 Application à la somme de n variables

Avec une notation redondante, le calcul de la somme de n variables est obtenu par les calculs successifs des sommes partielles $\sum_{i=1}^{k} x_i$ où chaque somme partielle est le résultat de l'ajout de la variable x_k, à la somme partielle précédente $\sum_{i=1}^{k-1} x_i$ déjà calculée. La somme partielle $\sum_{i=1}^{k-1} x_i$ étant représentée avec une notation redondante D^{k-1} et E^{k-1} (définis par les valeurs D_i^{k-1} et E_i^{k-1} pour $0 \leq i \leq m-1$), on ajoute $C^k = x_k$ à cette somme grâce aux formules (6.1) et (6.2). Cependant, dans ce cas particulier, toutes les valeurs de C^k $(1 \leq k \leq n)$ vérifient :

$$C_0^k = 0 \text{ ou } 1 \text{ et } C_i^k = 0 \quad \forall i \geq 1$$

On obtient donc une simplification des formules de sommation :

pour tout $1 \leq i \leq m-1$:

$$D_i^k \;=\; E_i^{k-1} \;\wedge\; D_{i-1}^{k-1}$$

$$E_i^k \;=\; E_i^{k-1} \;\oplus\; D_{i-1}^{k-1}$$

et pour la première et dernière case mémoire :

$$D_0^k \;=\; E_0^{k-1} \;\wedge\; x_k \quad \text{et} \quad D_m^k \;=\; 0$$

$$E_0^k \;=\; E_0^{k-1} \;\oplus\; x_k \;\text{ et }\; E_m^k \;=\; D_{m-1}^{k-1}$$

car par définition $D_{-1}^{k-1} \;=\; 0$ et $E_m^{k-1} \;=\; 0$. On obtient finalement pour tout $i \geq 0$:

$$D_i^k \;=\; E_i^{k-1} \;\wedge\; D_{i-1}^{k-1}$$

$$E_i^k \;=\; E_i^{k-1} \;\oplus\; D_{i-1}^{k-1}$$

avec les conventions $D_{-1}^{k-1} \;=\; x_k$ et $E_m^{k-1} \;=\; 0$.

II.3 Description de l'algorithme et coûts

L'obtention d'une somme de n variables, avec une notation redondante, suit la description du paragraphe précédent et demande le calcul successif des représentation (D^k, E^k) pour $0 \leq k \leq n$.

Lors du passage de (D^k, E^k) à (D^{k+1}, E^{k+1}), l'ajout de la variable x_{k+1} modifie le nombre de cases nécessaires au stockage de la représentation redondante.

Pour le stockage de la représentation redondante (D^k, E^k), au plus $[\,\log_2(k)\,]+1$ cases mémoire suffisent, car la somme totale représentée par $2D^k \;+\; E^k$ est au plus égale à k. Au vu de l'algorithme, il faut une case mémoire supplémentaire pour stocker (D^{k+1}, E^{k+1}) et ceci entraîne très vite de stocker trop d'informations.

On décide donc d'effectuer ces opérations sur un nombre fixe de cases mémoire.

Or, les sommes partielles ne peuvent dépasser la valeur n de la somme totale. Cette somme peut être stockée en $[\,\log_2(n)\,]+1$ cases mémoire et sa représentation redondante est définie sur $[\,\log_2(n)\,]$ cases pour D et sur $[\,\log_2(n)\,]+1$ cases sur E.

De plus, la valeur de la case $[\,\log_2(k)\,]$ de D^k est toujours nulle pour tout $0 \leq k \leq n$ et le reste à chaque transformation.

Il suffit donc de stocker D^k et E^k pour $0 \leq k \leq n$ sur $[\,\log_2(k)\,]+1$ cases mémoires. Ce choix permet de plus le stockage de la variable x_k sur la case supplémentaire du plan contenant D^k, ce qui permet d'effectuer le calcul en parallèle. De plus, le calcul de la valeur finale $2D^n \;+\; E^n$ nécessite pour être calculée et stockée $[\,\log_2(n)\,]+1$ case mémoire.

Ayant défini systématiquement la longueur de D^k et E^k pour $0 \leq k \leq n$ comme constante, l'ajout de la variable x_k à la représentation (D^{k-1}, E^{k-1}) est donné par les opérations suivantes :

$$\begin{aligned} D_i^k \;&=\; E_i^{k-1} \;\wedge\; D_{i-1}^{k-1} \\ E_i^k \;&=\; E_i^{k-1} \;\oplus\; D_{i-1}^{k-1} \quad \text{pour} \;\; 0 \leq i \leq [\,\log_2(n)\,] \end{aligned}$$

avec la convention $D_{-1}^{k-1} \;=\; x_k$ et en ayant initialiser avec $E_i^0 \;=\; D_i^0 \;=\; 0$ pour $0 \leq i \leq [\,\log_2(n)\,]$.

En définissant sur la rétine, les deux premiers plans par

$$\begin{aligned} p_0^1 &= x_k \text{ et } p_i^1 = D_{i-1}^{k-1} \quad \text{pour } 1 \leq i \leq [\log_2(n)] \\ p_i^2 &= E_i^{k-1} \quad \text{pour } 0 \leq i \leq [\log_2(n)] \end{aligned}$$

alors le calcul de D_i^k et E_i^k est donné par les opérations suivantes :

$$\begin{aligned} p_i^2 &= p_i^2 \oplus p_i^1 \\ p_i^1 &= \overline{p_i^2} \wedge p_i^1 \end{aligned}$$

qui ne demandent pas de mémoire supplémentaire. On a ensuite les égalités :

$$\begin{aligned} D_i^{k+1} &= p_i^1 \\ E_i^{k+1} &= p_i^2 \quad \text{pour } 0 \leq i \leq [\log_2(n)] \end{aligned}$$

L'ajout d'une variable à une somme partielle déjà calculée nécessite seulement deux opérations élémentaires et le calcul total de la somme coûte $2n$ o.e.

Cependant pour cette méthode, le stockage nécessaire est de $2([\log_2(n)] + 1)$ cases mémoire pour la représentation redondante. Par définition de la méthode, cette place mémoire est obligatoirement obtenu par une clustérisation. On en déduit la valeur du coefficient de clustérisation, qui est égale à $[\log_2(n)] + 1$.

Avec cette méthode le coût de la somme est de $2n$ o.e. et de $2([\log_2(n)] + 1)$ cases mémoire, répartis en deux plans avec un coefficient de clustérisation $[\log_2(n)] + 1$.

A partir de ce résultat et avec le même coefficient de clustérisation, il est possible de calculer une fonction seuil ou symétrique en utilisant trois plans mémoire. Le troisième plan sert pour stocker les valeurs de la fonction lors de chaque passage de la clustérisation. Nous exposons dans les paragraphes suivants trois approches distinctes pour obtenir la valeur de ces fonctions et comparons ensuite leurs coûts respectifs.

La première approche repose sur le calcul explicite de la somme à partir de la représentation redondante (dont on perd la connaissance) ; puis de la comparaison avec le ou les seuils. Cette méthode nécessite à chaque passage de la clustérisation le recalcul de la représentation redondante et le calcul explicite de la somme des variables, ce qui augmente notoirement son coût.

La deuxième est une variante de la première méthode et consiste à limiter le recalcul au minimum : c'est-à-dire à la partie variable de la somme. Pour cela, on conserve du passage précédent la valeur de la somme (sous forme non redondante) à laquelle on impute la partie variable.

La troisième approche quant à elle utilise le même principe. Mais, la somme précédente est alors conservée sous forme redondante.

III. Calcul direct de la somme

A partir de la représentation redondante (D, E) de la somme, on calcule effectivement la somme des variables. Pour cela, on utilise un algorithme de propagation de la retenue similaire à celui du rapport numéro 6. Si on note $A = 2D$ et $B = E$, le calcul effectif de $C = 2D + E$ est obtenu par la succession des opérations suivantes, effectuées sur chaque case mémoire en commençant par le bit de poids faible et en

allant vers le bit de poids fort :

$$
\begin{aligned}
c_0 &= a_0 \oplus b_0 \\
c_1 &= a_1 \oplus b_1 \oplus (a_0 \wedge b_0) \\
c_2 &= a_2 \oplus b_2 \oplus \Big([(a_1 \vee b_1) \wedge (a_0 \wedge b_0)] \vee (a_1 \wedge b_1) \Big) \\
&\cdot \\
c_i &= a_i \oplus b_i \oplus \Big((a_{i-1} \vee b_{i-1}) \wedge [(a_{i-2} \vee b_{i-2}) \wedge \cdots \vee (a_{i-2} \wedge b_{i-2})] \\
&\qquad\qquad \vee (a_{i-1} \wedge b_{i-1}) \Big) \\
&\cdot \\
c_{\nu-1} &= a_{\nu-1} \oplus b_{\nu-1} \oplus \Big((a_{\nu-2} \vee b_{\nu-2}) \wedge [(a_{\nu-3} \vee b_{\nu-3}) \wedge \cdots \vee \\
&\qquad\qquad (a_{\nu-3} \wedge b_{\nu-3})] \vee (a_{\nu-2} \wedge b_{\nu-2}) \Big) \\
c_\nu &= \Big((a_{\nu-1} \vee b_{\nu-1}) \wedge [(a_{\nu-2} \vee b_{\nu-2}) \wedge \cdots \vee \\
&\qquad\qquad (a_{\nu-2} \wedge b_{\nu-2})] \vee (a_{\nu-1} \wedge b_{\nu-1}) \Big)
\end{aligned}
$$

où $\nu = ([\log_2(n)] + 1)$. On simplifie l'écriture en utilisant une variable intermédiaire R_i qui représente la propagation de la retenue et on obtient :

$$
\begin{aligned}
c_i &= a_i \oplus b_i \oplus R_{i-1} \\
R_i &= (a_i \wedge b_i) \vee [(a_i \vee b_i) \wedge R_{i-1}]
\end{aligned}
$$

pour tout $1 \leq i \leq \nu - 1$ et avec les conditions initiales $C_0 = a_0 \oplus b_0$ et $R_0 = a_0 \wedge b_0$ et la condition finale $C_\nu = R_{\nu-1}$. Dans notre cas particulier, la somme $2D + E$ ne peut excéder n et il n'y a donc pas de retenue finale, d'où on a toujours $C_\nu = 0$. On transforme l'écriture de R_i :

$$
\begin{aligned}
R_i &= (a_i \wedge b_i) \vee [(a_i \vee b_i) \wedge R_{i-1}] \\
&= (a_i \wedge b_i) \vee \Big([(a_i \oplus b_i) \vee (a_i \wedge b_i)] \wedge R_{i-1} \Big) \\
&= (a_i \wedge b_i) \vee \Big((a_i \oplus b_i) \wedge R_{i-1} \Big).
\end{aligned}
$$

Finalement, il suffit de calculer :

$$
\begin{aligned}
c_i &= a_i \oplus b_i \oplus R_{i-1} \\
R_i &= (a_i \wedge b_i) \vee [(a_i \oplus b_i) \wedge R_{i-1}]
\end{aligned}
$$

pour tout $1 \leq i \leq \nu - 1$ et avec les conditions initiales $C_0 = a_0 \oplus b_0$ et $R_0 = a_0 \wedge b_0$.

