Projet de Semestre effectué dans la chaire ROSE du Professeur Dominique de Werra

Assistante du Projet : Daniela Bronner

Web Caching with Request Reordering

Louis Boppe

Lausanne, Février 2006

Table des matières

1 Introduction

Lorsque nous consultons une page web, nous faisons appel à des documents de toutes sortes (image, texte, vidéo, musique). Chaque appel à ces documents représente une requête et toutes les requêtes nécessitent un coût de chargement.

Nous nous demandons tout naturellement comment réduire le coût total d'une séquence de requêtes. Une idée est de garder en mémoire les documents dont nous faisons le plus appel. Pour ceci nous les mettons dans ce que nous appelerons le cache.

Dans ce projet nous nous intéressons au problème où nous ne connaissons pas à l'avance la séquence requêtes dont nous devons réduire le coût (problème online), et le cas où nous connaissons toutes les requêtes à l'avance (problème offline).

Nous présentons différents algorithmes appliqués à ces deux problèmes (online et offline). Ces algorithmes utilisent pour la plupart une technique de réordonnancement, présentée dans la suite de ce rapport.

2 Quelques Définitions et Notations

2.1 Un système de mémoire à deux niveaux

Dans un problème de Web Caching, nous distinguons deux types de mémoires. La mémoire rapide et la mémoire lente.

2.1.1 La mémoire rapide, le cache

La mémoire rapide est ce que nous appellons le cache d'un serveur web ou d'un client. Tous les documents appartenant au cache sont accessibles directement. C'est à dire que notre ordinateur peut les consulter sans coût. Nous définissons plus loin le concept de coût d'un document.

Remarque 1. *Pour maintenir l'intéret dans l'étude du problème du Web Caching, nous admettrons naturellement que le cache a une capacité restreinte, souvent fixée à K. En effet, si nous étions en présence d'un cache à capacité infinie, notre problème serait immédiatement résolu. Nous placerions tout simplement tous nos documents dans le cache, et nous n'aurions ainsi plus aucun coût de téléchargement.*

2.1.2 La mémoire lente

Contrairement à la mémoire rapide, la mémoire lente a une capacité infinie. En effet, la mémoire lente s'identifie avec l'ensemble de tous les documents accessibles sur le réseau, c'est à dire l'Internet.

2.2 Structure de la séquence de requêtes

Lorsque nous parlons d'algorithme, nous distinguons les données entrantes, input, aux données sortantes, output.

Les données entrantes soumisent à un algorithme de Web Caching sont constituées d'une suite de documents. Le but de l'algorithme étant de servir cette séquence de requêtes de maniere à ce que le coût de téléchargement soit aussi petit que possible.

Définition 1 (séquence de requêtes)**.** *Soit $m \in \mathbb{N}$. Nous disons que $\sigma = \sigma(1), \sigma(2), \cdots, \sigma(m)$ est une séquence de requêtes composée de m documents ($\sigma(i)$, $i = 1, \cdots, m$).*

Chaque document de la séquence de requêtes est caracterisé par sa taille et son coût.

La taille d'un document correspond à la place qu'il prend dans la mémoire, rapide ou lente. Généralement son unité est le *bit*. La taille d'un document est notée $size(d)$.

Dans la pratique, son coût est proportionnel à sa taille, et est noté $cost(d)$.

Remarque 2. *Si un document d appartient au cache, alors $cost(d) = 0$.*

Cette remarque découle du fait que, lorsqu'un document appartient au cache, nous n'avons plus besoin de le télécharger. C'est à dire que le coût lié au téléchargement du document est nul.

Nous définissons le coût d'un algorithme par le coût total engendré par les documents de la séquence de requêtes.

Formellement,

Définition 2. *Soit σ une séquence de requêtes.*
Nous noterons dorénavant le coût d'un algorithme par : NOMALGO(σ).

2.3 Compétitivité des algorithmes online

Nous évaluons la qualité d'un algorithme online en faisant une analyse de compétitivité.

L'idée est de comparer son coût au coût de l'un algorithme offline optimal. Un algorithme offline optimal connait l'integralité de la séquence de requêtes à l'avance, et il l'a sert avec un coût minimum. De plus, il faut que celui ci obtienne de bons résultats pour toutes les séquences de requêtes sur lesquelles il est appliqué.

Plus un algorithme online approche la solution de l'algorithme offline optimal, plus il est compétitif.

Définition 3 (c-compétitif)**.** *Soit σ une séquence de requêtes, A un algorithme online, OPT l'optimum de l'algorithme offline. Nous notons le coût de A et de OPT par $A(\sigma)$ et $OPT(\sigma)$ respectivement.*
Nous disons que l'algorithme A est c-compétitif s'il existe une constante a telle que :

$$A(\sigma) \leq c \cdot OPT(\sigma) + a \qquad \forall \sigma$$

Remarque 3. *Plus la variable c de la définition ci-dessus est petite, plus l'algorithme online A est compétitif.*

2.4 Principaux modèles

Lorsque nous nous intéressons aux performances d'un algorithme, nous l'appliquons à différents modèles caractéristiques. Avant d'appliquer un algorithme au modèle général, il est intéressant d'étudier son comportement dans des modèles plus spécifiques.

Le but de ce paragraphe est de définir les 4 principaux modèles utilisés, le modèle uniforme, le modèle bit, le modèle fault et le modèle général.

2.4.1 Le modèle uniforme

Dans le modèle uniforme, tous les documents ont la même taille, et nécessitent tous un coût de 1 lorsqu'ils ne sont pas dans le cache.

C'est à dire que :

$$size(d) = s \text{ et } cost(d) = 1 \qquad \forall \text{ document } d \text{ et avec } s \in \mathbb{R}^+$$

2.4.2 Le modèle bit

Dans le modèle bit, nous avons que :

$$cost(d) = size(d) \qquad \forall \text{ document } d$$

C'est à dire que nous mesurons le nombre de bits nécessaire au transfert de chaque document n'appartenant pas au cache.

2.4.3 Le modèle fault

Dans le modèle "fault", nous avons que :

$$cost(d) = 1 \qquad \forall \text{ document } d$$

C'est à dire que nous comptons le nombre de requêtes correspondant à un document n'appartenant pas au cache que nous servons.

2.4.4 Le modèle général

Dans le modèle général, nous attribuons à tous les document d, un coût, $cost(d)$, appartenant a $\mathbb{R}^+$.

3 Les Algorithmes Online

Dans notre étude des algorithmes online, nous nous restreindrons au modèle général.

Cette restriction est motivée par le fait que nous avons un algorithme déterministe

(qui produit un seul calcul) qui nous permet de résoudre le probleme de r-réordonnancement du modèle général, l'algorithme de Landlord modifié.

Nous présenterons tout d'abord l'algorithme de Landlord, qui s'applique dans le cas de non réordonnancement.

3.1 Sans Réordonnancement

Nous parlons d'un algorithme sans réordonnancement lorsque nous ne pouvons pas changer l'ordre des requêtes, c'est à dire que nous devons servir les requêtes dans leur ordre défini par la séquence initiale.

Nous nous intéressons à un algorithme sans réordonnancement en particulier, l'algorithme de Landlord.

3.1.1 L'Algorithme de Landlord [PLAXTON]

Pour cet algorithme, nous associons à chaque document appartenent au cache un *credit* .

$$credit(d) \in [0; cost(d)] \qquad \forall \quad d \in \text{ Cache}$$

Pour chaque requête g, procéder à cet algorithme :

Données : Une séquence de requêtes σ
si *la requête g n'est pas dans le cache* **alors**
  Soit $\triangle = \min_{f \in \text{Cache}} credit(f)/size(f)$;
  tant que *il n'y a pas de place pour g dans le cache* **faire**
    Pour tout document f appartenant au cache, $credit(f) = credit(f) - \triangle \cdot size(f)$;
    Enlever du cache tous les documents f avec $credit(f) = 0$;
  fin
  Mettre g dans le cache ;
  $credit(g) \leftarrow cost(g)$;
fin
sinon
  C'est à dire si g est déjà dans le cache. Redéfinir $credit(g)$ avec n'importe quelle valeur entre la sienne et $cost(g)$
fin

Algorithme 1 : Algorithme de Landlord

Remarque 4 (dernière étape de l'algorithme)**.** *Cette dernière étape n'est pas nécessaire dans notre analyse, mais elle est importante dans la pratique.*

Dans le cas d'une stratégie de least-recently-used (le document enlevé du cache est celui dont la dernière requête servie correspondant à ce document est la plus lointaine) , nous tendrons à augmenter le crédit le plus possible.

