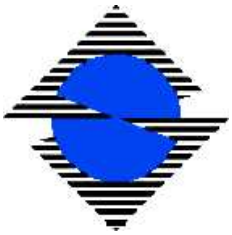
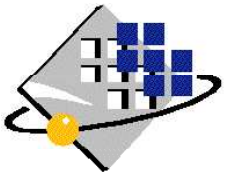




Numéro d'ordre :

UNIVERSITÉ PARIS XI
U.F.R. SCIENTIFIQUE D'ORSAY

THÈSE
présentée
pour obtenir

Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI ORSAY

PAR

M. Jérémy Barbay

SUJET :

Analyse fine :
bornes inférieures et algorithmes de calculs d'intersection pour moteurs de recherche

Soutenue le 24/09/2002 devant la Commission d'examen :

Mme Véronique Benzaken	
M Marek Karpinski	
Mme Claire Kenyon	*directrice*
M J. Ian Munro	*rapporteur*
Mme Michèle Soria	*rapporteur*

Remerciements

Bon vous êtes prévenus ça va être long et lourd. Je m'en voudrais d'oublier qui que ce soit...

Je remercie les professeurs du département d'informatique de Rouen (merci Philippe, Martine, Christophe et Thierry) de m'avoir convaincu de faire un détour par la licence de mathématiques avant de me mettre à l'informatique, et de m'avoir supporté pendant la Maitrise d'Ingénierie.

Je remercie le laboratoire d'Orsay, pour les secrétaires et concierges souriantes et efficaces, les collègues amicaux et serviables. Je remercie l'équipe de Miklos Santha, dont la diversité est une richesse, et l'unité (pas seulement autour de la machine à café) est une force qui doit beaucoup à son responsable. En particulier je remercie Jean-Pierre Tillich qui a eu une large influence sur mes enseignements, et Sophie Laplante pour ses conseils sur ma manière d'aborder la recherche.

Je remercie toute l'équipe du projet FFSS : Anis, Bennyben, féfé, Jal, kiki, Kindman et Mad. J'espère pouvoir encore participer à ce projet et à d'autres.

Je remercie David Faure sans qui je n'aurais pas commencé cette thèse, et Julien Sebot sans qui je ne l'aurais peut-être pas terminée, ou en tout cas autrement. Leur amitié restera parmi les découvertes fondamentales de ma thèse ;o).

Enfin je remercie ma "chef" Claire, qui a su diriger ma thèse comme un navire au long cours, à travers bourrasques, tempètes et passages à vide. Puisse-t-elle naviguer encore longtemps !

Table des matières

Introduction **1**

1 Contexte **3**
1.1 Algorithmes et Complexité 4
1.2 Algorithmes adaptatifs 15
1.3 Moteurs de recherche indexés 21

2 Intersection : État de l'art **23**
2.1 Analyse Classique 23
2.2 Un algorithme standard d'intersection 26
2.3 Analyse de Demaine, Lopez-Ortiz et Munro 30
2.4 Résultats pratiques 35
2.5 Résultats complémentaires 35

3 Alternance de l'intersection **39**
3.1 Mesure d'alternance 40
3.2 Borne inférieure 44
3.3 Complexité des algorithmes 49

4 Ensembles de Multiplicité **55**
4.1 Multiplicité et t-alternance 56
4.2 Calcul de l'ensemble de multiplicité fixée 57
4.3 Calcul de l'ensemble de multiplicité optimale 62

5 Généralisation **67**
5.1 Combinaison de problèmes de décision 68
5.2 Hypothèses 69
5.3 Bornes inférieures 71
5.4 Interunion 76

6 Projet FFSS **81**
6.1 Description générale du projet 81
6.2 Structure du système 82
6.3 Spécifications du moteur de recherche 83
6.4 Bilan du projet 84

Conclusion **87**
Apport de la thèse 87
Perspectives 88

Annexe **91**

Captures d'écran de FFSS **91**

Liste des algorithmes **99**

Liste des figures **102**

Bibliographie **103**

Index **105**

Introduction

Lors d'une requête conjonctive à un moteur de recherche indexé par mots-clés, celui-ci associe à chaque mot de la requête un tableau contenant une liste triée de références de documents. La réponse à la requête est alors calculée par le moteur de recherche en faisant l'intersection des tableaux. Une réponse rapide est bien sur souhaitable.

L'analyse de la complexité dans le pire des cas est une notion de base en informatique, nous la décrivons et rappelons quelques outils de base pour montrer des bornes inférieures de la complexité probabiliste (chapitre 1).

Une analyse usuelle dans le pire des cas de la complexité de calcul de l'intersection de tableaux de tailles fixées ne permet pas de comparer les algorithmes utilisés. On souhaite pouvoir différencier les algorithmes par une analyse de la complexité dans le pire des cas sur des groupes d'instances plus petits. Nous décrivons les travaux de Demaine, Lopez-Ortiz et Munro qui ont proposé à cette fin une analyse plus fine de la complexité déterministe de calcul de l'intersection [DLOM00], et ont conçu deux algorithmes déterministes pour effectuer ce calcul. L'un de ces algorithmes est optimal pour leur modèle d'analyse, mais l'autre est meilleur sur un échantillon pratique [DLOM01] (chapitre 2).

Nous définissons un modèle d'analyse différent de la complexité de calcul de l'intersection. Le meilleur algorithme pour ce modèle correspond au meilleur algorithme en pratique de Demaine, Lopez-Ortiz et Munro (Nous démontrons son optimalité chapitre 3).

Nous généralisons cette approche à divers problèmes tels que le calcul de l'*ensemble de multiplicité t* (chapitre 4), le calcul de l'*ensemble de multiplicité optimale* (chapitre 4), et le calcul de l'*"interunion"* (chapitre 5). Ces travaux sont liés au moteur de recherche intégré au système de partage de fichiers FFSS, présenté chapitre 6.

La borne inférieure du problème d'intersection du chapitre 3 a été publiée dans l'article [BK02], avec l'analyse de l'algorithme `Parallèle`. Le problème de l'ensemble de multiplicité fixée étudié dans le chapitre 4 et le problème de l'interunion étudié dans le chapitre 5 ont été définis dans ce même article [BK02].

La généralisation de ces résultats dans le chapitre 5 a été rédigée et soumise dans l'article [BR02].

Chapitre 1

Contexte

Dans ce premier chapitre nous introduisons les notions de base constituant le contexte de cette thèse. Nous rappelons la notion d'algorithme et de complexité algorithmique sur une instance et sur un problème. Les algorithmes sont représentés par des *arbres de décision* pour obtenir des résultats généraux sur tous les algorithmes déterministes possibles. En conjonction avec le principe du Minimax cela permet d'obtenir une borne inférieure à la complexité du meilleur algorithme probabiliste.

La mesure de la complexité d'un algorithme sur un problème dépend d'une partition des *instances*. Habituellement on les partitionne selon leur taille n. Cette partition peut être affinée pour préciser l'analyse de la complexité. C'est le sujet de l'étude des *algorithmes adaptatifs*, qui ont été étudiés pour divers problèmes.

Nous appliquons ces notions à l'analyse de problèmes d'intersections, qui se posent dans le cadre des moteurs de recherches indexés.

1.1 Algorithmes et Complexité

The White Rabbit put on his spectacles.
Where shall I begin, please your Majesty ? he asked.
Begin at the beginning, the King said gravely,
and go on till you come to the end :
then stop.

Lewis Carroll,
Alice's Adventures in Wonderland

1.1.1 Algorithmes

Dans cette thèse, nous parlerons à la fois d'algorithmes déterministes et d'algorithmes probabilistes. Nous rappelons la définition d'un algorithme.

Définition 1.1 (algorithme n. m.) *Un* algorithme *est une méthode de résolution d'un problème utilisant un nombre fini d'applications d'une règle. Par exemple l'algorithme d'Euclide permet de calculer le plus grand commun diviseur de deux nombres entiers.*

Un algorithme est déterministe *si les règles appliquées sont toutes déterministes. Dans le cas contraire c'est un* algorithme probabiliste.

Dans le contexte informatique, un algorithme est une description précise des étapes à parcourir pour effectuer un calcul ou une tâche précise. Les données reçues par l'algorithme forment l'*instance* à résoudre du problème.

Nous ne considérons que les algorithmes résolvant exactement le problème : ce sont les algorithme dit de "Las Vegas" dont le temps d'exécution peut-être aléatoire mais qui permettent toujours de trouver un résultat correct. Nous ne considérons pas les algorithmes probabilistes dits de "Monte-Carlo", permettant d'obtenir un résultat correct avec une probabilité fixée, et *toute référence ultérieure à un algorithme probabiliste concerne un algorithme de "Las Vegas".*

Le *problème des pièces cachées* est un problème abstrait qui servira à illustrer les différentes notions de ce chapitre :

Définition 1.2 (Problème des pièces cachées) *Soit une rangée de n cartes. Chaque carte dissimule une pièce, qui peut être une pièce de cuivre ou d'argent. Le* problème des pièces cachées *est de décider si la rangée contient au moins une pièce d'argent.*

Dans ce cas précis, un algorithme doit préciser quelles cartes retourner, dans quel ordre, selon les types des pièces découvertes. Sur un problème aussi simple, on peut restreindre l'étude aux algorithmes cessant de retourner les cartes aussitôt après avoir découvert une première pièce d'argent : toute carte retournée par la suite

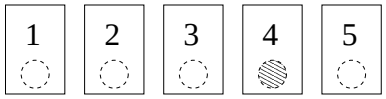


FIG. 1.1 – Une instance du problème des pièces cachées : une pièce d'argent parmi 4 pièces de cuivre dissimulées par des cartes.

serait superflue. Chacun de ces algorithmes est défini par l'ordre σ dans lequel il retourne les cartes tant qu'il n'y a pas de pièce d'argent.

1.1.2 Complexité d'un algorithme

Nous rappelons ici les bases de l'analyse de la complexité d'algorithmes.

Étant donné une fonction de coût sur les opérations effectuées par l'algorithme, la complexité d'un algorithme A sur une instance I est la somme des coûts des instructions correspondant à l'exécution de A sur I. Les algorithmes résolvant le même problème sont comparés par leur complexité. La *complexité en temps* $C(A, I)$ d'un algorithme A sur une instance I correspond au nombre d'instructions effectuées par A quand il est exécuté pour résoudre I.

Par exemple, dans une configuration ne comportant qu'une seule pièce d'argent cachée sous la deuxième carte à partir de la droite, l'algorithme retournant les cartes de gauche à droite en retournera $n - 1$, l'algorithme les retournant de droite à gauche n'en retournera que 2.

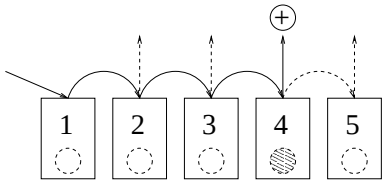


FIG. 1.2 – L'algorithme parcourant les cartes de gauche à droite : en trait pointillé les exécutions possibles, en trait plein l'exécution sur cette instance. La réponse de l'algorithme est positive car la rangée contient une pièce d'argent.

Soit $F = \{I_1, \ldots, I_{|F|}\}$ un ensemble fini d'instances, et A un algorithme pour ces instances. La complexité $C(A, F)$ de A sur F peut être définie de plusieurs manières. L'ensemble des valeurs prises par la complexité de A sur les instances de F est $\{C(A, I_1), \ldots, C(A, I_{|F|})\}$:

– leur maximum $C(A,F) = \max_{I \in F} C(A,I)$ correspond à la *complexité dans le pire des cas,*
– et la moyenne $C(A,F) = \sum_{I \in F} C(A,I)p(I)$ correspond à la *complexité en moyenne* selon une distribution de probabilité $p(I)$ sur les instances.

Par exemple la complexité dans le pire des cas de l'algorithme parcourant les cartes de gauche à droite est n.

La complexité d'un algorithme sur un nombre infini d'instances ne peut être définie de la même manière : le nombre de valeurs de complexité à considérer est potentiellement infini. Pour définir cette complexité l'ensemble des instances est partitionné en sous-ensembles de cardinalité finie indexés par $\mathbb{N}$, par exemple les instances pouvant être codées sur n bits, pour tout entier n. Pour tout entier n, la complexité $f(n)$ (dans le pire des cas, ou en moyenne) de l'algorithme sur le sous-ensemble d'index n est alors bien définie. La complexité $C(A)$ de l'algorithme A sur le problème est définie comme la fonction f qui à chaque entier n fait correspondre $f(n)$.

Par exemple, pour le problème des pièces cachées avec n pièces où n est fixé, l'ensemble des configurations est fini. L'algorithme de la figure 1.2 découvre la totalité des n cartes dans le pire des cas : sa complexité dans le pire des cas est donc $f(n) = n$. Sa complexité moyenne sur la distribution uniforme des instances ne comportant qu'une seule pièce d'argent est $(n+1)/2$.

Algorithme 1 Énoncé de l'algorithme parcourant les cartes de gauche à droite.

tant que des cartes restent à découvrir, et aucune pièce d'argent n'a été trouvée **faire**

Retourne la carte non retournée la plus à gauche ;

fin tant que

si une pièce d'argent a été découverte **alors**

répond positivement

sinon

répond négativement.

fin si

1.1.3 Arbres de décision et complexité d'un problème

Pour prouver une borne inférieure à la complexité d'un problème, il faut pouvoir raisonner sur l'ensemble de tous les algorithmes possibles. Pour cela, les algo-

rithmes sont représentés sous la forme d'arbres où chaque noeud est une instruction, chaque branche une exécution, chaque feuille un résultat : c'est le modèle des *arbres de décision*, utilisé pour tous les résultats de borne inférieure de cette thèse.

Définition 1.3 (Questionnaire) *Un* questionnaire *est un arbre dont les feuilles sont étiquetées par des classes et dont les noeuds internes sont étiquetés par des tests. Si le test a k résultats possibles, alors le noeud interne a k fils, et les k arcs reliant le noeud à ses fils sont étiquetés par ces résultats.*

Le questionnaire permet de décider à quelle classe appartient une instance donnée. L'instance est soumise à une suite de tests en commençant par le test contenu par le noeud à la racine. Suivant le résultat du test, la séquence de tests se poursuit dans le sous-arbre correspondant. Si le sous-arbre est une feuille, l'étiquette de cette feuille est la classe associée à l'instance.

Définition 1.4 (Arbre de décision) *Un* arbre de décision *est un questionnaire où chaque noeud interne correspond à une opération déterministe exécutable sur toute instance en un temps fini.*

Tout algorithme déterministe terminant en un temps fini peut être exprimé par un arbre de décision. Chaque feuille correspond alors à un résultat de l'algorithme. Le nombre d'opérations effectuées par l'algorithme sur cette instance est alors la longueur de la branche correspondante.

Définition 1.5 (Arbre de comparaison) *Un* arbre de comparaison *est un arbre de décision dont les tests sont des comparaisons entre les éléments internes à l'instance.*

Knuth [Knu73] utilise des arbres de comparaison pour obtenir une borne inférieure à la complexité d'algorithmes de tris basés sur les comparaisons. Son analyse exclut les algorithmes comme l'algorithme de tri par comptage, qui ne nécessite aucune comparaison.

Par exemple, les résultats sur l'intersection de tableaux triés du chapitre 2 sont exprimés dans le modèle des arbres de comparaison. Les résultats du chapitre 5 sur la combinaison de problèmes de décision sont eux dans le modèle des arbres de décision.

Par exemple, tout arbre de décision correspondant à un algorithme pour le problème des pièces cachées doit contenir au moins $n + 1$ feuilles : une pour chaque position possible de la première pièce d'argent détectée, et une pour la configuration sans pièce d'argent. Chaque test permet d'éliminer exactement une position potentielle pour la pièce d'argent. Dans ce cas précis, tout arbre de décision correspond à une chaîne. Sa hauteur est égale à n. La complexité dans le pire des cas de tout algorithme pour ce problème est donc $\Omega(n)$.

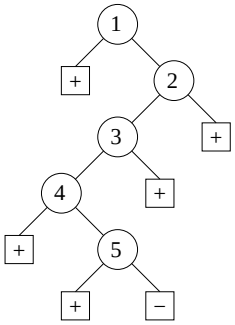


FIG. 1.3 – Un arbre de décision pour le problème des pièces cachées, quand il y a 5 pièces.

1.1.4 Algorithmes probabilistes

Le modèle des arbres de décision ne concerne que les algorithmes déterministes. Il s'étend aux algorithmes probabilistes par une distribution sur les arbres de décision déterministes :

Définition 1.6 (Algorithme probabiliste) *Pour un problème fixé, un* algorithme probabiliste *est défini par une distribution sur les algorithmes déterministes. De même, un* arbre de décision probabiliste *est une distribution sur les arbres de décision.*

La complexité d'un algorithme probabiliste R sur une instance I est la moyenne des complexités des algorithmes déterministes A sur I selon la distribution associée à R; $C(R, I) = \sum_A \Pr\{A\}C(A, I)$

Le modèle des algorithmes probabilistes est plus général que celui des algorithmes déterministes, et permet de meilleures performances. Par exemple, si une instance du problème des pièces cachées comprend $n/2$ pièces d'argent et $n/2$ pièces de cuivre, pour chaque algorithme déterministe il existe une configuration des pièces telle qu'il découvre $n/2$ pièces avant de découvrir une pièce d'argent. L'algorithme probabiliste choisissant un ordre σ des positions au hasard peut également découvrir $n/2$ pièces, mais avec une probabilité de $1/2^{n/2}$. En moyenne sur σ, l'algorithme probabiliste découvrira moins de deux pièces.

Définition 1.7 *La* complexité probabiliste d'un problème *est égale à la complexité moyenne du meilleur algorithme déterministe sur la pire distribution.*

Le théorème du Minimax permet d'obtenir des bornes inférieures à la complexité probabiliste d'un problème.

1.1.5 Théorème du Minimax et principe de Yao

Le théorème du Minimax est un outil théorique fondamental dans l'étude de la complexité probabiliste. Il est exprimé dans le contexte de la théorie des jeux, en particulier sur les jeux pour deux joueurs à somme nulle (jeux tels que la somme des gains des joueurs est toujours égale à zéro). Le principe de Yao applique ce théorème dans le contexte des algorithmes probabilistes, le premier joueur appliquant l'algorithme et le deuxième générant une instance de manière à maximiser la complexité de l'algorithme.

Soit Γ un jeu de somme nulle, à deux joueurs Alice et Bill tels que

- les ensembles $A = \{a_1, a_2, \ldots, a_m\}$ des stratégies déterministes possibles pour Alice et $B = \{b_1, b_2, \ldots, b_n\}$ des stratégies déterministes pour Bill soient finis,
- et le gain pour Alice quand elle applique la stratégie a_i et que Bill applique la stratégie b_j soit défini par l'élément $M_{i,j}$ de la matrice M.

L'ensemble des stratégies probabilistes (ou stratégies mixtes) est noté $\mathcal{A}$ pour Alice, $\mathcal{B}$ pour Bill. Ces stratégies probabilistes sont obtenues en combinant les stratégies déterministes (ou stratégies pures) selon un vecteur de probabilité :

$$\begin{aligned}\mathcal{A} = \quad & \{\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_m) \in [0,1]^m \text{ tel que } \textstyle\sum_{i=1}^m \alpha_i = 1 \\ & \text{et la stratégie } a_i \text{ est utilisée avec probabilité } \alpha_i \ \} \\ \mathcal{B} = \quad & \{\beta = (\beta_1, \beta_2, \ldots, \beta_n) \in [0,1]^n \text{ tel que } \textstyle\sum_{j=1}^n \beta_j = 1 \\ & \text{et la stratégie } b_j \text{ est utilisée avec probabilité } \beta_i\}\end{aligned}$$

Bien entendu les stratégies pures sont comprises dans l'ensemble des stratégies mixtes, pour des vecteurs de probabilité dont tout le poids est sur une seule composante. Par exemple a_1 correspond à la stratégie mixte de vecteur de probabilité $(1, 0, \ldots, 0)$. Une stratégie mixte est une combinaison linéaire de stratégies pures : $\alpha = \alpha_1 a_1 + \alpha_2 a_2 + \ldots + \alpha_n a_n$.

Remarque 1.1 *La performance d'une stratégie mixte α pour Alice face à une stratégie mixte β pour Bill se calcule par la formule suivante :*

$$\alpha^T M \beta = \sum_{i=1}^{m} \sum_{j=1}^{n} \alpha_i M_{i,j} \beta_j.$$

Un couple de stratégies (α^*, β^*) est un *équilibre de Nash* du jeu si, pour toutes stratégies α et β, on a $\alpha^T M \beta^* \leq \alpha^{*T} M \beta^* \leq \alpha^{*T} M \beta$. Pour certains problèmes il existe un équilibre de Nash parmis les stratégies pures. Le théorème du Minimax de von Neumann montre qu'il existe toujours un équilibre de Nash parmi les stratégies mixtes. Le lemme de Loomis montre que cet équilibre est atteint par un couple

formé d'une stratégie pure et d'une stratégie mixte. Le principe de Yao est une reformulation du théorème du Minimax, qui montre que la complexité en moyenne du meilleur algorithme déterministe sur la pire distribution est la complexité dans le pire des cas du meilleur algorithme probabiliste. Ces résultats sont prouvés dans les paragraphes suivants.

1.1.5.1 Théorème du Minimax

Le théorème du Minimax se démontre à l'aide du théorème du point fixe, et des lemmes 1.1 et 1.3. Les lemmes sont démontrés ici, et l'énoncé du théorème du point fixe est donné. Le début de la preuve du théorème du Minimax est inspiré de celle présentée par Borodin et El-Yaniv [BEY98]. Le lemme 1.1 est un résultat de base utilisé pour démontrer la double inégalité du théorème du Minimax (théorème 1.2)

Lemme 1.1 (Existence de $\tilde{\alpha}$ et $\tilde{\beta}$) *Soient ϕ et ψ définies sur $\mathbb{R}^m$ et $\mathbb{R}^n$ par*

$$\phi(\alpha) = \sup_{\beta} \alpha^T M \beta \text{ et } \psi(\beta) = \inf_{\alpha} \alpha^T M \beta$$

Alors :

1. *$\phi(\alpha) = \max_\beta \alpha^T M\beta$ et $\psi(\beta) = \min_\alpha \alpha^T M\beta$, et*
2. *il existe les stratégies mixtes $\tilde{\alpha}$ pour Alice et $\tilde{\beta}$ pour Bill telles que ϕ atteigne son minimum en $\tilde{\alpha}$ et ψ atteigne son maximum en $\tilde{\beta}$.*

Preuve.
Le premier point découle de la bilinéarité de M : Si α est fixé, la fonction qui à β associe $\alpha^T M\beta$ est linéaire, donc continue, et $\mathcal{B}$ étant compact elle atteint sa borne supérieure sur $\mathcal{B}$ (Th. 29 dans [Sch81]) : $\phi(\alpha) = \sup_\beta \alpha^T M\beta = \max_\beta \alpha^T M\beta$. Le même raisonnement s'applique pour ψ.

Pour montrer le deuxième point il suffit de montrer que ϕ et ψ sont continues : l'existence de $\tilde{\alpha}$ et $\tilde{\beta}$ est impliquée par la compacité de $\mathcal{A}$ et $\mathcal{B}$. Pour montrer la continuité de ϕ (le raisonnement s'applique également pour ψ), fixons $\alpha \in \mathcal{A}$ et $\varepsilon = (\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_m)$.

$$\begin{aligned}
\phi(\alpha+\varepsilon) &= \max_\beta (\alpha+\varepsilon)^T M\beta \\
&\leq \max_\beta \alpha^T M\beta + \max_\beta \varepsilon^T M\beta \\
&= \phi(\alpha) + \max_\beta \varepsilon^T M\beta.
\end{aligned}$$

Mais pour toute norme $|.|$, $|\varepsilon^T M\beta| \leq |\varepsilon| \times |M\beta|$. Comme $\mathcal{B}$ est de dimension n finie, $|M\beta|$ est fini, et $|\varepsilon^T M\beta|$ tend vers zéro quand $|\varepsilon|$ tend vers zéro. Donc $\lim_{|\varepsilon| \to 0} (\phi(\alpha+\varepsilon) - \phi(\alpha)) = 0$, et ϕ est continue en α. □

Le théorème du Minimax statue l'égalité entre $\min_\alpha \max_\beta \alpha^T M\beta$ et $\max_\beta \min_\alpha \alpha^T M\beta$. Il est prouvé en montrant une double inégalité, dont la première est vérifiée même sur les stratégies pures. Le lemme 1.2 ci-dessous est la partie "facile" du théorème du Minimax (théorème 1.2).