Dans le calcul de c_i et R_i, une partie peut être effectuée indépendamment entre chaque case mémoire, une autre, celle liée à R_{i-1}, est récursive. Dans cette méthode, le stockage des valeurs de A et de B, sur deux plans distincts permet d'envisager le calcul d'une partie des opérations de façon "parallèle". A partir de la représentation redondante (D, E), on définit A et B sur les deux plans p^1 et p^2 de la rétine par :

$$
\begin{aligned}
p_i^1 &= (A)_i = (2D)_i = \begin{cases} D_{i-1} & \text{pour } 1 \leq i \leq \nu - 1 \\ 0 & \text{pour } i = 0 \end{cases} \\
p_i^2 &= (B)_i = (E)_i = \begin{cases} E_i & \text{pour } 0 \leq i \leq \nu - 2 \\ 0 & \text{pour } i = \nu - 1. \end{cases}
\end{aligned}
$$

Dans ces conditions, le calcul de $a \oplus b$ et de $a \wedge b \ = \ \overline{(a \oplus b)} \ \wedge b$ peut être obtenu en effectuant les opérations suivantes :

$$\begin{aligned} p_i^1 \ &= \ p_i^1 \ \oplus \ p_i^2 \\ p_i^2 \ &= \ \overline{p_i^1} \ \wedge \ p_i^2. \end{aligned}$$

Le calcul coûte 2 o.e. et ne nécessite pas de place mémoire supplémentaire. A l'issue du calcul p^1 contient $a \oplus b$ et $p^2 \ \ a \wedge b$.

On va maintenant calculer récursivement les valeurs de R_i et c_i, toujours avec le moins de place mémoire et d'opérations élémentaires possibles. Si on suppose connaître R_{i-1} et que cette valeur est stockée dans la case mémoire p_{i-1}^1 (au premier rang, R_0 est bien contenu dans p_0^1 après la fin du calcul parallèle) le calcul de c_i, qui représente la somme, et de R_i nécessaire au prochain calcul peut s'effectuer par les opérations suivantes :

$$\begin{aligned} p_i^1 \ &= \ p_i^1 \ \oplus \ p_{i-1}^2 \ = \ a_i \ \oplus \ b_i \ \oplus \ R_{i-1} \ = \ c_i \\ p_{i-1}^2 \ &= \ \overline{p_i^1} \ \wedge \ p_i^2 \ = \ \overline{(a_i \ \oplus \ b_i \ \oplus \ R_{i-1})} \ \wedge \ R_{i-1} \ = \ (a_i \ \oplus \ b_i) \ \wedge \ R_{i-1} \\ p_i^2 \ &= \ p_i^2 \ \vee \ p_{i-1}^2 \ = \ R_i. \end{aligned}$$

On a bien obtenu les valeurs de c_i et de R_i (qui de plus est stocké dans la case attendue pour le calcul suivant). Il faut 3 o.e. pour calculer (c_i, R_i) à partir de $a_i \oplus b_i$, $a_i \wedge b_i$ et R_{i-1} et cela sans place mémoire supplémentaire.

Pour les premières valeurs (c_0, R_0), il n'est pas besoin d'effectuer ces opérations et pour la dernière valeur, il suffit d'une opération pour obtenir $c_{\nu-1}$. Finalement, le calcul de la somme à partir de la représentation redondante nécessite : 2 opérations pour la partie parallèle et $3([\ \log_2(n)\ ] + 1 - 2) + 1$ pour la partie récursive, soit au total $3([\ \log_2(n)\ ])$.

Il est maintenant possible de déterminer une fonction seuil ou symétrique à partir de la somme explicite des variables. Il suffit de comparer avec le ou les seuils définissant la fonction. Cependant, la détermination de la fonction sur l'ensemble de la rétine ne se fait pas de façon simultanée. Le résultat est obtenu par passage successif puisque le nombre de mémoires nécessaires au calcul de la somme est obtenu par une clustérisation. Il faut recalculer, à chaque passage de la clustérisation, la somme des variables, puis les comparaisons et stocker le résultat.

Les résultats des paragraphes précédents et du rapport numéro 6 pour les comparaisons égalités nous permettent d'obtenir le coût en o.e. et en plans mémoire nécessaires à l'écriture de n'importe quelle fonction seuil $T_{n,p}$ de cette manière. On le résume dans le tableau suivant :

	Calcul de la somme	Comp. supérieure	Fonction $T_{n,p}$
$N_{o.e.}^1(n)$ passage	$2n + 3[\log_2(n)]$	$[\log_2(n)] + 1$	$2n + 4[\log_2(n)] \ + 1$
$N_{o.e.}^1(n)$ Total	$\Big(2n + 3[\log_2(n)]\Big)$ $\times \ ([\log_2(n)] + 1)$	$\Big([\log_2(n)] + 1\Big)$ $\times \ ([\log_2(n)] + 1)$	$\Big(2n + 4[\log_2(n)] + 1\Big)$ $\times \ ([\log_2(n)] + 1)$
$N_{cases}(n)$ passage	$2([\log_2(n)] + 1)$	$[\log_2(n)] + 1$	$2([\log_2(n)] + 1)$ $+1$ pour résultat
$N_{cases}(n)$ Total	$2([\log_2(n)] + 1)$	$[\log_2(n)] + 1$	$3([\log_2(n)] + 1)$
$N_{plans}(n)$	2	1	3

Dans la première partie du tableau, nous présentons le coût opératoire par passage (qui est fixe) et le coût total égal au produit du coefficient de clustérisation (égal à $[\log_2(n)]+1$) par le coût par passage. Dans la deuxième partie, nous exposons le nombre de cases mémoire nécessaires à cette détermination. Il est égal à $2([\log_2(n)]+1)$ cases mémoire pour le calcul et une case pour le stockage du résultat de chaque passage. Cette méthode réutilise à chaque passage le même espace de calcul. Seul, la place mémoire pour le stockage du résultat doit être différent à chaque passage. On obtient donc pour le stockage total :

$$\begin{aligned} N_{cases}(n) &= N_{cases\ calcul}(n) + N_{cases\ stockage}(n) \times Coeff\ de\ clusterisation \\ &= 2([\log_2(n)]+1) + 1 \times ([\log_2(n)]+1) \\ &= 3([\log_2(n)]+1). \end{aligned}$$

Cependant, ce nombre de cases mémoire est obtenu par clustérisation et il ne faut que trois plans mémoire pour l'atteindre.

On remarque que le coût en o.e. est indépendant du seuil p de la fonction et que la partie la plus dispendieuse est à chaque passage le calcul de la somme (la comparaison est négligeable en coût d'o.e.).

On peut étendre ce résultat aux fonctions symétriques. Pour cela, il suffit de comparer (ici ce sont des comparaisons égalités) la somme des variables à une succession de seuils $(p_l)_{1\leq l\leq l_0}$. Comme pour les fonctions symétriques, les paragraphes précédents nous permettent d'obtenir le coût en o.e. et en plans mémoire nécessaires à l'écriture de toutes les fonctions symétriques.

On le résume dans le tableau suivant :

	Calcul de la somme	Comp. égalité	Fonction symétrique
$N^1_{o.e.}(n)$ passage	$2n+3[\log_2(n)]$	$l_0\ [\log_2(n)]$	$2n+3[\log_2(n)]$ $+\ l_0\ [\log_2(n)]$
$N^1_{o.e.}(n)$ Total	$\Big(2n+3[\log_2(n)]\Big)$ $\times\ ([\log_2(n)]+1)$	$\Big(l_0[\log_2(n)]\Big)$ $\times\ ([\log_2(n)]+1)$	$\Big(2n+3[\log_2(n)]$ $+\ l_0\ [\log_2(n)]\ \Big)$ $\times\ ([\log_2(n)]+1)$
$N_{cases}(n)$ passage	$2([\log_2(n)]+1)$	$[\log_2(n)]+1$	$2([\log_2(n)]+1)$ $+1$ pour résultat
$N_{cases}(n)$ Total	$2([\log_2(n)]+1)$	$[\log_2(n)]+1$	$3([\log_2(n)]+1)$
$N_{plans}(n)$	2	1	3

Comme précédemment, la partie la plus importante du coût est le calcul de la somme des variables. Nous présentons dans le paragraphe suivant une deuxième approche qui réduit ce coût.

IV. Calcul par ajout de la partie variable avec une notation classique

Lors d'une clustérisation après le premier passage, le calcul de la somme des variables peut être obtenu par la modification de la somme précédente et non par son recalcul total. En effet, entre deux passages de la clustérisation, il suffit de rajouter à la somme précédente les nouvelles variables entrant dans la définition de la fonction et d'enlever les variables n'en faisant plus partie. Nous allons utiliser cette méthode pour obtenir la valeur de la somme.

Si pour un curseur de taille n, il y a n_0 variables à modifier entre deux passages de la clustérisation (voir Annexe A pour la définition et l'obtention de n_0). L'ensemble de ces n_0 variables est composé d'autant de variables à ajouter que de variables à retrancher. On définit alors l'obtention de la nouvelle somme par l'égalité suivante :

$$Somme\ Nouvelle\ =\ Somme\ Precedente\ -\ \Sigma\, x_{ret}\ +\ \Sigma\, x_{nouv}$$

où x_{ret} et x_{nouv} sont respectivement les variables à retrancher et à ajouter à la somme. Il suffit donc d'estimer le coût (en o.e. et en place mémoire) des opérations d'ajout et de retrait d'une variable à la somme.