Dans le cas d'une stratégie de first-in-first-out (le document enlevé du cache est celui qui y est depuis le plus longtemps) , nous n'augmenterons pas le crédit.

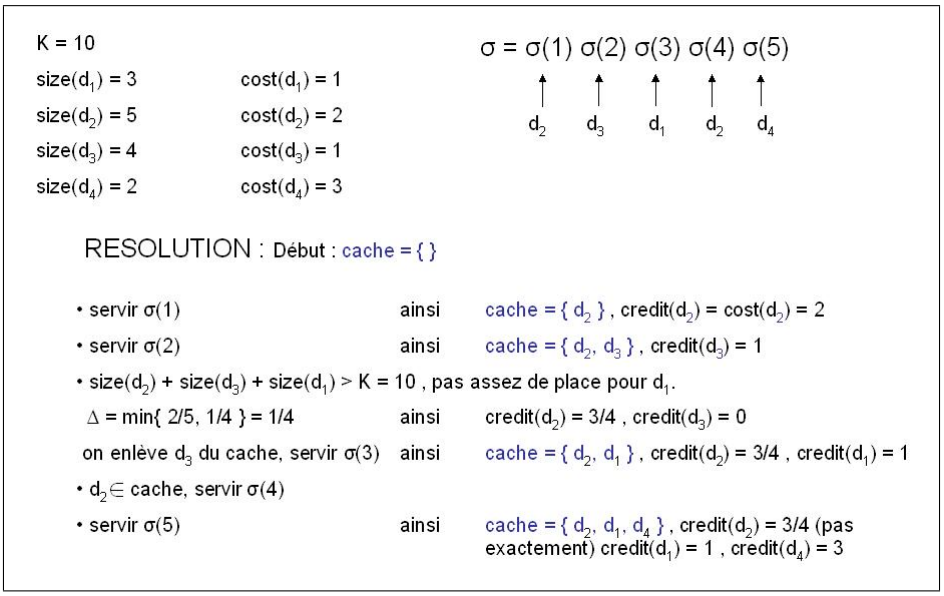

FIG. 1 – Exemple Algorithme de Landlord

Dans l'exemple ci-dessus nous avons choisi un cache de taille $K = 10$ et vide au début.

Nous servons les requêtes dans leur ordre d'arrivée.

3.2 Avec Réordonnancement

Dans le cas d'algorithme online avec réordonnancement, la séquence de requêtes n'est pas obligatoirement servie selon l'ordre d'arrivée des documents. Cependant, il n'est naturellement pas souhaitable qu'un document reste non servi trop longtemps. C'est pourquoi nous définissons le r-réordonnancement.

Définition 4 (r-réordonnancement)**.** *Soit $r \in \mathbb{N}$, A un algorithme online et σ une séquence de requêtes. Pour un algorithme A appliqué à un problème de r-réordonnancement, le document $\sigma(j)$ peut être servi avant le document $\sigma(i)$ si $j - i < r$.*

Remarque 5. *Soit $\sigma(i)$ le premier document non servi de la séquence de requête. Pour qu'un probleme de r-réordonnancement soit justifié, nous admettrons que les documents $\sigma(j)$ avec $j - i < r$ sont connus et que les documents $\sigma(j)$ avec $j - i \geq r$ sont inconnus.*

Regardons maintenant un exemple d'un problème de 3-réordonnancement.

Exemple 1 (r=3)**.** *Soit σ la séquence de requêtes.*

Supposons que le premier document non servi soit $\sigma(3)$.

Ainsi les documents connus et pouvant être servis avant $\sigma(3)$ sont $\sigma(4)$ et $\sigma(5)$. Les documents $\sigma(j)$ avec $j \geq 6$ ne sont pas connus et ne peuvent pas être servis avant $\sigma(3)$.

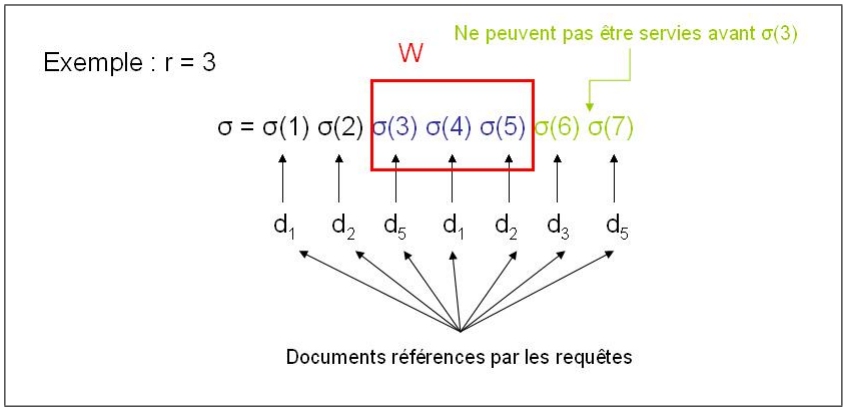

FIG. 2 – Exemple de 3-réordonnancement

3.2.1 L'Algorithme Modifié de Landlord

L'algorithme présenté ci-dessous est une amélioration de l'Algorithme de Landlord décrit plus haut.

Cet algorithme est déterministe, c'est à dire que son résultat n'est pas du au hasard, n'est pas aléatoire. A partir d'un input particulier, il produira toujours le même output. Il permet d'étudier des probleme de r-réordonnancement dans le cas du modele général.

Initialement,

$$credit(d) = 0 \qquad \forall \quad d$$

De plus, étant donné une séquence de requêtes σ, l'algorithme va maintenir une fenêtre W contenant r requêtes consécutives de σ.

Au début W contient les r premières requêtes de la séquence σ.

Données : Une séquence de requêtes σ et une fenêtre W contenant r requête consécutives de σ

1 Tout d'abord, nous servons toutes les requêtes dans W qui correspondent à un document appartenant au cache ;
2 **si** *la première requête dans* W *n'est pas servie* **alors**
3 Soit d le document correspondant à cette première requête ;
4 Servir toutes les requêtes qui correspondent à d dans W ;
5 $credit(d) \leftarrow cost(d)$ et $C \leftarrow \{d\} \cup \{d' \mid d' \in cache\}$;
6 **tant que** $\sum_{d' \in C} size(d') > K$ **faire**
7 Soit $\triangle = \min_{d' \in C} credit(d')/size(d')$;
8 Pour tous les documents $d' \in C$, $credit(d') = credit(d') - \triangle \cdot size(d')$;
9 Enlever de C et du cache tous les documents d' avec $credit(d') = 0$;
10 **fin**
11 **si** $credit(d) > 0$ **alors**
12 Mettre d dans le cache ;
13 **fin**
14 **fin**
15 Déplacer la fenêtre W d'une position vers la droite ;

Algorithme 2 : Algorithme Modifié de Landlord

Remarque 6. *Il est important de remarquer que C n'est pas le cache. En effet, C peut être considéré comme étant un cache intermédiaire. Dans l'algorithme MLL, nous avons besoin de C pour calculer le rapport Δ. La différence entre C et le cache réside dans le fait que le document candidat au cache appartient a C avant même de savoir s'il sera mis dans le cache ou pas.*