Lemme 1.2 (Inégalité simple)

$$\min_\alpha \max_\beta \alpha^T M\beta \geq \max_\beta \min_\alpha \alpha^T M\beta$$

Preuve.
Soient $\tilde{\alpha}$ et $\tilde{\beta}$ définies par le lemme 1.1 :

$$\min_\alpha \max_\beta \alpha^T M\beta = \max_\beta \tilde{\alpha}^T M\beta \geq \tilde{\alpha}^T M\tilde{\beta} \geq \min_\alpha \alpha^T M\tilde{\beta} = \max_\beta \min_\alpha \alpha^T M\beta.$$

□

Le lemme ci-dessous est utilisé pour démontrer le théorème du Minimax (théorème 1.2).

Lemme 1.3 (Lemme du Minimax) *Dans le jeu Γ les deux conditions suivantes sont équivalentes :*

(i) $\min_\alpha \max_\beta \alpha^T M\beta = \max_\beta \min_\alpha \alpha^T M\beta$

(ii) *Il existe un réel v et des stratégies mixtes $\tilde{\alpha}$ et $\tilde{\beta}$ telles que :*

$$\begin{aligned} \tilde{\alpha}^T M b_j &\leq v \quad \forall j = 1, 2, \ldots, n \\ a_i^T M\tilde{\beta} &\geq v \quad \forall i = 1, 2, \ldots, m. \end{aligned}$$

Preuve.
$(ii) \Rightarrow (i)$ Supposons l'existence du réel v et des stratégies mixtes $\tilde{\alpha}$ et $\tilde{\beta}$ de (ii) : par linéarité de M, pour tout $\beta \in \mathcal{B}$, on a $\tilde{\alpha}^T M\beta \leq v$. Donc, en particulier, $\max_\beta \tilde{\alpha}^T M\beta \leq v$, et par définition du minimum, $\min_\alpha \max_\beta \alpha^T M\beta \leq v$. De même pour $\tilde{\beta}$, $v \geq \max_\beta \min_\alpha \alpha^T M\beta$. Donc $\min_\alpha \max_\beta \alpha^T M\beta \leq v \leq \max_\beta \min_\alpha \alpha^T M\beta$. D'autre part le lemme 1.2 implique l'inégalité inverse $\min_\alpha \max_\beta \alpha^T M\beta \geq \max_\beta \min_\alpha \alpha^T M\beta$: d'où (i).

$(i) \Rightarrow (ii)$ Supposons qu'on ait l'égalité de (i). Soit $v = \min_\alpha \max_\beta \alpha^T M\beta$. utilisant le lemme 1.1, il existe un $\tilde{\alpha} \in \mathcal{A}$ tel que $\max_\beta \tilde{\alpha}^T M\beta = v$. Par définition, $\forall \beta \in \mathcal{B}$ $\tilde{\alpha}^T M\beta \leq v$, et donc a fortiori l'inégalité $\tilde{\alpha}^T M b_j \leq v$ de (ii) est vérifiée pour toute valeur de j de 1 à n. De même il existe un $\tilde{\beta} \in \mathcal{B}$ tel que $\min_\alpha \alpha^T M\tilde{\beta} = v$, donc $\forall \alpha \in \mathcal{A}$ $\alpha^T M\tilde{\beta} \geq v$, et par conséquent l'inégalité $a_i^T M\tilde{\beta} \leq v$ de (ii) est vérifiée pour toute valeur de i de 1 à m. □

La preuve du théorème du Minimax présentée ici repose sur le théorème de point fixe de Brouwer.

Théorème 1.1 (Point fixe de Brouwer [BEY98, page 112]) *Soit X un ensemble compact et convexe dans $\mathbb{R}^n$. Toute fonction continue $\phi : X \to X$ admet un point fixe $x \in X$ tel que $\phi(x) = x$.*

Théorème 1.2 (Théorème du Minimax de von Neumann) *Étant donné le jeu Γ défini par la matrice M :*

$$\min_{\alpha} \max_{\beta} \alpha^T M \beta = \max_{\beta} \min_{\alpha} \alpha^T M \beta$$

Preuve.
Étant donné les stratégies mixtes de chaque joueur α et β, soient

$$\begin{array}{rcccl} p_i & = & p_i(\alpha,\beta) & = & a_i^T M\beta - \alpha^T M\beta \quad \forall i = 1,\ldots,m \\ q_j & = & q_j(\alpha,\beta) & = & \alpha^T M\beta - \alpha^T M b_j \quad \forall j = 1,\ldots,n. \end{array}$$

Soit la fonction $\Phi : \mathcal{A}* \mathcal{B} \to \mathcal{A}* \mathcal{B}$ qui à chaque couple (α, β) fait correspondre le couple de stratégies mixtes (ξ, η) définies par

$$\begin{array}{rl} \forall i = 1,\ldots,m \quad \xi_i = & \dfrac{\alpha_i + \max\{p_i, 0\}}{1 + \sum_{k=1}^{n} \max\{p_k, 0\}} \\ \forall j = 1,\ldots,n \quad \eta_j = & \dfrac{\beta_j + \max\{q_j, 0\}}{1 + \sum_{k=1}^{m} \max\{q_k, 0\}} \end{array}$$

La fonction Φ est la somme de fonctions continues : elle est continue. $\mathcal{A}$ et $\mathcal{B}$ sont des espaces vectoriels compacts et convexes. Le théorème 1.1 implique que Φ admet un point fixe $(\tilde{\alpha}, \tilde{\beta})$. Soit p_i considéré au point $(\tilde{\alpha}, \tilde{\beta})$: $p_i = p_i(\tilde{\alpha}, \tilde{\beta})$ pour tout i. On a :

$$\begin{array}{rcl} \sum_i \tilde{\alpha}_i p_i & = & \sum_i \tilde{\alpha}_i (a_i^T M\tilde{\beta} - \tilde{\alpha}^T M\tilde{\beta}) \\ & = & (\sum_i \tilde{\alpha}_i a_i)^T M\tilde{\beta} - (\sum_i \tilde{\alpha}_i)\tilde{\alpha}^T M\tilde{\beta} \\ & = & \tilde{\alpha}^T M\tilde{\beta} - \tilde{\alpha}^T M\tilde{\beta} \\ & = & 0. \end{array}$$

Les termes d'une somme nulle soit sont tous nuls, soit comprennent au moins un terme négatif et un terme positif.

- Si les termes sont tous nuls, pour tout i $\tilde{\alpha}_i = 0$ ou $p_i = 0$. Supposons $p_i \neq 0$. Par définition du point fixe $\tilde{\alpha}_i \sum_{k=1}^{n} \max\{p_k, 0\} = \max\{p_i, 0\}$: on a $\max(p_i, 0) = 0$ et $p_i \leq 0$. Donc $p_i \leq 0$ pour tout i.

– Si les termes ne sont pas tous nuls, il existe un terme $\tilde{\alpha_i} p_i$ strictement négatif. $\tilde{\alpha}$ étant une distribution de probabilité, $\tilde{\alpha_i}$ est positif, ce qui implique que p_i est strictement négatif. Par définition du point fixe, $\tilde{\alpha_i} \sum_{k=1}^{n} \max\{p_k, 0\} = \max\{p_i, 0\}$, ce qui est nul puisque $p_i < 0$. Les termes de la somme nulle de terme positifs $\max\{p_k, 0\}$ sont tous nuls, donc $p_k \leq 0$ pour tout k.

Les q_j sont montrés tous négatifs ou nuls de manière symétrique. Le point (ii) du lemme 1.3 est vérifié en posant $v = \tilde{\alpha}^T M \tilde{\beta}$, ce qui prouve le théorème. □

1.1.5.2 Lemme de Loomis et Principe de Yao

Le lemme de Loomis est l'argument essentiel pour passer du théorème du Minimax au principe de Yao. La preuve présentée ici est celle donnée par Borodin et El-Yaniv [BEY98].

Lemme 1.4 (Lemme de Loomis [BEY98, lemme 8.2]) *Étant donné α une stratégie mixte fixée pour Alice, il existe une stratégie déterministe optimale b_j pour Bill quand Alice applique la stratégie α :*

$$\max_{\beta} \alpha^T M \beta = \alpha^T M b_j$$

De même, étant donné β une stratégie mixte fixée pour Bill, il existe une stratégie déterministe optimale a_i pour Alice quand Bill applique la stratégie β :

$$\min_{\alpha} \alpha^T M \beta = a_i^T M \beta$$

Preuve.
Pour α fixé, $\alpha^T M$ est une fonction linéaire sur $\mathcal{B}$, et à ce titre atteint son maximal en au moins une stratégie pure b_j, correspondant à un coefficient maximal de $\alpha^T M$. Par conséquent, $\max_\beta \alpha^T M \beta = \alpha^T M b_j$. Le même raisonnement tient pour Bill. □

L'égalité du théorème du Minimax peut se récrire en appliquant le lemme de Loomis à des stratégies optimales $\tilde{\alpha}$ et $\tilde{\beta}$:

$$\min_{\alpha} \max_{j} \alpha^T M b_j = \max_{\beta} \min_{i} a_i^T M \beta \tag{1.1}$$

Ces résultats de théorie des jeux sont appliqués à l'analyse de la complexité des algorithmes. Soient B l'ensemble *fini* des instances de taille s, $\mathcal{B}$ l'ensemble des distributions aléatoires sur ces instances, A un ensemble *fini* d'algorithmes déterministes résolvant le problème pour une taille de données fixée s, et $\mathcal{A}$ l'ensemble des algorithmes probabilistes définis par une distribution de probabilité sur A.

Alors $\alpha^T M b_j$ est la complexité moyenne de l'algorithme probabiliste $\alpha \in \mathcal{A}$ sur l'instance $b_j \in B$, et $\max_j \alpha^T M b_j$ est la complexité moyenne de cet algorithme sur la pire instance. De manière symétrique, $a_i^T M \beta$ est la complexité moyenne de l'algorithme déterministe $a_i \in A$ sur la distribution d'instances $\beta \in \mathcal{B}$ et $\max_\beta a_i^T M \beta$ est la complexité de cet algorithme sur la pire distribution d'instances imaginable.

L'inégalité suivante, dite "inégalité de Yao" est obtenue en appliquant cette interprétation à la formule 1.1. Cette inégalité permet d'obtenir des bornes inférieures à la complexité d'algorithmes probabilistes.

Théorème 1.3 (Principe de Yao [Yao77]) *La complexité du meilleur algorithme déterministe sur la pire distribution d'instances $\beta \in \mathcal{B}$ est égale à la complexité du meilleur algorithme probabiliste sur la pire instance.*

$$\max_\beta \min_i a_i^T M \beta = \min_\alpha \max_j \alpha^T M b_j$$

Cette approche permet d'obtenir des bornes inférieures pour des ensembles *finis* d'algorithmes déterministes, sur des ensembles d'instances finies. Par exemple : le modèle de calcul basé sur les comparaisons se réduit aux algorithmes n'effectuant pas deux fois la même comparaison. Le nombre de tels algorithmes est polynomial dans le nombre d'éléments à comparer. Il y a donc un nombre fini d'algorithmes sur des instances de tailles finies.

Par exemple dans le problème des pièces cachées, une unique pièce d'argent étant cachée uniformément, tout algorithme déterministe retourne en moyenne $n/2$ cartes. Le principe de Yao implique que le meilleur algorithme probabiliste retourne en moyenne $n/2$ cartes pour au moins une configuration. Cette différence d'un facteur $\frac{1}{2}$ n'est pas très importante, mais des différences de performances plus importantes apparaissent entre les algorithmes déterministes et probabilistes lors d'une analyse plus fine de la complexité, comme dans le cadre des algorithmes adaptatifs.

Dans cette thèse, nous utiliserons le principe de Yao de manière systématique dans toutes les preuves de borne inférieure.

1.2 Algorithmes adaptatifs

Le fait de grouper les instances par taille permet de borner la taille des ensembles d'instances. Mais parmi les instances de même taille, certaines instances sont plus "faciles" que d'autres. Par exemple, la complexité des algorithmes de tri dépend de l'ordre initial du tableau à trier. Parmi les tableaux de n éléments, les tableaux déjà presque triés sont "faciles", les tableaux désordonnés sont "difficiles". Pour mesurer cette difficulté, Burge introduit [Bur58] la notion de mesure de difficulté :

Définition 1.8 (Mesure de difficulté) *Une* mesure de difficulté *est une fonction à valeurs réelles définie sur les instances du problème.*

Une mesure de difficulté définit un ordre partiel sur les instances, et on s'attend à ce que la complexité d'un algorithme croisse avec la difficulté des instances.

Par exemple, l'analyse du problème des pièces cachées a été faite pour un nombre total de pièces fixé. Fixer le nombre de pièces d'argent en plus du nombre de pièces total permet une analyse plus fine. Celle-ci démontre entre autre la supériorité des algorithme probabilistes. La configuration ne comprenant pas de pièces d'argent est la pire pour tous les algorithmes. C'est la configuration de l'analyse classique dans le pire des cas. Les configurations comprenant plus de pièce d'argent sont plus faciles à résoudre. Sur une configuration comprenant x pièces d'argent, un algorithme déterministe retourne au plus $n - x + 1$ cartes, et l'algorithme probabiliste qui retourne les cartes dans un ordre σ aléatoire retourne en moyenne $\sum_{j=1}^{n-x+1} C_{n-j}^{x-1}/C_n^x$ cartes. Pour $x = n/2$, tout algorithme déterministe retourne $\lceil (n+1)/2 \rceil$ cartes dans le pire des cas, alors qu'un algorithme probabiliste en retourne en moyenne 2. Autrement dit, on obtient une complexité linéaire en $\Theta(n)$ pour les algorithmes déterministes, et une complexité constante $\Theta(1)$ pour les algorithmes probabilistes.

Kirkpatrick et Seidel [KS86] explorent cette approche pour l'analyse du calcul de l'enveloppe convexe d'un ensemble de points. Ils fixent la taille de la solution en plus de la taille de l'instance. La difficulté à trier un tableau est mesurée par le *désordre* de ce tableau. Le problème a été étudié pour diverses mesures de désordre [PM95, Bur58, CK80, Man85], en cherchant à définir pour chaque mesure un "algorithme adaptatif" s'adaptant au désordre de l'instance. Estivill-Castro et Wood [ECW92] rassemblent dans leur étude une liste d'algorithmes adaptatifs de tri et les analysent selon plusieurs mesures. Demaine, Lopez-Ortiz et Munro [DLOM00] appliquent cette notion d'algorithme adaptatif au calcul de l'union de B-arbres, et généralisent leur analyse au calcul de l'intersection de tableaux triés. Les études d'algorithmes dits "adaptatifs" sont plus basées sur une nouvelle manière d'analyser la complexité de l'algorithme que sur la conception d'algorithmes particuliers.

Nous introduisons le terme d'"analyse fine" de la complexité, qui nous semble mieux convenir à ce type d'étude.

1.2.1 Enveloppe convexe d'un ensemble de points

L'*enveloppe convexe* d'un ensemble de points du plan est le polygone de surface minimale contenant ces points. Les sommets de ce polygone appartiennent à l'ensemble de points donné en entrée. Selon les variantes du problème, un algorithme calculant l'enveloppe convexe d'un ensemble de points doit retourner ces points, de manière ordonnée ou pas. Kirkpatrick et Seidel [KS86] analysent la complexité de calcul de cette enveloppe convexe selon un partitionnement plus fin des instances du problème.

La complexité de leur algorithme est donnée en fonction de la taille de l'ensemble de points donné en entrée, et du nombre H de sommets de l'enveloppe convexe. Elle est de $O(n \log\ H)$. Dans le pire des cas, l'enveloppe convexe est composée des n points ($H = n$), et la complexité de l'algorithme est $O(n \log n)$, ce qui correspond à la performance des algorithmes antérieurs.

Leur algorithme est prouvé optimal en montrant une borne inférieure de $\Omega(n \log H)$ en fonction de ces mêmes critères. De même que précédemment, dans le pire des cas $H = n$ et la borne inférieure se réduit à la borne inférieure basée sur une analyse de la complexité dans le pire des cas pour seul n fixé.

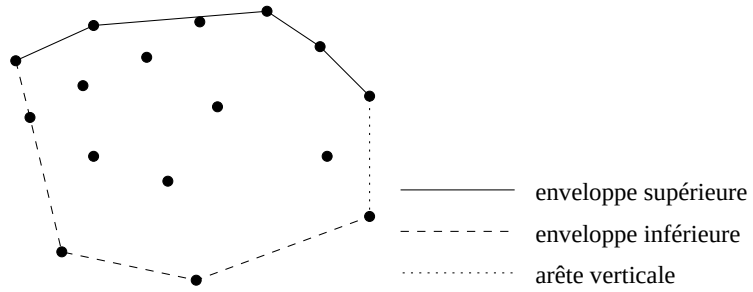


FIG. 1.4 – Décomposition de l'enveloppe convexe de taille 9 d'un ensemble de 16 points.

Le problème de l'enveloppe convexe sur les instances de taille n est de complexité $\Theta(n \log(n))$.

Leur borne inférieure est limitée au modèle de calcul basé sur les *arbres de décision algébriques de degré fixé*, qui sont des arbres de décision dont les noeuds ne comprennent que des calculs algébriques qui sont de degré fixé. Elle est obtenue par une réduction au problème de vérification de la taille de l'enveloppe convexe

qui consiste à vérifier si H est la taille de l'enveloppe convexe de S étant donnés S et H. Elle est basée sur un théorème général du domaine des calculs géométriques :

Théorème 1.4 ([BO83]) *Soit $W \subset \mathbb{R}^n$ un ensemble de points comprenant au moins N composantes connexes disjointes. Tout algorithme basé sur un arbre de décision algébrique de degré fini effectue $\Omega(\log N - n)$ comparaisons dans le pire des cas pour décider de l'appartenance d'un point à W.*

1.2.2 Tri de tableaux

Estivill-Castro et Wood [ECW92] rassemblent dans leur étude un état de l'art des algorithmes adaptatifs de tri, étudiés précédemment par Burge [Bur58], Cook et Kim [CK80], Dijkstra [Dij82], Guibas et al. [GMPR77], Knuth [Knu73], Mehlhorn [Meh79, Meh84] et Mannila [Man85]. Ils présentent les notions et concepts de base du tri adaptatif.

Le concept de base de l'étude des algorithmes adaptatifs de tri est que la performance d'un algorithme de tri dépend fortement de l'ordre initial de la séquence d'éléments à trier. Pour chaque analyse, la complexité du problème de tri est exprimée en fonction d'une mesure du "désordre" de la séquence d'éléments à trier. Les analyses sont effectuées pour la complexité dans le pire des cas ou pour la complexité en moyenne.

Des résultats de théorie de l'information donnent une borne inférieure à la complexité algorithmique du problème de tri, ce qui permet de définir l'optimalité d'un algorithme de tri pour une mesure :

Définition 1.9 (Optimalité par rapport à une mesure) *Soit M une mesure de désordre et A un algorithme de tri effectuant $C_A(I)$ comparaisons sur l'instance I. A est* optimal *par rapport à M (A est M-*optimal*) s'il existe une constance positive $c > 0$ telle que, pour toute instance I, $C_A(I)$ est soit inférieur à c fois la borne inférieure $\log_2 |\{J, |J| = n \text{ et } M(J) \leq M(I)\}|$ relative à la mesure M ; soit inférieur à c fois la taille de l'instance $|I|$.*

Chaque mesure correspond à un aspect différent de la difficulté de l'instance. Par exemple le nombre de couples (x_i, x_j) tels que $x_i > x_j$ pour $i < j$ (nombre d'inversions dans la séquence) est une mesure de désordre aux résultats intéressants, mais ne rend pas compte de l'ordre de la séquence $\mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}, \mathit{1}, \mathit{2}, \mathit{3}, \mathit{4}$.

Ces mesures sont ordonnées selon un ordre partiel selon qu'elles partitionnent l'ensemble des instances en partitions plus ou moins fines :

Définition 1.10 (Comparaison de mesures) *Soient $M_1, M_2 : \mathbb{N}^{\mathbb{N}} \to \mathbb{R}$ deux mesures de désordre.*

- *M_1 est* algorithmiquement plus fine *que M_2 ($M_1 <_{alg} M_2$) si et seulement si tout algorithme optimal pour M_1 est optimal pour M_2.*

– M_1 *et* M_2 *sont* algorithmiquement équivalentes *(*$M_1 =_{alg} M_2$*) si et seulement si* $M_1 <_{alg} M_2$ *et* $M_2 <_{alg} M_1$.

Onze mesures de désordre sont ordonnées selon un ordre partiel dans le contexte de l'analyse dans le pire des cas. L'adaptivité est également étudiée dans le contexte de la complexité en moyenne, et sur une variante du problème de tri, le tri externe.

1.2.3 Calcul de l'union et de l'intersection de tableaux triés

Demaine, Lopez-Ortiz et Munro [DLOM00] définissent des mesures de difficulté pour le calcul de l'union et de l'intersection de tableaux triés. Ces mesures sont basées sur la taille moyenne d'encodage d'un certificat du résultat de l'algorithme. Ils donnent des algorithmes adaptatifs et prouvent leur optimalité par rapport aux mesures définies. Leur analyse du calcul de l'intersection est détaillée dans le chapitre 2 sur l'intersection, section 2.3. Les principaux résultats de l'analyse du calcul de l'union sont donnés ici.

Le principal résultat est basé sur les remarques suivantes :

– Une description implicite de l'union de tableaux triés peut quelquefois être calculée rapidement même si l'union est de grande taille.
– Cette description de l'union correspond également à un *certificat* de l'union U, un ensemble de comparaisons vérifiées uniquement par les instances dont l'union est U.

Ils montrent une borne inférieure de la complexité du calcul de la description de l'union pour chaque instance de difficulté fixée, et donnent un algorithme qui atteint cette borne inférieure.

1.2.3.1 Borne inférieure

La *signature* d'une instance composée de k tableaux est le vecteur de $k+1$ entiers formé par k et les tailles de ces tableaux [DLOM00, section 2]. La *taille* n d'une instance est la somme des tailles des tableaux la composant.

L'union triée de tableaux triés est décrite par une suite de blocs composant les tableaux triés de l'instance : par exemple l'union de $A = \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{9}, \mathbf{10}$ et $B = 4, 5, 6, 7, 8$ est composée du bloc $\mathbf{1}, \mathbf{2}, \mathbf{3}$ de A, du bloc $4, 5, 6, 7, 8$ de B puis enfin du bloc $\mathbf{9}, \mathbf{10}$ de A.

$$\boxed{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{9}, \mathbf{10}} \cup \boxed{4, 5, 6, 7, 8} = \boxed{\mathbf{1}, \mathbf{2}, \mathbf{3}, 4, 5, 6, 7, 8, \mathbf{9}, \mathbf{10}}$$

FIG. 1.5 – Décomposition d'une union en blocs

Pour *vérifier* ce résultat, il faut et il suffit d'effectuer deux comparaisons, entre **3** (la fin du premier bloc de A) et *4* (le début du bloc de B), et entre *8* (la fin du bloc de B) et **10** (le début du deuxième bloc de A). Toutes les instances de deux tableaux vérifiant une inégalité stricte pour ces deux comparaisons se décomposent en blocs similaires. Les unions de ces instances sont décrites de manière identiques. Pour plus de deux tableaux, et dans le cas ou tous les éléments ne sont pas distincts, le concept reste le même : l'union de toute instance peut être exprimée en blocs et vérifiée par des comparaisons (inégalités ou égalités) entre les éléments aux extrémités de ces blocs.

Par définition de la correction d'un algorithme, tout algorithme calculant l'union sans erreur en vérifie un certificat. Le mot $c_u(P)$, de longueur $|c_u(P)|$, est défini comme codant un certificat P par la description des blocs composant l'union. La *difficulté* $\mathcal{C}_u(I)$ de l'instance I est par définition la taille minimale d'une représentation d'un certificat.

$$\mathcal{C}_u(I) = \min_{P \text{ certificat pour } I} |c_u(P)|.$$

Un algorithme calculant l'union doit distinguer toutes les instances à traiter. Dans le modèle des arbres de comparaisons, $\log N$ comparaisons sont nécessaires pour distinguer une instance parmi N. Avec le lemme suivant, cela permet de montrer une borne inférieure de la complexité de calcul de l'union :

Lemme 1.5 (Construction d'une famille d'instances [DLOM00], Th. 3.2)
Pour toute instance I, il existe $2^{\mathcal{C}_u(I)}$ instances I' de même signature et de coût $\mathcal{C}_u(I')$ inférieur à celui de I.