Cette méthode est dévéloppée pour une notation classique avec un algorithme de propagation de retenue. Pour ne pas dépasser n, et donc avoir besoin de plus de place mémoire, on effectue les retraits avant les ajouts.

L'algorithme de l'ajout d'une variable x à une somme A (inférieure ou égale à n et représentée par les valeurs $a_0, a_1, \cdots, a_{\nu-1}$ avec $\nu = ([\log_2(n)] + 1)$) a été présenté dans le rapport numéro 6 et peut se résumer par :

$$\begin{array}{llll} c_0 = a_0 \oplus x & & R_0 = a_0 \wedge x & \\ c_i = a_i \oplus R_{i-1} & & R_i = a_i \wedge R_{i-1} & 1 \leq i \leq \nu - 2 \\ c_{\nu-1} = a_{\nu-1} \oplus R_{\nu-2} & & & \end{array}$$

car on se restreint au cas où on ne dépasse pas la valeur n. Il est possible d'obtenir cette sommation en $2\nu - 1$ o.e. et avec une seule case mémoire supplémentaire.

Le retrait d'une variable x à la somme A peut s'écrire de façon similaire :

$$\begin{array}{llll} c_0 = a_0 \oplus x & & R_0 = \overline{a_0} \wedge x & \\ c_i = a_i \oplus R_{i-1} & & R_i = \overline{a_i} \wedge R_{i-1} & 1 \leq i \leq \nu - 2 \\ c_{\nu-1} = a_{\nu-1} \oplus R_{\nu-2}. & & & \end{array}$$

La séquence de retrait est identique à celle de l'ajout à la négation des valeurs de A près. On peut donc obtenir le retrait avec le même nombre d'o.e. et le même place mémoire que l'ajout.

Cependant, il est possible d'améliorer ces algorithmes en tenant compte de la nature du stockage. Le calcul d'un ajout ou d'un retrait ne change l'algorithme que par le choix entre les valeurs de a_i et de $\overline{a_i}$. On présente donc les améliorations pour le retrait.

Toutes les valeurs de la somme A sont stockées sur le même plan. On peut donc effectuer le calcul de c_i de façon parallèle dès que les valeurs de R_i sont connues et si elles sont elles aussi stockées sur un même plan. On va calculer les valeurs des retenues R_i $(0 \leq i \leq \nu - 2)$ de façon récurrente.

A l'initialisation, les valeurs de A sont stockées sur le premier plan et on met la valeur de x dans p_0^2. On a donc :

$$\begin{array}{l} p_i^1 = a_i \qquad 0 \leq i \leq \nu - 1 \\ p_0^2 = x \quad \text{et} \quad p_i^2 = 0 \quad \text{pour} \quad i \geq 1 \end{array}$$

On effectue séquentiellement les o.e. suivantes :

$$\begin{array}{l} p_1^2 = \overline{p_0^1} \wedge p_0^2 = \overline{a_0} \wedge x = R_0 \\ p_i^2 = \overline{p_{i-1}^1} \wedge p_{i-1}^2 = \overline{a_{i-1}} \wedge R_{i-2} = R_{i-1} \quad 1 \leq i \leq \nu - 1. \end{array}$$

Il faut une o.e. pour chaque calcul de R_i, soit $(\nu-1)$ o.e. en tout. Ensuite connaissant toutes les valeurs de R_0 à $R_{\nu-2}$, on effectue l'o.e. suivante :

$$\begin{aligned} p_i^2 &= p_i^2 \oplus p_i^1 \qquad \text{pour} \quad 0 \le i \le \nu-1. \\ &= \begin{cases} a_0 \oplus x & \text{pour } i=0 \\ a_i \oplus R_i & \text{pour } i \ge 1 \end{cases} \\ &= c_i \end{aligned}$$

On a obtenu le résultat final (retrait ou ajout d'une variable) en ν o.e. et sans mémoire supplémentiare, ce qui divise le coût par deux. Le recalcul de la somme entre chaque passage de la clustérisation nécessite donc n_0 $([\log_2(n)]+1)$ o.e. On peut donc maintenant estimer les coûts (en o.e. et en mémoire) de cette nouvelle méthode, que l'on résume dans le tableau suivant pour une fonction seuil $T_{n,p}$ dont le curseur comporte n_0 variables à modifiées à chaque passage.

	Calcul de la somme	Comp. supérieure	Fonction $T_{n,p}$
$N^2_{o.e.}(n)$ 1er passage	$2n+3[\log_2(n)]$	$[\log_2(n)]+1$	$2n+4[\log_2(n)]\ +1$
$N^2_{o.e.}(n)$ passage	$n_0([\log_2(n)]+1)$	$[\log_2(n)]+1$	$(n_0+1)([\log_2(n)]\ +1)$
$N^2_{o.e.}(n)$ Total	$2n+3[\log_2(n)]+\Big(n_0([\log_2(n)]+1)\Big)\times([\log_2(n)])$	$\Big([\log_2(n)]+1\Big)\times([\log_2(n)]+1)$	$2n+3[\log_2(n)]+\Big((n_0+1)([\log_2(n)])+1\Big)\times([\log_2(n)]+1)$
$N_{cases}(n)$ passage	$2([\log_2(n)]+1)$	$[\log_2(n)]+1$	$2([\log_2(n)]+1)$ $+1$ pour résultat
$N_{cases}(n)$ Total	$2([\log_2(n)]+1)$	$[\log_2(n)]+1$	$3([\log_2(n)]+1)$
$N_{plans}(n)$	2	1	3

Dans ce tableau, le coût total en o.e. est égal à la somme de l'ensemble des coûts de chaque passage de la clustérisation. Le premier passage est le plus coûteux ; car il représente l'obtention de la sommme complète. Le coût de chaque passage suivant est égal à celui de la modification de la partie variable. Quant à la place mémoire, son nombre reste inchangé.

En comparant les coûts (en o.e.) des méthodes de calcul direct et de la partie variable, on obtient la différence :

$$\Delta \ = \ N^1_{o.e.}\ (n) \ - \ N^2_{o.e.}\ (n) \ = \ [\log_2(n)]\Big(2n \ + \ (3-n_0)([\log_2(n)]+1)+3\Big).$$

Cette différence est positive pour la majorité des valeurs de n et n_0 et on en déduit que :

$$N^1_{o.e.}(n) \ \ge \ N^2_{o.e.}(n)$$

pour toutes les valeurs de n et de n_0.

Finalement le coût total de cette nouvelle méthode est toujours inférieure à la précédente. Comme précédemment, on peut étendre cette définition aux fonctions symétriques et avec les mêmes notations on obtient le tableau suivant :

	Calcul de la somme	Comp. égalité	Fonction symétrique
$N^2_{o.e.}(n)$ 1er passage	$2n + 3[\log_2(n)]$	$l_0\ [\log_2(n)]$	$2n + 3[\log_2(n)]$ $+\ l_0\ [\log_2(n)]$
$N^2_{o.e.}(n)$ passage	$n_0([\log_2(n)] + 1)$	$l_0\ [\log_2(n)]$	n_0+ $(n_0 +\ l_0)\ [\log_2(n)]$
$N^2_{o.e.}(n)$ Total	$2n + n_0$ $([\log_2(n)])(3 + n_0)$	$l_0[\log_2(n)]$ $\times\ ([\log_2(n)] + 1)$	$2n + n_0 + \Big(n_0$ $+\ l_0\ ([\log_2(n)]\ + 1)$ $+3\Big) \times\ ([\log_2(n)])$
$N_{cases}(n)$ passage	$2([\log_2(n)] + 1)$	$[\log_2(n)] + 1$	$2([\log_2(n)] + 1)$ $+1$ pour résultat
$N_{cases}(n)$ Total	$2([\log_2(n)] + 1)$	$[\log_2(n)] + 1$	$3([\log_2(n)] + 1)$
$N_{plans}(n)$	2	1	3

Pour les fonctions symétriques, cette méthode est encore moins coûteuse que le recalcul de la somme présentée au paragraphe III.

V. Calcul par ajout de la partie variable avec une notation redondante

Nous développons maintenant la méthode par ajout de la partie variable à une somme représentée par une notation redondante. Si le principe est identique, le passage à une notation redondante nécessite une adaptation importante pour le retrait d'une variable.

L'ajout d'une variable à une somme représentée par une notation redondante est présente dans la première partie de ce rapport. L'algorithme et les coûts de cette opération sont connus.

Quant au retrait d'une variable en notation redondante, il faut écrire un nouvel algorithme. Contrairement à une notation classique, le principe de la notation redondante ne permet pas de faire propager facilement une retenue de façon rétrograde. Pour obtenir ce nouvel algorithme, on utilise donc l'égalité suivante (voir Annexe B pour sa démonstration) :

$$A - x \ = \ \overline{\overline{A} + x}$$

où les opérateurs + et - sont les opérateurs arithmétiques et qui transforme un retrait en une somme.

Cependant, pour utiliser cette propriété avec une notation redondante, il faut tout d'abord définir, en fonction de la représentation redondante d'un nombre, la représentation redondante de sa négation ; soit obtenir (B', C') tel que :

$$B' + C' \ = \ \overline{A}$$

en fonction de (B, C) défini par $B + C \ = \ A$.

On obtient (voir Annexe B), la définition de la représentation redondante (D', E') de la négation de A en fonction de la représentation (D, E) de A. Si A est représenté par (D, E) tel que $A = 2D + E$ et si D et E sont inférieurs ou égaux à $2^p - 1$, alors la négation de A est représentée par (D', E') définies par l'égalité suivante :

$$2D' + E' \;=\; 2\overline{D} + \overline{E} + 2^p + 2. \tag{7}$$

Pour obtenir l'unique représentation (D', E'), il suffit donc de sommer $\overline{D}, \overline{E}$ et $X = 2^p + 2$ avec l'algorithme habituel.

Pour obtenir $A - x$, il faut ensuite sommer la variable x à la représentation (D', E') pour obtenir une nouvelle représentation redondante dont on prendra la négation grâce à l'équation (7). Cependant, il est plus judicieux d'effectuer les deux opérations de sommation simultanément en ajoutant en lieu et place de X, le nombre $X' \;=\; X + x \;=\; 2^p + 2 + x$. On a ainsi décrit l'obtention du retrait de x à une somme A représentée par une notation redondante.

On en présente maintenant l'algorithme détaillé du retrait d'une variable à une somme dans une rétine et on en déduit son coût opératoire.