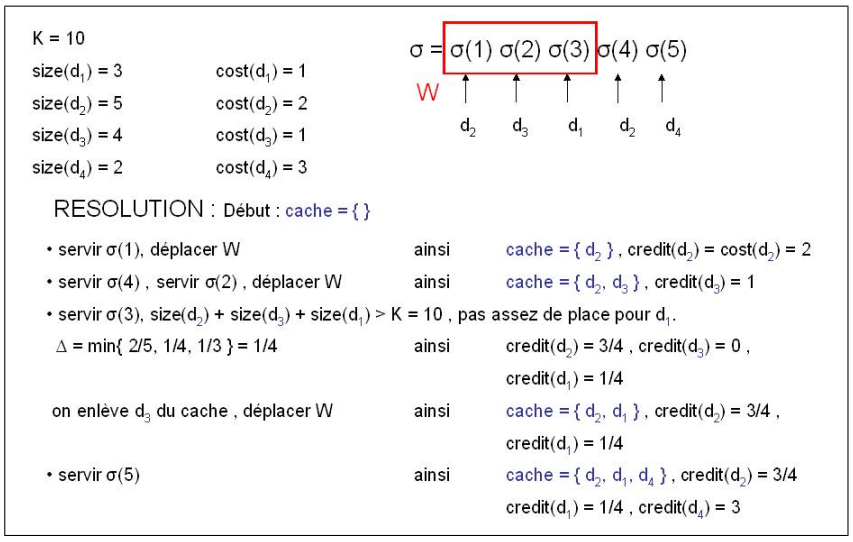

FIG. 3 – Exemple Algorithme de Landlord Modifié

séquence des requêtes servies : $\sigma(1)$, $\sigma(4)$, $\sigma(2)$, $\sigma(3)$, $\sigma(5)$

Définition 5 (Potentiel)**.** *Définissons la fonction* Φ *tel que :*

$$\Phi = k \sum_{d' \in D} (credit(d')) + (k+1) \sum_{d' \in OPT \cup S} (cost(d') - credit(d'))$$

Où :

D : l'ensemble de tous les documents référencés par σ*, la séquence de requêtes*

OPT : l'ensemble de tous les documents dans le cache de l'OPT, l'OPT étant un algorithme offline optimal.

S : l'ensemble de tous les documents qui ne sont pas dans OPT, mais pour lesquels au moins une requête a déjà été servie par l'algorithme OPT mais pas par l' algorithme MLL.

Comme $credit(d') \in [0, cost(d')]$ *alors la fonction* $\Phi \geq 0$*.*

Nous allons maintenant tester l'efficacité de l'algorithme MLL par rapport à l'algorithme offline optimal. Cette efficacité est déterminée par une k-compétitivité. Cette même compétitivité sera étudiée grâce à notre fonction de potentiel définie plus haut.

Nous voulons démontrer le théorème suivant :

Théoreme 1 (compétitivité de l'algorithme MLL)**.** *L'algorithme Modifié de Landlord, MLL, est* $(k+1)$*-competitif.*

Pour cela nous allons tout d'abord démontrer deux résultats intermédiaires.

Proposition 1. *Soit une séquence de requêtes* σ *et OPT un algorithme offline optimal.*

Si OPT sert les requêtes correspondant à un document d avec un coût égal à $cost(d)$*, alors la fonction* Φ *augmente d'au plus* $(k+1) \cdot cost(d)$*.*

Toutes les autres actions de OPT ne font pas augmenter Φ*.*

Remarque 7. *Dans notre analyse, nous supposerons que l'algorithme OPT sert les requêtes en premier et MLL en deuxième.*

Preuve :

Supposons que OPT sert une requête correspondant à un document d avec un coût égal à $cost(d)$, cela implique que d n'est pas dans l'ensemble OPT, étant le cache de l'algorithme OPT, avant d'être servi.

Ensuite, ce document peut être mis dans le cache ou non.

Dans le cas où il est mis dans le cache, $d \in OPT$.

Dans le cas où il n'est pas mis dans le cache, nous avons $d \in S$. Car la requête

qui vient d'être servie correspond au document d.

Ainsi après que OPT sert la requête correspondant au document d,

$$d \in OPT \text{ ou } d \in S \Rightarrow \quad d \in OPT \cup S.$$

Ceci entraine une variation sur Φ de

$$(k+1) \cdot (cost(d) - credit(d)) \leq (k+1) \cdot cost(d) \text{ car } credit(d) \in [0, cost(d)]$$

Supposons maintenant que OPT sert une requête avec un coût nul. Alors le document $d \in OPT$, et donc la fonction potentiel n'augmente pas.

Supposons enfin que OPT retire un document du cache. Ceci ne peut que diminuer la valeur de la fonction Φ.

□

Intéressons nous maintenant au comportement de l'algorithme MLL.

Remarque 8. *Nous rappelons ici une propriété importante de l'algorithme MLL :*

$$credit(d) = 0 \quad \forall \quad d \notin \ cache$$

Nous devons démontrer le résultat suivant pour l'algorithme MLL.

Proposition 2. *Soit une séquence de requêtes σ et l'algorithme MLL l'algorithme modifié de Landlord.*

Si MLL sert une requête correspondant à un document d avec un coût égal à $cost(d)$, alors la fonction Φ diminue d'au moins $cost(d)$.

Plus généralement, tout le long de l'algorithme MLL, la fonction Φ diminue d'au moins $cost(d)$.

Preuve :

A la première ligne de l'algorithme MLL, il n'y a aucun changement positif pour Φ, car le seul changement possible est que d ne soit plus dans S. Ce qui entraine une diminution de la valeur de Φ.

Juste avant la quatrième ligne, $d \in OPT \cup S$, car $\sigma(t)$ correspondant au documend d a été servie par OPT.

Pendant la quatrième ligne, $d \notin S$, ce qui implique que Φ diminue de $(k+1) \cdot cost(d)$ (à ce moment $credit(d) = 0$).

A la cinquième ligne, $credit(d) = cost(d)$, ce qui implique que Φ augmente

de $k \cdot cost(d)$.

Donc après les cinq premières lignes de l'algorithme MLL, la fonction potentiel Φ a une variation de :

$$k \cdot cost(d) - (k+1) \cdot cost(d) = -cost(d)$$

Ce qui correspond à une diminution de $cost(d)$.

Si nous montrons que Φ n'augmente pas durant la boucle de l'algorithme, alors nous avons démontré notre resultat.

Définissons pour une partie $X \subseteq D$

$$size(X) = \sum_{d' \in X} size(d').$$

A la huitième ligne, Φ subit une variation de

$$\triangle \cdot \text{-}k \cdot size(C) + \triangle \cdot (k+1) \cdot size(C \cap (OPT \cup S))$$

Or nous avons $C \cap S = \varnothing$ car quand la huitième ligne est éxécutée, l'algorithme MLL a servi toutes les requêtes dans W qui font référence à un document appartenant à C et OPT ne peut pas servir des requêtes antérieures à celles de W. En d'autres termes, il ne peut plus y avoir de documents dans C qui n'ont pas été servies par l'algorithme MLL.

Comme

$$C \cap (OPT \cup S) = (C \cap OPT) \cup (C \cap S) = C \cap S$$

et que

$$size(C) + size(C \cap OPT) = size(C \setminus OPT)$$

Alors, Φ subit une variation de

$$\triangle \cdot \text{-}k \cdot size(C \setminus OPT) + \triangle \cdot size(C \cap OPT) \leq \triangle \cdot \text{-}k \cdot size(C \setminus OPT) + \triangle \cdot K$$

Remarquons ensuite que $C \setminus OPT \neq \varnothing$.

En effet, si $d \notin OPT$, alors $d \in C \setminus OPT$.

Par contre, si $d \in OPT$, alors il devrait y avoir un document d' tel que $d' \in C$, $d' \neq d$, avec $d' \notin OPT$. Car sinon $C \subseteq OPT$, ce qui impliquerait que $size(C) \leq K$, et donc que la boucle de notre algorithme n'aurait jamais démarée !

Finalement durant la boucle, Φ subit une variation de

$$\triangle(\text{-}k \cdot D_{min} + K) = 0 \text{ donc } \triangle(\text{-}k \cdot D_{min} + K) \leq 0$$

avec $k = \frac{K}{D_{min}}$.

Ceci termine notre démonstration car à la neuvième ligne, tous les documents retirés de C ont un crédit nul, ce qui signifie aucun changement pour la fonction potentiel Φ.

□

Revenons maintenant à la démonstration du théorème :

Preuve :

de la compétitivité de l'algorithme MLL.

A l'aide des deux dernières propositions, la démonstration est naturelle.

En effet, d'un coté l'algorithme OPT fait augmenter Φ d'au plus $(k+1)\cdot cost(d)$.

De l'autre coté, l'algorithme MLL fait diminuer Φ d'au moins $cost(d)$.

$$\frac{(k+1)\cdot cost(d)}{cost(d)} = k+1$$

D'où la $(k+1)$ -compétitivité de l'algorithme MLL.