Théorème 1.5 (Borne inférieure en moyenne [DLOM00], Cor. 3.2) *Sur l'ensemble des instances* de coût $\mathcal{C}_u(I) \leq c$, tout algorithme déterministe calculant l'union effectue en moyenne au moins c comparaisons.*

1.2.3.2 Borne supérieure

Pour une difficulté fixée, cette borne inférieure est atteinte par un algorithme adaptatif :

Théorème 1.6 (Borne supérieure pour l'union [DLOM00], Th. 5-2) *Sur toute instance I du problème de calcul de l'union, l'algorithme de [DLOM00] effectue $O(\mathcal{C}_u(I))$ comparaisons.*

*L'énoncé original est "Any algorithm for the union problem requires at least $\mathcal{D}$ comparisons in the average case", où $\mathcal{D} = \mathcal{C}_u(I)$. Il est nécessaire de préciser sur quel ensemble d'instances la moyenne est effectuée.

1.2.4 Analyse fine

L'approche utilisée par Kirkpatrick et Seidel [KS86], Estivill-Castro et Wood [ECW92] et Demaine, Lopez-Ortiz et Munro [DLOM00] forme une nouvelle manière d'analyser la complexité de l'algorithme. Cela permet parfois d'orienter la création de nouveaux algorithmes, mais s'applique aussi aux algorihmes connus en permettant une analyse plus fine de leur complexité. La dénomination d'algorithme adaptatif masque la vraie dynamique de cette approche centrée sur l'analyse du problème plutôt que sur de nouveaux algorithmes. L'ordre partiel défini sur les instances par la mesure de difficulté est de peu d'utilité : le seul intérêt est de partitionner l'ensemble des instances. Ce partitionnement peut être défini par toute fonction à valeurs dans un espace vectoriel de dimension finie, définie sur l'ensemble des instances. Par la suite nous ferons référence à cette approche par le terme "analyse fine".

Un grand nombre de mesures de difficulté peuvent être définies pour un problème donné. Potentiellement, un grand nombre d'algorithmes peuvent être définis, chacun optimal pour une mesure de difficulté. Il est nécessaire de définir des critères permettant de juger de la qualité d'une mesure de difficulté. Certaines mesures de difficulté peuvent correspondre au contexte particulier d'une application pratique. Le manque d'informations sur les distributions réelles des instances dans l'étude de la complexité en moyenne témoigne de la difficulté de cette approche. Une manière de comparer des mesures de difficulté est de considérer pour chaque mesure l'écart entre la complexité dans le pire des cas et la complexité dans le meilleur des cas. Si l'écart est suffisamment réduit, un algorithme optimal pour cette mesure est alors optimal pour toute mesure, dans le pire et le meilleur des cas, et en moyenne.

Dans le problème des pièces cachées par exemple, les complexités probabilistes dans le pire des cas et dans le meilleur des cas pour un nombre de pièces d'argent fixé sont égales : l'analyse est optimale pour la complexité probabiliste.

Les Chapitres 2 et 3 présentent des analyses fines de la complexité probabiliste dans le pire des cas du calcul de l'intersection de tableaux triés. Les Chapitre 4 et 5 présentent des analyses fines de la complexité probabiliste dans le pire des cas d'autres calculs sur les tableaux triés. Ces calculs apparaissent notamment dans les moteurs de recherche indexés, dont les principes sont exposés dans la section suivante.

1.3 Moteurs de recherche indexés

Soit le méta-problème suivant : dans un ensemble de documents, étant donnés k critères, trouver tous ceux satisfaisant une combinaison de ces critères.

Les données accessibles sur Internet sont par essence peu ordonnées, et nombreuses. Les moteurs de recherche sont des outils incontournables pour y trouver des documents pertinents.

Les moteurs de recherche disponibles sur Internet proposent de résoudre ce "problème" sur une partie des documents textuels accessibles sur le réseau. Dans le contexte des *requêtes conjonctives* un document doit correspondre à chacun des critères. Les critères sont des mots-clés, et peuvent relever de la présence du mot-clé dans le document, ou de la présence du mot-clé dans un lien hyper-texte vers ce document. En particulier les critères sont indépendants : étant donnés deux mots-clés a et b la présence de a n'entraîne pas la présence de b.

Chaque moteur de recherche précalcule un index. Un index est une structure associant à chaque mot-clé une liste de références aux documents associés à ce mot-clé.

Chaque requête composée de k mots-clés est résolue par :

– une recherche dans l'index pour chaque mot-clé, pour trouver la liste associée à ce mot-clé ; puis
– une intersection entre les k listes ainsi obtenues.

L'accent est mis sur le grand nombre de documents indexés et de requêtes quotidiennes, ainsi que sur la rapidité des réponses. Le processus de réponse doit être optimisé pour réduire le temps d'attente de l'utilisateur, mais aussi pour pouvoir répondre à plus de requêtes sur le même serveur.

Trier les listes de références sous la forme de tableaux permet d'optimiser le processus de réponse[†]. Ceci permet de choisir des algorithmes d'intersection plus rapides tirant parti de cet ordre.

Les Chapitres 2 et 3 présentent des études d'algorithmes d'intersection de tableaux triés, qui ont pour but d'optimiser la résolution des requêtes conjonctives. Les Chapitre 4 et 5 généralisent ces résultats à d'autres combinaisons de critères (requêtes plus complexes). Le Chapitre 6 présente l'index d'un moteur de recherche existant.

[†]Bien sûr, maintenir ces listes triées se fait au détriment des processus de construction et de mise à jour de l'index.

Chapitre 2

Intersection : État de l'art

Dans ce second chapitre nous présentons de manière formelle le problème du calcul de l'intersection et prouvons une borne inférieure de la complexité de calcul de l'intersection. Nous exposons un algorithme standard `Deux-à-deux` qui atteint cette borne (section 2.1). Nous exposons les algorithmes d'intersection `Parallèle` (section 2.3), et `Séquentiel` (section 2.4) donnés par Demaine, Lopez-Ortiz et Munro [DLOM00, DLOM01], et présentons de nouvelles preuves (inspirées des résultats sur la mesure d'alternance [BK02] présentées dans le chapitre 3) de leur borne inférieure démontrant l'optimalité de l'algorithme `Parallèle` pour les algorithmes déterministes. Nous complétons ces résultats et montrons comment les étendre à la complexité probabiliste (section 2.5).

2.1 Analyse Classique

Définition 2.1 (Intersection) *Soit $\mathcal{U}$ un espace totalement ordonné. Une instance du problème d'intersection est composée de k tableaux triés $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$ et composés d'éléments de $\mathcal{U}$. L'*intersection $A_1 \cap \ldots \cap A_k$ *de cette instance est le tableau trié des éléments présents dans tous les tableaux.*

Définition 2.2 (signature et taille) *La* signature *d'une instance est un vecteur composé du nombre de tableaux k, et des tailles des tableaux $n_1, \ldots, n_k$. La* taille *d'une instance est la somme $n = n_1 + \ldots + n_k$ des tailles de ces tableaux.*

Théorème 2.1 (Borne inférieure) *Pour tout n, tout algorithme calculant l'intersection effectue au moins $n/2 = \Omega(n)$ comparaisons dans le pire des cas pour les instances de taille n.*

Preuve.
Pour n fixé, soit I l'instance composée de deux tableaux triés identiques de taille $n/2$. Un algorithme calculant l'intersection doit vérifier chacune des $n/2$ égalités, ce qui implique une borne inférieure de $\Omega(n)$ comparaisons. □

Par exemple, dans l'exemple suivant, tout algorithme doit *vérifier* 5 égalités pour conclure sur l'intersection de A et B.

$A =$	1	2	3	4	5
$B =$	1	2	3	4	5

L'intersection de l'instance construite précédemment comprend $n/2$ éléments. La remarque suivante montre que la difficulté de l'instance ne provient pas du cardinal de sa solution : il existe une instance d'intersection de cardinal arbitraire qui nécessite $\Omega(n)$ comparaisons.

Remarque 2.1 *Soit $s \leq \lfloor n/2 \rfloor$. Tout algorithme calculant l'intersection effectue au moins $n/2 = \Omega(n)$ comparaisons dans le pire des cas pour les instances de taille n et dont l'intersection comprend s éléments.*

Preuve.
Pour n fixé, et un entier s entre 0 et $\lfloor n/2 \rfloor$ inclus, soient s nombres pairs distincts c_i choisis entre 0 et n exclus. Soit I l'instance composée de deux tableaux triés construits comme suit :

- Le premier tableau A contient les entiers impairs entre 1 et $n+1$ exclus ;
- Le second tableau B contient les entiers pairs entre 0 et n à l'exception des nombres $(c_i)_{i \leq s}$, remplacés par les nombres $(c_i + 1)_{i \leq s}$.

La solution de cette instance est de taille s. Tout algorithme résolvant cette instance doit effectuer au moins $n/2$ comparaisons pour distinguer les instances générées par les différents choix possibles de s et $(c_i)_{i \leq s}$. D'où une borne inférieure de $\Omega(n)$ du nombre de comparaisons. □

Par exemple, dans l'exemple suivant, l'intersection est de taille 1, l'élément choisi étant $c_1 = 4$, remplacé par 5. Tout algorithme doit effectuer 9 comparaisons pour conclure sur l'intersection de A et B.

$A =$	1	3	**5**	7	9
$B =$	0	2	**5**	6	8

Cette borne inférieure est aisément atteinte par un algorithme naïf.

Lemme 2.1 *L'algorithme* `Intersection naïve` *effectue $O(n)$ comparaisons pour calculer l'intersection d'une instance de taille n.*

Algorithme 2 `Intersection naïve`$(A_1, \ldots, A_k)$

Étant donné k tableaux triés non vides $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$, l'algorithme retourne leur intersection.

si $k = 1$ **alors**
  retourner A_1
sinon si $k = 2$ **alors**
  $I \leftarrow \emptyset$; $a \leftarrow 1$; $b \leftarrow 1$
  tant que $a \leq \#A$ et $b \leq \#B$ **faire**
    si $A[a] < B[b]$ **alors**
      $a \leftarrow a + 1$
    sinon si $A[a] > B[b]$ **alors**
      $b \leftarrow b + 1$
    sinon
      $I \leftarrow I \cup \{m\}$
      $a \leftarrow a + 1$; $b \leftarrow b + 1$
    fin si
  fin tant que
  retourner I
sinon
  $I \leftarrow$ `Intersection naïve`(A_1, A_2)
  retourner `Intersection naïve`$(I, A_3, \ldots, A_k)$
fin si

Preuve.
Si $k = 1$, l'algorithme n'effectue que le test d'égalité de k avec 1. Si $k = 2$, chaque élément est comparé au plus une fois : l'algorithme effectue au plus $O(n)$ comparaisons.

Si $k > 2$, et que les tableaux sont de tailles respectives $n_1, \ldots, n_k$, l'algorithme calcule l'intersection des deux premiers tableaux, puis récursivement l'intersection des $k-1$ tableaux restants. Comme la taille de l'intersection de deux tableaux est inférieure à la taille de chaque tableau, l'algorihme effectue au plus $n_1+n_2+\min(n_1,n_2)+n_3+\min(n_1,n_2,n_3)+n_4+\ldots+\min(n_1,\ldots,n_{k-1})+n_k$ comparaisons. L'algorithme effectue donc plus $2(n_1+\ldots+n_k) = O(n)$ comparaisons. □

En pratique, on utilise des algorithmes plus sophistiqués qui ont une meilleure performance sur les instances réelles.

2.2 Un algorithme standard d'intersection

L'algorithme d'intersection `SvS` est un algorithme standard utilisé dans des moteurs de recherches [DLOM01, ligne 3, page 3]. L'algorithme d'intersection `Deux-à-deux` (algorithme 6) en est une variante reposant sur le même principe : il fait appel aux algorithmes de `Recherche doublante` (algorithme 3 [Meh84]), de `Recherche dichotomique` (algorithme 4 [M.L75]) et d'`Intersection simple` (algorithme 5) qui sont décrits ici et utilisés dans les autres algorithmes d'intersection.

Lemme 2.2 *L'algorithme* `Recherche doublante` *effectue* $\lceil \log_2(r+1) \rceil$ *comparaisons pour trouver un intervalle contenant un élément x à partir de la position p si le rang d'insertion de x est $p+r$.*

Preuve.
L'algorithme cherche le couple $(gauche, droite)$ à partir de la position p incluse, de la gauche vers la droite, en doublant la taille de l'intervalle à chaque comparaison : il effectue exactement $\lceil \log_2(r+1) \rceil$ comparaisons si le rang d'insertion de l'élément est $p+r$. □

Lemme 2.3 *Pour chercher un élément dans un tableau trié entre les positions g et d l'algorithme* `Recherche dichotomique` *effectue* $\lceil \log_2(d-g+1) \rceil$ *comparaisons.*

Preuve.
L'invariant de boucle est $A[gauche] < x \leq A[droite]$. Initialement l'intervalle est de taille strictement positive. À chaque comparaison, la taille $d-g$ de cet

Algorithme 3 `Recherche doublante`(x, A, p)

Étant donné un élément x, un tableau A et une position de départ $p < \#A$, l'algorithme retourne un couple de valeurs $(gauche, droite)$ telles que $p \leq gauche < droite \leq \#A$ et $A[gauche] < x \leq A[droite]$.

$pas \leftarrow 1$; $position \leftarrow p$

tant que $position \leq \#A$ et $A[position] < x$ **faire**

$pas \leftarrow 2 \times pas$

$position \leftarrow position + pas$

fin tant que

retourner $(position - pas, \min(\#A, position))$

Algorithme 4 `Recherche dichotomique`(x, A, g, d)

Étant donné un élément x, un tableau A et deux positions $gauche$, $droite$ telles que $A[gauche] < x \leq A[droite]$, l'algorithme retourne la valeur d telle que $A[d-1] < x \leq A[d]$: c'est la "position d'insertion" de x dans A.

si $droite - gauche = 1$ **alors**

retourner $droite$

sinon

$milieu \leftarrow \lfloor (gauche + droite)/2 \rfloor$

si $x \leq A[milieu]$ **alors**

retourner `recherche dichotomique`$(x, A, gauche, milieu)$

sinon

retourner `recherche dichotomique`$(x, A, milieu, droite)$

fin si

fin si

intervalle est divisée par deux. L'algorithme termine quand la taille de cet intervalle est réduite à un : il effectue donc $\lceil \log_2(d-g+1) \rceil$ comparaisons. □

Algorithme 5 `Intersection simple`(A, B)

Étant donné deux tableaux triés A et B, l'algorithme retourne leur intersection. I ne contient pas de répétition si ni A ni B n'en contiennent.

$I \leftarrow \emptyset$; $a \leftarrow 1$; $b \leftarrow 1$
tant que $a \leq \#A$ et $b \leq \#B$ **faire**
  $m \leftarrow \max\{A[a], B[b]\}$
  si $m = A[a]$ **alors**
    $(g, d) \leftarrow$ `Recherche doublante`(m, B, b)
    $b \leftarrow$ `Recherche dichotomique`(m, B, g, d)
    si $m = B[b]$ **alors**
      $I \leftarrow I \cup \{m\}$
      $a \leftarrow a + 1$; $b \leftarrow b + 1$
    sinon
      $a \leftarrow a + 1$;
    fin si
  sinon
    Faire de même dans A.
  fin si
fin tant que
retourner I

Lemme 2.4 *L'algorithme* `Intersection simple` *effectue* $O(n_1 + n_2)$ *comparaisons pour calculer l'intersection de deux tableaux de tailles* n_1 *et* n_2.

Preuve.
L'algorithme parcourt les deux tableaux de gauche à droite par une succession de recherches doublantes et dichotomiques, jusqu'à ce que l'un des tableaux soit totalement parcouru.

À tout instant, tous les éléments de l'intersection inférieurs à m sont dans I. Les valeurs prises par m sont strictement croissantes. L'algorithme effectue donc

au plus $n_1 + n_2$ recherches doublantes, et autant de recherches dichotomiques, soit $2(n_1 + n_2)$ comparaisons. □

Algorithme 6 `Deux-à-deux`$(A_1, \ldots, A_k)$ [DLOM01]

Étant donné k tableaux triés non vides $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$, l'algorithme retourne leur intersection.

si $k = 1$ **alors**
  retourner A_1
sinon
  Ordonner les tableaux par taille croissante.
  $I \leftarrow$ `Intersection simple`(A_1, A_2)
  retourner `Deux-à-deux`$(I, A_3, \ldots, A_k)$
fin si

Théorème 2.2 *Pour tout n, l'algorithme* `Deux-à-deux` *effectue $O(n+k\log(k))$ comparaisons sur une instance de taille n composée de k tableaux, dont $O(n)$ comparaisons d'éléments.*

Preuve.
Soient $n_1 \leq \ldots \leq n_k$ les tailles des tableaux à intersecter. Le tri des tableaux par taille croissante nécessite $k \log k$ comparaisons de tailles. Par récurrence sur k, on montre que le calcul de l'intersection de $(A_1, \ldots, A_k)$ nécessite au plus $kn_1 + n_2 + \ldots + n_k$ comparaisons d'éléments.

Si $k = 1$, l'algorithme ne compare aucun élément, l'hypothèse de récurrence est vérifié. Si $k > 1$, l'algorithme `Intersection simple` effectue moins de $n_1 + n_2$ comparaisons, et l'intersection I est de taille au plus n_1. Par récurrence, le calcul de l'intersection de $(I, A_3, \ldots, A_k)$ nécessite moins de $(k-1)n_1 + n_3 + \ldots + n_k$ comparaisons. Soit en tout moins de $kn_1 + n_2 + \ldots + n_k$ comparaisons.

Comme n_1 est la plus petite des tailles de tableaux, l'algorithme effectue $kn_1 + n_2 + \ldots + n_k \leq n_1 + 2n_2 + \ldots 2n_k < 2n = O(n)$ comparaisons d'éléments.

L'algorithme effectue donc en tout $O(n + k \log k)$ comparaisons, dont $O(n)$ comparaisons d'éléments. □

L'algorithme `Deux-à-deux` est optimal dans le pire des cas pour les instances de taille n fixée. Cet algorithme tire parti de la répartition des tailles des tableaux. Il peut être orienté par ces tailles vers un calcul compliqué, sur des instances faciles à résoudre.

Dans l'exemple suivant, l'algorithme `Deux-à-deux` calcule l'intersection $A \cap B \cap C$ avant de considérer le tableau D, effectuant 10 comparaisons quand une simple comparaison entre le dernier élément du tableau A et le premier de D, $A[5] < D[1]$, prouve que l'intersection est vide.

$A =$	1	2	3	4	5	
$B =$	1	2	3	4	5	
$C =$	1	2	3	4	5	
$D =$	6	7	8	9	10	11

Se basant sur cette remarque, Demaine, Lopez-Ortiz et Munro ont introduit une mesure de difficulté plus fine pour l'intersection, et un algorithme adapté à cette mesure.

2.3 Analyse de Demaine, Lopez-Ortiz et Munro

Demaine, Lopez-Ortiz et Munro [DLOM00] transposent leur analyse de l'union (résumée section 1.2.3) à l'intersection de tableaux triés. Comme pour l'union, ils définissent la notion de "certificat" pour l'intersection, et de "coût" d'une instance. Le *coût* d'une instance est une mesure de difficulté définie comme la taille minimale d'"encodage" d'un certificat de l'instance.

Un argument de stratégie d'adversaire permet d'obtenir une borne inférieure qui est atteinte par l'algorithme `Parallèle` (algorithme 7).

Un *argument* est une suite finie d'égalités et d'inégalités symboliques de la forme "$A_s[i] < A_t[j]$" ou "$A_s[i] = A_t[j]$". Une instance *satisfait* un argument si chaque comparaison de l'argument est vérifiée par cette instance.

Définition 2.3 (Certificat) *Un argument P est un* certificat *pour une instance I du problème d'intersection si I satisfait P et que P caractérise l'intersection : les éléments de l'intersection de toute instance satisfaisant P sont aux même positions que ceux de l'intersection de I. [DLOM00, section 2.1]*

Pour une instance I fixée, un certificat P est minimal *s'il n'existe pas de certificat pour I comprenant moins de comparaisons que P [DLOM00, section 3.1].*

Demaine, Lopez-Ortiz et Munro limitent tout d'abord leur étude aux instances d'intersection vide. Les certificats de ces intersections ne comprennent que des comparaisons, et aucune égalité. Un code $c_i(P)$ est associé à chaque certificat P pour le coder. Les comparaisons sont ordonnées par les indices des tableaux concernés et les indices des éléments comparés. Le code de chaque comparaison $A_s[i] : A_t[j]$ se divise en deux parties identifiant les deux tableaux s et t contenant les éléments à comparer, et les positions i et j à comparer.

Les indices des tableaux concernés sont codés par $\log_2(k(k-1))$ bits pour chaque comparaison. La somme des tailles d'encodage des indices de tableaux des comparaisons du certificat est notée $s(P)$. Les positions à comparer sont encodées par tableau de la manière suivante : pour chaque tableau A_i, soit l le nombre d'éléments de A_i intervenant dans P, soient $p_1, \ldots, p_l$ leurs positions, et soient $p_0 = 0$, $p_{l+1} = |A_i| + 1$. Pour $j = 1, \ldots, l+1$ soient $g_j = 1 + p_j - p_{j-1}$ les nombres d'éléments séparant ces positions . La longueur du code précisant les positions des éléments comparés par P dans A_i est $g(P, A_i) = \sum_j \log_2(g_j) - \max_j \log_2(g_j)$. La longueur du code précisant les positions des éléments comparés dans P est ainsi $g(P) = \sum_i g(P, A_i)$.

Au total, le code $c_i(P)$ est de longueur $|c_i(P)| = s(P) + g(P)$.

Définition 2.4 (Coût $\mathcal{C}_i(I)$ [DLOM00, section 3.1]) *Le* coût $\mathcal{C}_i(I)$ *d'une instance d'intersection vide I est $\mathcal{C}_i(I) = \min_P |c_i(P)|$.*

Cette mesure correspond au nombre maximal de bits nécessaires pour exprimer un certificat. C'est une borne inférieure du nombre moyen de comparaisons que doit effectuer un algorithme pour calculer l'intersection. Cette borne inférieure n'est pas optimale : un certificat ne caractérise pas l'instance comme dans l'union, et peut être décrit de manière courte.

La mesure de dispersion permet d'obtenir une meilleure borne par un argument de stratégie d'adversaire.

Définition 2.5 (Dispersion $\mathcal{G}(I)$ [DLOM00, section 3.1]) *La* dispersion $\mathcal{G}(I)$ *d'une instance d'intersection vide est la longueur minimale de codage des positions des éléments comparés par un certificat pour I : $\mathcal{G}(I) = \min_P g(P)$.*

Cette mesure correspond à la "dispersion" des éléments comparés par le certificat : elle augmente logarithmiquement avec l'espacement des éléments comparés.

Théorème 2.3 (Borne inférieure élémentaire) *Étant donnés $k > 1$ et $g > \log_2 k$, tout algorithme* déterministe *calculant l'intersection effectue $\Omega(kg)$ comparaisons dans le pire des cas pour les instances sur k tableaux de dispersion $O(g)$.*

Preuve.
Soit un adversaire construisant des instances composées chacune d'un tableau A_1 contenant un unique élément x, et de $k-1$ tableaux triés $A_2, \ldots, A_k$ contenant chacun $2^g/k$ éléments, dont $k-2$ contiennent x et un seul ne contient pas x. L'hypothèse $g > \log_2 k$ permet d'affirmer qu'aucun de ces tableaux n'est vide. Pour tout $1 < i < k$, les éléments inférieurs à x de A_i sont tous plus petits que les éléments de A_{i+1}. De manière symétrique, les éléments supérieurs à x de A_{i+1} sont tous plus grands que les éléments de A_i.

L'intersection de ces instances est nulle. De plus, chacune de ces instances est de dispersion $O(g)$: au plus $\log\ k$ bits sont nécessaires pour décrire l'indice du tableau ne contenant pas x, et $\log\ (2^g/k) = g - \log k$ bits sont nécessaires pour décrire la position de x dans ce tableau.

Initialement toutes les instances sont possibles. Si l'algorithme pose une question ne concernant pas A_1, du type "$A_s[i] : A_t[j]$" avec $s < t$, l'adversaire répond

- "=" si $A_t[j] = x = A_s[i]$,
- ">" si $A_t[j] \leq x \leq A_s[i]$ mais $A_t[j] \neq A_s[i]$,
- "<" si $A_s[i] \leq x \leq A_t[j]$ ou $A_s[i] \leq\ A_t[j] \leq\ x$ ou $x \leq A_s[i] \leq\ A_t[j]$ mais $A_s[i] \neq A_t[j]$.

Si l'algorithme pose une question concernant A_1, l'adversaire choisit le résultat de la comparaison de manière à maximiser le nombre d'instances cohérentes avec l'ensemble des résultats des comparaisons déjà effectuées. Le nombre d'instances possibles se réduit au fur et à mesure des comparaisons effectuées par l'algorithme et des résultats choisis par l'adversaire. L'algorithme conclut quand il ne reste qu'une seule instance.