La somme de n variables étant définie par sa représentation redondante (D, E) (connus par les valeurs D_i et E_i pour $0 \leq i \leq \nu = ([\log_2(n)] + 1))$, le retrait d'une variable x à cette somme est donné par l'obtention successive des représentations (D', E') (définie par D'_i et E'_i) vérifiant l'égalité :

$$2D' + E' \;=\; 2\overline{D} + \overline{E} + 2^\nu + 2 + x \tag{8}$$

et puis la représentation $(D", E")$ (définie par $D"_i$ et $E"_i$) vérifiant :

$$2D" + E" \;=\; 2\overline{D'} + \overline{E'} + 2^\nu + 2. \tag{9}$$

Les valeurs D'_i et E'_i sont donc définies par :

$$\begin{aligned} E'_i &= \overline{E}_i \;\oplus\; \overline{D_{i-1}} \;\oplus\; (2^\nu + 2 + x)_i \qquad \text{pour } \; 0 \leq i \leq \nu - 1 \\ D'_i &= (\overline{E}_i \;\wedge\; \overline{D}_{i-1}) \;\vee\; (\overline{E}_i \;\wedge\; (2^\nu + 2 + x)_i) \vee\; (\overline{D}_{i-1} \;\wedge\; (2^\nu + 2 + x)_i) \end{aligned}$$

où $(2^\nu + 2 + x)_i$ est la représentation du nombre $(2^\nu + 2 + x)$. Il est évident que la valeur 2^ν qui correspond à un 1 sur la case ν ne rentre pas ici dans le calcul de E'_i et de D'_i. Ces calculs se simplifient en :

$$\begin{aligned} E'_0 &= \overline{E}_0 \;\oplus\; x \\ E'_1 &= \overline{E}_1 \;\oplus\; D_0 \\ E'_i &= \overline{E}_i \;\oplus\; \overline{D}_{i-1} \qquad \text{pour } \; 2 \leq i \leq \nu - 1 \end{aligned}$$

et

$$\begin{aligned} D'_0 &= \overline{E}_0 \;\vee\; x \\ D'_1 &= \overline{E}_1 \;\vee\; \overline{D}_0 \\ D'_i &= \overline{E}_i \;\wedge\; \overline{D}_{i-1} \qquad \text{pour } \; 2 \leq i \leq \nu - 1 \end{aligned}$$

De même, on peut ensuite exprimer les valeurs de $D"_i$ et de $E"_i$ en fonction de celles de D'_i et E'_i :

$$\begin{aligned} E"_i &= \overline{E'}_i \;\oplus\; \overline{D'_{i-1}} \;\oplus\; (2^\nu + 2)_i \qquad \text{pour } \; 0 \leq i \leq \nu - 1 \\ D"_i &= (\overline{E'}_i \;\wedge\; \overline{D'}_{i-1}) \;\vee\; (\overline{E'}_i \;\wedge\; (2^\nu + 2)_i) \vee\; (\overline{D'}_{i-1} \;\wedge\; (2^\nu + 2)_i). \end{aligned}$$

ce qui se simplifie en :

$$\begin{aligned} E"_0 &= \overline{E'}_0 \oplus x \\ E"_1 &= \overline{E'}_1 \oplus D'_0 \\ E"_i &= \overline{E'}_i \oplus \overline{D'}_{i-1} \qquad \text{pour } 2 \leq i \leq \nu - 1 \end{aligned}$$

et

$$\begin{aligned} D"_0 &= \overline{E'}_0 \\ D"_1 &= \overline{E'}_1 \vee \overline{D'}_0 \\ D"_i &= \overline{E'}_i \wedge \overline{D'}_{i-1} \qquad \text{pour } 2 \leq i \leq \nu - 1. \end{aligned}$$

On peut ensuite implanter ces opérations sur la rétine de la façon suivante. En définissant sur la rétine, les deux premiers plans par

$$\begin{aligned} p_0^1 &= \overline{x} \text{ et } p_i^1 = D_{i-1} \quad \text{pour } 1 \leq i \leq \nu - 1 \\ p_i^2 &= E_i \quad \text{pour } 0 \leq i \leq \nu - 1 \end{aligned}$$

alors le calcul de D'_i et E'_i est donné par les opérations suivantes :

$$\begin{aligned} p_i^2 &= \overline{p}_i^2 \oplus \overline{p}_i^1 \\ p_i^1 &= \overline{p_i^2} \wedge p_i^1 \quad \text{pour } 0 \leq i \leq \nu - 1. \end{aligned}$$

Il faut ensuite modifier les valeurs particulières p_1^1 et p_1^2 :

$$\begin{aligned} p_1^2 &= p_1^1 \vee p_1^2 = (\overline{E}_1 \wedge \overline{D}_0) \vee (\overline{E}_1 \oplus \overline{D}_0) = \overline{E}_1 \vee \overline{D}_0 \\ p_1^1 &= \overline{p}_1^2 \oplus p_1^1 = \overline{(\overline{E}_1 \vee \overline{D}_0)} \oplus (\overline{E}_1 \wedge \overline{D}_0) = \overline{E}_1 \wedge D_0. \end{aligned}$$

On a donc les égalités :

$$\begin{aligned} p_i^1 &= D'_i \\ p_i^2 &= E'_i \qquad \text{pour } 0 \leq i \leq \nu - 1 \end{aligned}$$

et pour le calcul de $E"$ et de $D"$, on modifie le stockgae de p^1 comme suit :

$$p_0^1 = 1 \quad \text{et} \quad p_i^1 = D'_{i-1} \quad \text{pour } 1 \leq i \leq \nu - 1$$

Le calcul de $E"_i$ et $D"_i$ est alors donnée par une succession d'o.e. identique :

$$\begin{aligned} p_i^2 &= \overline{p}_i^2 \oplus \overline{p^1}_i \\ p_i^1 &= \overline{p}_i^2 \wedge p_i^1 \qquad \text{pour } 0 \leq i \leq \nu - 1. \end{aligned}$$

Il faut de même modifier deux valeurs particulières :

$$\begin{aligned} p_1^2 &= p_1^1 \vee p_1^2 = (\overline{E'}_1 \wedge \overline{D'}_0) \vee (\overline{E'}_1 \oplus \overline{D'}_0) = \overline{E'}_1 \vee \overline{D'}_0 \\ p_1^1 &= \overline{p}_1^2 \oplus p_1^1 = \overline{(\overline{E'}_1 \vee \overline{D'}_0)} \oplus (\overline{E'}_1 \wedge \overline{D'}_0) = \overline{E'}_1 \wedge D_0. \end{aligned}$$

A la fin de ces opérations, on a obtenu ainsi les valeurs de $E"$ et de $D"$ qui sont contenus dans les plans p^1 et p^2 et qui représentent le retrait d'une variable à une somme. Le coût de ce retrait est donc de 8 o.e. et ne demande pas de place mémoire supplémentaire.

Le recalcul d'une somme comprenant n_0 variables à modifier, entre chaque passage de la clustérisation nécessite l'ajout de $\frac{n_0}{2}$ variables et le retrait de $\frac{n_0}{2}$ variables. Il en coûte donc $\frac{8n_0}{2}$ o.e. pour les retraits et $\frac{2n_0}{2}$ o.e. pour les ajouts, soit au total $5n_0$ o.e. sans place mémoire supplémentaire.

A l'issue de l'obtention de la notation redondante, il est nécessaire de calculer la somme pour pouvoir effectuer les comparaisons. Le passage à une notation classique explicité dans la paragraphe III coûte $3([\log_2(n)])$ o.e.

On peut donc maintenant estimer les coûts (en o.e. et en mémoire) de cette nouvelle méthode, que l'on résume dans le tableau suivant pour une fonction seuil $T_{n,p}$ dont le curseur comporte n_0 variables à modifiées à chaque passage.

	Calcul de la somme	Comp. supérieure	Fonction $T_{n,p}$
$N^3_{o.e.}(n)$ 1er passage	$2n + 3[\log_2(n)]$	$[\log_2(n)] + 1$	$2n + 4[\log_2(n)] + 1$
$N^3_{o.e.}(n)$ passage	$5n_0 + 3[\log_2(n)]$	$[\log_2(n)] + 1$	$5n_0 + 4[\log_2(n)] + 1$
$N^3_{o.e.}(n)$ Total	$2n + 3[\log_2(n)] + \Big(5n_0 + 3[\log_2(n)]\Big) \times ([\log_2(n)])$	$\Big([\log_2(n)] + 1\Big) \times ([\log_2(n)] + 1)$	$2n + 3[\log_2(n)] + \Big(5n_0 + 4[\log_2(n)] + 2\Big) \times ([\log_2(n)]) + 1$
$N_{cases}(n)$ passage	$2([\log_2(n)] + 1)$	$[\log_2(n)] + 1$	$2([\log_2(n)] + 1)$ +1 pour résultat
$N_{cases}(n)$ Total	$2([\log_2(n)] + 1)$	$[\log_2(n)] + 1$	$3([\log_2(n)] + 1)$
$N_{plans}(n)$	2	1	3

Dans ce tableau, le coût total en o.e. est égal à la somme de l'ensemble des coûts de chaque passage de la clustérisation. Le premier passage est le plus coûteux ; car il représente l'obtention de la sommme complète. Le coût de chaque passage suivant est égal à celui de la modification de la partie variable. Quant à la place mémoire, son nombre reste inchangé.

En comparant les coûts (en o.e.) des méthodes de calcul de la partie variable, on obtient la différence :

$$\Delta \;=\; N^3_{o.e.}\,(n) \;-\; N^2_{o.e.}\,(n) \;=\; [\log_2(n)] \;\Big(([\log_2(n)])\;(n_0 - 3) \;-\; 4n_0\Big).$$

Cette différence ne permet pas 'affirmer qu'une méthode est toujours plus performante qu'une autre.