□

Remarque 9. *Dans la section des algorithmes online avec réordonnancement, nous n'avons pas étudié les modèles cas par cas.*

Ceci est du au fait que l'Algorithme Modifie de Landlord est appliquable directement au modèle général.

4 Les Algorithmes Offline

Lors de notre étude sur la performance des algorithmes online, nous avons trouvé une borne supérieure qui dépend de la performance du résultat d'un algorithme offline optimal.

Notre motivation pour l'étude des algorithmes offline découle directement de ce résultat. Plus nos résultats sur les algorithmes offline seront meilleurs, plus la borne supérieure des algorithmes online sera basse.

Dans cette partie, nous construirons des algorithmes à temps polynomial (l'algorithme fournit le résultat en un nombre fini d'opérations élémentaires). Ceux ci sont basés sur une technique qui transforme le problème de r-réordonnancement en un problème de planification de tâches en batch.

Nous allons présenter cette technique de processus en batch et ensuite nous développerons cette technique à travers des algorithmes dans nos différents modèles.

4.1 Les processus en batch

Dans cette technique, nous conservons la fenêtre W présentée dans les problèmes de réordonnancement. Par conséquent, toutes les requêtes qui sont à gauche de W ont déjà été servies, et les requêtes se trouvant dans W peuvent être réordonnées.

Nous disons qu'un algorithme A sert une séquence de requêtes σ selon un processus en batch si :

$\forall \quad i = 0, 1, \cdots, \lfloor m/r \rfloor$: Quand $\sigma(ir+1)$ est la première requête de W , alors A sert toutes les requêtes $\sigma(ir+1), \sigma(ir+2), \cdots, \sigma(min\{ir+r, m\})$

Définition 6. *On appelle un batch i, une suite (de taille $\leq r$) de requêtes consécutives dans la séquence de requêtes σ.*

Dans notre cas, $\sigma(ir+1), \sigma(ir+2), \cdots, \sigma(min\{ir+r, m\})$ forment le batch i.

Lorsque $\sigma(ir+1)$ est la première requête de W, aucune des requêtes du batch i n'est servie. L'algorithme sert toutes les requêtes du batch avant de déplacer la fenêtre W à $\sigma((i+1)r+1)$ si $i < \lfloor m/r \rfloor$.

Voici une illustration de ceci à l'aide d'un découpage en batch d'une séquence de requêtes, avec un 4-réordonnancement.

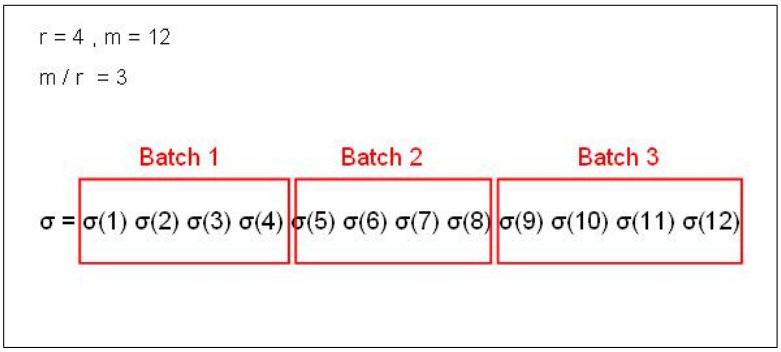

FIG. 4 – Découpage en batch de σ avec un 4-réordonnancement

Maintenant que nous connaissons cette technique de processus en batch, nous aimerions trouver un résultat pour notre problème de r-réordonnancement.

Nous montrerons le résultat suivant :

Lemme 1. *Soit A un algorithme servant une séquence de requêtes σ selon le modèle de r-réordonnancement standard, avec $A(\sigma) = C$ (un coût).*

Alors il existe un algorithme A' servant une séquence de requêtes σ selon le processus en batch, avec $A'(\sigma) \leq 2 \cdot C$.

Preuve :

L'idée est de transformer notre algorithme de base A, en un algorithme A' avec les propriétés désirées. C'est à dire que A' fonctionnera selon le processus en batch.

Définissons tout d'abord quelques ensembles :

Soit B_i l'ensemble des documents référencés par les requêtes du batch i.

Soit S_i l'ensemble des documents qui sont dans le cache de A quand $\sigma(ir+1)$ est la première requête de W $\forall \quad 0 \leq i \leq \lfloor m/r \rfloor$. Supposons, sans perte de généralité, qu'au début $S_0 = \varnothing$. De même, $S_{\lfloor m/r \rfloor + 1}$ correspond à la configuration finale du cache de A.

Soit $D_i \subseteq B_i$ l'ensemble des documents tel que si $d \in D_i$, alors $d \notin S_i \cup S_{i+1}$. C'est à dire que les documents appartenant à D_i sont soit chargés dans le cache pendant le batch i et dans ce cas ils sont enlevés avant la fin du batch, soit jamais chargés dans le cache pendant le batch i.

Au cours du batch i, seuls les documents $d \in D_i$ et $d \in S_{i+1} \setminus S_i$ engendrent un coût pour l'algorithme A.

Ainsi,

$$A(\sigma) \geq \sum_{i=0}^{\lfloor m/r \rfloor} \Big(\sum_{d \in S_{i+1} \setminus S_i} cost(d) + \sum_{d \in D_i} cost(d) \Big)$$

A présent, construisons l'algorithme A'.

Mettons nous dans le cas où $\sigma(ir+1)$ est la première requête de W. Alors, A' sert les requêtes correspondant aux documents appartenant à S_i. Cela n'engendre aucun coût.

Puis A' sert les requêtes correspondant aux documents appartenant à $S_{i+1} \setminus S_i$. Cela engendre un coût de $\sum_{d \in S_{i+1} \setminus S_i} cost(d)$.

De plus A' sert les requêtes correspondant aux documents appartenant à D_i, sans les mettre dans le cache. Cela engendre un coût de $\sum_{d \in D_i} cost(d)$.

Enfin, si $i > 0$, A' sert les requêtes correspondant aux documents appartenant à $D_{i-1} \cup (S_{i-1} \setminus S_i)$, sans les mettre dans le cache. $D_{i-1} \cup (S_{i-1} \setminus S_i)$ sont les documents qui sont dans le cache ou non au début du batch $i-1$ et qui ne le sont pas au début du batch i. Cela engendre un coût de $\sum_{d \in D_{i-1}} cost(d) + \sum_{d \in S_{i-1} \setminus S_i} cost(d)$.

Remarquons que A sert toutes les requêtes de σ et qu'il ne peut servir que les r prochaines requêtes (en partant par exemple de $\sigma(ir+1)$). Donc tous les

documents référencés par des requêtes du batch i sont dans $S_i \cup S_{i+1} \cup D_i$ ou dans $S_{i-1} \cup D_{i-1}$ si $i > 0$.

C'est à dire que

$$B(i) = \begin{cases} S_i \cup S_{i+1} \cup D_i, \\ S_{i-1} \cup D_{i-1}, \qquad \text{si } i > 0 \end{cases}$$

Ce qui montre que A' sert bien toutes les requêtes de σ. L'algorithme est bien construit.

Il ne nous reste plus qu'a montrer que

$$A'(\sigma) \leq 2 \cdot C$$

Or

$$A'(\sigma) = \sum_{i=0}^{\lfloor m/r \rfloor} (\sum_{d \in D_i} cost(d) + \sum_{d \in S_{i+1} \setminus S_i} cost(d)) + \sum_{i=1}^{\lfloor m/r \rfloor} (\sum_{d \in D_{i-1}} cost(d) + \sum_{d \in S_{i-1} \setminus S_i} cost(d))$$

Donc si nous montrons que :

$$A'(\sigma) \leq 2 \cdot \sum_{i=0}^{\lfloor m/r \rfloor} (\sum_{d \in D_i} cost(d) + \sum_{d \in S_{i+1} \setminus S_i} cost(d))$$

Nous avons fini puisque pour montrer l'inégalité ci-dessus, il faut montrer l'inégalité suivante :

$$\sum_{i=1}^{\lfloor m/r \rfloor} \sum_{d \in S_{i-1} \setminus S_i} cost(d) \leq \sum_{i=0}^{\lfloor m/r \rfloor} \sum_{d \in S_{i+1} \setminus S_i} cost(d)$$

Nous remarquons que l'ensemble $S_{i-1} \setminus S_i$ correspond aux documents que nous enlevons du cache et que l'ensemble $S_{i+1} \setminus S_i$ correspond aux documents que nous mettons dans le cache.