Tout algorithme doit prouver que x n'est pas dans l'intersection. Pour cela il doit trouver le tableau ne contenant pas x et le rang d'insertion de x dans ce tableau. Dans chaque tableau, l'adversaire force l'algorithme à effectuer $\log(2^g/k) = g - \log k$ comparaisons avant d'indiquer la présence ou l'absence de x. L'adversaire force l'algorithme à chercher le rang de x dans $k-1$ tableaux avant d'indiquer l'absence de x dans le dernier tableau recherché. L'algorithme effectue donc en tout $k(g - \log k) = \Omega(kg)$ comparaisons. □

L'instance générée par l'adversaire est de taille $1 + \frac{k-1}{k} 2^{\ g}$. Ce nombre est réduit à $O(p(1+k2^{g/p}))$ par le corollaire suivant. Le corollaire 2.2 dans la section 2.5 réduit ce nombre à $O(g+k)$.

Corollaire 2.1 (Borne inférieure [DLOM00, théorème 4.1]) *Pour tout $k > 1$, g et $p \leq g$, tout algorithme* déterministe *calculant l'intersection effectue $\Omega(kg)$ comparaisons dans le pire des cas pour les instances sur k tableaux de dispersion $O(g)$ et de taille $O(p(1+k2^{g/p}))$.*

Preuve.
Soit un adversaire construisant p instances comme dans le théorème 2.3, chacune de dispersion g/p et telle que tous les éléments de la première instance soient inférieurs à ceux de la seconde, tous ceux de la seconde inférieurs à ceux de la troisième, et ainsi de suite jusqu'à la dernière instance.

On considère les instances obtenues en prenant comme $i$ème tableau l'union des $i$èmes tableaux de ces p sous-instances. Ces instances sont d'intersection vide, elles comprennent $p(1 + \frac{k-1}{k} 2^{\frac{q}{p}})$ éléments et leur dispersion est $O(g)$.

Tout algorithme doit prouver que les éléments du premier tableau ne sont pas dans l'intersection. Résoudre cette instance correspond à résoudre les p sous-instances de manière indépendante. L'adversaire fait effectuer à l'algorithme $\Omega(kg/p)$ comparaisons sur chaque sous-instance, c'est à dire en tout $\Omega(kg)$ comparaisons. □

Cette borne inférieure est atteinte par l'algorithme d'intersection `Parallèle` (algorithme 7). Cet algorithme utilise les algorithmes de recherche doublante et dichotomique qui ont été définis section 2.1 (algorithme 3 et 4). Demaine, Lopez-Ortiz et Munro donnent un algorithme [DLOM00] un peu plus compliqué, parcourant les tableaux à la fois de gauche à droite et de droite à gauche, mais dont la performance et l'analyse dans le pire des cas sont les mêmes.

Théorème 2.4 (Complexité de l'algorithme [DLOM00, Th. 5.1]) *Pour tout k, l'algorithme* `Parallèle` *effectue $O(k\mathcal{G}(I))$ comparaisons sur toute instance I sur k tableaux.*

Preuve.
À tout moment l'algorithme teste l'appartenance de m à l'intersection et tous les éléments de l'intersection inférieurs à m sont dans I. La suite des valeurs successives de m est strictement croissante. Un élément de valeur m est cherché dans $k-1$ tableaux en parallèle, de gauche à droite, par recherche doublante. Chaque recherche doublante qui se termine est suivie d'une recherche dichotomique de m. L'algorithme termine quand aucun élément supérieur à m ne peut plus appartenir à l'intersection, c'est à dire quand tous les tableaux ont leur dernier élément inférieur à m.

Soit une instance I de dispersion $\mathcal{G}(I) = g$. Appelons "comparaisons doublantes" les comparaisons effectuées lors d'une recherche doublante, et "comparaisons dichotomiques" celles effectuées lors d'une recherche dichotomique.

Toute recherche dichotomique est effectuée sur un intervalle de taille 2^j obtenu par une recherche doublante de j pas. Le nombre de comparaisons effectué lors de cette recherche doublante est j, celui effectué lors de la recherche dichotomique est également j. Le nombre total de comparaisons dichotomiques étant inférieur ou égal au nombre de comparaisons doublantes, il suffit de borner le nombre de comparaisons doublantes effectué par l'algorithme.

Soit P un certificat de l'instance telle que $\mathcal{G}(P) = g$. Dans le tableau A_i, soit $(x^i_j)_j$ la suite indexée par j des éléments comparés dans le certificat, triés par ordre croissant. Pour chaque élément x^i_j, soit t^i_j le nombre d'éléments le séparant du plus proche élément à sa gauche, ou du premier élément de A_i. Pour chacune de ces valeurs, soit $g^i_j = \log t^i_j : g = \sum_{i,j} g^i_j$.

Algorithme 7 `Parallèle`$(A_1, \ldots, A_k)$

Étant donné k tableaux triés non vides $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$, l'algorithme retourne leur intersection I.

$I \leftarrow \emptyset$; **pour tout** i **faire** $pas_i \leftarrow 1$; $p_i \leftarrow 1$; $marque_i = FAUX$ **fin pour**
$M \leftarrow \min\{A_1[n_1], \ldots, A_k[n_k]\}$; $m \leftarrow \max\{A_1[1], \ldots, A_k[1]\}$
$i \leftarrow j$ tel que $m = A_j[1]$; $marque_i = VRAI$
tant que $m \leq M$ **faire**
  Soit A_i le prochain tableau non marqué dans l'ordre cyclique.
  $pas_i \leftarrow 2 \times pas_i$; $p_i \leftarrow p_i + pas_i$
  si $m \leq A_i[p_i]$ **alors**
    $p_i \leftarrow$ `Recherche dichotomique`$(m, A_i, p_i - pas_i, p_i)$
    $pas_i \leftarrow 1$
    si $m = A_i[p_i]$ **alors**
      $marque_i \leftarrow VRAI$
      si $\forall i \, marque_i = VRAI$ **alors**
        $I \leftarrow I \cup \{m\}$
        pour tout i **faire** $p_i \leftarrow p_i + 1$ **fin pour**
      fin si
    fin si
    si $m \in I$ ou $m \notin A_i$ **alors**
      $m \leftarrow A_i[p_i]$
      pour tout i **faire** $marque_i \leftarrow FAUX$ **fin pour**
      $i \leftarrow j$ tel que $m = A_j[p_j]$; $marque_i = VRAI$
    fin si
  fin si
fin tant que
retourner I

Si x_j^i n'est pas dans l'intersection, l'algorithme effectue au plus g_j^i comparaisons doublantes dans A_i pour le trouver, c'est à dire au total au plus kg_j^i comparaisons doublantes. Si x_j^i est dans l'intersection, l'algorithme effectue au plus g_j^i comparaisons doublantes dans A_i.

Au total l'algorithme effectue donc au plus $k\sum_{i,j}\ g_j^i$ comparaisons doublantes, soit $O(kg)$. □

2.4 Résultats pratiques

Demaine, Lopez-Ortiz et Munro testent leur algorithme d'intersection sur des instances obtenues à partir de requêtes au moteur de recherche `Google`. Ils testent également l'algorithme `SvS`, une variante de l'algorithme d'intersection `Deux-à-Deux`, et comparent les résultats des deux algorithmes.

Comme la différence de performance entre les algorithmes `SvS` et `Parallèle` (algorithmes 6 et 7) n'est pas significative, ils étudient les différences entre les deux algorithmes et produisent un troisième algorithme d'intersection, l'algorithme `Séquentiel` (algorithme 8).

À tout moment l'algorithme teste l'appartenance de m à l'intersection et tous les éléments de l'intersection inférieurs à m sont dans I. La suite des valeurs successives de m est strictement croissante. L'algorithme effectue une recherche doublante à la fois, toujours suivie d'une recherche dichotomique. L'algorithme termine quand aucun élément supérieur à m ne peut plus appartenir à l'intersection, c'est à dire quand tous les tableaux ont leur dernier élément inférieur à m.

Cet algorithme est plus efficace sur les instances de l'étude, mais n'est pas optimal par rapport aux mesures de dispersion ou de coût. Dans la section suivante nous généralisons les résultats de la section 2.3 et montrons que les algorithmes `Deux-à-deux` et `Séquentiel` ne sont pas optimums.

2.5 Résultats complémentaires

Dans le corollaire 2.1 [DLOM00, théorème 4.1], la taille des instances construites est exponentielle en g/p. Le corollaire suivant montre que la taille des instances n'est pas la cause de leur difficulté :

Corollaire 2.2 (Borne inférieure) *Pour tout $k > 1$, $g > k$ et $n = \Omega(g + k)$, tout algorithme* déterministe *calculant l'intersection effectue $\Omega(kg)$ comparaisons dans le pire des cas pour les instances sur k tableaux de dispersion $O(g)$ et de taille $O(n)$.*

Algorithme 8 `Séquentiel`$(A_1, \ldots, A_k)$[DLOM01]

Étant donné k tableaux triés non vides $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$, l'algorithme retourne leur intersection I.

$I \leftarrow \emptyset$; $o \leftarrow 1$
pour tout i **faire** $p_i \leftarrow 1$ **fin pour**
$M \leftarrow \min\{A_1[n_1], \ldots, A_k[n_k]\}$
$m \leftarrow \max\{A_1[1], \ldots, A_k[1]\}$
$i \leftarrow j$ tel que $m = A_j[1]$
tant que $m \leq M$ **faire**
  $i \leftarrow (i-1)$ modulo $k+1$
  $(g, d) \leftarrow$ `Recherche doublante`(m, A_i, p_i)
  $p_i \leftarrow$ `Recherche dichotomique`(m, A_i, g, d)
  si $m = A_i[p_i]$ **alors**
    $o \leftarrow o + 1$.
    si $o = k$ **alors**
      $I \leftarrow I \cup \{m\}$
      pour tout i **faire** $p_i \leftarrow p_i + 1$ **fin pour**
    fin si
  fin si
  si $o = k$ ou $m \neq A_i[p_i]$ **alors**
    $m \leftarrow A_i[p_i]$
    $o \leftarrow 1$
  fin si
fin tant que
retourner I

Preuve.
Dans le corollaire 2.1 la taille des instances générées est $p(1 + k2^{\frac{g}{p}})$. Cette valeur est minimale pour $p = \lceil g \ln 2 \rceil$, égale à $p + k = \lceil g \ln 2 \rceil + k = \Omega(g + k)$. Les valeurs plus grandes de p permettent d'obtenir les valeurs plus grandes de n. □

Chaque instance construite par l'adversaire dans le théorème 2.3 et le corollaire 2.1 est difficile pour un grand nombre d'algorithmes déterministes : ces instances sont difficiles pour les algorithmes probabilistes aussi. Les mêmes techniques que celles utilisées dans le chapitre 3 [BK02] permettent de généraliser la borne inférieure de $\Omega(kg)$ aux algorithmes probabilistes. La preuve n'est pas développée ici parce que redondante avec celle du chapitre 3.

Remarque 2.2 (Borne inférieure probabiliste) *Pour tout $k > 1$, $g > k$ et $n = \Omega(g + k)$, tout algorithme* probabiliste *calculant l'intersection effectue $\Omega(kg)$ comparaisons dans le pire des cas pour les instances de taille $O(n)$, sur k tableaux et de dispersion $O(g)$.*

La non-optimalité des algorithmes `Deux-à-Deux` et `Séquentiel` est prouvée par un contre-exemple :

Remarque 2.3 *Les algorithmes* `Deux-à-deux` *et* `Séquentiel` *ne sont pas optimaux par rapport aux mesures de coût et de dispersion.*

Preuve.
Dans l'instance suivante composée de trois tableaux A, B, C, tous les éléments de A peuvent être insérés à une unique position dans B ou C. L'intersection est vide, et le certificat $P =$ "$C[1] < A[1], A[6] < C[2]$" est minimal pour les mesures de coût et de dispersion.

$A =$	10	11	12	13	14	15		
$B =$	1	2	3	4	16	17	18	19
$C =$	1	16	17	18	19	20	21	22

Les algorithmes `Deux-à-deux` et `Séquentiel` parcourent entièrement B pour trouver la position d'insertion de 10. Ils effectuent 3 comparaisons. Mais l'algorithme `Parallèle` parcourt B et C en parallèle et trouve la position d'insertion de 10 dans C après seulement 2 comparaisons : les algorithmes `Deux-à-deux` et `Séquentiel` ne sont pas optimaux. □

L'algorithme `Parallèle` est optimal pour les mesures de dispersion et de coût, mais celles-ci ne mesurent pas bien la difficulté des instances pratiques, où l'algorithme `Séquentiel` est meilleur.

Dans le chapitre suivant nous définissons la mesure d'*alternance* qui mesure différemment la difficulté d'une instance, et pour laquelle l'algorithme `Séquentiel` est optimal.

Chapitre 3

Alternance de l'intersection

Dans ce chapitre nous présentons une nouvelle mesure de difficulté des instances d'intersection de tableaux triés, l'*alternance*. Cette mesure est liée à la *complexité non déterministe* du problème. L'analyse est effectuée pour les instances de *taille* (somme des tailles des tableaux) et d'alternance fixées, puis pour les instances de *signature* (vecteur des tailles individuelles des tableaux) et d'alternance fixées. Ces mesures sont de plus en plus fines : sur les instances d'alternance fixée, toute borne inférieure pour les instances de signature fixée implique une borne inférieure pour les instances de taille fixée.

Nous donnons une borne inférieure de la complexité probabiliste de l'intersection dans le pire des cas pour les instances de *signature* et d'alternance fixée, qui implique une borne inférieure pour les instances de *taille* et d'alternance fixée.

L'algorithme `Deux-à-deux` n'est pas optimal dans le pire des cas, ni pour les instances de taille et d'alternance fixées, ni pour les instances de signatures et d'alternance fixées. L'algorithme `Parallèle` est optimal dans le pire des cas pour les instances de *taille* et d'alternance fixées, mais n'est pas optimal pour les instances de *signature* et d'alternance fixées. L'algorithme `Séquentiel` est optimal dans le pire des cas pour les instances de signature et d'alternance fixées, et donc pour les instances de taille et d'alternance fixées.

3.1 Mesure d'alternance

Définition 3.1 (Partition Certificat) *Une* partition certificat $P = (I_j)_{1 \leq j \leq \delta}$ *d'une instance composée de k tableaux triés $A_1, \ldots, A_k$ est une partition de $\mathbb{N}$ telle que :*

- *tout singleton $\{x\}$ de la partition correspond à un élément x de l'intersection,*
- *tout autre intervalle I_j de la partition est d'intersection vide avec au moins un tableau :*

$$\forall j \ \exists i, \ A_i \cap \ I_j = \emptyset$$

Définition 3.2 (Alternance) *L'*alternance $\delta(I)$ *d'une instance I est la taille minimale d'une partition certificat pour cette instance.*

Par exemple, la partition $\{\,]-\infty, 10[,\ [10, 16[,\ [16, +\infty[\,\}$ est une partition certificat de l'instance suivante, d'alternance $\delta(A, B) = 3$:

$A =$	10	11	12	13	14	15		
$B =$	1	2	3	4	16	17	18	19

L'algorithme `Alternance` (algorithme 9) mesure l'alternance d'une instance. Cet algorithme est similaire à un algorithme de Demaine, Lopez-Ortiz et Munro [DLOM00] pour calculer le certificat de longueur minimale. À tout moment l'algorithme teste l'appartenance de la valeur courante de m à l'intersection, et tous les éléments de l'intersection inférieurs à cette valeur sont déjà dans I. Les valeurs prises par m sont strictement croissantes, à chaque nouvelle valeur de m l'algorithme effectue les recherche doublantes et dichotomiques de m dans tous les tableaux. L'algorithme termine quand m prend une valeur supérieure au plus grand élément M d'un tableau.

Contrairement à l'algorithme `Séquentiel` (algorithme 8), l'algorithme `Alternance` ne nécessite pas un ordre sur les tableaux : m est cherché dans tous les tableaux quels que soient les résultats des recherches. Dans l'algorithme `Séquentiel`, au contraire, m n'est cherché que dans un tableau à la fois, de manière à faire progresser le pivot selon le résultat de la recherche.

Soit une partition certificat de taille minimale, un pivot m non dans l'intersection, et I l'intervalle contenant ce pivot. Par définition de la partition certificat, il y a au moins un tableau A_i d'intersection vide avec I. Parcourir la liste des tableaux dans un ordre fixé garanti de trouver un tel tableau A_i en au plus k recherches.

L'alternance est liée à la notion de certificat vue au chapitre 2 par le lemme suivant :

Algorithme 9 `Alternance`$(A_1, \ldots, A_k)$

Étant donné k tableaux triés non vides $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$, l'algorithme retourne l'intersection I et l'alternance δ de l'instance.

$I \leftarrow \emptyset$; $\delta \leftarrow 1$
pour tout i **faire** $p_i \leftarrow 1$ **fin pour**
$M \leftarrow \min\{A_1[n_1], \ldots, A_k[n_k]\}$
$m \leftarrow \max\{A_1[1], \ldots, A_k[1]\}$
tant que $m \leq M$ **faire**
  Soit i tel que $m = A_i[p_i]$
  pour tout $j \neq i$ **faire**
    $(g, d) \leftarrow$ `Recherche doublante`(m, A_j, p_j)
    $p_j \leftarrow$ `Recherche dichotomique`(m, A_j, g, d)
  fin pour
  $m' \leftarrow \max_j\{A_j[p_j]\}$
  si $m = m'$ **alors**
    $I \leftarrow I \cup \{m\}$
    pour tout i **faire** $p_i \leftarrow p_i + 1$ **fin pour**
    $m \leftarrow \max_j\{A_j[p_j]\}$
  sinon
    $m \leftarrow m'$
  fin si
  $\delta \leftarrow \delta + 1$
fin tant que
retourner (I, δ)

Lemme 3.1 (Correspondance certificat/alternance [BK02, Théorème 3.1]) *Une instance d'intersection I admet une partition certificat de taille δ si et seulement si elle admet un certificat de $\delta - 1 + (k-1)\#I$ comparaisons.*

Preuve.
Soit $P = (I_j)_{1 \leq j \leq \delta}$ une partition certificat de taille δ, ordonnée de façon que pour tout $j = 1, \ldots, \delta - 1$ les éléments de I_j soient inférieurs aux éléments de I_{j+1}. Soit C_+ l'argument comprenant, pour chaque singleton $\{x\}$ de la partition, $k-1$ égalités de la forme "$A_2[q_2] = A_1[q_1], \ldots, A_k[q_k] = A_1[q_1]$" où $A_1[q_1] = x$.

Pour chaque intervalle non singleton I_j de P, soit i_j un indice de tableau tel que A_{i_j} soit d'intersection vide avec I_j : $A_{i_j} \cap I_j = \emptyset$. Soient p_j et q_j tels que $A_{i_j}[p_j] = \min(A_{i_j} \cap I_{j+1})$ et $A_{i_{j+1}}[q_j] = \max(A_{i_{j+1}} \cap I_j)$. Soit $C_- = \{$"$A_{i_{j+1}}[q_j] < A_{i_j}[p_j]$" pour tout $j < \delta$ tel que I_j non singleton$\}$. C_- est un argument de longueur $\delta - 1$, et $C = C_+ \cup C_-$ est un certificat de longueur $\delta - 1 + (k-1)\#I$ de l'instance.

Inversement, soit $C = \{$ "$A_{i_{j+1}}[q_j] : A_{i_j}[p_j]$" $\forall\, j = 1, \ldots, \delta - 1\}$ un certificat de longueur $\delta - 1 + (k-1)\#I$ de l'instance. Une partition certificat est construite par récurrence.

Si l'un des tableaux est vide, la partition $P = \{]-\infty, \infty[\}$ est une partition certificat de l'instance. Si $C = \emptyset$ alors un des tableaux est vide, et on a bien une partition de $\delta = 1$ intervalles. Si $C \neq \emptyset$, alors $\delta = \#C + 1 - (k-1)\#I > 1$; soit $a = \max\{A_1[1], \ldots, A_k[1]\}$ et i tel que $a = A_i[1]$.

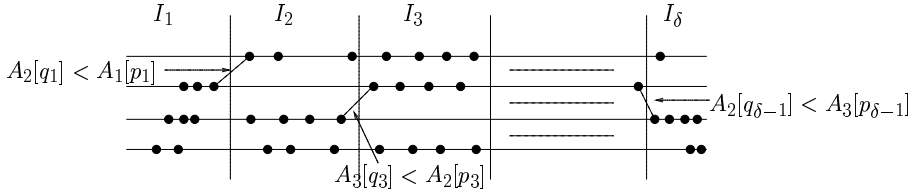


FIG. 3.1 – Une instance, un certificat et une partition certificat. Chaque axe représente un tableau, où chaque élément x est noté par un point d'abscisse x.

Si $a \in I$, soit $I_\delta =]-\infty, a[$, d'intersection vide avec le tableau $A_{i_\delta} = A_i$. Soit $\tilde{A}_1, \ldots, \tilde{A}_k$ la nouvelle instance définie par $\tilde{A}_i = \{x \in A_i | x > a\} \forall i \in \{1, \ldots, k\}$. Soit $\tilde{C}$ obtenu à partir de C en retirant toute comparaison impliquant un élément inférieur à a, et en transcrivant C sur l'instance $\tilde{A}_1, \ldots, \tilde{A}_k$ en mettant à jour les rangs des éléments comparés. L'argument $\tilde{C}$ est un certificat pour $(\tilde{A}_1, \ldots, \tilde{A}_k)$, de taille au plus $\#C - 1 = \delta - 2$: par hypothèse de récurrence

$(\tilde{A}_1, \ldots, \tilde{A}_k)$ admet une partition certificat $\tilde{P} = (\tilde{I}_j)_{1 \leq j \leq \delta - 1}$. Soit $I_j = \tilde{I}_j \backslash I_\delta$ pour $j = 1, \ldots, \delta - 1$. Alors $P = (I_j)_{1 \leq j \leq \delta}$ est une partition certificat de taille δ.

Si $a \notin I$, un raisonnement similaire permet d'obtenir une partition certificat de taille δ.

□

La taille minimale d'un certificat est liée à la complexité non déterministe de l'instance (le nombre minimal de comparaisons qu'un algorithme non déterministe doit effectuer pour résoudre l'instance). Une instance d'alternance δ n'admet pas de partition certificat de moins de δ intervalles : toute instance d'alternance δ est de complexité non déterministe $\delta - 1$. La complexité non déterministe forme une première borne inférieure de la complexité du calcul de l'intersection, mais une meilleure borne est obtenue en analysant la difficulté de trouver une partition certificat.

3.2 Borne inférieure

Les constructions utilisées sont similaires à celles utilisées dans le chapitre 2 [DLOM00, Lemme 4.4] pour donner une borne inférieure de la complexité dans le pire des cas pour les algorithmes déterministes. La borne inférieure du théorème 3.2 pour les algorithmes probabilistes est obtenue par l'analyse de la performance moyenne des algorithmes déterministes et par le principe de Yao.

Théorème 3.1 (Pire distribution élémentaire [BK02, Théorème 2.2]) *Pour tout $k > 1$, $n_1, \ldots, n_k$, x et g, d tels que $g < x - \sum n_i$ et $d > x + 1 + \sum n_i$, il existe une distribution aléatoire sur les instances de signature $(k, n_1, \ldots, n_k)$, d'alternance au plus 4, dont tous les éléments sont compris entre g et d, telle que tout algorithme déterministe effectue $\frac{1}{4}\sum_i \log(n_i) - \frac{3}{4}k = \Omega(\sum_i \log(n_i))$ comparaisons en moyenne pour décider si x appartient à l'intersection.*

Preuve.
Soit x un élément fixé. $\mathcal{D}$ est une distribution sur des instances composées chacune de k tableaux triés $A_1, \ldots, A_k$ de tailles respectives $n_1, \ldots, n_k$, dont $k-1$ contiennent x et un seul ne contient pas x. Pour tout $i = 1, \ldots, k-1$, ces instances sont telles que les éléments de A_i inférieurs à x sont tous plus petits que les éléments de A_{i+1}. De manière symétrique, les éléments de A_{i+1} supérieurs à x sont tous plus grands que les éléments de A_i.

$\mathcal{D}$ est construite comme suit :

- pour chaque tableau A_i une position p_i est choisie uniformément dans $\{1, \ldots, n_i\}$;
- un tableau A_w est choisi au hasard avec probabilité $\alpha_w = \log n_w / \sum_l \log n_l$;
- soit $y = x + 1$; $A_w[p_w] = y$ et $A_i[p_i] = x$ pour tout $i \neq w$;
- les autres éléments des tableaux sont dans $]g, x[$ ou $]y, d[$ et sont placés de telle façon que pour tout $i < i'$, si $a \in A_i$ et $a' \in A_{i'}$ alors $a < a'$ si a et a' sont tous deux inférieurs à x, ou si a et a' sont tous deux supérieurs à y.