Comme précédemment, on peut étendre cette définition aux fonctions symétriques et avec les mêmes

notations on obtient le tableau suivant :

	Calcul de la somme	Comp. égalité	Fonction symétrique
$N^3_{o.e.}(n)$ 1er passage	$2n + 3[\log_2(n)]$	$l_0\ [\log_2(n)]$	$2n + 3[\log_2(n)]$ $+\ l_0\ [\log_2(n)]$
$N^3_{o.e.}(n)$ passage	$5n_0 + 3([\log_2(n)])$	$l_0\ [\log_2(n)]$	$5n_0+$ $(3 +\ l_0)\ [\log_2(n)]$
$N^3_{o.e.}(n)$ Total	$2n + 5n_0$ $6([\log_2(n)])$	$l_0[\log_2(n)]$ $\times\ ([\log_2(n)] + 1)$	$2n + \Big(5n_0$ $+ 2l_0\ + 6\Big)$ $\times\ ([\log_2(n)])$
$N_{cases}(n)$ passage	$2([\log_2(n)] + 1)$	$[\log_2(n)] + 1$	$2([\log_2(n)] + 1)$ $+1$ pour résultat
$N_{cases}(n)$ Total	$2([\log_2(n)] + 1)$	$[\log_2(n)] + 1$	$3([\log_2(n)] + 1)$
$N_{plans}(n)$	2	1	3

V. Conclusion

Des trois méthodes présentées, il semble que le calcul de la somme à chaque passage de la clustérisation par ajout de la partie variable en notation redondante soit la méthode la plus performante pour la majorité des fonctions seuils et symétriques.

Annexe A : Partie variable d'une somme

Entre deux passages successifs d'une clustérisation, la partie variable d'une somme est composée des nouvelles variables entrant dans la définition de la fonction et des variables n'en faisant plus partie. Cette partie variable vérifie principalement quatre propriétés :

Propriété 1

Elle est composée d'autant de variables à enlever que de variables à ajouter.

En effet, la taille du curseur qui détermine les variables entrant dans la définition de la somme est fixe d'un passage à l'autre. Si lors du nouveau passage, n_1 variables communes au précédent curseur n'entrent plus dans la définition de la somme, il faut n_1 nouvelles variables. On a donc autant de variables "entrantes " que de variables "sortantes". La variation est toujours paire, de la forme $2n_1$.

Propriété 2

La variation de la somme est indépendante de l'emplacement de calcul de la fonction dans le curseur.

La variation est due au décalage du curseur entre deux passages et non à l'emplacement de calcul de la fonction dans ce curseur. De même,

Propriété 3

La variation est identique pour chaque passage de la clustérisation.

Cependant, la variation du curseur est liée à la géométrie.

Propriété 4

La variation de la somme est fortement dépendante de la géométrie du curseur

On peut établir pour certaines formes géométriques les plus courantes, les variations du curseur. Pour un curseur rectiligne, il est évident que quel que soit sa longueur, le nombre de variables qui changent est égal à 2 : une sortante et une entrante. Pour un curseur carré de taille n, il y a $\sqrt{n}$ variables entrantes et $\sqrt{n}$ variables sortantes. Il est aussi possible de déterminer la partie variable pour un curseur cruciforme.

On donne dans le tableau suivant une estimation de la partie variable pour des curseurs rectilignes et carrés :

n	Théorique	4	9	16	25	36
Curseur rectiligne	2	2	2	2	2	2
Curseur carré	$2\sqrt{n}$	4	6	8	10	12

Annexe B

1. Calcul de $(A - x)$

Pour tout nombre A inférieur ou égal à $2^p - 1$ (c'est-à-dire défini par les p valeurs $(a_i)_{0 \leq i \leq p-1}$), alors le complément de A noté $\overline{A}$ est défini par :

$$a_i \ \wedge \ \overline{a}_i \ = \ 1 \quad 0 \leq i \leq p-1$$

ce que l'on traduit par l'égalité :

$$A + \overline{A} \ = \ 2^p - 1 \tag{1}$$

où + est l'opération arithmétique. On déduit de (1) l'égalité suivante :

$$\overline{A} \ = \ 2^p - 1 - A$$

En ajoutant à A, une variable x (de longueur 1), on a l'égalité suivante :

$$A + \overline{A} \ = \ (A - x) + \overline{A} + x \ = \ 2^p - 1$$

d'où :

$$(A - x) \ = \ 2^p - 1 - (\overline{A} + x)$$
$$(A - x) \ = \overline{\overline{A} + x}.$$

On a obtenu finalement une égalité transformant un retrait en une somme.

2. Représentation redondante d'une négation

Pour obtenir la définition d'une représentation redondante de la négation d'un nombre, il faut au préalable établir l'égalité suivante :

$$\overline{2D} \ = \ 2\overline{D} + 1. \tag{2}$$

Pour un nombre D, de longueur p (défini par les valeurs $p_i \ \ 0 \leq i \leq p-1$), le nombre $2D$ est défini par $(p+1)$ valeurs telles que :

$$(2D)_0 \ = \ 0$$
$$(2D)_i \ = \ D_{i-1} \quad 0 \leq i \leq p+1.$$

On en déduit que :

$$\begin{aligned} \overline{(2D)}_0 &= 1 \\ \overline{(2D)}_i &= \overline{D}_{i-1} = (2\overline{D})_i \end{aligned}$$

d'où finalement :

$$\overline{2D} = 2\overline{D} + 1.$$

La représentation redondante de la négation d'un nombre A est maintenant donnée par le lemme suivant.

Lemme 1. – *Soit (D, E) une représentation redondante du nombre A définie par : $2D + E = A$ où les nombres D et E sont inférieurs ou égaux à $2^p - 1$ et donc définies par les valeurs d_i et e_i pour $0 \leq i \leq p-1$, alors la négation de A, $\overline{A}$ est représentée par la notation redondante (D', E') vérifiant :*

$$2D' + E' = \overline{A} = 2\overline{D} + \overline{E} + 2^p + 2.$$

Ce procédé est constructif, car il suffit d'utiliser l'algorithme de sommation entre $2\overline{D}$, $\overline{E}$ et $2^p + 2$ pour obtenir (D', E').

Le nombre $A = 2D+E$ peut être de longueur $(p+2)$ et il est défini par les valeurs a_i pour $0 \leq i \leq p+1$. De plus, il vérifie l'égalité :

$$A + \overline{A} = \overline{2D + E} + 2D + E = 2^{p+2} - 1.$$

On en déduit :

$$\begin{aligned} \overline{2D + E} &= 2^{p+2} - 1 - 2D - E \\ &= (2^{p+1} - 2D) + (2^p - E - 1) + 2^p \end{aligned} \tag{3}$$

Or le nombre E est défini par p valeurs, d'où :

$$\overline{E} = 2^p - 1 - E$$

et le nombre $2D$ est défini par $(p + 1)$ valeurs, d'où :

$$\overline{2D} = 2^{p+1} - 1 - 2D.$$

En reportant dans (3) et en utilisant l'égalité (2), on obtient :

$$\overline{2D + E} = 2\overline{D} + \overline{E} + 2^p + 2.$$

Remarque: Le nombre $\overline{2D + E}$ est défini par $(p + 2)$ valeurs.

Chapitre VIII

Comparaison des diverses méthodes

I. Introduction

Pour la détermination des fonctions seuils et symétriques, nous avons présenté dans les précédents rapports différentes méthodes et avons estimé leurs coûts opératoires et la place mémoire nécessaires à leur mise en œuvre. Pour l'ensemble de ces méthodes, nous résumons ici leurs principales propriétés et inconvénients et donnons leurs coûts.

Dans le deuxième paragraphe, nous définissons les variables générales et les notions nécessaires à la définition de l'ensemble des méthodes. Nous présentons dans le troisième paragraphe, les différentes méthodes appliquées aux fonctions seuils et donnons dans le paragraphe quatre les estimations numériques.

II. Notations

On définit de façon générale les variables suivantes :

- n : nombre de variables entrant dans la définition des fonctions seuils ou symétriques ;

- p : valeur du seuil d'une fonction seuil ;

- l_0 $(1 \geq l_0 \geq [n/2])$: nombre de seuils distincts d'une fonction symétrique ;

- S : somme des n variables : $S = \sum_{i=1}^{n} x_i$.

On définit ensuite les fonctions seuils et symétriques.

La fonction seuil $T_{n,p}\ (x_1, \ldots, x_n)$ est définie par :

$$T_{n,p} = \begin{cases} 1 \text{ si } \sum_{i=1}^{n} x_i = p \\ 0 \text{ sinon} \end{cases}$$

Une fonction symétrique est décrite par l'ensemble des cas (portant sur la somme) où la fonction vaut 1, c'est-à-dire

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} 1 \text{ si } \sum_{i=1}^{n} x_i = p_l \\ 0 \text{ sinon} \end{cases}$$

où $(p_l)_{1 \leq l \leq l_0}$ est un des seuils de la fonction.

On définit ensuite les coûts des méthodes par :

- $N_{o.e.}$ est le coût de calcul en opérations élémentaires (o.e.) qui décompte le nombre d'opérations élémentaires $\vee$, $\wedge$ et $\oplus$ entrant dans l'algorithme d'une méthode. En revanche, les négations et les déplacements des variables ne sont pas pris en compte.

- N_{plans} est le nombre de plans nécessaires (avec ou sans clustérisation) au calcul et au stockage d'une fonction par une méthode. Ce nombre ne tient pas compte du plan contenant l'image.

- $\nu = [\log_2(n)] + 1$ est le nombre de cases mémoire nécessaires pour stocker le résultat de la somme de n variables. Ces cases mémoire peuvent être aussi bien sur le même plan que sur ν plans distincts. Cependant, les algorithmes de calculs de la somme auront alors des coûts différents.

Pour une clustérisation, il est nécessaire de définir les notions suivantes :

- C_{clus} est le coefficient de clustérisation qui détermine le nombre de cases mémoire mis en commun pour effectuer le calcul. Le nombre de cases mémoire est égal au coefficient de clustérisation multiplié par le nombre de plans mémoire disponibles pour chaque pixel de l'image. Le coefficient de clustérisation est aussi le nombre de passages nécessaires pour calculer la même fonction sur l'ensemble de l'image d'une rétine.

- n_0 est le nombre de variables de la somme à modifier entre deux passages successifs dans une clustérisation. En choisissant un clusteur linéaire, la valeur n_0 est fixe et peut être déterminée facilement.

III. Tableau général pour les fonctions seuils

On détermine tout d'abord les avantages et les inconvénients des méthodes sans clustérisation pour les fonctions seuils.