Cette inegalité découle du fait qu'à chaque fois que nous mettons un document dans le cache, nous en enlevons au plus un.

□

4.2 Le modèle uniforme

Rappelons que dans le modèle uniforme, $size(d) = s \in \mathbb{N} \qquad \forall \quad d$ et $cost(d) = 1$.

Définition 7. *Considérons un algorithme qui sert une séquence de requêtes selon un processus en batch.*

Nous définissons, pour tout document d et à tout moment de l'algorithme, l'index du batch où nous aurons la prochaine requête correspondant au document d :

L'index d'un document est noté, b(d).

Par convention, si un document d n'est plus demandé, alors son index $b(d) = \lfloor m/r \rfloor + 1$ avec m : la longueur de la séquence de requêtes.

Définition 8. *Nous appellerons* "service operation" *une opération par laquelle un algorithme sert une requête correspondant à un document dans le batch actuellement servi.*

Données : Une séquence de requêtes σ servie selon un processus en batch
pour tous les *batches* $i = 0, \cdots, \lfloor m/r \rfloor$ **faire**
    Tout d'abord, nous servons toutes les requêtes du batch i qui correspondent à un document appartenant au cache ;
    tant que *il reste un document d non servi dans le batch* **faire**
        Servir toutes les requêtes du batch i correspondant au document d ;
        Définir $b = max_{d' \in S} b(d')$ avec S l'ensemble des documents actuellement dans le cache ;
        si $b(d) < b$ **alors**
            Mettre d dans le cache ;
            Retirer tous les documents d' avec $b(d') = b$;
        fin
    fin
fin

Algorithme 3 : Algorithme BMIN

Remarque 10. *Notons que l'algorithme BMIN n'enlève pas forcément du cache le document dont la prochaine requête non servie est la plus lointaine, mais n'importe quel document ayant sa prochaine requête dans le même batch de cette plus lointaine prochaine requête.*

Nous montrerons que l'algorithme BMIN est optimal parmi ceux qui procèdent par batch.

Pour cela, nous avons besoin d'une lemme technique que nous énoncerons sans démonstration.

Lemme 2. *Soit A un algorithme procédant selon un processus en batch, et σ une séquence de requêtes quelconque.*

S : l'ensemble des documents actuellement dans le cache.
C : coût de A après le temps t jusqu'a la fin de σ
Soient $d \in S$ et $d' \notin S$ deux documents avec $b(d') \leq b(d)$.

Alors il existe un algorithme A' qui commence au temps t avec le même ensemble de requêtes servies que A et avec $Cache = (S - \{d\}) \cup \{d'\}$. $A'(\sigma) \leq C$ (A' commence au temps t, donc le coût C correspond au coût de t à la fin de σ).

A' opère aussi en batch, les mêmes batches utilisés pour A.

Théoreme 2. *Pour toute séquence de requêtes σ, le coût engendré par BMIN est le plus petit parmi ceux engendrés par des algorithmes procédant selon un processus en batch.*

Preuve :

Soit une séquence de requêtes σ quelconque et A l'algorithme engendrant le coût minimum.

Supposons que les l premières *service operation* de BMIN et de A sont les mêmes, mais que la $(l+1)$-ème est différente.

C'est à dire que A et BMIN ont le même cache après les l premières *service operation* et ils ont servis les mêmes requêtes.

Au départ, l peut être égal à 0.

Introduisons une notation :

$(l+1)^{ALGO}$ est la $(l+1)$-ème *service operation* d'ALGO.

Nous allons montrer qu'il existe un algorithme A' avec $(l+1)^{A'} = (l+1)^{BMIN}$ et où $A'(\sigma) \leq A(\sigma)$.

En faisant une récurrence sur l, le résultat du théorème sera prouvé.

Supposons que la $(l+1)$-ème *service operation* de BMIN : $(l+1)^{BMIN} \in Cache$, alors nous avons fini, car le coût engendré est nul.

Supposons maintenant que $(l+1)^{BMIN} \notin Cache$. Donc A et BMIN ont déjà servis toutes les requêtes correspondant à des documents appartenant au cache. Admettons que les deux algorithmes servent le même document d pendant la $(l+1)$-ème *service operation*.

L'algorithme A' est construit tel que après les $(l+1)$ premières *service operation*, le cache de A et de A' diffère d'un document. Soit $d_1 \in Cache_A$ et $d'_1 \in Cache_{A'}$.

Nous sommes face à trois différents cas.

Nous allons montrer que dans chacun de ces cas, $b(d'_1) \leq b(d_1)$. Ainsi grâce au résultat du lemme précédent, le théorème sera prouvé.

- BMIN et A mettent tous les deux le document d dans leur cache en enlevant toutefois un document différent. Ici BMIN et A' enlèvent d_1 de leur cache, tandis que A enlève d'_1. Mais comme BMIN enlève d_1 au lieu de d'_1, alors forcément $b(d'_1) \leq b(d_1)$.
- BMIN met d dans son cache, A non. Ici BMIN et A' enlèvent d_1 de leur cache, tandis que A le garde. Soit $d'_1 = d$. Dès lors, nous avons par

définition de BMIN que $b(d'_1) < b(d_1)$.
- A met d dans son cache, BMIN non. Ici A enlève d'_1, et de même nous avons par définition de BMIN que $b(d'_1) < b(d_1) = b(d)$.

□

Le résultat suivant suit directement du Lemme 1 et du Théorème démontrés plus haut.

Corollaire 1. *L'algorithme BMIN est* 2 *- compétitif.*

4.3 Le modèle bit

Rappelons que le modèle bit est tel que $cost(d) = size(d) \quad \forall \quad d$.

Le prochain algorithme que nous allons présenter, l'algorithme BBMIN, est un algorithme qui donne premièrement une solution "fractionnaire", où les documents peuvent être dans le cache que partiellement et dans un deuxième temps l'algorithme arrondit la solution partielle à une solution entière admissible.

Définition 9. *Soit d un document. Nous définissons le nombre de bits de d présent dans le cache (fraction du document d) comme une valeur comprise entre 0 et $size(d)$, notée $c(d)$.*

$$0 < c(d) < size(d) \quad \forall \quad d$$

.

Remarque 11. *Le coût pour servir une requête correspondant à un document d devient :*

$$cost(d) = cost(d) - c(d)$$

La stratégie pour concevoir les solutions partielles découle directement de l'algorithme BMIN. En effet, comme pour l'algorithme BMIN, nous définissons un index pour tout document de la séquence de requêtes.

Définition 10. *Soit une séquence de requêtes σ. Pour tout document d dans le batch i, $b(d, i)$ est le plus petit index j, avec $j > i$, tel qu'il existe une requête correspondant à d appartenant au batch j.*

Par convention, si un document d n'est plus demandé, alors $b(d, i) = \lfloor m/r \rfloor + 1$ avec m : longueur de la séquence de requêtes.

Données : Une séquence de requêtes σ servie selon un processus en batch
pour tous les *batches* $i = 0, \cdots, \lfloor m/r \rfloor$ **faire**
    Tout d'abord, nous servons toutes les requêtes dans le batch i qui correspondent à un document appartenant au cache, en entier ou en partie ;
    Servir les requêtes correspondants aux documents qui ne sont pas dans le cache ;
    pour tous les *document d qui n'est pas entièrement dans le cache* **faire**
        Soit S l'ensemble des documents en entier ou en partie dans le cache. ;
        Définir $b = \max_{d' \in S} b(d', i)$;
    fin
    tant que $b(d, i) < b$ *et* $c(d) < size(d)$ **faire**
        Retirer $\beta = \min\{size(d) - c(d), c(d')\}$ bits d'un document d' avec $b(d', i) = b$;
        Mettre β bits manquants de d dans le cache ;
        Réinitialiser $b = \max_{d' \in S} b(d', i)$;
    fin
fin

Algorithme 4 : Algorithme BBMIN

Nous voulons à présent montrer que l'algorithme BBMIN est le plus performant de tous les algorithmes qui procèdent en batch donnant une solution fractionnaire.