L'intersection de ces instances est vide. Tout algorithme doit prouver que x n'est pas dans l'intersection. Pour cela il doit nécessairement trouver le tableau A_w ne contenant pas x et trouver le rang d'insertion p_w de x dans ce tableau.

On définit un type de requête "$A_i[p] : x, y$", correspondant à la comparaison d'une position p d'un tableau A_i avec x et y, qui admet quatre résultats possibles $A_i[p] < x, A_i[p] = x, A_i[p] = y, A_i[p] > y$.

Pour tout algorithme déterministe B calculant l'intersection d'une instance de $\mathcal{D}$, il existe un algorithme A décidant de l'appartenance de x à l'intersection, qui est limité aux requêtes $A_i[p] : x, y$, et qui effectue au plus deux fois plus de requêtes que B de comparaisons.

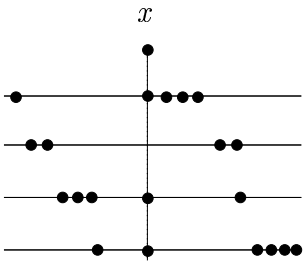


FIG. 3.2 – Une instance selon $\mathcal{D}$.

Pour chaque comparaison $A_i[p] : A_{i'}[p']$ effectuée par B considérons l'algorithme A qui effectue les requêtes "$A_i[p] : x, y$" et "$A_{i'}[p'] : x, y$". Le résultat de la comparaison de B est déterminé par le résultat des requêtes de A :

- si $i = i'$, le résultat de la comparaison découle de l'ordre de p et p', la comparaison est superflue ;
- si $A_i[p]$ est égal à x ou y, le résultat découle de la position de $A_{i'}[p']$ par rapport à x, et si $A_{i'}[p']$ est égal à x ou y, le résultat découle de la position de $A_i[p]$ par rapport à x
- si x et y séparent $A_i[p]$ et $A_{i'}[p']$, le résultat est déduit des positions de $A_i[p]$ et $A_{i'}[p']$ par rapport à x par transitivité ;
- si $A_i[p]$ et $A_{i'}[p']$ sont tous les deux inférieurs à x ou tous les deux supérieurs à y, le résultat découle de l'ordre de i et i'.

Une borne inférieure C de la complexité des algorithmes à requêtes comme A implique donc une borne inférieure $C/2$ pour la complexité d'algorithmes à comparaisons comme B. Dans la suite de la preuve, seuls sont considérés des algorithmes basés, comme A, sur des requêtes $A_i[p] : x, y$.

Soit A un tel algorithme décidant l'appartenance de x à l'intersection par des requêtes $A_i[p] : x, y$. A termine dès qu'il a trouvé y dans le tableau A_w à la position p_w. Soit A' un algorithme poursuivant les requêtes après A de manière à trouver toutes les positions p_i restantes.

Pour chaque instance aléatoire tirée de $\mathcal{D}$ et les algorithmes A et A', soient

- X_i le nombre de requêtes effectuées par A dans le tableau A_i ;
- Y_i le nombre de requêtes effectuées par A' dans le tableau A_i ;
- ξ_i une variable indiquant si la position p_i de x dans A_i a été découverte par A.

L'algorithme A effectue $C = \sum_i X_i$ requêtes. Il effectue $\sum_i X_i \xi_i = \sum_i Y_i \xi_i \leq C$ requêtes dans les tableaux où il trouve le rang d'insertion de x. Une manipulation

élémentaire permet de séparer l'étude de Y_i de celle de ξ_i :

$$\begin{aligned}\Pr\{Y_i\xi_i \geq a_i\} &= \Pr\{Y_i \geq a_i \text{ et } \xi_i = 1\} \\ &= 1 - \Pr\{Y_i < a_i \text{ ou } \xi_i = 0\} \\ &\geq 1 - \Pr\{Y_i < a_i\} - \Pr\{\xi_i = 0\} \\ &= \Pr\{\xi_i = 1\} - \Pr\{Y_i < a_i\} \qquad (3.1)\end{aligned}$$

L'analyse de Y_i se fait par un argument de type "théorie de l'information", comme suit : A' correspond à un arbre de décision dont les noeuds sont de degré 4. Au plus 4^{a_i} feuilles sont de profondeur inférieure à a_i. À chaque feuille correspond une position p_i, tirée uniformément avec probabilité $1/n_i$. La probabilité totale de ces feuilles est donc inférieure à $4^{a_i}/n_i$. Pour $a_i = (\log_4 n_i) - 1$, cela implique que $\Pr\{Y_i < (\log_4 n_i) - 1\} \leq 1/4$.

Nous nous tournons maintenant vers l'étude de ξ_i. Pour $(p_1, \ldots, p_k)$ fixés, soit σ l'ordre dans lequel A puis A' découvrent les rangs p_i (Cf figure 3.3). Les k choix possibles pour w correspondent aux k seules instances possibles une fois fixés $(p_1, \ldots, p_k)$.

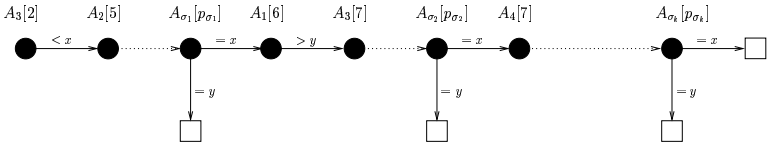


FIG. 3.3 – L'arbre de décision de A restreint aux instances telles que $(p_1, \ldots, p_k)$ soit fixé.

L'algorithme A termine quand il a trouvé y, dans le tableau A_w à la position p_w. À ce moment l'algorithme a trouvé toutes les positions p_i telles que i précède w dans l'ordre σ :

$$\sigma_i \leq \sigma_w \Leftrightarrow \xi_i = 1.$$

w est choisi égal à j avec probabilité $\Pr\{w = j\} = \alpha_j$:

$$\Pr\{\xi_i = 1 | (p_1, \ldots, p_k)\} = \sum_{j:\sigma_j \geq \sigma_i} \alpha_j.$$

Par définition $\alpha_i = \log n_i / \sum_l \log n_l$:

$$\Pr\{\xi_i = 1\} = \sum_{p_1,\ldots,p_k} \Pr\{\xi_i = 1 | p_1, \ldots, p_k\} \Pr\{p_1, \ldots, p_k\}$$

$$= \sum_{p_1,\ldots,p_k} \Pr\{p_1,\ldots,p_k\} \sum_{w:w\geq_\sigma i} \frac{\log_4 n_w}{\sum_l \log_4 n_l}.$$

Nous pouvons maintenant calculer $E(C)$: la linéarité de la moyenne, l'inégalité de Markov et l'équation (3.1) permettent d'en déduire une minoration de la complexité moyenne de la complexité de A :

$$\begin{aligned}
E(C) &\geq \sum_i E(Y_i\xi_i) \\
&\geq \sum_i (\log_4 n_i - 1)\Pr\{Y_i\xi_i \geq (\log_4 n_i - 1)\} \\
&\geq \sum_i (\log_4 n_i - 1) \times (\Pr\{\xi_i = 1\} - \Pr\{Y_i < (\log n_i - 1)\}) \\
&\geq \sum_i (\log_4 n_i - 1) \times (\Pr\{\xi_i = 1\} - \frac{1}{4}) \\
&= \sum_i (\log_4\ n_i \Pr\{\xi_i = 1\} - \Pr\{\xi_i = 1\} - \frac{1}{4}\log_4\ n_i + \frac{1}{4}) \\
&\geq \sum_i \log_4 n_i \Pr\{\xi_i = 1\} - \frac{1}{4}\sum_i \log_4 n_i - \frac{3}{4}k. \qquad (3.2)
\end{aligned}$$

Le premier terme de l'équation (3.2) est

$$\sum_i \log_4 n_i \Pr\{\xi_i = 1\} = \sum_{p_1,\ldots,p_k} \Pr\{p_1,\ldots,p_k\} \sum_{i,w:\sigma_w\geq\sigma_i} \frac{\log_4 n_w \log_4 n_i}{\sum_l \log_4 n_l}.$$

L'ordre σ dépend de $(p_1,\ldots,p_k)$, mais la somme peut être majorée de manière indépendante de p :

$$\sum_{i,w:\sigma_w\geq\sigma_i} \log_4 n_w \log_4 n_i \geq \frac{1}{2}((\sum_i \log_4 n_i)^2 + \sum_i \log_4^2 n_i).$$

Ceci permet de conclure :

$$\begin{aligned}
\sum_i \log_4 n_i \Pr\{\xi_i = 1\} &\geq \frac{1}{2}\frac{1}{\sum_l \log n_l}\left((\sum_i \log n_i)^2 + \sum_i \log^2 n_i\right) \\
&\geq \frac{1}{2}\frac{(\sum_i \log_4 n_i)^2}{\sum_l \log_4 n_l} \\
&= \frac{1}{2}\sum_i \log_4 n_i.
\end{aligned}$$

La borne inférieure du théorème est obtenue en plongeant ce résultat dans l'équation (3.2) : $E(C) \geq \frac{1}{4}\sum_i \log_4 n_i - \frac{3}{4}k$. □

Par symétrie, le même résultat est valable pour l'union :

Corollaire 3.1 (Borne inférieure élémentaire union) *Pour tout $k > 1$, $n_1, \ldots, n_k$, x et g, d tels que $(g < x - \sum n_i)$ et $(d > x + 1 + \sum n_i)$, il existe une distribution aléatoire sur les instances de signature $(k, n_1, \ldots, n_k)$, d'alternance au plus 4, dont tous les éléments sont compris entre g et d, telle que tout algorithme déterministe effectue $\frac{1}{4}\sum_i \log(n_i) - \frac{3}{4}k = \Omega(\sum_i \log(n_i))$ comparaisons en moyenne pour décider si x appartient à l'union.*

Preuve.
Il suffit de prendre la même distribution : y appartient à l'union si et seulement si x *n'appartient pas* à l'intersection. Un algorithme devant décider de l'appartenance de y à une instance de la distribution du théorème 3.1 effectue $\frac{1}{4}\sum_i \log(n_i) - \frac{3}{4}k = \Omega(\sum_i \log(n_i))$ comparaisons en moyenne pour décider de l'appartenance de x à l'intersection, et donc de l'appartenance de y à l'union. □

Le principe de Yao permet de déduire du théorème 3.1 et du corollaire 3.1 une borne inférieure sur la complexité des algorithme probabilistes :

Corollaire 3.2 (Borne inférieure élémentaire) *Pour tout $k > 1$, $n_1, \ldots, n_k$, et x, tout algorithme probabiliste effectue $\Omega(\sum_i \log(n_i))$ comparaisons dans le pire des cas pour décider de l'appartenance de x à l'intersection ou l'union d'une instance de signature $(k, n_1, \ldots, n_k)$.*

Théorème 3.2 (Borne inférieure [BK02, corollaire 3.1]) *Pour tout $k > 1$, $n_1, \ldots, n_k$ et δ, tout algorithme probabiliste calculant l'intersection effectue $\Omega(\delta \sum_i \log(n_i/\delta))$ comparaisons dans le pire des cas pour les instances de signature $(k, n_1, \ldots, n_k)$ et d'alternance δ.*

Preuve.
Soit $p = \lfloor \delta/4 \rfloor$. Soient p éléments $x_1, \ldots, x_p$ tels que pour tout $j = 1, \ldots, k$, $x_{j+1} - x_j > \sum_i n_i/2p$, et $p+1$ éléments $g_0, \ldots, g_p$ tels que $g_0 = -\infty$ et pour tout $j = 1, \ldots, p$ on ait $g_j = (x_j + x_{j+1})/2$.

La distribution est basée sur p sous-instances générées par le corollaire 3.2, chacune de signature $(k, n_1/p, \ldots, n_k/p)$, faisant chercher x_j entre g_{j-1} et g_j pour chaque sous-instance j.

Pour chaque sous-instance, une partition de 4 intervalles au plus suffit à montrer que x_j n'appartient pas à T_i : chaque sous-instance est d'alternance au plus 4. Les instances obtenues en prenant comme $i$ème tableau l'union des $i$èmes tableaux de ces p sous-instances sont d'intersection vide, de signature $(k, n_1, \ldots, n_k)$ et d'alternance au plus $4p = O(p)$.

Tout algorithme doit prouver que les éléments $\{x_1, \ldots, x_p\}$ ne sont pas dans l'intersection. Résoudre cette instance correspond à résoudre les p sous-instances

de manière indépendante. Le Corollaire 3.2 implique que tout algorithme probabiliste effectue $\Omega(\sum_i \log(n_i/p))$ comparaisons dans le pire des cas sur chaque sous-instance. Ces sous-instances sont indépendantes. Tout algorithme probabiliste effectue donc en tout $\Omega(p\sum_i \log(n_i/p)) = \Omega(\delta \sum_i \log(n_i/\delta))$ comparaisons dans le pire des cas. □

Corollaire 3.3 (Borne inférieure) *Pour tout $k > 1$, n et δ, tout algorithme probabiliste calculant l'intersection effectue $\Omega(\delta\ k \log(n/\delta\ k))$ comparaisons dans le pire des cas pour les instances de taille n sur k tableaux et d'alternance δ.*

Preuve.
En l'absence de contraintes sur les tailles individuelles des tableaux, ils suffit d'appliquer le théorème 3.2 aux instances telles que tous les tableaux soient de taille $n_1 = \ldots = n_k = \lfloor n/k \rfloor$. □

3.3 Complexité des algorithmes

L'algorithme `Parallèle` est optimal dans le pire des cas pour les instances de taille et d'alternance fixées.

Théorème 3.3 (Complexité de l'algorithme `Parallèle`[BK02, Th. 3.3])
Pour tout $k > 1$, n et δ, l'algorithme `Parallèle` *effectue $O(\delta\ k \log(n/\delta\ k))$ comparaisons sur une instance de taille n, sur k tableaux, d'alternance δ.*

Preuve.
Soit une instance de taille n, sur k tableaux, d'alternance δ. Appelons "comparaisons doublantes" les comparaisons effectuées lors d'une recherche doublante, et "comparaisons dichotomiques" celles effectuées lors d'une recherche dichotomique.

Soit $P = (I_j)_{j \leq \delta}$ une partition certificat de l'instance. Lors d'une exécution de l'algorithme `Séquentiel`, les valeurs successives de m forment une suite croissante. Découpons cette exécution en δ phases, une comparaison appartenant à la phase j si $m \in I_j$.

Lors de chaque phase, l'algorithme effectue au plus $k - 1$ recherches doublantes :

- si $I_j = \{x\}$ l'algorithme cherche x dans les $k - 1$ tableaux restants ;
- sinon I_j est un intervalle et l'algorithme cherche un tableau ne contenant pas m. Il y en a au moins un, par définition de la partition certificat, et l'algorithme parcourant les tableaux dans un ordre fixe le trouvera après au plus $k - 1$ recherches doublantes.

Pour chaque tableau A_i et chaque phase j, soit g_j^i le nombre de comparaisons doublantes effectuées dans le tableau A_i pendant la phase j. Le nombre total de comparaisons doublantes est $\sum_{i,j} g_j^i = \sum_{i,j} \log 2^{g_j^i}$. Par concavité du logarithme, ceci est inférieur à $\delta k \log(\sum_{i,j} 2^{g_j^i}/k\delta)$ où $2^{g_j^i}$ est le nombre d'éléments "sautés" par la recherche doublante dans A_i pendant la phase j. La somme de ces nombres n'excède pas le nombre d'éléments dans l'instance : $\sum_{i,j} 2^{g_j^i} \leq n$. Par conséquent, le nombre total de comparaisons doublantes effectuées est inférieur à $k\delta \log(n/k\delta)$.

Toute recherche dichotomique est effectuée sur un intervalle de taille 2^j obtenu par une recherche doublante de j pas. Le nombre de comparaisons effectué lors de cette recherche doublante est j, celui effectué lors de la recherche dichotomique est également j. Le nombre total de comparaisons dichotomiques est égal au nombre de comparaisons doublantes. Le nombre total de comparaisons effectuées dans l'instance est donc inférieur à $2\delta\ k \log(n/k\delta) = \Omega(\delta\ k \log(n/k\delta))$. □

Si les logarithmes des tailles des tableaux sont de même ordre, l'algorithme `Parallèle` est également optimal dans le pire des cas pour les instances de taille, de nombre de tableaux et d'alternance fixés. Si un tableau est de taille très supérieure aux autres, alors l'algorithme `Parallèle` n'est pas optimal.

Remarque 3.1 *Pour tout $k \geq 3$, et tout $x > 1$ il existe une instance de signature $(k, 2^{x+1}, x, \ldots, x)$ telle que l'algorithme* `Parallèle` *effectue kx comparaisons.*

Preuve.
Soit x fixé. L'instance suivante est composée de k tableaux triés.

$A =$	1	...	2^x	$2^x + 2x + 1$	...	$2^{x+1} + 2x + 1$
$B_1 =$	$2^x + 1$	$2^x + 3$	...	$2^x + 2x - 1$		
...						
$B_{\lfloor \frac{k-1}{2} \rfloor} =$	$2^x + 1$	$2^x + 3$	...	$2^x + 2x - 1$		
$C_1 =$	$2^x + 2$	$2^x + 4$	...	$2^x + 2x$		
...						
$C_{\lceil \frac{k-1}{2} \rceil} =$	$2^x + 2$	$2^x + 4$	...	$2^x + 2x$		

Tous les entiers de 1 à 2^x et de $2^x + 2x + 1$ à $2^{x+1} + 2x + 1$ sont éléments du tableau A, la taille de ce tableau est 2^{x+1}. Les tableaux $B_1, \ldots, B_{\lfloor \frac{k-1}{2} \rfloor}$ sont identiques et ont pour éléments tous les entiers impairs strictement compris entre 2^x et $2^x + 2x + 1$. Les tableaux $C_1, \ldots, C_{\lceil \frac{k-1}{2} \rceil}$ sont identiques et ont pour éléments tous les entiers pairs strictement compris entre 2^x et $2^x + 2x + 1$. Les tableaux $B_1, \ldots, B_{\lfloor \frac{k-1}{2} \rfloor}$ et $C_1, \ldots, C_{\lceil \frac{k-1}{2} \rceil}$ sont tous de taille x.

L'intersection de cette instance est vide. Une partition certificat de cette instance est $]-\infty, 2^x+1[, [2^x+1, 2^x+2x+1[, [2^x+2x+1, \infty[$: l'alternance est inférieure à 3.

Supposons que les tableaux soient considérés dans l'ordre $A, B_1, B_{\lfloor \frac{k-1}{2} \rfloor}, C_1, \ldots, C_{\lceil \frac{k-1}{2} \rceil}$. Les valeurs prises par m lors de l'exécution de l'algorithme `Parallèle` sont toutes dans $[2^x+1, 2^x+2x+1]$. L'algorithme effectue une recherche doublante dans A en même temps qu'une succession de recherches doublantes et dichotomiques dans les autres tableaux. Il effectue x comparaisons dans A avant de trouver la position d'insertion d'une valeur de m, et x comparaisons dans chacun des autres tableaux pour prouver que leur intersection est vide. L'algorithme `Parallèle` effectue donc en tout kx comparaisons pour calculer l'intersection de cette instance. □

Le théorème 3.2 donne une borne inférieure $\Omega(x + (k-1)\log x) = \Omega(x)$ du nombre de comparaisons nécessaires pour calculer l'intersection dans le pire des cas sur des instances de même alternance et signature que celle-ci. L'algorithme `Parallèle` n'est donc pas optimal dans le pire des cas pour les instances de signature et d'alternance fixées.

L'algorithme `Séquentiel` est lui optimal dans le pire des cas pour les instances de signature et d'alternance fixées, *et* dans le pire des cas pour les instances de taille et d'alternance fixées.

Théorème 3.4 (Complexité de l'algorithme `Séquentiel`) *Pour tout $k > 1$, $n_1, \ldots, n_k$ et δ, l'algorithme `Séquentiel` effectue $O(\delta \sum_i \log(n_i/\delta))$ comparaisons sur une instance de signature $(k, n_1, \ldots, n_k)$ et d'alternance δ.*

Preuve.
Soit $A_1, \ldots, A_k$ une instance de signature $(k, n_1, \ldots, n_k)$ et d'alternance δ. Appelons "comparaisons doublantes" les comparaisons effectuées lors d'une recherche doublante, et "comparaisons dichotomiques" celles effectuées lors d'une recherche dichotomique.

Soit $P = (I_j)_{j \leq \delta}$ une partition certificat de l'instance. Lors d'une exécution de l'algorithme `Séquentiel`, les valeurs successives de m forment une suite croissante. Découpons cette exécution en δ phases, une comparaison appartenant à la phase j si $m \in I_j$.

Pour chaque tableau A_i et chaque phase j, soit g_j^i le nombre de comparaisons doublantes effectuées dans le tableau A_i pendant la phase j. Le nombre total de comparaisons doublantes dans le tableau A_i est $\sum_j g_j^i = \sum_j \log 2^{g_j^i}$. Par concavité du logarithme, ceci est inférieur à $\delta \log(\sum_j 2^{g_j^i}/\delta)$, où $2^{g_j^i}$ est le nombre

d'éléments "sautés" par la recherche doublante dans A_i pendant la phase j. La somme de ces nombres n'excède pas le nombre d'éléments dans A_i : $\sum_j 2^{g_j^i} \leq n_i$. Par conséquent, le nombre total de comparaisons doublantes effectuées dans le tableau A_i est inférieur à $\delta \log(n_i/\delta)$. Le nombre total de comparaisons doublantes effectuées dans l'instance est inférieur à la somme de ces nombres $\sum_i \delta \log(n_i/\delta)$.

Toute recherche dichotomique est effectuée sur un intervalle de taille 2^j obtenu par une recherche doublante de j pas. Le nombre de comparaisons effectué lors de cette recherche doublante est j, celui effectué lors de la recherche dichotomique est également j. Le nombre total de comparaisons dichotomiques est égal au nombre de comparaisons doublantes. Le nombre total de comparaisons effectuées dans l'instance est donc inférieur à $2\sum_i \delta \log(n_i/\delta) = \Omega(\sum_i \delta \log(n_i/\delta))$. □

Corollaire 3.4 (Complexité de l'algorithme `Séquentiel`) *Pour tout $k > 1$, n et δ, l'algorithme* `Séquentiel` *effectue $O(\delta\, k \log(n/\delta\, k))$ comparaisons sur une instance de taille n sur k tableaux et d'alternance δ.*

Preuve.
En l'absence de contraintes sur les tailles individuelles des tableaux, ils suffit d'appliquer le théorème 3.4 aux instances telles que tous les tableaux soient de taille $n_1 = \ldots = n_k = n/k$. □

La non-optimalité de l'algorithme `Deux-à-deux` est montrée par un exemple :

Remarque 3.2 *L'algorithme* `Deux-à-deux` *n'est pas optimal pour les instances de taille et d'alternance fixées.*

Preuve.
Dans l'instance suivante composée de trois tableaux A, B, C, tous les éléments de A ou de B peuvent être insérés à une unique position dans C. L'intersection est vide, et le certificat $P =$ "$C[4] < A[1], A[5] < C[5]$" est de longueur minimale : l'alternance de l'instance est $\delta = 3$.

L'algorithme `Deux-à-deux` effectue $5 = n/k$ comparaisons pour calculer l'intersection de A et B, avant de chercher le point d'insertion de $A[1]$ dans C.

$A =$	5	6	7	8	9	
$B =$	5	6	7	8	9	
$C =$	1	2	3	10	12	13

□

L'algorithme `Parallèle` est optimal pour les instances de dispersion fixée, mais pas pour les instances d'alternance et de signature fixée, et l'algorithme `Séquentiel` est optimal pour les instances d'alternance et de signature fixées, mais

pas pour les instances de dispersion fixée. Ceci prouve l'indépendance de l'analyse dans le pire des cas pour les instances d'alternance et de signature fixée par rapport à l'analyse dans le pire des cas pour les instances de dispersion fixée (chapitre 2). D'autre part, tout algorithme optimal dans le pire des cas pour les instances de signature fixées est optimal dans le pire des cas pour les instances de taille fixée. Ces relations entre les analyses sont résumées sur la figure 3.4.