Méthodes	Avantages/Inconvénients/Coûts
$*$ F.N.D. (Forme Normale Disjonctive) Description : énumération de tous les monômes à n variables où la fonction vaut 1 (voir rapport 1).	Inconvénients - Coût opératoire exponentiel ; - Enumération des monômes sans aucune mise en commun de facteurs ; - Coût dépendant de la valeur du seuil. Avantages - Coût en mémoire minimum ; - Mise en œuvre facile. Coûts $N_{plans} = 2$ $N_{o.e.} = n(C_n^p + C_n^{p+1} + \cdots + C_n^n)$
$*$ F.N.D. Réduite Description : on réduit le nombre des monômes à l'énumération de tous ceux à p variables (voir rapport 3).	Inconvénients / Avantages Identique à la précédente méthode Coûts $N_{plans} = 2$ $N_{o.e.} = p\ C_n^p$
A.V.B. (Ajout des Variables par Blocs) Description : les n variables sont découpées par blocs consécutifs égaux. Dans chaque bloc, on détermine un partie suffisante de la somme pour décider si le seuil p est atteint (voir rapport 3).	Inconvénients - Description fastidieuse de $S(p,k,m)$ - Difficulté de l'estimation du coût opératoire a priori ; - Coût opératoire dépend fortement du choix de la taille des blocs. Avantages - Coût opératoire très inférieur à la FND réduite Coûts $N_{plans} = 3$ $N_{o.e.}$: Voir ci-après.

Le nombre d'opérations $N_{o.e.}$, obtenu en découpant les variables par blocs de k, est au plus égal à :

$\star$ si $n = k(m-1)+b$ avec $0 < b < k$,

$$\begin{aligned}
&\text{si } m = 1, \quad N_T \leq p\binom{b}{p} ; \\
&\text{si } m = 2, \quad N_T \leq \sum_{p_m=0}^{b} (p-p_m)\binom{k}{p-p_m} + \sum_{p_m=0}^{b} p_m \binom{b}{p_m} ; \\
&\text{si } m > 2, \quad N_T \leq (m-1) \sum_{p_m=0}^{b} \sum_{p_1=0}^{\min(k,p-p_m)} s(p-p_m-p_1,k,m-2)\ p_1\binom{k}{p_1} \\
&\qquad\qquad + \sum_{p_m=0}^{b} s(p-p_m,k,m-1)\ p_m\binom{b}{p_m}.
\end{aligned}$$

$\star$ si $n = km$,

$$\begin{aligned}
&\text{si } m = 1, \quad N_T \leq p\binom{k}{p} ; \\
&\text{si } m > 1, \quad N_T \leq m \sum_{p_1=0}^{\min(k,p)} s(p-p_1,k,m-1)\ p_1\binom{k}{p_1}.
\end{aligned}$$

où $s(p,k,m)$ est le cardinal de l'ensemble défini par :

$$S(p,k,m) = \{(p_1,\ldots,p_m) ; \qquad \forall j = 1,\ldots,m,\ p_j \leq k,\ \sum_{1}^{m} p_j = p\}.$$

Méthodes	Avantages/Inconvénients/Coûts
$*$ A.V.B. optimisé Description : le principe est identique à la méthode précédente ; on utilise seulement une mémoire supplémentaire pour stocker des résultats intermédiaires (voir rapport 4).	Inconvénients / Avantages Identique à la précédente méthode Coûts $N_{plans} = 4$ $N_{o.e.}$ = entre 90% et 70% de la valeur précédente.

Méthodes	Avantages/Inconvénients/Coûts
$*$ Calcul et Comparaison Description : cette méthode repose sur le calcul explicite de la somme des variables et la comparaison avec le seuil (voir rapport 6).	Avantages - Coût opératoire relativement faible et indépendant de la valeur du seuil p. Inconvénients - Coût mémoire dépendant du nombre de variables et donc très vite trop important. Coûts $N_{plans} = [\log_2(n)] + 2$ $N_{o.e.} = n + 2\,([\log_2(n)\,]) \times (n + \frac{3}{2}) + 5 - 2^{[\log_2(n)\,]+2}$

Méthodes avec clusterisation	Avantages/Inconvénients/Coûts
$*$ Calcul Somme en notation classique Description : on calcule la somme en notation classique et on la compare au seuil à chaque passage de la clustérisation. Deux choix possibles pour l'utilisation de cette méthode.	Inconvénients - Coût mémoire important Avantages - Coût opératoire relativement faible et indépendant de la valeur du seuil p. - Facile à mettre en œuvre Coûts 1) $C_{clus} = [\log_2(n)] + 1$ $N_{plans} = 2$ $N_{o.e.} = n\Big((2\,\nu + 1)\nu\Big)$ 2) $C_{clus} = [\log_2(n)] + 1$ $N_{plans} = 3$ $N_{o.e.} = n\Big((\nu + 1)\nu\Big)$

Il est possible de réduire le coût du premier choix en limitant la propagation de la retenue (voir rapport 6).

Méthodes avec clusterisation	Avantages/Inconvénients/Coûts
$*$ Calcul Somme en notation redondante Description : le principe est identique à la méthode précédente avec une notation redondante de la somme (voir rapport 7).	Inconvénients / Avantages Identique à la précédente méthode Coûts $C_{clus} = [\log_2(n)] + 1$ $N_{plans} = 3$ $N_{o.e.} = \Big((2n + 4[\log_2(n)] + 1\Big)\nu$

Méthodes avec clusterisation	Avantages/Inconvénients/Coûts
$*$ Calcul par Ajout de la Partie Variable (en notaion classique) Description : on se restreint à recalculer la partie variable de la somme entre chaque passage (voir rapport 7).	Avantages Coût plus faible. Coûts $C_{clus} = [\log_2(n)] + 1$ $N_{plans} = 3$ $N_{o.e.} = 2n + 3[\log_2(n)] +$ $\Big((n_0 + 1)[\log_2(n)] + 1\Big)\nu$
$*$ Calcul par Ajout de la Partie Variable (en notation redondante) Description : même méthode que la précédente ; mais le calcul est effectué en notation redondante (voir rapport 7).	Avantages Coût plus faible. Coûts $C_{clus} = [\log_2(n)] + 1$ $N_{plans} = 3$ $N_{o.e.} = 2n + 1 +$ $\Big(4n_0 + 4[\log_2(n)] + 5\Big)$ $\times[\log_2(n)]$

IV. Estimations numériques

Nous présentons maintenant les estimations numériques de l'ensemble des méthodes pour les fonction seuils $T_{n,p}$ pour n variant de 4 à 36. Pour certaines méthodes, le coût est indépendant de la valeur du seuil p. Pour d'autres (méthode AVB, FND, ...), il est dépendant de la valeur p et est estimé pour le "pire" des cas, c'est-à-dire $p = [\frac{n}{2}]$. Pour les méthodes CAPVc et CAPVr, on définit la valeur n_0 par le tableau suivant :

n	4	9	12	16	25	32	36
n_0	4	6	8	8	10	12	12

en considérant le curseur comme carré pour les valeurs de $n = 4, 9, 16, 25, 36$ et comme repésentant une croix pour $n = 12$ et 32.

Pour désigner les méthodes, on définit les abréviations suivantes :

- FND : Forme Normale Disjonctive réduite ;

- AVB : Ajout des Variables par Blocs ;

- AVBopt : Ajout des Variables par Blocs optimisé ;

- CC : Calcul et Comparaison ;

- CSc : Calcul Somme en notation classique (CSc^1 et CSc^2 représentent respectivement les choix 1 et 2) ;

- CSr : Calcul Somme en notation redondante ;

- CAPVc : Calcul par Ajout de la Partie Variable en notation classique ;

- CAPVr : Calcul par Ajout de la Partie Variable en notation redondante.

On présente le coût (en o.e.) de chaque méthode pour l'ensemble des fonctions dans le tableau suivant :

n	4	9	12	16	25	32	36
FND	11	503	5543	102 959	62×10^6	19×10^9	163×10^9
AVB	12	107	384	2024	38 084	241 664	672 628
AVBopt	11	98	338	1722	28 190	178 833	497 762
CC	15	45	66	97	178	241	288
CSc^1	45	180	264	485	890	1205	1728
CSc^2	48	180	240	480	750	960	1512
CSr	55	124	153	245	335	410	558
CAPVc	47	115	145	229	287	341	483
CAPVr	67	142	172	245	295	341	438

Sans tenir compte du coût mémoire, la méthode la moins coûteuse en o.e. pour les fonctions seuils est presque toujours la méthode CC (Calcul et Comparaison). Cependant, pour l'implantation dans une rétine, nous recherchons un compromis entre le coût opératoire et le coût mémoire. Nous présentons dans le tableau

suivant le coût des différentes méthodes en fonction de leurs coûts mémoire.

Nbre plans mémoire	2	3	4	5
$T_{4,2}$	11 (FND) 45 (CSc^1)	12 (AVB) 47 (CAPVc) 48 (CSc^2) 55 (CSr) 67 (CAPVr)	11 (AVBopt) 15 (CC)	
$T_{9,4}$	180 (CSc^1) 503 (FND)	107 (AVB) 115 (CAPVc) 124 (CSr) 142 (CAPVr) 180 (CSc^2)	98 (AVBopt)	45 (CC)
$T_{12,6}$	264 (CSc^1) 5543 (FND)	145 (CAPVc) 153 (CSr) 172 (CAPVr) 240 (CSc^2) 384 (AVB)	338 (AVBopt)	66 (CC)

Nbre plans mémoire	2	3	4	6/7
$T_{16,8}$	485(CS c^1) 102 959 (FND)	229(CAPVc) 245 (CSr) 245 (CAPVr) 480 (CS c^2) 2024 (AVB)	1722(AVBopt)	97(CC) (6 plans)
$T_{25,12}$	890(CS c^1) 62×10^6 (FND)	287(CAPVc) 295 (CAPVr) 335 (CSr) 750 (CS c^2) 38 084 (AVB)	28190(AVBopt)	178(CC) (6 plans)
$T_{32,16}$	1205(CS c^1) 19×10^9 (FND)	341(CAPVc) 341(CAPVr) 410 (CSr) 960 (CS c^2) 241 664 (AVB)	10^5 (AVBopt)	241(CC) (6 plans)
$T_{36,18}$	1728(CS c^1) 1×10^{11} (FND)	438(CAPVr) 483(CAPVc) 558 (CSr) 1512 (CS c^2) 672 628(AVB)	10^5 (AVBopt)	288(CC) (7 plans)

V. Conclusion

Il est évident que certaines méthodes sont totalement inadaptées. La FND est trop coûteuse en o.e. (sauf cas exceptionnel). La méthode AVBopt apporte un gain trop faible en o.e. pour l'ajout d'un plan mémoire supplémentaire. La méthode CC est trop dispendieuse en place mémoire dans le cas général. Il reste donc cinq méthodes avec clustérisation (CS c^1, CS c^2, CAPVc, CAPVr, CSr) et une méthode sans clustérisation (AVB).