Les algorithmes procédant en batch et ayant une solution entière sont notés OPT^B. Ainsi le résultat devient,

Lemme 3.

$$BBMIN(\sigma) \leq OPT^B(\sigma) \quad \forall \sigma$$

Avant de commencer la preuve de ce Lemme, nous montrerons un autre résultat qui nous sera utile.

Proposition 3. *La solution construite par l'algorithme BBMIN est la même que celle construite par l'algorithme BMIN.*

En d'autres termes, BBMIN et BMIN ont, après chaque batch, des caches identiques.

Preuve :

Considérons un batch i quelconque. BMIN n'impose pas de regle en ce qui concerne l'ordre de service entre les requêtes correspondantes à des bits appartenant au cache et celles correspondantes à des bits n'appartenant pas au cache.

Cependant, nous pouvons décrire deux parties lors de l'éxécution d'un batch par l'algorithme BMIN.

La partie 1 du batch i : Elle correspond au service des requêtes correspondant à des bits appartenant au cache. Elles ne sont pas servies selon l'ordre croissant de la valeur de leur $b(d, i)$, d étant le document auquel les bits demandés appartiennent. De plus, les requêtes correspondantes à des bits appartenant au même document sont regroupées.

La partie 2 du batch i : Elle correspond au service des requêtes correspondant à des bits n'appartenant pas au cache. Les bits appartenant à des documents intervenant dans la partie 1 sont servis en priorité. Elles non plus ne sont pas servies selon l'ordre croissant de la valeur de leur $b(d, i)$, et les requêtes correspondant à des bits appartenant au même document sont aussi regroupées.

Considérons maintenant les documents d intervenant à la fois dans la partie 1 et dans la partie 2 du batch i.

Ces documents doivent être en partie dans le cache au début du batch. Notons que BMIN n'enleve jamais du cache des bits d'un document d' au profit d'un document d, si l'index du prochain batch ou intervient d' est inférieur à celui de d. C'est à dire si $b(d', i) < b(d, i)$.

Ainsi, BMIN et BBMIN ont les mêmes règles en ce qui concerne le retrait de bits appartenant au cache.

Nous en déduisons que la configuration de leur cache est la même après le batch i.

Enfin, comme nous avons pris un batch i quelconque, $BBMIN(\sigma) = BMIN(\sigma)$.
□

Preuve : du Lemme

Soit $OPT_F^B(\sigma)$ le coût optimal d'un algorithme procédant en batch et ayant une solution fractionnaire.

Nous avons clairement que

$$OPT_F^B(\sigma) \leq OPT^B$$

car toute solution entière peut être considérée comme une solution fractionnaire.

Construisons un modèle bit complet, c'est à dire que tous les documents d sont constitués de $size(d)$ bits correspondants à autant de requêtes dans la séquence de requêtes. Soit $OPT_B^B(\sigma)$ le coût optimal des algorithmes selon le modèle bit complet.
Nous avons,

$$OPT_B^B(\sigma) \leq OPT_F^B$$

Or le résultat du Lemme 2 implique que,

$$BMIN(\sigma) = OPT_B^B(\sigma)$$

Finalement, grâce à la proposition précédente, nous avons le résultat souhaité.
□

Nous allons à présent nous intéresser à un algorithme générant une solution à valeurs entières à partir d'une solution fractionnaire provenant de BBMIN. Il a pour nom Algorithme Rounding.

Tout d'abord reprenons la définition de $c(d)$, représentant le nombre de bits présent dans le cache, et modifions la légèrement.

Définition 11. *Soit d un document. Nous définissons le nombre de bits de d présent dans le cache après le batch i comme une valeur comprise entre 0 et $size(d)$, notée $c(d, i)$.*

$$0 < c(d, i) < size(d) \qquad \forall \quad d$$

Définition 12. *Soit B_i l'ensemble des documents référencés par les requêtes du batch i.*

Nous aimerions modifier BBMIN de telle facon à ce que la proportion d'un document d présente dans le cache ne change pas pendant le batch i et le batch j où $j = b(d, i)$.

Pour ceci nous allons contraindre $c(d, j)$ à etre égal à $c(d, b(d, i) - 1)$ $\forall j = i, \cdots, b(d, i) - 2$. De plus nous libérons de la mémoire dans le cache sans augmenter le coût.

D'autre part, nous avons la propriété suivante :

Propriété 1. *Soit d et d' deux documents avec $d \in B_i$, $d' \in B_j$ et $i \leq j$.*

Si $b(d', j) < b(d, i)$ et $c(d, l) > 0 \qquad \forall \quad l = i, \cdots, b(d, i) - 1.$

Alors $c(d', l) = size(d') \qquad \forall \quad l = j, \cdots, b(d', j) - 1.$

Cette propriété est dûe au fait qu'entre deux documents, BBMIN privilégie au niveau du retrait du cache celui dont la prochaine requête intervient le plus tôt.

La solution de l'algorithme Rounding nécessite une mémoire du cache d'au plus $K + \delta D_{max}$, où on a noté par D_{max} la taille maximale de l'ensemble des documents, et où $0 < \delta \leq 1$.

Voici l'Algorithme Rounding :

Données : Une séquence de requêtes σ servie selon un processus en batch

1 $Extra \leftarrow 0$;
2 **pour** $i \leftarrow 0$ *a* $\lfloor m/r \rfloor$ **faire**
3 **pour tous les** $d \in B_i$ *avec* $(1 - \delta) \cdot size(d) < c(d,i) < size(d)$ **faire**
4 **si** $Extra + size(d) - c(d,i) \leq \delta \cdot D_{max}$ **alors**
5 $Extra \leftarrow Extra + size(d) - c(d,i)$;
6 $c(d,i) \leftarrow size(d)$, pour $j = i \ , \cdots \ , b(d,i) - 1$;
7 **sinon**
8 $Extra \leftarrow Extra - (c(d,i) - (1 - \delta) \cdot size(d))$;
9 $c(d,i) \leftarrow (1 - \delta) \cdot size(d)$, pour $j = i \ , \cdots \ , b(d,i) - 1$;
10 **fin**
11 **fin**
12 **fin**
13 **pour tous les** $c(d,i)$ *avec* $0 < c(d,i) \leq (1 - \delta) \cdot size(d)$ **faire**
14 $c(d,i) \leftarrow 0$;
15 **fin**

Algorithme 5 : Algorithme Rounding

Remarque 12. *Un document d est arrondi vers le haut si le document final occupe une mémoire supplémentaire d'au plus δD_{max}.*

Sinon le document d est arrondi vers le bas à $(1 - \delta)size(d)$

Toutes les valeurs de $c(d,i) \in (0, (1 - \delta)size(d)]$, sont arrondis vers le bas à 0.

Théoreme 3. *Soit σ une séquence de requêtes quelconque.*

Pour tous $\varepsilon \geq 0$, nous pouvons construire une solution engendrant un coût d'au plus $(1 + \varepsilon)BBMIN(\sigma)$ et nécessitant une mémoire supplémentaire d'au plus $D_{max}/(1 + \varepsilon) = D_{max}\delta$.

Preuve :

Montrons d'abord que la solution nécessite une mémoire supplémentaire d'au plus $D_{max}/(1 + \varepsilon)$.

De la ligne 4 de l'algorithme Rounding découle l'inegalité $0 \leq Extra \leq \delta D_{max}$.

Si un document est arrondi vers le bas, alors

$$Extra + size(d) - c(d,i) > \delta D_{max}$$

Ce qui implique que

$$Extra + (1 - \delta)size(d) - c(d,i) > \delta D_{max} - \delta size(d) \geq 0$$

Remarquons que si la valeur de $Extra$ est augmentée à la ligne 5 de l'algorithme, alors le document d est arrondi vers le haut. La mémoire supplémentaire nécessaire est $size(d) - c(d,i)$, ce qui équivaut a l'augementation de la valeur de $Extra$.

De même, si la valeur de $Extra$ est diminuée à la ligne 8, alors le document d est arrondi vers le bas. Et nous liberons $c(d,i) - (1-\delta)size(d)$ de place.