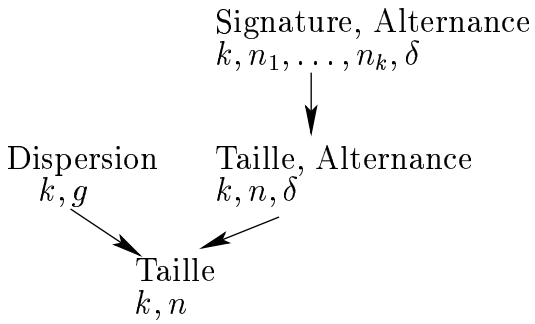


FIG. 3.4 – Ordre partiel sur les analyses. Une flèche d'une analyse A vers une analyse B signifie que tout algorithme optimal pour l'analyse A est optimal pour l'analyse B.

Les résultats sur l'intersection sont généralisés à une relaxation du problème d'intersection dans le chapitre suivant.

Chapitre 4

Ensembles de Multiplicité

La réponse à une requête conjonctive dans un moteur de recherche indexé est l'intersection des ensembles de référence associés à chaque mot de la requête. Quand l'intersection est vide, cela se traduit par une réponse nulle à la requête. La seule information fournie dans ce cas est une information négative indiquant que la requête est trop spécifique. Nous proposons ici une solution qui d'une part n'est pas plus coûteuse que la réponse usuelle dans le cas où l'intersection n'est pas vide, et qui d'autre part a l'avantage de retourner une information non nulle dans tous les cas.

L'*ensemble de multiplicité* t de k tableaux triés (en anglais "t-Threshold set" [BK02]) est l'ensemble des éléments présents dans au moins t tableaux. La *t-alternance* est une mesure sur les instances. L'analyse dans le pire des cas pour les algorithmes probabilistes et les instances de signature $(k, n_1, \ldots, n_k)$ et de t-alternance δ_t montre une borne inférieure de la complexité *pour* $t \geq \lfloor k/2 \rfloor$. Nous donnons un algorithme déterministe `Multiplicité` asymptotiquement optimal pour calculer l'ensemble de multiplicité t.

La *multiplicité* d'une instance composée de k tableaux triés est le nombre maximal de tableaux ayant un élément en commun. L'*ensemble de multiplicité optimale* d'une instance de multiplicité t est l'ensemble des éléments dont la multiplicité est t. L'analyse pour les algorithmes calculant l'ensemble de multiplicité optimale se déduit de l'analyse pour les algorithmes calculant l'ensemble de multiplicité t. Nous donnons un algorithme déterministe `Multiplicité optimale` pour calculer l'ensemble de multiplicité optimale.

4.1 Multiplicité et t-alternance

Définition 4.1 (Ensemble de multiplicité t [BK02, déf. 4.1: t-threshold]) *Soit $\mathcal{U}$ un espace totalement ordonné. Une instance du problème de l'ensemble de multiplicité t est composée de k tableaux triés $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$ et composés d'éléments de $\mathcal{U}$. L'*ensemble de multiplicité *t d'une instance $(A_1, \ldots, A_k)$ est l'ensemble $T_t(A_1, \ldots, A_k)$ des éléments de cette instance présents dans au moins t tableaux distincts.*

Pour $t = 1$, c'est l'union des ensembles, et pour $t = k$ c'est l'intersection : c'est une généralisation de ces deux problèmes. L'ensemble T_2 de multiplicité 2 spécifie l'union : il délimite les blocs composant l'union et précise les redondances, comme dans l'analyse de l'union vue dans la section 1.2.3 [DLOM00].

Définition 4.2 (Partition t-certificat) *Une partition t-*certificat *est une partition composée d'intervalles de l'espace des éléments $\mathcal{U}$, telle que tout singleton correspond à un élément de l'ensemble de multiplicité t, et que tout intervalle soit d'intersection vide avec au moins $k - t + 1$ tableaux.*

Définition 4.3 ($t-$alternance) *La $t-$*alternance *d'une instance est la taille minimale d'une partition t-certificat.*

Par exemple, l'ensemble de multiplicité 5 de l'instance (A, B, C, D, E) suivante est vide.

$A =$	3	4	5	6	7
$B =$	5	6	7	10	11
$C =$	0	1	2	10	11
$D =$	0	1	2	8	9
$E =$	3	4	5	8	9

La partition $\{\,]-\infty, 3[,\ [3, 10[,\ [10, +\infty[\,\}$ est une partition 5-certificat de cette instance, de 5-alternance $\delta_5(A, B, C, D, E) = 3$.

Pour $t = 4$, l'instance a également un ensemble de multiplicité vide. Pour le vérifier, il suffit de représenter l'ensemble des éléments de chaque tableau sur un axe en prenant sa valeur pour abscisse :

$A =$				3	4	5	6	7				
$B =$						5	6	7			10	11
$C =$	0	1	2								10	11
$D =$	0	1	2						8	9		
$E =$				3	4	5			8	9		

La partition $\{\,]-\infty, 3[,\ [3, 8[,\ [8, 10[,\ [10, +\infty[\,\}$ est une partition 4-certificat de cette instance, de 4-alternance $\delta(A, B, C, D, E) = 4$. Aucun élément n'est présent dans 4 tableaux à la fois.

4.2 Calcul de l'ensemble de multiplicité fixée

Lemme 4.1 (Borne inférieure élémentaire) *Soit $k > 1$, t, et $n_1, n_2, \ldots, n_k$ donnés. On considère les instances de signature $(k, n_1, \ldots, n_k)$ et de t-alternance au plus $2\lceil k/(t-1)\rceil + 2$. Alors, tout algorithme probabiliste effectue $\Omega(\frac{t}{k}\sum_i \log\ n_i)$ comparaisons pour décider de l'appartenance de x à l'ensemble de multiplicité t.*

Preuve.
Sans perte de généralité, les tableaux de l'instance sont supposés ordonnés par taille de sorte que $n_1 \geq \ldots \geq n_k$.

Le Corollaire 3.2 implique une borne inférieure $\Omega(\sum_{i=1}^{t} \log(n_i))$ pour décider de l'appartenance de x à l'intersection des t plus grands tableaux. Tout élément appartenant à l'intersection de ces t tableaux appartient à l'ensemble de multiplicité t : cette borne vaut pour l'appartenance de x à T_t.

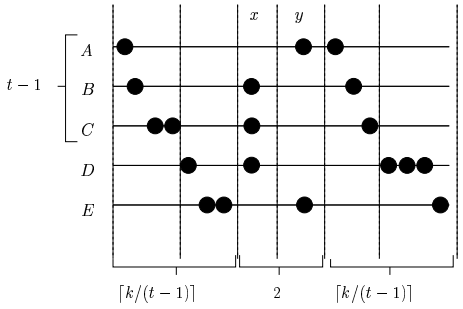


FIG. 4.1 – Chaque intervalle ne peut intersecter plus de $t-1$ tableaux. Chaque instance est de t-alternance au plus $2\lceil k/(k-t+1)\rceil + 2$.

L'ensemble de multiplicité t des éléments inférieurs à x est certifié par au plus $\lceil k/(t-1)\rceil$ intervalles ; et l'ensemble de multiplicité t des éléments supérieurs à y est certifié par au plus $\lceil\ k/(t-1)\rceil$ intervalles supplémentaires. L'appartenance de x et y à T_t est certifiée par au plus deux intervalles supplémentaires. En tout une partition de $2\lceil k/(t-1)\rceil + 2$ certifie les instances générées par le Corollaire 3.2 : ces instances sont de t-alternance au plus $2\lceil k/(t-1)\rceil + 2$. Les tableaux sont triés par taille décroissante. On obtient donc $\sum_{i=1}^{t} \log(n_i) \leq \frac{t}{k}\sum_{i=1}^{k} \log(n_i)$. □

Remarque 4.1 (Borne inférieure élémentaire) *Si $t \geq k/2$, le Lemme 4.1 implique une borne inférieure $\Omega(\sum_{i=1}^{k} \log\ n_i)$ pour les instances d'alternance au plus* 6.

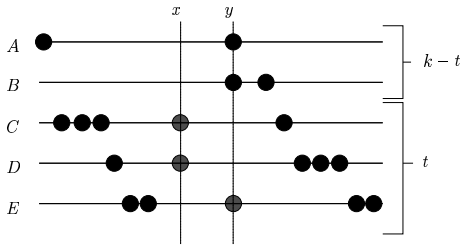


FIG. 4.2 – Si $t \geq k/2$, on considère les instances telles que les $k-t$ plus petits tableaux ne contiennent pas x.

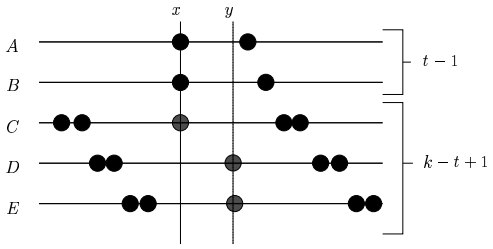


FIG. 4.3 – Si $t < k/2$, on considère les instances telles que les $t-1$ plus petits tableaux contiennent x.

Dans le cas où t prend une petite valeur indépendante de k, la même borne inférieure peut être trouvée, mais avec des instances de t-alternance dépendant de t. Le Corollaire 3.2 implique une borne inférieure dans le pire des cas de $\Omega(\sum_{i=1}^{k-t+1} \log(n_i))$ comparaisons pour décider de l'appartenance de x à l'*union* des $k-t+1$ plus grands tableaux. Tout élément présent dans les $t-1$ plus petits tableaux appartient à T_t si et seulement si il appartient à l'union des $k-t+1$ tableaux restants : cette borne vaut également pour la complexité de décision de l'appartenance de x à T_t. Comme les tableaux sont triés par taille décroissante on a $\sum_{i=1}^{k-t+1} \log(n_i) \leq \frac{k-t+1}{k} \sum_{i=1}^{k} \log(n_i) = \Omega(\sum_{i=1}^{k} \log(n_i))$. Certaines instances ainsi construites sont d'alternance $2k/(t-1)+2$: on ne peut construire une instance générale d'alternance finie indépendante de t et k avec ces instances élémentaires, quand t prend une petite valeur indépendante de k.

Théorème 4.1 (Borne inférieure) *Pour tout $k > 1$, $t \geq \lfloor k/2 \rfloor$, $n_1, n_2, \ldots, n_k$ et δ_t, tout algorithme probabiliste calculant l'ensemble de multiplicité t effectue $\Omega(\delta_t \sum_{i=1}^{k} \log(n_i/\delta_t))$ comparaisons dans le pire des cas pour les instances de signature $(k, n_1, \ldots, n_k)$ et de $t-$alternance au plus δ_t.*

Preuve.
De manière similaire à la preuve du théorème 3.2, soit $p = \lfloor \delta_t/6 \rfloor$. Soient p éléments $x_1, \ldots, x_p$ tels que pour tout $j = 1, \ldots, k$, on ait $x_{j+1} - x_j > \sum_i n_i/2p$. Soient $p+1$ éléments $g_0, \ldots, g_p$ tels que $g_0 = -\infty$ et $g_j = (x_j + x_{j+1})/2$ pour tout $j = 1, \ldots, p$.

La pire instance est basée sur p sous-instances générées par le lemme 4.1, chacune de signature $(k, n_1/p, \ldots, n_k/p)$, de t-alternance au plus 6, et faisant vérifier l'appartenance de x_j à T_t entre g_{j-1} et g_j pour chaque sous-instance j.

Considérons les instances obtenues en prenant comme $i$ème tableau l'union des $i$èmes tableaux de chacune de ces p sous-instances. Ces instances ont un ensemble

de multiplicité t vide, sont de signature $(k, n_1, \ldots, n_k)$ et de t-alternance au plus $6p = 6\lfloor \delta_t/6 \rfloor \leq \delta_t$.

Tout algorithme doit prouver que les éléments $\{x_1, \ldots, x_p\}$ ne sont pas dans T_t. Résoudre cette instance correspond à résoudre les p sous-instances de manière indépendante. La remarque 4.1 implique que tout algorithme probabiliste effectue $\Omega(\sum_i \log(n_i/p))$ comparaisons dans le pire des cas sur chaque sous-instance. Ces sous-instances étant indépendantes, tout algorithme probabiliste effectue en tout $\Omega(p \sum_i \log(n_i/p)) = \Omega(\delta \sum_i \log(n_i/\delta))$ comparaisons dans le pire des cas. □

De la même manière que pour l'analyse par alternance du problème d'intersection, cette borne inférieure de la complexité dans le pire des cas pour les instances de *signature* et t-alternance fixées implique une borne inférieure pour les instances de *taille* et de t-alternance fixées.

Pour tout $t \geq \lfloor k/2 \rfloor$ l'algorithme `Multiplicité` (algorithme 10) est optimal dans le pire des cas pour les instances de signature et t-alternance fixées, et donc également dans le pire des cas pour les instances de taille et de t-alternance fixées.

Théorème 4.2 (Complexité de l'algorithme `Multiplicité`) *Pour tout k>1, t, $n_1, n_2, \ldots, n_k$ et δ_t, l'algorithme `Multiplicité` effectue $O(\delta_t \sum_i \log(n_i/\delta_t) + k \log(k) + \delta_t \; k \log(k - t + 1))$ comparaisons sur une instance de signature $(k, n_1, \ldots, n_k)$ et de t-alternance δ_t.*

Preuve.
À tout instant l'algorithme teste l'appartenance de la valeur courante de m à T. La suite des valeurs successives de m est strictement croissante, et tous les éléments de l'ensemble de multiplicité t inférieurs à la valeur courante de m sont déjà dans T. Les marques $contient_i \in \{OUI, NON, POSSIBLE\}$ notent si m est présent dans A_i. Les nombres de tableaux marqués sont notés $\#OUI$, $\#NON$ et $\#POSSIBLE$ pour chaque marque. H est un tas permettant d'obtenir à tout moment le minimum de $A_i[p_i]$ sur les tableaux marqués NON. L'algorithme effectue une recherche doublante à la fois, toujours suivie d'une recherche dichotomique. L'algorithme termine quand aucun élément supérieur à m ne peut plus appartenir à l'ensemble de multiplicité t, c'est à dire quand $k - t + 1$ tableaux ont leur dernier élément inférieur à m.

Soit une instance I de signature $(k, n_1, \ldots, n_k)$ et de t-alternance δ_t. Appelons "comparaisons doublantes" les comparaisons effectuées lors d'une recherche doublante, "comparaisons dichotomiques" celles effectuées lors d'une recherche dichotomique, et "comparaisons de tas" celles effectuées lors d'un ajout ou d'un retrait de H.

Soit $P = (I_j)_{j \leq \delta_t}$ une partition t-certificat de l'instance. Lors d'une exécution de l'algorithme `Multiplicité`, les valeurs successives de m forment une suite

Algorithme 10 `Multiplicité`$(t, A_1, \ldots, A_k)$

Étant donné un entier t et k tableaux triés non vides $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$, l'algorithme retourne l'ensemble T de multiplicité t correspondant.

$M \leftarrow$ le $t$ième plus grand élément de $\{A_1[n_1], A_2[n_2], \ldots, A_k[n_k]\}$
$m \leftarrow$ le $t$ième plus petit élément de $\{A_1[1], A_2[1], \ldots, A_k[1]\}$
$T \leftarrow \emptyset$; $H \leftarrow \emptyset$
pour tout i **faire**
  $p_i \leftarrow 1$
  si $A_i[1] = m$ **alors** $contient_i \leftarrow OUI$
  si $A_i[1] < m$ **alors** $contient_i \leftarrow POSSIBLE$
  si $A_i[1] > m$ **alors** $contient_i \leftarrow NON$; $H \leftarrow H \cup A_i[1]$
fin pour
tant que $m \leq M$ **faire**
  tant que $\#OUI \geq t$ ou $\#NON = k - t + 1$ **faire**
    si $\#OUI \geq t$ **alors** $T \leftarrow T \cup \{m\}$ **fin si**
    pour tout i tel que $contient_i = OUI$ **faire**
      $p_i \leftarrow p_i + 1$
      si $\#POSSIBLE < t - 1$ **alors** $contient_i \leftarrow POSSIBLE$
      sinon $H \leftarrow H \cup \{A_i[p_i]\}$; $contient_i \leftarrow NON$ **fin si**
    fin pour
    $m \leftarrow \min H$
    Retirer tous les tableaux de H tels que $m = A_i[p_i]$ et les marquer OUI
  fin tant que
  Soit A_i le prochain tableau marqué $POSSIBLE$ dans l'ordre fixé.
  $(g, d) \leftarrow$ `Recherche doublante`(m, A_i, p_i)
  $p_i \leftarrow$ `Recherche dichotomique`(m, A_i, g, d)
  si $m = A_i[p_i]$ **alors** $contient_i \leftarrow OUI$
  sinon $H \leftarrow H \cup \{A_i[p_i]\}$; $contient_i \leftarrow NON$ **fin si**
fin tant que
retourner T

croissante. Découpons cette exécution en δ_t phases, une comparaison appartenant à la phase j si $m \in I_j$.

Pour chaque tableau A_i et chaque phase j, soit g_j^i le nombre de comparaisons doublantes effectuées dans le tableau A_i pendant la phase j. Le nombre total de comparaisons doublantes dans le tableau A_i est $\sum_j g_j^i = \sum_j \log 2^{g_j^i}$. Par concavité du logarithme, ceci est inférieur à $\delta_t \log(\sum_j 2^{g_j^i}/\delta_t)$ où $2^{g_j^i}$ est le nombre d'éléments "sautés" par la recherche doublante dans A_i pendant la phase j. La somme de ces nombres n'excède pas le nombre d'éléments dans A_i : $\sum_j 2^{g_j^i} \leq n_i$. Par conséquent le nombre total de comparaisons doublantes effectuées dans le tableau A_i est inférieur à $\delta_t \log(n_i/\delta_t)$. Le nombre total de comparaisons doublantes effectuées dans l'instance est inférieur à la somme de ces nombres $\sum_i \delta_t \log(n_i/\delta_t)$.

Toute recherche dichotomique est effectuée sur un intervalle de taille 2^j, obtenu par une recherche doublante de j pas. Le nombre de comparaisons effectuées lors de cette recherche doublante est j, tout comme le nombre de comparaisons effectuées lors de la recherche dichotomique. Le nombre total de comparaisons dichotomique est égal au nombre de comparaisons doublantes.

Le calcul de m et M, l'initialisation du marquage et l'initialisation du tas H nécessitent au plus $2k \log k$ comparaisons : il suffit de trier $\{A_1[n_1], \ldots, A_k[n_k]\}$ et $\{A_1[1], \ldots, A_k[1]\}$. À tout instant, le tas comprend au plus $k-t+1$ éléments, et permet de calculer son élément minimal en temps constant : chaque insertion dans le tas coûte au plus $\log(k-t+1)$ comparaisons. À chaque itération de la boucle principale, un élément est inséré dans le tas dans deux circonstances :

- Si $\#OUI \geq t$ ou $\#NON \geq k-t+1$, le tas est complété jusqu'à contenir $k-t+1$ éléments. Si $\#NON = k-t+1$ aucun élément n'est rajouté au tas, si $\#OUI \geq t$ alors au plus $k-t$ éléments sont rajoutés à H, et un élément est rajouté à T. Comme $\#T < \delta_t$, au plus $\delta_t(k-t+1)\log(k-t+1)$ comparaisons sont effectuées pour les insertions dans le tas dans ce cas.
- Si l'élément m recherché n'est pas dans le tableau A_i, le premier élément de A_i supérieur à m est inséré dans H. Ce type d'insertion n'est effectué qu'après une recherche doublante. Comme au plus $k-1$ recherches doublantes sont exécutées durant chaque phase, au plus $\delta_t(k-1)\log(k-t+1)$ comparaisons sont dues à ce type d'insertion.

En tout cela fait moins de $2k\log(k) + 2\delta_t\ k\log(k) = O(k\log(k) + \delta_t\ k\log(k-t+1))$ comparaisons dues à la gestion du tas H.

Le nombre total de comparaisons doublantes, dichotomiques et de tas est donc inférieur à

$$2\delta_t \sum_i \log(n_i/\delta_t) + k\log(k) + 2\delta_t\ k\log(k-t+1)$$

$$= \quad O\left(\delta_t \sum_i \log(n_i/\delta_t) + k\log(k) + \delta_t\, k\log(k-t+1)\right).$$

□

Les termes $k\log(k)$ et $\delta_t\ k\log(k-t+1)$ deviennent négligeables par rapport au terme $\delta\sum_i \log(n_i/\delta)$ quand la taille des tableaux augmente : l'algorithme `Multiplicité` est asymptotiquement optimal. C'est seulement pour les instances de grande t-alternance ($\delta_t = \Omega(n_i/k)$) que le dernier terme est le plus fort et que la complexité de l'algorithme est en $O(\delta_t k\log(k-t+1))$ (le terme en $k\log k$ est alors négligeable). La performance de l'algorithme est alors à un facteur $\log(k-t+1)/\log(\max_i\ n_i/\delta_t)$ de la borne inférieure. Comme $\max_i\ n_i > \delta_t$, ce facteur est petit en pratique : il est inférieur à $\log(k-t+1)$, donc inférieur à k qui correspond au nombre de mots-clés d'une requête, et est généralement inférieur à 8 [DLOM01].

4.3 Calcul de l'ensemble de multiplicité optimale

Dans le cadre d'un moteur de recherche, l'ensemble de multiplicité t est moins précis que l'intersection. La relaxation de l'intersection à l'ensemble de multiplicité t n'a de sens que si l'intersection est vide. L'ensemble de multiplicité apportant le plus d'informations est le plus petit ensemble non vide.

Définition 4.4 (Multiplicité d'une instance) *La* multiplicité *d'une instance composée de k tableaux est le nombre maximal de tableaux ayant un élément en commun.*

Définition 4.5 (Ensemble de multiplicité optimale) *Soit $\mathcal{U}$ un espace totalement ordonné. Une instance du problème de l'ensemble de multiplicité optimale est composée de k tableaux triés $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$ et composés d'éléments de $\mathcal{U}$. L'*ensemble de multiplicité optimale *d'une instance de multiplicité t est l'ensemble des éléments présents dans t tableaux.*

C'est aussi le plus petit ensemble non nul parmi $\{T_k, \ldots, T_1\}$.

Définition 4.6 (Alternance optimale) *L'*alternance optimale *d'une instance I de multiplicité t est la t-alternance δ_t de l'instance.*

Par exemple, l'instance (A, B, C, D, E) de la section précédente (page 56) est de multiplicité 3. Pour le vérifier, il suffit de représenter comme dans la section précédente l'ensemble des éléments de chaque tableau sur un axe en prenant sa

valeur pour abscisse :

$A =$				3	4	**5**	6	7				
$B =$						**5**	6	7			10	11
$C =$	0	1	2								10	11
$D =$	0	1	2						8	9		
$E =$				3	4	**5**			8	9		

La partition $\{\,]-\infty, 3[,\ [3.5[,\ \{\mathbf{5}\},\ ]5, 8[,\ [8, 10[,\ [10, +\infty[\,\}$ est une partition 3-certificat de cette instance, de 3-alternance $\delta_3(A, B, C, D, E) = 6$. Seul l'élément 5 est présent dans 3 tableaux à la fois, et aucun élément n'est présent dans plus de 3 tableaux : la *multiplicité* de l'instance est 3, l'*ensemble de multiplicité optimale* est $T_3 = \{5\}$, et l'*alternance optimale* est $t = 3$.

Corollaire 4.1 (Borne inférieure) *Pour tout $k > 1$, $t \geq \lfloor k/2 \rfloor$, $n_1, n_2, \ldots, n_k$ et δ_t, tout algorithme probabiliste calculant l'ensemble de multiplicité optimale effectue $\Omega(\delta_t \sum_i \log(n_i/\delta_t))$ comparaisons dans le pire des cas pour les instances de signature $(k, n_1, \ldots, n_k)$, de multiplicité t et d'alternance optimale δ_t.*

Preuve.
C'est une simple application du théorème 4.1 : la complexité de l'algorithme trouvant la multiplicité t de l'instance et calculant T_t est supérieure à celle d'un algorithme calculant T_t pour le même t donné. □

L'algorithme `Incrémental` construit une preuve de l'intersection. Si l'intersection est vide, il construit à partir de celle-ci une preuve pour l'ensemble de multiplicité $k-1$. Il continue ainsi séquentiellement jusqu'à trouver un ensemble de multiplicité non vide. Dans le pire des cas, l'algorithme termine en calculant l'ensemble de multiplicité 1, égal à l'union des tableaux et jamais vide.

La construction incrémentale d'une preuve de t-multiplicité exige de conserver un maximum d'informations sur les recherches passées. Il faut mémoriser la liste des *pivots* (valeurs prises par m), et pour chacun conserver la liste des tableaux où ces pivots ont été cherchés, qu'ils aient été trouvés ou non. Cette liste de pivots, associée aux indices des tableaux et des positions où l'algorithme a vérifié leur absence ou leur présence, forme un *t-certificat*.