Pour un faible nombre de variables (jusqu'à 9), la méthode AVB est plus performante que les autres méthodes et son coup mémoire est identique. Pour les fonctions seuils comprenant plus de variables, la méthode AVB n'est plus compétitive et dans les méthodes restantes, les méthodes utilisant la notation redondante (CAPVr, CAPVs, CSr) s'avèrent les mieux adaptées. Elles sont d'autant plus performantes que le nombre de variables grandit (l'écart en o.e. par rapport à CS c^1 et CS c^2 se creuse).

En conclusion :

- il est toujours possible d'implanter les fonctions seuils sur deux plans mémoire pour un coût relativement faible avec une méthode de clustérisation (CS c^1) ;

- il est plus avantageux pour les fonctions seuils d'utiliser des méthodes nécessitant un plan mémoire supplémentaire :

  - la méthode AVB jusqu'à 9 variables ;

  - la méthode CAPVc jusqu'à 32 variables

  - la méthode CAPVr au delà.

Ces deux dernières méthodes utilisent la clustérisation.

Chapitre IX

Algorithme majoritaire

I. Introduction

Dans les premiers chapitres, la fonction seuil $T_{9,3}$ était décrite par une méthode tenant compte des propriétés géométriques du curseur. Dans ce chapitre, on rappelle cette méthode et on décrit son implantation sur une rétine. On en déduit une généralisation de l'algorithme à l'ensemble des fonctions seuils $T_{9,p}$. On présente ensuite les coûts opératoires et mémoires de cet algorithme pour les fonctions $T_{9,p}$ et on en déduit ses principales propriétés et limitations.

II. Description de l'algorithme pour la fonction $T_{9,3}$

On définit l'ensemble des variables du curseur de la fonction $T_{9,3}$ par un carré de côté 3 et on estime la valeur de la fonction au centre du carré.

Le calcul de la fonction $T_{9,3}$ repose sur l'algorithme suivant :

- calcul simultané des sommes des trois colonnes et stockage du résultat dans les cases mémoires liées au pixel central de la colonne ;

- ajout des résultats des deux colonnes extrêmes avec une saturation du résultat à trois et stockage du résultat dans les cases mémoires liées au pixel central du curseur ;

- ajout successif des valeurs des pixels de la colonne centrale à la somme précédemment calculée avec saturation.

On décrit ces trois étapes par les opérations élémentaires suivantes.

*** Calcul des sommes des colonnes**

Soit une colonne de trois pixels notées x_1, x_2, x_3 et leur somme S représentée par (b_1, b_2) respectivement l'unité et la dizaine de la représentation en base 2, alors on peut représenter l'obtention de cette somme par les opérations élémentaires suivantes :

$$\begin{aligned} b_1 &= x_1 \oplus x_2 \\ b_2 &= x_1 \wedge x_2 \end{aligned}$$

et

$$\begin{aligned} b_1 &= b_1 \oplus x_3 \\ b_2 &= b_2 \oplus (b_1 \wedge x_3). \end{aligned}$$

*** Ajout des deux colonnes avec saturation**

Soit (b_2^g, b_1^g) et (b_2^d, b_1^d) les représentations respectives en base deux des sommes des colonnes gauche et droite du curseur, leur ajout avec une saturation à trois est donné par les opérations élémentaires suivantes :

$$\begin{aligned} b_2 &= b_2^g \vee b_2^d \vee (b_1^g \wedge b_1^d) \\ b_1 &= b_1^g \oplus b_1^d \vee (b_2^g \wedge b_2^d) \vee (b_2^g \wedge b_1^g \wedge b_1^d) \vee (b_2^d \wedge b_1^d \wedge b_1^g). \end{aligned}$$

*** Ajout d'une variable avec saturation**

Soit x la variable à ajouter à la somme S représentée par (b_2, b_1), l'obtention de la nouvelle somme avec saturation à la valeur trois est donnée par la succession d'opérations élémentaires suivantes :

$$\begin{aligned} b_2 &= b_2 \vee (b_1 \wedge x) \\ b_1 &= (b_1 \oplus x) \vee (b_2 \wedge b_1 \wedge x). \end{aligned}$$

L'implantation de cet algorithme sur une rétine nécessite deux plans mémoire pour stocker le résultat du calcul de la somme des colonnes, auxquels il faut ajouter au moins deux autres plans mémoire pour obtenir la fusion de ces résultats. Avec quatre plans mémoire notés respectivement (p^1, p^2, p^3, p^4), il est possible d'obtenir le calcul de $T_{9,3}$. Les opérations élémentaires suivantes :

$$\begin{aligned} p_i^1 &= x_1 \oplus x_2 \\ p_i^2 &= x_1 \wedge x_2 \end{aligned}$$

et

$$\begin{aligned} p_i^1 &= p_i^1 \oplus x_3 \\ p_i^2 &= p_i^2 \oplus (p_i^1 \wedge x_3) \end{aligned}$$

déterminent l'obtention de la somme des colonnes du curseur en 5 o.e. et de façon parallèle.

On suppose maintenant que les représentations des sommes de la colonne gauche et droite sont stockées dans les cases mémoire suivantes :

$$\begin{aligned} p_{i-1}^1 &= b_1^g \qquad & p_{i+1}^1 &= b_1^d \\ p_{i-1}^2 &= b_2^g \qquad & p_{i+1}^2 &= b_2^d \end{aligned}$$

et on obtient la somme des deux colonnes par les opérations suivantes :

$$\begin{aligned} p_i^3 &= p_{i-1}^1 \wedge p_{i+1}^1 = b_1^g \wedge b_1^d \\ p_i^3 &= p_i^4 \end{aligned}$$

et

$$\begin{aligned} p_i^3 &= p_i^3 \wedge p_{i+1}^2 = b_1^g \wedge b_1^d \wedge b_2^d \\ p_i^4 &= p_i^4 \wedge p_{i-1}^2 = b_1^g \wedge b_1^d \wedge b_2^g \end{aligned}$$

Puis,

$$\begin{aligned} p_i^3 &= p_i^3 \vee p_i^4 = (b_1^g \wedge b_1^d \wedge b_2^d) \vee (b_1^g \wedge b_1^d \wedge b_2^g) \\ p_i^4 &= p_{i-1}^1 \oplus p_{i+1}^1 = b_1^g \oplus b_1^d \end{aligned}$$

et

$$\begin{aligned} p_i^3 &= p_i^3 \vee p_i^4 = b_1^g \oplus b_1^d \vee (b_1^g \wedge b_1^d \wedge b_2^d) \vee (b_1^g \wedge b_1^d \wedge b_2^g) = b_2^s \\ p_i^4 &= \overline{p^4}_i \wedge p_{i+1}^1 = b_1^g \wedge b_1^d \end{aligned}$$

et

$$p_i^4 = p_i^4 \vee p_{i+1}^2 = (b_1^g \wedge b_1^d) \vee b_2^d$$

et

$$p_i^4 = p_i^4 \vee p_{i-1}^2 = (b_1^g \wedge b_1^d) \vee b_2^d \vee b_2^g = b_1^s.$$

La somme et la saturation ont coûté 9 o.e. L'ajout d'une variable x à la somme est donné par cette suite

d'opérations élémentaires :

$$\begin{aligned} p_i^1 &= p_i^3 \wedge x = b_1^s \wedge x \\ p_i^2 &= p_i^1 \wedge p_i^4 = b_2^s \wedge b_1^s \wedge x \end{aligned}$$

et

$$p_i^3 = p_i^3 \oplus x = b_1^s \oplus x$$

et

$$\begin{aligned} p_i^3 &= p_i^3 \vee p_i^2 = (b_1^s \oplus x) \vee (b_2^s \wedge b_1^s \wedge x) = b_1^S \\ p_i^4 &= p_i^4 \vee p_i^1 = b_2^s \vee (b_1^s \wedge x) = b_2^S. \end{aligned}$$

et coûte 5 o.e. pour chaque variable.

Finalement, la valeur de la fonction $T_{9,3}$ est donnée par le calcul de $b_1^S \wedge b_2^S$, ce que l'on obtient en une opération élémentaire supplémentaire. Il faut au total 30 o.e. et quatre plans mémoire pour obtenir la fonction $T_{9,3}$

III. Généralisation de l'algorithme pour les fonctions seuil $T_{9,p}$

L'algorithme de cette méthode est généralisable au calcul d'une fonction seuil $T_{9,p}$. Comme précédemment, le calcul est alors effectué en trois étapes distinctes :

- calcul simultané des sommes des valeurs des pixels des trois colonnes avec saturation de la somme à la valeur p et stockage du résultat dans les cases mémoire liées au pixel central de la colonne ;

- ajout des résultats des deux colonnes extrêmes avec une saturation du résultat à la valeur p et stockage du résultat dans les cases mémoire liées au pixel central du curseur ;

- ajout successif des valeurs des pixels de la colonne centrale à la somme précédemment calculée avec saturation à la valeur p.

Si l'algorithme repose toujours sur les mêmes opérations, ces différentes opérations nécessitent une description distincte pour chaque fonction $T_{9,p}$, car la saturation est directement liée à la valeur de p. Pour chaque fonction $T_{9,p}$, pour $p = 1, 2, 4$, on présente dans le tableau suivant les coûts opératoire et mémoire :

Methodes	$T_{9,1}$	$T_{9,2}$	$T_{9,3}$	$T_{9,4}$
FND	8	71	251	503
AVB	9	36	72	108
Geometrique	6	25	30	59
Plans memoire	2	4	4	5

D'une part, les coûts opératoires comparés avec les méthodes FND et AVB s'avèrent plus intéressants, mais il est possible de trouver des méthodes moins coûteuses pour les fonctions seuil $T_{9,(1,2)}$. D'autre part, le coût mémoire est plus important.

La généralisation de cette méthode est limitée par l'obligation de redéfinir les opérations élémentaires nécessaires au calcul de chaque partie de l'algorithme pour chaque changement de valeur de p. De plus, son coût opératoire ne peut être estimé a priori et son coût mémoire devient vite exorbitant.

IV. Conclusion

Cette méthode est difficilement généralisable et s'avère très coûteuse en place mémoire.