Cette mémoire liberée est à disposition jusqu'à la fin de batch $b(d,i) - 1$. Dès lors, lorsqu'un document d est arrondi vers le bas, il peut exister des documents d' arrondis vers le haut (utilisant cette mémoire liberée).

Cependant, nous devons respecter la règle de BBMIN. C'est à dire que le batch dans lequel ces documents d' sont demandés, disons j, doit être $\leq b(d,i)$. En effet, BBMIN aurait préféré enlever des bits de ces documents au lieu d'enlever ceux du document d.

Ainsi, les documents arrondis vers le haut ne demandent aucune mémoire supplémentaire jusqu'à la fin du batch $b(d,i) - 1$. Et cette mémoire supplémentaire est bornée supérieurement par

$$Extra + (1-\delta)size(d) - c(d,i) < Extra \leq \delta D_{max}$$

Puisqu'à la ligne 14, nous réduisons $c(d,i)$ à 0, alors la place supplémentaire nécessaire n'augmente pas. Et nous pouvons conclure.

Montrons maintenant que le coût engendré par la solution est d'au plus

$$(1+\varepsilon)BBMIN(\sigma)$$

.

Lors de l'éxécution des lignes 1 à 9, la valeur de $Extra$ est égal au coût relatif sauvé par $BBMIN(\sigma)$.

Si un document d est arrondi vers le haut, le coût sauvé lors de sa prochaine requête est égal à $size(d) - c(d,i)$.

Si un document d est arrondi vers le bas, le coût supplémentaire est $c(d,i) - (1-\delta)size(d)$.

Mais comme $Extra$ a toujours une valeur positive, alors juste avant l'éxécution de la ligne 13, le coût de la solution est borné supérieurement par $BBMIN(\sigma)$.

A la ligne 14, le coût peut augmenter d'un facteur de $1/\delta = 1 + \varepsilon$. Et donc nous pouvons conclure.

□

Corollaire 2. *Soit σ une séquence de requêtes quelconque.*

Pour tous $\varepsilon \geq 0$, nous pouvons construire une solution engendrant un coût d'au plus $(2+\varepsilon)$ fois la solution optimale et nécessitant une mémoire supplémentaire d'au plus $D_{max}/(1+\varepsilon/2)$.

4.4 Le modèle fault

Rappelons que le modèle fault est tel que $cost(d) = 1 \quad \forall \quad d$.

L'approche du modéle fault est différente des autres modèles. En effet, pour notre modèle fault, nous allons formuler notre problème de web caching en un programme linéaire. Nous procédons de cette façon parce qu'aucun algorithme combinatoire simple fonctionnant selon le modèle en batch n'est connu pour construire une solution fractionnaire acceptable.

4.4.1 Avec Réordonnancement

Nous allons formuler le programme linéaire du problème de web caching. Pour ceci nous utilisons des notations déjà utilisées dans le modèle bit.

Définition 13 ($b(d,i)$ et $c(d,i)$.)**.** *Pour tout document d, $b(d,i)$ est le plus petit index j, avec $j > i$, tel qu'il existe une requête correspondant à d appartenant au batch j.*

Pour tout document d, $c(d,i)$ est son nombre de bits présent dans le cache à la fin du batch i. Par convention, le nombre de bits présent initialement dans le cache est noté $c(d,-1)$.

Tout d'abord nous allons supposer raisonnablement que le nombre de bits présent dans le cache d'un document ne change pas entre la fin du batch dans lequel il est demandé et le début du batch dans lequel il est redemandé une nouvelle fois. C'est à dire que :

$$c(d,j) = c(d,b(d,i)-1) \quad \forall j = i, \cdots, b(d,i)-2 \text{ et } \forall d \in B_i$$

Le coût d'un document $d \in B_i$ est égal à la fraction de d non présente dans le cache à la fin du batch qui précède. C'est à dire :

$$cost(d) = 1 - c(d,i-1)/size(d) \quad \forall d \in B_i$$

Ainsi, le programme linéaire peut être formulé comme suit :

$$\text{Min} : \sum_{i=0}^{\lfloor m/r \rfloor} \sum_{d \in B_i} (1 - c(d,i-1)/size(d))$$

$$\begin{aligned}
\text{S.c} : \quad & c(d,i) = c(d,b(d,i)-1) && \forall \quad j = i, \cdots, b(d,i)-2 \text{ et } \forall \quad d \in B_i \\
& \textstyle\sum_{d \in D} c(d,i) \leq K && \forall \quad i \\
& 0 \leq c(d,i) \leq size(d) && \forall \quad i,d
\end{aligned}$$

Nous pouvons résoudre ce programme linéaire dans un temps polynomial.

Notons par $OPT^B_{PL}(\sigma)$ le coût du programme linéaire. Nous pouvons admettre que c'est une borne inférieure au coût minimum d'une solution à valeurs entières

d'un algorithme procédant en batch.

Fixons $\varepsilon > 0$. Nous allons classer les documents en différentes classes de la manière suivante :

$$d \in C_k \qquad \text{si} \qquad D_{max}(1+\varepsilon)^{-k} \geq size(d) > D_{max}(1+\varepsilon)^{-(k+1)}$$

$$\forall \quad 0 \leq k \leq \lfloor log_{1+\varepsilon} D_{max} \rfloor + 1.$$

Nous voulons arriver à ce que toutes les classes C_k satisfassent la propriété déjà utilisée dans le modèle bit.

Propriété 2. *Soit d , $d' \in C_k$ deux documents avec $d \in B_i$, $d' \in B_j$ et $i \leq j$.*

Si $b(d', j) < b(d, i)$ et $c(d, l) > 0 \qquad \forall \quad l = i, \cdots, b(d, i) - 1.$

Alors $c(d', l) = size(d') \qquad \forall \quad l = j, \cdots, b(d', j) - 1.$

Tant que cette propriété n'est pas satisfaite entièrement, nous modifions $c(d, l)$ et $c(d', l)$ tels que $c(d, l) = 0$ ou $c(d', l) = size(d')$.

Nous pouvons être encore plus précis. Notons $\gamma = minc(d, i), size(d') - c(d', j)$, c'est à dire le minimum entre le nombre de bits du document d présent dans le cache après le batch i et le nombre de bits du document d' qui ne sont pas dans le cache après le batch j.

Nous augmentons $c(d', l)$ de γ pour $l = j, \cdots, b(d', j) - 1$, et nous diminuons $c(d, l)$ de γ pour $l = i, \cdots, b(d, i) - 1$.

Notons OPT^* la solution obtenue après toutes ces modifications. Et appliquons l'Algorihtme Rounding à toutes les classes C_k séparemment. Suivant les notations de l'algorithme nous avons $\delta = 1$ et $D_{max} = D^k_{max}$ (la taille maximale d'un document de la classe C_k).

Nous avons maintenant tous les outils pour parvenir au résultat suivant.

Théoreme 4. *Pour toute séquence de requêtes σ et $\varepsilon > 0$, nous pouvons construire une solution appliquée à σ et procédant en batch telle que son coût est borné supérieurement par $(1+\varepsilon)OPT^B_{PL}(\sigma)$ et qui nécessite au plus $D_{max}(1 + 1/\varepsilon)$ de mémoire supplémentaire.*

Preuve :

Montrons d'abord qu'il nous faut au plus $D_{max}(1+1/\varepsilon)$ de mémoire supplémentaire.

La solution OPT^* ne demande aucune mémoire supplémentaire. Cependant, lorsque nous appliquons l'Algorithme Rounding aux classes C_k, nous avons besoin de D^k_{max} mémoire supplémentaire par classe. Et comme, $\sum_{k\geq 0} D^k_{max} \leq \sum_{k\geq 0} D_{max}(1+\varepsilon)^{-k} < D_{max}(1+1/\varepsilon)$, alors nous avons bien qu'il nous faut au plus $D_{max}(1 + 1/\varepsilon)$ de mémoire supplémentaire.

Montrons maintenant que le coût de la solution est borné supérieurement par $(1+\varepsilon)OPT^B_{PL}(\sigma)$.

Pour ceci, nous définissons R^u_k comme étant le nombre par lequel l'ensemble des documents de C_k ont été arrondis par le haut.