Le certificat est une séquence de couples (P_+, P_-) où chaque couple correspond à un point clef du certificat. La composante P_+ est une séquence de couples $\{(i_1, q_1), \ldots, (i_k, q_l)\}$ telle que $A_{i_1}[q_1] = \ldots = A_{i_l}[q_l] = M$. La composante P_- est une séquence de couples (i, q) tels que $M < A_i[1]$ si $q = 1$, $M > A_i[n_i]$ si $q = \infty$ et $A_i[q-1] < M < A_i[q]$ sinon. L'algorithme termine quand m prend une valeur strictement supérieure à la borne supérieure M passée en paramètre.

Algorithme 11 `Incrémental`$(t, A_1, \ldots, A_k, p_1, \ldots, p_k, S, M)$

L'algorithme reçoit en paramètre k tableaux triés non vides $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$, des positions dans ces tableaux (passées par adresse) $p_1, \ldots, p_k$, un ensemble S d'indices de tableaux exclus de la recherche, et une borne supérieure M. Il retourne les éléments inférieurs à M de l'intersection des tableaux non exclus par S, et un certificat P.

$I \leftarrow \emptyset$; $P \leftarrow \emptyset$; $o \leftarrow 1$

$m \leftarrow \max\{A_i[p_i], i \notin S\}$

tant que $m \leq M$ **faire**

Soit A_i le tableau suivant tel que $i \notin S$

$(g, d) \leftarrow$ `Recherche doublante`(m, A_i, p_i)

$p_i \leftarrow$ `Recherche dichotomique`(m, A_i, g, d)

si $m = A_i[p_i]$ **alors**

$o \leftarrow o + 1$

si $o = t$ **alors** $I \leftarrow I \cup \{m\}$

fin si

si $o = t$ ou $m \neq A_i[p_i]$ **alors**

$P_+ \leftarrow \emptyset$; $P_- \leftarrow \emptyset$

pour tout i **faire**

si $i \in S$ ou $A_i[p_i] > m$ **alors**

$P_- \leftarrow P_- \cup \{(i, p_i)\}$

sinon si $A_i[p_i] = m$ **alors**

$P_+ \leftarrow P_+ \cup \{(i, p_i)\}$; $p_i \leftarrow p_i + 1$

fin si

fin pour

$P \leftarrow P \cup \{(P_+, P_-)\}$; $m \leftarrow A_i[p_i]$; $o \leftarrow 1$

fin si

fin tant que

retourner (I, P)

Algorithme 12 `Multiplicité optimale`$(A_1, \ldots, A_k)$

Étant donnés k tableaux triés non vides $A_1, \ldots, A_k$ de tailles $n_1, \ldots, n_k$, l'algorithme retourne leur ensemble de multiplicité optimale T.

$t \leftarrow k$
pour tout i **faire** $p_i \leftarrow 1$ **fin pour**
$M \leftarrow \max\{A_1[n_1], \ldots, A_k[n_k]\}$
$(T, P) \leftarrow$`Incrémental`$(t, A_1, \ldots, A_k, p_1, \ldots, p_k, \emptyset, M)$
tant que $T = \emptyset$ **faire**
  $t \leftarrow t - 1$; $P' \leftarrow \emptyset$; $T' \leftarrow \emptyset$
  pour tout i **faire** $p_i \leftarrow 1$ **fin pour**
  pour tout $(P_+, P_-) \in P$ **faire**
    $S \leftarrow$ les indices de tableaux apparaissant dans P_-
    $M \leftarrow A_i[p_i]$ tel que $(i, p_i) \in P_+$
    $(T", P") \leftarrow$`Incrémental`$(t, A_1, \ldots, A_k, p_1, \ldots, p_k, S, M)$
    $P' \leftarrow P' \cup P"$; $T' \leftarrow T' \cup T"$
  fin pour
  $T \leftarrow T'$; $P \leftarrow P'$
fin tant que
retourner (t, T)

Théorème 4.3 (Complexité de l'algorithme `Multiplicité optimale`**)** *Pour tout $k > 1$, t, et $n_1, n_2, \ldots, n_k$, l'algorithme* `Multiplicité optimale` *effectue $O(\delta_t \sum_i \log(n_i/\delta_t) + \delta_t k \log(k - t))$ comparaisons sur une instance de signature $(k, n_1, \ldots, n_k)$, de multiplicité t et d'alternance optimale δ_t.*

Preuve.
À tout instant l'algorithme calcule l'ensemble de multiplicité t, pour t initialisé à k et décroissant au cours de l'exécution de l'algorithme. Le premier ensemble de multiplicité non vide est retourné : c'est l'ensemble de multiplicité optimale.

Soit $A_1, \ldots, A_k$ une instance de signature $(k, n_1, \ldots, n_k)$ de multiplicité t et d'alternance optimale δ_t. Appelons "comparaisons doublantes" les comparaisons effectuées lors d'une recherche doublante, et "comparaisons dichotomiques" celles effectuées lors d'une recherche dichotomique. Chaque recherche doublante étant suivie d'une recherche dichotomique de même nombre de comparaisons, il suffit

de compter les comparaisons doublantes.

Soit $P = (I_j)_{j \leq \delta_t}$ une partition t-certificat de l'instance. Lors d'une exécution de l'algorithme `Incrémental`, les valeurs successives de m forment une suite croissante. Découpons chaque exécution en δ_t phases, une comparaison appartenant à la phase j si $m \in I_j$. Sur l'ensemble des exécutions de l'algorithme `Incrémental`, chaque tableau A_i est l'objet d'au plus *une* recherche doublante par phase. Comme il y a δ_t phases, chaque tableau est l'objet en tout d'au plus δ_t recherches doublantes.

Pour chaque phase $j = 1, \ldots, \delta_t$ et chaque exécution $l = 1, \ldots, k - t + 1$ de l'algorithme `Incrémental` soit $g^i_{j,l} \in \{0, 1\}$ le nombre de comparaisons doublantes effectuées dans le tableau A_i pendant la phase j de la $l$ième exécution de `Incrémental`. Le nombre total de comparaisons doublantes dans le tableau A_i est $\sum_{j,l} g^i_{j,l} = \sum_{j,l} \log\ 2^{g^i_{j,l}}$, où $2^{g^i_{j,l}}$ est la taille de la recherche doublante dans le tableau A_i pendant la phase j de la $l$ième exécution de `Incrémental`. Par concavité du logarithme, l'algorithme `Multiplicité optimale` n'effectue pas plus de $\delta_t \log(\sum_{j,l} 2^{g^i_{j,l}}/\delta_t)$ comparaisons doublantes.

Contrairement au cas de l'algorithme d'intersection `Séquentiel`, chaque élément peut être parcouru par plusieurs recherches doublantes, une par exécution de l'algorithme `Incrémental`. Par conséquent, la somme des tailles des recherches doublantes dans un tableau A_i peut excéder la taille n_i de A_i. Cependant, pour chaque exécution de l'algorithme `Incrémental`, la somme des tailles des recherches doublantes dans le tableau A_i n'excède pas n_i. L'algorithme `Multiplicité optimale` effectuant au plus $k - t + 1$ fois l'algorithme `Incrémental`, la somme des tailles des recherches doublantes dans le tableau A_i est inférieure à $(k-t+1)n_i$. L'algorithme n'effectue pas plus de $\delta_t \log((k-t+1)n_i/\delta)$ comparaisons dans le tableau A_i. Le résultat du théorème est obtenu en sommant sur i et en sortant le terme en $\log(k - t + 1)$. L'algorithme effectue donc $O(\delta_t \sum_i \log(n_i/\delta_t) + \delta_t\ k \log(k - t + 1))$ comparaisons. □

L'ensemble de multiplicité t est une généralisation de l'intersection et de l'union, et l'ensemble de multiplicité optimale en est une variante. Dans le cadre des moteurs de recherche indexés, chaque tableau est une liste de références associées à un mot-clé, et l'ensemble de multiplicité un ensemble de références correspondant à un maximum de mots-clés. Les mêmes résultats tiennent si les références correspondant à un mot-clé donné doivent être *exclues*. Diverses autres combinaisons sont possibles. La chapitre suivant généralise ces résultats aux *combinaisons de problèmes de décision.*

Chapitre 5

Généralisation

Dans le cadre des moteurs de recherche indexés, les requêtes conjonctives sur mots-clés correspondent à l'intersection. Certains moteurs proposent des requêtes conjonctives sur les *facteurs de mots-clés* : chaque mot de la requête est associé à l'ensemble des références associées à un mot-clé contenant ce mot. Ces requêtes correspondent au problème d'*interunion* : l'intersection d'unions de tableaux.

Pour étudier ce problème, nous décomposons le *problème élémentaire d'intersection* en une conjonction de problèmes de recherche, et plus généralement nous décomposons le *problème élémentaire de multiplicité* t en une combinaison par fonction de seuil t (une fonction booléenne positive si et seulement si au moins t des valeurs données en entrée sont positives) de k problèmes de recherche. Cette approche est généralisée aux combinaisons par fonction de seuil t de problèmes de décision respectant certaines contraintes. Les contraintes sont, pour chaque problème Pb_i, d'admettre une borne inférieure $\alpha_i f_i(n)$ de la complexité probabiliste dans le pire des cas, et d'admettre une borne supérieure $f_i(n_i)$ de la complexité déterministe dans le pire des cas. Nous montrons [BR02] alors une borne inférieure $\frac{(\min_i \alpha_i)^2}{2} \sum_{i=1}^{k} f_i(n_i)$ de la complexité probabiliste dans le pire des cas, et une borne supérieure $\sum_{i=1}^{k} f_i(n_i)$ de la complexité déterministe dans le pire des cas.

Le théorème s'appliquant à son propre résultat, il peut être appliqué plusieurs fois récursivement et prouve une borne inférieure $\Omega(\sum_{i=1}^{k} f_i(n_i))$ de la complexité probabiliste dans le pire des cas de la combinaison de problèmes de décision respectant les conditions déjà citées, par fonction booléenne *à lecture unique* (pouvant être calculées en ne lisant qu'une fois chaque variable). Le problème de décision de l'appartenance d'un élément à l'interunion rentre dans ce cadre, et cette borne inférieure permet de montrer une borne inférieure de la complexité probabiliste dans le pire des cas pour le problème d'interunion.

5.1 Combinaison de problèmes de décision

Définition 5.1 (ϕ-combinaison de problèmes de décision) *Soient $Pb_1, \ldots, Pb_k$ une famille de k problèmes de décision, et $\phi(x_1, \ldots, x_k)$ une fonction booléenne sur k variables. Une* instance combinée *est un vecteur formé d'une instance de chaque problème. Soient $x_1, \ldots, x_k$ tels que $\forall\ i\ \ x_i = 1$ si et seulement si I_i est une instance positive de Pb_i. La* ϕ-combinaison de problèmes de décision *$\phi(Pb_1, \ldots, Pb_k)$ est le problème de décision sur les instances combinées pour lequel une instance $I = (I_1, \ldots, I_k)$ est positive si et seulement si $\phi(x_1, \ldots, x_k) = 1$. L'ensemble des requêtes admises sur ce problème est l'union des ensembles de requêtes admises sur chaque sous-problème.*

Définition 5.2 (Signature) *Si la signature des instances de chaque sous-problème est définie, alors la* signature *d'une instance combinée est le vecteur formé du nombre de sous-problèmes et des signatures de ces sous-problèmes.*

Définition 5.3 (Fonction de seuil) *Pour tout k et $t \leq k$, la* fonction de seuil *t sur k variables $x_1, \ldots, x_k$ est la fonction booléenne de valeur positive si et seulement si le nombre de variables positives est supérieur ou égal à t.*

Définition 5.4 (Fonction à lecture unique) *Une* formule à lecture unique *est une formule sur un ensemble fini X de variables booléennes dans laquelle chaque variable de X apparaît exactement une fois. Une fonction booléenne est* à lecture unique *si elle peut être exprimée par une formule à lecture unique. La* hauteur *d'une telle fonction est la profondeur syntaxique minimale d'une formule à lecture unique exprimant cette fonction.*

Dans le cadre des moteurs de recherche indexés par *facteurs de* mots-clés, chaque mot de la requête correspond à une famille de mots-clés contenant ce mot. Chaque mot-clé correspond à son tour à un tableau contenant une liste de références associées au mot-clé. La liste des références associées à un mot de la requête est l'union des tableaux associés aux mots-clés contenant ce mot. La réponse à la requête conjonctive sur ces mots est l'intersection des listes de références associées à chaque mot. C'est l'*interunion* des familles de tableaux :

Définition 5.5 (Interunion) *Soit $\mathcal{U}$ un espace totalement ordonné. Une instance du problème d'interunion est composée de k familles de tableaux triés $A_1, \ldots, A_k$ comprenant respectivement $l_1, \ldots, l_k$ tableaux $A_1^1, \ldots, A_1^{l_1}, \ldots, A_k^1, \ldots A_k^{l_k}$ de tailles respectives $n_1^1, \ldots, n_1^{l_1}, \ldots, n_k^1, \ldots n_k^{l_k}$ et composés d'éléments de $\mathcal{U}$. L'*interunion *$A_1 \cap \ldots \cap\ A_k = (A_1^1 \cup \ldots \cup A_1^{l_1}) \cap \ldots (A_k^1 \cup \ldots \cup A_k^{l_k})$ de cette instance est le tableau trié contenant les éléments présents dans toutes les familles.*

Ce problème est similaire à celui de l'intersection. Comme pour l'intersection, un adversaire peut faire effectuer $\sum_{i=1}^{k} \sum_{j=1}^{l_i} \log n_i^j$ comparaisons à tout algorithme déterministe décidant de l'appartenance d'un élément x à l'interunion.

Définition 5.6 (Recherche) *Étant donné un élément x et un tableau A, le* problème de recherche *de x dans A consiste à décider de la présence de x dans le tableau A.*

Le *problème élémentaire d'intersection* consiste à décider de l'appartenance d'un élément x à l'intersection. Il se décompose en une conjonction de problèmes de recherche : x est dans l'intersection s'il est dans chacun des tableaux. Le *problème élémentaire d'union* consiste à décider de l'appartenance d'un élément x à l'union. Il se décompose en une disjonction de problèmes de recherche : x est dans l'union s'il est dans au moins un des tableaux. Le *problème élémentaire de multiplicité t* consiste à décider de l'appartenance d'un élément x à l'ensemble de multiplicité t. Il se décompose en une fonction de seuil de problèmes de recherche : x est dans l'ensemble de multiplicité t s'il est dans au moins t des tableaux.

Le théorème 3.1, à la base des résultats concernant l'analyse par alternance de l'intersection, repose sur deux propriétés essentielles de la complexité de calcul du problème de recherche d'un élément dans un tableau trié. Soit $C(A, x)$ la complexité de calcul d'un algorithme déterministe pour le problème de recherche d'un élément x dans un tableau trié A de taille n. Il existe une distribution sur les instances du problème de recherche telle que

$$E(C(A, x)) \geq \log n \tag{5.1}$$

$$\Pr\{C(A, x) < a\} \leq 4^a/n \tag{5.2}$$

Ce sont des propriétés très fortes, auxquelles correspondent des propriétés équivalentes dans l'étude des combinaisons de problèmes de décisions.

5.2 Hypothèses

Toute instance positive P du problème de recherche de x dans A correspond à au moins une instance négative du problème de recherche de x dans A' telle que A et A' ne diffèrent qu'en un seul élément, x. Travailler sur les *paires d'instances* permet de construire des instances positives et négatives du problème combiné.

Définition 5.7 (Complexité de différenciation) *Soit A un arbre de décision, P une instance positive et N une instance négative. La* complexité de différenciation *$D(A, P, N)$ de l'algorithme déterministe correspondant à A sur la paire d'instances (P, N) est le nombre de noeuds de l'arbre de décision A traversés par chacune des exécutions de A sur P et N.*

Définition 5.8 (Borne inférieure de la complexité de différenciation) *Un problème de décision admet une* borne inférieure $f(n)$ de la complexité de différenciation *sur les instances de taille n si il existe une distribution sur les paires d'instances (P, N) telle que P soit une instance positive, N soit une instance négative, que les tailles de P et N soient égales à n, telles que, pour tout algorithme déterministe A, la complexité de différenciation moyenne de A est au moins $f(n)$:*

$$E(D(A, P, N)) \geq f(n).$$

Cette hypothèse correspond à la propriété (5.1) $E(C(A, x)) \geq \log\ n$ du problème de recherche.

La deuxième propriété du problème de recherche utilisée pour l'analyse du problème d'intersection est (5.2) $\Pr\{C(A, x) < a\} \leq 4^a/n$. Son équivalent sur les problèmes de recherche est une borne inférieure de la complexité de différenciation *tronquée et normalisée* : les lemmes 5.1 et 5.2 montrent la correspondance entre la complexité de différenciation et la complexité tronquée et normalisée.

Définition 5.9 (Complexité tronquée et normalisée) *Soit un paramètre entier F. La* complexité tronquée et normalisée $X(A, P, N, F)$ *d'un arbre de décision A sur une paire d'instances (P, N) est le nombre de noeuds de profondeur inférieure ou égale à F qui sont traversés à la fois par l'exécution de A sur P et par l'exécution de A sur N, divisé par F : (Par définition $X(A, P, N, F) \leq 1$)*

$$X(A, P, N, F) = \begin{cases} \frac{D(A,P,N)}{F} & \text{si } D(A, P, N) \leq F \text{ ;} \\ 1 & \text{sinon.} \end{cases}$$

Définition 5.10 (Borne inférieure de la complexité tronquée et normalisée) *Un problème de décision admet une* borne inférieure α de la complexité tronquée et normalisée par $f(n)$ *sur les instances de taille n si il existe une distribution sur les paires d'instances (P, N) telle que P soit une instance positive, N soit une instance négative, que les tailles de P et N soient égales à n, et que pour tout arbre de décision A, la complexité tronquée et normalisée moyenne soit au moins α :*

$$E(X(A, P, N, f(n))) \geq \alpha.$$

Lemme 5.1 (Initial) *Un problème de décision qui admet une borne inférieure $\alpha f(n)$ de la complexité de différenciation, et un algorithme de complexité dans le pire des cas inférieure à $f(n)$, admet une borne inférieure $\alpha/2$ de la complexité tronquée et normalisée par $f(n)$.*

Preuve.
Soient A et B deux algorithmes pour ce problème, B étant de complexité dans le

pire des cas inférieure à $f(n)$, et soit $\mathcal{D}$ la distribution de la borne inférieure de la complexité de différenciation. Soit A' l'algorithme obtenu en simulant alternativement A et B :

$$\begin{aligned} D(A', P, N) &= 2 * \min(D(A, P, N), D(B, P, N)) \\ &\leq 2 * \min(D(A, P, N), f(n)). \end{aligned}$$

Par hypothèse, $E_{\mathcal{D}}(D(A', P, N)) \geq \alpha\ f(n)$. Combinant ces deux inégalités, la linéarité de la moyenne permet de déduire que

$$\begin{aligned} E_{\mathcal{D}}(\min(D(A, P, N), f(n))) &\geq \frac{1}{2} * E(D(A', P, N)) \\ &\geq \frac{\alpha}{2} f(n). \end{aligned}$$

D'où le résultat du lemme :

$$E_{\mathcal{D}}(X(A, P, N, f(n))) \geq \frac{\alpha}{2}.$$

□

Lemme 5.2 (Final) *Un problème de décision qui admet une borne inférieure α de la complexité tronquée et normalisée par $f(n)$ admet également une borne inférieure $\alpha f(n)$ de la complexité de différenciation.*

Preuve.
Par définition, la complexité tronquée et normalisée est inférieure ou égale à la complexité de différenciation. □

5.3 Bornes inférieures

Théorème 5.1 (Propagation) *Soient $k \in \mathbb{N}$, $t \in \{1, \ldots, k\}$. Soit $(Pb_1, \ldots, Pb_k)$ une famille de problèmes de décision indépendants, et $(f_1, \ldots, f_k)$ une famille de fonctions entières telles que chaque sous-problème Pb_i admet une borne inférieure α_i de la complexité tronquée et normalisée par $f_i(n_i)$. Alors la complexité tronquée et normalisée de la combinaison à seuil $Pb = T_t(Pb_1, \ldots, Pb_k)$ admet une borne inférieure $\alpha = (\min_i \alpha_i)^2/2$ de la complexité tronquée et normalisée par $f(n) = \sum_i\ f_i(n_i)$.*

Preuve.
Fixons $(n_1, \ldots, n_k)$, pour tout i soit $F_i = f_i(n_i)$, et $F = \sum_i F_i$. Pour chaque i, soit $\mathcal{D}_i$ une distribution donnant la borne inférieure de la complexité de différenciation F_i pour le sous-problème Pb_i sur les instances de taille n_i.

La distribution $\mathcal{D}$ est définie comme suit : une paire d'instances (P_i, N_i) est tirée selon la distribution $\mathcal{D}_i$ pour chaque sous-problème *Pb*$_i$. Notons $\tilde{P} = (P_1, \ldots, P_k)$ et $\tilde{N} = (N_1, \ldots, N_k)$. La paire d'instances de distribution $\mathcal{D}$ est composée d'une instance positive P, dont exactement t sous-problèmes sont satisfaits, et d'une instance négative N, dont exactement $t-1$ sous-problèmes sont satisfaits. Ces instances diffèrent en un seul sous-problème, appelé le *témoin* : tout algorithme distinguant les deux instances doit résoudre ce problème.

$$\mathcal{D} = \left\{ \begin{array}{l} \text{Soient } (P_1, N_1), \ldots, (P_k, N_k) \text{ tirés selon } \mathcal{D}_1, \ldots, \mathcal{D}_k; \\ \text{Soit } w = i \text{ avec probabilité } F_i/F \text{ pour i=1,...,k;} \\ \text{Soit } S \text{ un sous-ensemble aléatoire de taille } t-1 \text{ de } \{1, \ldots, k\} \backslash \{w\}; \\ \text{Pour tout } j \neq w, \text{ soit } I_j = \left\{ \begin{array}{ll} P_j & \text{si } j \in S \\ N_j & \text{sinon} \end{array} \right. \\ \text{Alors } \left\{ \begin{array}{lllll} P & = & P(w,S) & = & (I_1, \ldots, I_{w-1}, P_w, I_{w+1}, \ldots, I_k) \\ N & = & N(w,S) & = & (I_1, \ldots, I_{w-1}, N_w, I_{w+1}, \ldots, I_k). \end{array} \right. \end{array} \right\}$$

Décomposition de la complexité de différenciation d'un arbre de décision fixé A :
Pour tout $j = 1, \ldots, k$, appelons j-noeud tout noeud correspondant à une requête concernant le sous-problème *Pb*$_j$. Pour $\tilde{P}$ et $\tilde{N}$ fixés, soit $D_j(w)$ le nombre de j-noeuds dans l'arbre de décision, traversés à la fois par l'exécution de A sur P et par celle de A sur N : $D(A,P,N) = \sum_j D_j(P,N)$. De manière similaire, soit $X_j(w,S)$ tel que

$$X_j(w,S) = \left\{ \begin{array}{ll} D_j(A,P,N)/F_j & \text{si } D_j(w,S) \leq F_j \\ 1 & \text{sinon.} \end{array} \right.$$

Alors par définition $X(A,P,N,F) = 1/F \min(\sum_j F_j, \sum_j D_j(w,S))$, et par propriété du min,

$$X(A,P,N,F) \geq 1/F \sum_j \min(F_j, D_j(w,S)) = \sum_j \frac{F_j}{F} X_j(w,S).$$

La moyenne sur $\mathcal{D}$ est égale par linéarité à $E_{\mathcal{D}}(X(A,P,N,F)) \geq \sum_i \sum_j \frac{F_j}{F} E_{\tilde{P},\tilde{N},S}(X_j(w,S)|w = i) \Pr\{w = i\}$, c'est à dire $\sum_i \sum_j \frac{F_i F_j}{F^2} E_{\tilde{P},\tilde{N},S}(X_j(i,S))$.

Réordonnancement des termes de la somme :
Soit S' un autre sous-ensemble de $\{1, \ldots, k\}$ dépendant de S, i et j tels que

$$S' = \left\{ \begin{array}{ll} S \backslash \{j\} \cup \{i\} & \text{si } j \in S \\ S & \text{sinon.} \end{array} \right.$$

Nous réordonnons les termes de la somme et les groupons par deux.