Chapitre X

Méthode par arbre de décomposition

I. Introduction

Nous présentons dans ce dixième rapport, pour la description des fonctions seuil, une méthode introduite par Antoine Manzanera : elle allie les propriétés géométriques du curseur à une décomposition de la valeur du seuil similaire à celle de la méthode AVB. Nous exposons dans le premier paragraphe, cette nouvelle méthode et l'algorithme associé. Nous présentons en exemple la décomposition de la fonction $T_{25,12}$ sous forme d'un arbre. Nous estimons ensuite son coût opératoire dans les différentes configurations de plans mémoire et nous abordons ensuite les voies possibles pour l'optimisation de cette méthode.

II. Description de la méthode

Cette nouvelle méthode repose sur deux principes :

- sur la décomposition du curseur en une série de sous-ensembles géométriquement identiques (par exemple, les lignes ou les colonnes si le curseur est rectangulaire) formant un recouvrement de l'ensemble du curseur ;

- sur les décompositions de la valeur du seuil en une combinaison nécessaire et suffisante de seuils partiels pouvant être obtenus sur ces sous-ensembles géométriques.

La combinaison de ces deux principes permet d'obtenir la valeur de la fonction seuil en un nombre réduit d'opérations élémentaires.

En effet, le premier principe permet de tirer partie de la géométrie du curseur et d'effectuer simultanément les mêmes calculs sur l'ensemble des sous-ensembles géométriques. On peut ainsi estimer si la somme des variables de chaque sous-ensemble est plus grande qu'un seuil p_j.

Le deuxième principe est équivalent à rassembler et à combiner les résultats obtenus sur les sous-ensembles pour savoir si le seuil total p est atteint.

Pour cela, on établit préalablement (par exemple, sous forme d'arbre) toutes les décompositions de la valeur du seuil p en une combinaison de seuils partiels p_j obtenus sur ces sous-ensembles géométriques, en ne gardant que les décompositions suffisantes et nécessaires.

Du fait de l'interchangeabilité de ces sous-ensembles, il n'est pas nécessaire de différencier, comme dans la méthode AVB, les décompositions du seuil p comportant les mêmes seuils partiels p_j, mais définis sur des sous-ensembles différents. Il est donc aisé d'obtenir en peu d'opérations les conditions du type "deux parmi cinq sous-ensembles ont des sommes supérieures à trois" lorsqu'on sait si, sur chaque sous-ensemble, la condition "la somme est supérieure à trois" est ou n'est pas vérifiée.

Une fois l'ensemble de ces décompositions établi, il suffit de le parcourir en calculant successivement les conditions définies sur les sous-ensembles.

On obtient ainsi la description de la fonction seuil $T_{n,p}$. Sans perte de généralité, on considère le curseur comme carré et on découpe le curseur en colonnes. Soit n le nombre des variables du curseur et p la valeur du seuil. On établit l'ensemble des décompositions de p en au plus $\sqrt{n}$ valeurs comprises entre 1 et $\sqrt{n}$. Chaque décomposition d^i se décrit par une suite de couples (p_l^i, n_l^i) telle que :

$$p = \sum_{l=1}^{n_0} p_l^i n_l^i$$

où les p_l^i, n_l^i et n_0 sont au plus égaux à $\sqrt{n}$. En considérant que les valeurs de p_l^i représentent les seuils partiels, il suffit, pour connaître une décomposition d^i, de conserver les valeurs de $(n_l^i)_{1\leq l\leq\sqrt{n}}$ en les ordonnant par ordre décroissant de valeur de p_l. On peut ainsi ordonner l'ensemble E où chaque décomposition d^i est donnée par un $\sqrt{n}$-uplet $(n_1^i,\ldots,n_{\sqrt{n}}^i)$ tel que :

$$p \;=\; \sum_{l=1}^{\sqrt{n}} n_l^i \;\; (\sqrt{n}-l+1).$$

Pour définir une décomposition d^i, il est nécessaire d'introduire les définitions suivantes :

- $(C_j)_{1\leq j\leq\sqrt{n}}$ est la j-ème colonne du curseur ;
- $S_{c_j} \;=\; \sum_{x_l\in c_j} x_l$ est la somme des variables de la j-ème colonne ;

et de définir les conditions sur les sommes des sous-ensembles par les fonctions $R_{p_l}^{p_l}$ suivantes :

$$R_{n_l}^{p_l}(C_1,C_2,\ldots,C_{\sqrt{n}}) \;=\; \begin{cases} 1 \;\text{ si }\; \exists \;\; j_1,j_2,\ldots,j_{n_l} \;\;/\;\; S_{c_{j_k}}\geq p_l \\ 0 \;\text{ sinon} \end{cases}$$

Cette fonction détermine la condition "parmi tous les sous-ensembles, il y a au moins n_l sommes supérieures à p_l". Alors, on peut décrire une décomposition d^i comme une succession de fonctions R_p^n ; soit :

$$\begin{aligned} f_{d^i}(x_1,x_2,\ldots,x_n) \;&=\; R_{n^i_{\sqrt{n}}}^{\sqrt{n}} \wedge R_{n^i_{\sqrt{n}}+n^i_{\sqrt{n}-1}}^{\sqrt{n}-1} \wedge\cdots\wedge R_{n^i_{\sqrt{n}}+\cdots+n^i_{\sqrt{n}-l}}^{\sqrt{n}-l} \\ &=\; \bigwedge_{1\leq l\leq\sqrt{n}} R_{n^i_{\sqrt{n}}+\cdots+n^i_{\sqrt{n}-l+1}}^{\sqrt{n}-l+1} \end{aligned}.$$

La décomposition de la fonction $T_{n,p}$ étant le parcours de l'ensemble des décompositions, elle correspond au "ou" du résultat de chaque décomposition :

$$T_{n,p}(x_1,x_2,\ldots,x_n) \;=\; \bigvee_{1\leq i\;\leq n_d} f_{d^i}(x_1,x_2,\ldots,x_n)$$

où n_d est le cardinal de l'ensemble E.

A partir de cette décomposition, il est possible d'obtenir l'algorithme suivant :

- Construction de l'ensemble E des décompositions $(d^i)_{1\leq i\leq n_d}$ de p ;
- Pour chaque $d^i=(n_l^i)_{1\leq l\leq\sqrt{n}}\in E$, il faut calculer le ET du résultat des fonctions $R_{n^i_{\sqrt{n}}+\cdots+n^i_{\sqrt{n}-l+1}}^{\sqrt{n}-l+1}$.

On calcule successivement les fonctions $R_{n^i_{\sqrt{n}}+\cdots+n^i_{\sqrt{n}-l+1}}^{\sqrt{n}-l+1}$. Pour cela, il faut établir sur les sous-ensembles géométriques et de façon simultanée, les comparaisons entre les sommes partielles des variables et le seuil partiel $(\sqrt{n}-l+1)$. Ce résultat peut être obtenu en utilisant une F.N.D. sur les variables composant les sous-ensembles géométriques. Une fois obtenues, ces comparaisons sont combinées pour donner le résultat de $R_{n^i_{\sqrt{n}}+\cdots+n^i_{\sqrt{n}-l+1}}^{\sqrt{n}-l+1}$. Chaque résultat de $R_{n^i_{\sqrt{n}}+\cdots+n^i_{\sqrt{n}-l+1}}^{\sqrt{n}-l+1}$ est ajouté avec l'ensemble des résultats $R_{n^i_{\sqrt{n}}+\cdots+n^i_{\sqrt{n}-l+1}}^{\sqrt{n}-l+1}$ déjà obtenus pour cette décomposition en utilisant une opération ET.

- Le résultat final d'une décomposition est alors ajouté aux résultats précédents, car la valeur de la fonction est le "OU" du résultat de chaque décomposition d^i de l'ensemble E.

On présente maintenant l'exemple de la fonction $T_{25,12}$ et on calcule son coût opératoire.

III. Fonction $T_{25,12}$

On suppose que le curseur de la fonction $T_{25,12}$ est carré et on choisit une décomposition en colonnes.

On établit pour le seuil $p = 12$ l'ensemble E des décompositions d^i. Cet ensemble E est décrit sous la forme d'un arbre de 20 branches en annexe. Chaque décomposition correspond à une branche et est définie par le 5-uplet $(n_1, n_2, \ldots, n_5)$ où n_i est la "multiplicité" de $\sqrt{n} - l + 1 = 6 - l$. On définit ensuite chaque fonction f_{d^i} pour la succession des conditions $R^{\sqrt{n}-l+1}_{n^i_{\sqrt{n}}+\cdots+n^i_{\sqrt{n}-l+1}}$ qui sont données pour chaque branche dans l'annexe, par la condition sur $\sqrt{n} - l + 1 = 6 - l$. A partir de cette description, on peut obtenir la décomposition de la fonction $T_{25,12}$.

Pour cela, on va calculer successivement, pour chaque branche, la fonction f_{d^i}, en ne tenant compte que des conditions non nulles. Le résultat de chaque fonction f_{d^i} est ensuite ajouté (par une op'ration OU) aux précédents résultats.

On présente maintenant la définition en o.e. du calcul de la première condition de la branche 3. On calcule simultanément sur toutes les colonnes du curseur la comparaison entre la somme des variables et le seuil partiel $p = 5$ en effectuant les opérations suivantes sur les cinq variables $(x_1, x_2, \ldots, x_5)$ de chaque colonne :

$$c_j \;=\; x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5.$$

A partir de ces résultats $C_1, C_2, \ldots, C_5$ (un par colonne), on calcule la fonction

$$R^5_1 \;=\; C_1 \vee C_2 \vee C_3 \vee C_4 \vee C_5.$$

Pour obtenir le résultat de f_{d^3}, il suffit ensuite de calculer R^4_2 et R^3_3 en utilisant des FND sur les variables pour obtenir C_j et sur les résultats C_j pour obtenir R.

A partir de cette description, il est possible d'obtenir un coût en o.e. pour cette fonction dans les cas de cinq places mémoire (850 o.e.) ou de neuf places mémoire (470 o.e.). Dans les autres configurations mémoire, le coût dépend fortement de l'optimisation du parcours de l'arbre.

IV. Conclusion

Cette nouvelle méthode permet d'obtenir les fonction seuil en tirant partie de la géométrie du curseur et avec des coûts en o.e. moins importants. Cependant, elle nécessite pour chaque valeur de (n, p), l'obtention d'un arbre et dans le cas d'un nombre de places mémoire interm'diaire (entre 5 et 9), il est nécessaire d'optimiser le parcours de l'arbre selon plusieurs critères.