De la même manière, R^d_k est le nombre par lequel l'ensemble des documents de C_k ont été arrondis par le bas.

Nous disons qu'un document est arrondi par le haut de γ si nous avons augmenté $c(d,i)$ de γ. Ceci correpond à un gain de coût de $\gamma/size(d)$ quand le document d sera à nouveau servi.

De la même manière, nous disons qu'un document est arrondi par le bas de γ si nous avons diminué $c(d,i)$ de γ. Ceci correpond à un coût supplémentaire de $\gamma/size(d)$ quand le document d sera à nouveau servi.

La solution OPT^* et l'Algorithme Rounding implique que :

$$R^u_k \geq R^d_k$$

De plus

$$OPT^B_{PL}(\sigma) \geq \sum_{k\geq 0} R^u_k/D^k_{max}$$

Et comme la taille des documents entre deux classes varie d'au plus un facteur de $1+\varepsilon$, alors notre solution finale nécessite un coût supplémentaire d'au plus

$$\sum_{k\geq 0} R^d_k(1+\varepsilon)/D^k_{max}$$

Or

$$\begin{aligned}\textstyle\sum_{k\geq 0} R^d_k(1+\varepsilon)/D^k_{max} &\leq \textstyle\sum_{k\geq 0} R^u_k(1+\varepsilon)/D^k_{max} \\ &\leq (1+\varepsilon)OPT^B_{PL}(\sigma)\end{aligned}$$

□

Grâce au lemme du paragraphe processus en batch, nous avons le corollaire suivant :

Corollaire 3. *Pour toute séquence de requêtes σ et $\varepsilon > 0$, nous pouvons construire une solution appliquée à σ telle que son coût est borné supérieurement par $(2+\varepsilon)OPT^B_{PL}(\sigma)$ et qui nécessite au plus $D_{max}(1+2/\varepsilon)$ de mémoire supplémentaire.*

4.5 Le modèle général

4.5.1 Sans Réordonnancement

La solution pour le modèle général sans réordonnancement que nous allons exposer ci-dessous ici est basée sur une approximation de l'algorithme de Bar-Noy et al.

Dans son article, Bar-Noy et al. [BARNOY] ont montré que le problème du web caching dans le cas général sans réordonnancement peut être résolu comme un problème de "perte minimale".

Ici, nous ne ferons qu'exposer ce problème.

4.5.2 Le probleme de "perte minimale"

Notre objectif est de minimiser les pertes dues aux activités non programmées.

Nous avons besoin de définir les termes du problèmes à l'aide des définitions suivantes.

Définition 14. *Nous définirons :*

$\ell(t)$ *comme étant l'ensemble des intervalles contenant le temps* t*.*

$width(t)$ *comme étant le nombre de ressource disponibles au temps* t*.*

$w(I)$ *comme étant le "width" de l'intervalle* I*, représentant le nombre de ressource nécessaire au temps* t*.*

$p(I)$ *comme étant la pénalité de l'intervalle* I *,* $\forall I \in \ell(t)$*, représentant le coût si* I *n'est pas programmé.*

Ainsi, notre but est de trouver un ensemble d'intervalles $S \subset I$ programmés tel que :

$$\sum_{I \in S} w(I) \leq width(t) \quad \forall t$$

Et tel que :

$$\text{Le total des pénalités } \sum_{I \in \ell(t) \setminus S} p(I) \text{ soit le plus petit possible.}$$

4.5.3 Avec Réordonnancement

Nous montrerons dans ce paragraphe que nous pouvons aussi nous ramener à un problème de perte minimale dans le cas du modèle général avec réordonnancement.

Notre raisonnement reste basé sur l'étude d'algorithmes, appliqués à une séquence de requêtes, fonctionnant selon un processus en batch.

Nous reprenons la définition suivante :

Définition 15 ($b(d,i)$)**.** *Pour tout document d, $b(d,i)$ est le plus petit index j, avec $j > i$, tel qu'il existe une requête correspondant à d appartenant au batch j.*

Pour tous document d et pour tous batch i, tels que d correspond à une requête du batch i, nous définissons l'intervalle I :

$$I_{(d,i)} = [i, b(d,i) - 1]$$

cet intervalle représente le cas où d appartient au cache et y reste entre la fin du batch i et la fin du batch $b(d,i) - 1$.

Si le document d n'était pas présent dans le cache avant de commencer le batch i, alors il y aurait été mis lors de son déroulement.

Nous avons deux possibilités, soit l'intervalle $I_{(d,i)}$ est programmé, soit il ne l'est pas.

Si $I_{(d,i)}$ est programmé, alors les requêtes du batch $b(d,i)$ faisant référence au document d ont un coût nul.

Par contre, si $I_{(d,i)}$ n'est pas programmé, leur coût est égal à $cost(d)$. Et nous définissons la pénalité, $p(I_{(d,i)}) = cost(d)$. Ainsi que $w(I_{(d,i)}) = size(d)$ et $width(t) = K$.

Finalement nous avons définis les termes nécessaires à l'application du problème de perte minimale, et donc nous pouvons l'appliquer à notre situation. De ce fait nous obtenons les résultats suivants.

Théoreme 5. *Soit σ une séquence de requêtes. Pour toutes σ, nous pouvons construire une solution servant σ selon un processus en batch, A, nécessitant un coût d'au plus 4 fois celui de la solution optimale servant σ selon un processus en batch et qui ne nécessite pas de mémoire supplémentaire dans le cache.*

C'est à dire que :

$$A(\sigma) \leq 4 \cdot OPT_{batch}(\sigma)$$

Corollaire 4. *Soit σ une séquence de requêtes.*

Pour toutes σ, nous pouvons construire un algorithme, B , donnant une solution nécessitant un coût d'au plus 8 fois celui de la solution optimale et qui ne nécessite pas de mémoire supplémentaire dans le cache.

C'est à dire que :

$$B(\sigma) \leq 8 \cdot OPT_{sans\ batch}(\sigma)$$

Preuve :

Soit A un algorithme procédant en batch et soit B un algorithme ne procédant pas en batch.

La preuve découle d'un résultat que nous avons démontré dans la partie traitant du processus en batch qui nous dit que :

$$B(\sigma) = C \Longrightarrow \exists \quad A \text{ tel que } A(\sigma) \leq 2 \cdot C$$

Ainsi,

$$\begin{aligned} B(\sigma) &\leq A(\sigma) \\ &\leq 4 \cdot OPT_{\text{batch}}(\sigma) \\ &\leq 4 \cdot 2 \cdot OPT_{\text{sans batch}}(\sigma) \\ &= 8 \cdot OPT_{\text{sans batch}}(\sigma) \end{aligned}$$

□

5 Conclusion

Les algorithmes présentés dans ce rapport sont modifiables suivant le problème que nous sommes ammenés à résoudre. Au lieu d'en chercher des améliorations éventuelles, j'ai préféré étudier les cas précédents qui ont amenés aux résultats présentés.

La compétitivité de l'algorithme utilisé pour le problème online que nous avons trouvée ($k+1$ avec $k = K/D_{min}$) semble bonne. Pour ce qui est des algorithmes offline, nombreux sont les articles qui pensent que nous sommes encore loin d'un résultat optimal.

J'ai eu plaisir à effectuer ce projet de semestre dans la chaire ROSE du Professeur Dominique de Werra avec le suivi de Daniela Bronner. Ce projet m'a apporté en particulier des connaissances sur l'étude des algorithmes.

Références

[PLAXTON] Greg Plaxton, 2003. *Online File Caching.* University of Texas at Austin.

[ALBERS] Susanne Albers, 2003. *New Results on Web Caching with Request Reordering.* Albert-Ludwigs-Universitat Freiburg.

[FEDER1] Tomas Feder et al. *Combining request scheduling with web caching.*

[YOUNG] Neal E. Young. *Online File Caching.*

[BARNOY] Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, Baruch Schieber. *A unified Approah to Approximating Resource Allocation and Scheduling.*

[FEDER2] Tomas Feder, Rajeev Motwani, Rina Panigraphy, An Zhu. *Web Caching With Request Reordering.*

Pages Internet :

[1] http ://cne.gmu.edu/modules/vm/yellow/repol.html

[2] http ://wwwsi.supelec.fr/yb/projets/algogen/NP-Complet.htm