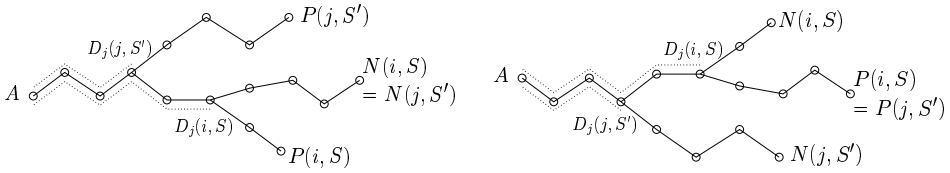
$$\begin{aligned} & E_{\mathcal{D}}(X(A,P,N,F)) \\ \geq \quad & \sum_i \frac{F_i^2}{F^2} E_{\tilde{P},\tilde{N},S}(X_i(i,S)) \\ & + \sum_i \sum_{j:j<i} \frac{F_i F_j}{F^2} E_{\tilde{P},\tilde{N},S}(X_j(i,S) + X_i(j,S')). \end{aligned} \tag{5.3}$$

Pour tout $i = 1,\ldots,k$, soient $(P_1,\ldots,P_{i-1},P_{i+1},\ldots,P_k)$ et $(N_1,\ldots,N_{i-1},N_{i+1},\ldots,N_k)$ fixés. Si (P_i,N_i) est choisi selon la distribution $\mathcal{D}_i$, par hypothèse $E(X_i(i,S)) \geq \alpha_i$. La moyenne sur $\mathcal{D}_1 \times \ldots \times \mathcal{D}_{i-1} \times \mathcal{D}_{i+1} \times \ldots \times \mathcal{D}_k$ donne une borne inférieure pour la première somme de l'équation 5.3 :

$$E_{\tilde{P},\tilde{N},S}(X_i(i,S)) \quad \geq \quad \alpha_i. \tag{5.4}$$

Borne inférieure pour $X_j(i,S) + X_i(j,S')$:
Les termes correspondant aux comparaisons sur un sous-problème non entièrement résolu par l'algorithme ne sont pas importants : il existe au moins un algorithme qui n'en effectue pas. Pour chaque paire de termes de la somme sur j, au moins un des termes de la paire peut ainsi être ignoré.

Soient i, j, $\tilde{P}$, $\tilde{N}$ et S fixés. Par définition, S ne contient pas i (et S' ne contient pas j). Si S ne contient pas j non plus, alors $S = S'$, et $N(i,S) = N(j,S')$. Si S contient j, alors S et S' diffèrent sur le statut de i et j, mais par construction $P(i,S) = P(j,S')$ (voir Figure 5.1).



cas où $j \notin S$ cas où $j \in S$

FIG. 5.1 – Chemins communs dans l'arbre de décision.

Supposons que S ne contiennent pas j. Les trois feuilles $N(i,S) = N(j,S')$, $P(j,S')$ et $P(i,S)$ correspondent à trois branches. Chaque noeud concernant un unique sous-problème, les branches se séparent en différents noeuds. Appelons "branche principale" la branche correspondant à la feuille $N(i,S) = N(j,S')$:

– Si la branche de feuille $P(j,S')$ se sépare de la branche principale avant la branche de feuille $P(i,S)$, alors $D_j(i,S) \geq D_j(j,S')$, et

$X_j(i,S) \geq X_j(j,S')$. D'autre part $X_i(j,S') \geq 0$, aussi a-t-on $X_j(i,S) + X_i(j,S') \geq X_j(j,S')$.

– De la même manière, si la branche de feuille $P(j,S')$ se sépare de la branche principale après la branche de feuille $P(i,S)$, alors $X_j(i,S) + X_i(j,S') \geq X_i(i,S)$.

La combinaison de ces inégalités permet de montrer une borne inférieure de $X_i(j,S') + X_j(i,S)$:

$$X_i(j,S') + X_j(i,S) \geq \min\left(X_i(i,S), X_j(j,S')\right).$$

La même inégalité est obtenue dans le cas où S contient j, en considérant la branche de feuille $P(i,S) = P(j,S')$ comme étant la branche principale, et selon l'ordre dans lequel les branches de feuilles $N(j,S')$ et $N(i,S)$ se séparent de la branche principale.

Pour i et j fixés, et pour toute instance N, soit $\mathit{chem}(N)$ l'ensemble des noeuds du chemin dans l'arbre de décision A, de la racine à la feuille correspondant à l'instance N. Nous définissons

$$\mathit{Opt}_i(q) = \inf_{A_q} E_{\mathcal{D}}(X_i(A_q,i,S) | q \in \mathit{chem}(N)),$$

où l'infimum est pris sur tous les arbres de décision A_q égaux à A, en dehors du sous-arbre enraciné en q. (Opt_j est défini de manière analogue.) Par hypothèse, si r_A est la racine de A, alors $\mathit{Opt}_i(r_A) \geq \alpha_i$. Pour tout noeud q, nous prouvons par récurrence que $E_{\tilde{P},\tilde{N},S}(\min(X_i(i,S), X_j(j,S')) | q \in \mathit{chem}(N)) \geq \mathit{Opt}_i(q)\mathit{Opt}_j(q)$.

Supposons que q soit une feuille. Soit $h_i(q)$ le nombre de i-noeuds de hauteur inférieure à F_i dans $\mathit{chem}(q)$, et $h_j(q)$ le nombre de j-noeuds de hauteur inférieure à F_j. Alors, $\min(X_i(i,S), X_j(j,S')) = \min(h_i(q)/F_i, h_j(q)/F_j)$. Par définition de la complexité tronquée et normalisée, ces termes sont inférieurs à 1. Par conséquent, ils sont tous deux inférieurs à leur produit, égal à $\mathit{Opt}_i(q)\mathit{Opt}_j(q)$, ce qui prouve les conditions initiales du raisonnement par récurrence.

Supposons maintenant que q soit un noeud interne, par exemple un i-noeud (l'autre cas est analogue). Notons "$q \to p$" si q est le père du noeud p. Par définition de *Opt*, fixer q est une contrainte supplémentaire sur les sous-arbres considérés dans l'infimum, et

$$\mathit{Opt}_i(q) \leq \sum_{p:q\to p} \mathit{Opt}_i(p) \Pr(p \in \mathit{chem}(N) | q \in \mathit{chem}(N)).$$

De plus, q étant un i-noeud et les distributions des sous-instances étant indépendantes, $\mathit{Opt}_j(q) = \mathit{Opt}_j(p)$ pour tout fils p de q. On obtient ainsi

$$E_{\tilde{P},\tilde{N},S}\left(\min(X_i(i,S), X_j(j,S') | q \in \mathit{chem}(N)\right)$$

$$\begin{aligned}
&\geq \sum_{p:q\to p} Opt_i(p)Opt_j(p)\ \Pr(p \in chem(N) | q \in chem(N)) \\
&\geq Opt_j(q) \sum_{p:q\to p} Opt_j(q)\ \Pr(p \in chem(N) | q \in chem(N)) \\
&\geq Opt_i(q)Opt_j(q).
\end{aligned}$$

On obtient une borne inférieure de la moyenne de la somme des paires de termes en appliquant cette inégalité à la racine de A :

$$E_{\tilde{P},\tilde{N},S}(X_i(i,S) + X_j(j,S')) \quad \geq \quad \alpha_i\alpha_j. \tag{5.5}$$

Conclusion :
La complexité moyenne de l'algorithme est bornée en combinant les équations (5.4) et (5.5) dans l'équation (5.3) :

$$\begin{aligned}
E_{\mathcal{D}}(X(A,P,N,F)) &\geq \sum_i \sum_{j:j\leq i} \frac{F_iF_j}{F^2}\alpha_i\alpha_j \\
&\geq \frac{1}{2}\left(\frac{\sum_i \alpha_iF_i}{F}\right)^2 \\
&\geq \frac{(\min_i \alpha_i)^2}{2}
\end{aligned}$$

□

Théorème 5.2 (Borne inférieure-fonctions de seuil) *Soit k fixé, $t \leq k$, et $\phi_t(x_1,\ldots,x_k)$ la fonction de seuil t sur k variables. Soient $Pb_1,\ldots,Pb_k$ une famille de k problèmes de décision indépendants, et $(f_1,\ldots,f_k)$ une famille de fonctions entières telles que chaque problème Pb_i admette sur les instances de taille n_i un algorithme déterministe de complexité $f_i(n_i)$ dans le pire des cas, et une borne inférieure $\alpha_i f_i(n_i)$ de la complexité de différenciation.*

Alors le problème $Pb = \phi_t(Pb_1,\ldots,Pb_k)$ admet sur les instances de signature $(k,n_1,\ldots,n_k)$ un algorithme déterministe de complexité $f(n) = \sum_i f_i(n_i)$ dans le pire des cas, et une borne inférieure $\frac{\alpha^2}{8}\sum_i f_i(n_i) = \Omega(f(n)$ de la complexité de différenciation.

Preuve.
La borne supérieure est la complexité de l'algorithme résolvant tous les sous-problèmes. Le lemme 5.1 appliqué à chaque sous-problème montre une borne inférieure $\alpha/2$ de la complexité tronquée et normalisée par $f_i(n_i)$, où $\alpha = \min\{\alpha_i\}$.

Le théorème 5.1 permet d'en déduire une borne inférieure de $\alpha^2/8$ de la complexité tronquée et normalisée par $\sum_i f_i(n_i)$ du problème combiné. Le lemme 5.2 fournit le résultat final : une borne inférieure $\frac{\alpha^2}{8} \sum_i f_i(n_i)$ de la complexité de différentiation pour le problème combiné. □

Corollaire 5.1 (Borne inférieure-fonctions à lecture unique) *Soit k fixé, et $\phi(x_1, \ldots, x_k)$ une fonction booléenne à lecture unique sur k variables. Soient $Pb_1, \ldots, Pb_k$ une famille de k problèmes de décision indépendants, et $(f_1, \ldots, f_k)$ une famille de fonctions entières telles que chaque problème Pb_i admette sur les instances de taille n_i un algorithme déterministe de complexité $f_i(n_i)$ dans le pire des cas, et une borne inférieure $\alpha_i f_i(n_i)$ de la complexité de différenciation.*

Alors le problème $Pb = \phi(Pb_1, \ldots, Pb_k)$ admet sur les instances de signature $(k, n_1, \ldots, n_k)$ un algorithme déterministe de complexité $f(n) = \sum_i f_i(n_i)$ dans le pire des cas, et une borne inférieure $\Omega(f(n)$ de la complexité de différenciation.

Preuve.
La borne supérieure est la complexité de l'algorithme résolvant tous les sous-problèmes.

Soit A un arbre d'évaluation à lecture unique de ϕ. Sans perte de généralité, supposons que les négations de A sont toutes au niveau des feuilles. A est alors à lecture unique et composé uniquement de disjonction et de conjonction sur $X' = (x_1, \ldots, x_k, \overline{x_1}, \ldots, \overline{x_k})$, où aucune variable x n'est présente si sa négation $\overline{x}$ l'est.

Le lemme 5.1 appliqué à chaque sous-problème permet d'obtenir une borne inférieure $\alpha_i/2$ de la complexité tronquée et normalisée par $f_i(n_i)$. Soit $\alpha' = \min_i \alpha_i/2$. Le théorème 5.1, appliqué à chaque noeud de A correspondant à une disjonction ou une conjonction, des feuilles aux branches, prouve une borne inférieure $\alpha = 2(\frac{\alpha'}{2})^{2^h})$ de la complexité tronquée et normalisée par $f(n) = \sum_i f_i(n_i)$ pour le problème combiné. Le facteur $\alpha > 0$ dépend uniquement des sous-problèmes et de la hauteur de l'arbre d'évaluation de ϕ. Cette hauteur est inférieure à k puisque ϕ est à lecture unique. La complexité de différentiation pour le problème combiné admet donc une borne inférieure de $\alpha\ f(n) = \Omega(f(n))$. □

5.4 Interunion

Définition 5.11 (Multi-signature) *La* multi-signature *d'une instance composée de k familles, comprenant respectivement $l_1, \ldots, l_k$ tableaux triés, de tailles respectives $n_1^1, \ldots, n_1^{l_1}, \ldots, n_k^1, \ldots n_k^{l_k}$ est le vecteur composé de ces nombres $(k, l_1, \ldots, l_k, n_1^1, \ldots, n_1^{l_1}, \ldots, n_k^1, \ldots\ n_k^{l_k})$.*

Définition 5.12 (Multi-alternance) *Une* partition certificat $P=(I_j)_{1\leq j\leq\delta}$ *d'une instance composée de k familles de tableaux triés $\{A_1^1,\ldots,A_1^{l_1}\},\ldots,\{A_k^1,\ldots,A_k^{l_k}\}$ est une partition de l'espace des éléments telle que :*

1. *tout singleton corresponde à un élément de l'interunion,*
2. *et tout intervalle soit d'intersection vide avec au moins une famille, c'est à dire avec tous les tableaux de cette famille.*

La multi-alternance *d'une instance est la taille minimale d'une telle partition.*

Lemme 5.3 (Borne inférieure du problème de recherche) *La complexité de différenciation du problème de recherche d'un élément x dans un tableau trié A de taille n admet une borne inférieure* $\log n$.

Corollaire 5.2 (Borne inférieure élémentaire) *Pour tout $k>1$, $l_1,\ldots,l_k$, $n_1^1,\ldots,n_1^{l_1},\ldots,n_k^1,\ldots n_k^{l_k}$, tout algorithme probabiliste effectue $\Omega(\sum_{i,j}\log(n_i^j))$ comparaisons dans le pire des cas pour décider de l'appartenance d'un élément x à l'interunion d'une instance de multi-signature $(k,l_1,\ldots,l_k,n_1^1,\ldots,n_1^{l_1},\ldots,n_k^1,\ldots\ n_k^{l_k})$.*

Preuve.
On obtient le résultat voulu en appliquant le théorème 5.1 et le lemme 5.3. □

Théorème 5.3 (Borne inférieure) *Pour tout $k>1$, $l_1,\ldots,l_k$, $n_1^1,\ldots,n_1^{l_1},\ldots,n_k^1,\ldots n_k^{l_k}$ et δ, tout algorithme probabiliste calculant l'interunion effectue $\Omega(\delta\sum_{i,j}\log(n_i^j/\delta))$ comparaisons pour les instances de multi-signature $(k,l_1,\ldots,l_k,n_1^1,\ldots,n_1^{l_1},\ldots,n_k^1,\ldots\ n_k^{l_k})$ et de multi-alternance δ.*

Preuve.
La preuve est similaire à celle du théorème 3.2 : une instance de multi-alternance δ est construite à partir d'instances élémentaires de multi-alternance finie.

Soit $p=\lceil\delta/3\rceil$. Pour toute instance $I=(A_1^1,\ldots,A_k^{l_k})$ de signature $(k,l_1,\ldots,l_k,n_1^1/p,\ldots,n_k^{l_k}/p)$ obtenue par le corollaire 5.2, l'instance $I'=(\{x\},A_1^1,\ldots,A_k^{l_k})$ de signature $(k+1,1,l_1,\ldots,l_k,1,n_1^1/p,\ldots,n_k^{l_k}/p)$ est de multi-alternance inférieure ou égale à 3. La partition $]-\infty,x[,[x,x+\varepsilon[,[x+\varepsilon,\infty[$ certifie l'interunion. L'instance I' équivaut à l'instance I : x est dans l'interunion de I_1 si et seulement si l'interunion de I_2 est égale à $\{x\}$. Par conséquent, tout algorithme probabiliste effectue $\Omega(\sum_{i,j}\log(n_i^j/p))$ comparaisons pour résoudre l'instance I'.

Soient p instances $I_1,\ldots,I_p$ obtenues par le corollaire 5.2, et les p instances correspondantes $I'_1,\ldots,I'_p$. De la même manière que pour le théorème 3.2, les instances élémentaires $I_1,\ldots,I_p$ peuvent être combinées pour obtenir une instance

I du problème d'interunion de signature $(k, 1, l_1, \ldots, l_k, p, n_1^1, \ldots, n_k^{l_k})$. Les instances élémentaires $I'_1, \ldots, I'_p$ se combinent elles en une instance I' du problème d'interunion de signature $(k+1, 1, l_1, \ldots, l_k, p, n_1^1, \ldots, n_k^{l_k})$ et d'alternance au plus $3p \leq \delta$.

La résolution de l'instance I' équivaut à la résolution des p instances élémentaires $I'_1, \ldots, I'_p$, et nécessite donc $\Omega(\sum_{i,j} \log(n_i^j/p))$ comparaisons. La résolution de l'instance I nécessite au moins autant de comparaisons que l'instance I', d'où la borne inférieure de $\Omega(\sum_{i,j} \log(n_i^j/p))$ comparaisons. □

Théorème 5.4 (Complexité de l'algorithme `Interunion`) *Pour tout $k > 1$, $l_1, \ldots, l_k$, $n_1^1, \ldots, n_1^{l_1}, \ldots, n_k^1, \ldots n_k^{l_k}$ et δ, l'algorithme* `Interunion` *effectue $O(\delta \sum_{i=1}^{k} \sum_{j=1}^{l_i} \log(n_i^j/\delta))$ comparaisons sur une instance de multi-signature $(k, l_1, \ldots, l_k, n_1^1, \ldots, n_1^{l_1}, \ldots, n_k^1, \ldots\ n_k^{l_k}$ et de multi-alternance δ.*

Preuve.
À tout instant l'algorithme teste l'appartenance de m à l'interunion, et tous les éléments de l'intersection inférieurs à m sont dans I. La suite des valeurs successives de m est strictement croissante. L'algorithme effectue une recherche doublante à la fois, toujours suivie d'une recherche dichotomique. L'algorithme termine quand aucun élément supérieur à m ne peut plus appartenir à l'interunion, c'est à dire quand tous les tableaux ont leur dernier élément inférieur à m.

Soit $A_1^1, \ldots, A_1^{l_1}, \ldots, A_k^1, \ldots A_k^{l_k}$ une instance de signature $(k, l_1, \ldots, l_k, n_1^1, \ldots, n_1^{l_1}, \ldots, n_k^1, \ldots\ n_k^{l_k})$ et de multi-alternance δ. Notons $l = \sum_{i=1}^{k} l_i$.

Soit $P = (I_j)_{j \leq \delta}$ une partition certificat de l'instance. Lors d'une exécution de l'algorithme `Interunion`, les valeurs successives de m forment une suite croissante. Découpons cette exécution en δ phases, une comparaison appartenant à la phase j si $m \in I_j$.

Appelons "comparaisons doublantes" les comparaisons effectuées lors d'une recherche doublante, "comparaisons dichotomiques" celles effectuées lors d'une recherche dichotomique, et "comparaisons internes" celles effectuées pour calculer l'élément minimal ou maximal d'un ensemble de valeurs.

Comparaisons internes : Le calcul de chaque M_i et m_i nécessite $l_i - 1$ comparaisons, c'est à dire $2(l - k)$ comparaisons pour calculer toutes les valeurs. Les calculs de M et de la valeur initiale de m nécessitent $k - 1$ comparaisons chacun. En tout $2(l-1)$ comparaisons internes sont nécessaires pour l'initialisation de l'algorithme. À chaque changement de valeur de m, la nouvelle valeur est choisie dans une famille A_i, parmi au plus l_i valeurs, en au plus $2l_i$ comparaisons internes. m change au plus $k - 1$ fois de valeur pendant une même phase, et cette valeur

Algorithme 13 `Interunion`$(l_1, \ldots, l_k, A_1^1, \ldots, A_1^{l_1}, \ldots, A_k^1, \ldots A_k^{l_k})$

Étant donné $l = \sum_{i=1}^{k} l_i$ tableaux triés non vides $A_1^1, \ldots, A_1^{l_1}, \ldots, A_k^1, \ldots A_k^{l_k}$ de tailles respectives $n_1^1, \ldots, n_1^{l_1}, \ldots, n_k^1, \ldots n_k^{l_k}$, l'algorithme retourne leur interunion I.

$I \leftarrow \emptyset$; $o \leftarrow 1$
pour tout $i = 1, \ldots, k$ **faire**
  $M_i \leftarrow \max_j A_i^j[n_i^j]$; $m_i \leftarrow \min_j A_i^j[1]$
  pour tout $j = 1, \ldots, l_i$ **faire** $p_i^j \leftarrow 1$ **fin pour tout**
fin pour
$M \leftarrow \min_i\{M_i\}$; $m \leftarrow \max_i\{m_i\}$
$(i, j_i) \leftarrow (i', j')$ tel que $m = A_{i'}^{j'}[1]$
tant que $m \leq M$ **faire**
  $i \leftarrow (i-1)$ modulo $k+1$; $j \leftarrow 1$
  tant que $j \leq l_i$ et $A_i^j[p_i^j] \neq m$ **faire**
    $(g, d) \leftarrow$ `Recherche doublante`(m, A_i^j, p_i^j)
    $p_i^j \leftarrow$ `Recherche dichotomique`(m, A_i^j, g, d)
  fin tant que
  si $m = A_i^j[p_i^j]$ **alors**
    $o \leftarrow o + 1$; $j_i \leftarrow j$
    si $o = k$ **alors**
      $I \leftarrow I \cup \{m\}$
      pour tout $i = 1. \ldots, k$ **faire** $p_i^{j_i} \leftarrow p_i^{j_i} + 1$ **fin pour**
    fin si
  fin si
  si $o = k$ ou $m \neq A_i^j[p_i^j]$ **alors**
    $m \leftarrow \min_j \max(m, A_i^j[p_i^j])$
    $o \leftarrow 1$
  fin si
fin tant que
retourner I

est toujours choisie dans une famille différente. Pendant une phase donnée, l'algorithme effectue au plus $2l$ comparaisons internes. En tout, l'algorithme effectue $2\delta l + 2(l-1) = O(\delta l)$ comparaisons internes.

Comparaisons doublantes : Dans chaque tableau A_i^j, l'algorithme effectue au plus δ recherches doublantes. Soient $(g_m)_{m=1,\ldots,\delta}$ la suite des tailles des recherche doublantes effectuées, le nombre de comparaisons doublantes effectuées par l'algorithme est inférieur à $\sum_{m=1}^{\delta} \log\ g_m \leq \delta \log(\sum_{m=1}^{\delta}\ g_m/\delta) \leq \delta \log(2n_i^j/\delta)$. Sur l'ensemble des tableaux, l'algorithme effectue au plus $\delta \sum_{i=1}^{k} \sum_{j=1}^{l_i} \log(2n_i^j/\delta)$ comparaisons doublantes.

Comparaisons dichotomiques : Pour chaque recherche doublante de m comparaisons, l'algorithme effectue une recherche dichotomique sur un intervalle de taille au plus 2^m, qui nécessite au plus m comparaisons. Il y a donc moins de comparaisons dichotomiques que de comparaisons doublantes.

En tout, l'algorithme effectue donc au plus $2\delta\ l + 2(l-1) + 2\delta \log(2n_i^j/\delta)$ comparaisons, c'est à dire $O(\delta \sum_{i=1}^{k} \sum_{j=1}^{l_i} \log(n_i^j/\delta))$ comparaisons. □

Le problème de calcul de l'interunion se pose notamment dans le moteur de recherche intégré au projet FFSS, exposé dans le chapitre suivant.

Chapitre 6

Projet FFSS

6.1 Description générale du projet

Les moteurs de recherche disponibles sur les réseaux locaux reposent sur le parcours à intervalles réguliers du réseau pour construire la liste des documents disponibles. Cette solution est coûteuse en terme de charge réseau, et inefficace : si la liste ne change pas entre deux parcours du réseau, alors le second parcours est inutile, et si la liste change beaucoup, alors le moteur de recherche peut avoir donné des réponses inexactes.

Il est plus efficace de faire signaler tout changement dans la liste des documents disponibles. Le moteur de recherche n'effectue alors des parcours que quand c'est nécessaire, et son index est toujours à jour.

Cette idée a donné lieu à l'intégration d'un moteur de recherche au projet étudiant `FFSS` (acronyme pour `Fleming File Sharing System`), développé en résidence universitaire. Ce projet a pour vocation de remplacer le protocole de partage de fichier `SMB` sur les réseaux locaux de taille importante. Il permet d'indexer sur une machine centrale les fichiers partagés sur le réseau. La fréquence de mise à jour de son index est proportionnelle à la fréquence des changements dans les listes de fichiers partagés sur chaque machine. Il intègre un nouveau protocole de partage de fichier, un moteur d'index et de recherche, et les applications les mettant en œuvre.

Le protocole et l'idée d'intégrer le moteur de recherche au protocole de partage de fichier sont issus des spécifications originales de Christophe Calmejane, David Majorel et Julien Sébot. La structure de l'index et le moteur de recherche ont été définis par Jérémy Barbay, Christophe Calmejane et Julien Sébot. Christophe Calmejane a codé la plus grande partie des applications associées, à l'exception des clients `linux` et `cgi` qui ont été implémentés par Benoit Bourdin et du client

windows qui a été implémenté par David Majorel. Loic Jaquemet travaille sur le pilote linux, Aurélien Merle sur le pilote windows. Anis Affès a été d'un grand secours pour tester les diverses implémentations et rédiger la documentation.

6.2 Structure du système

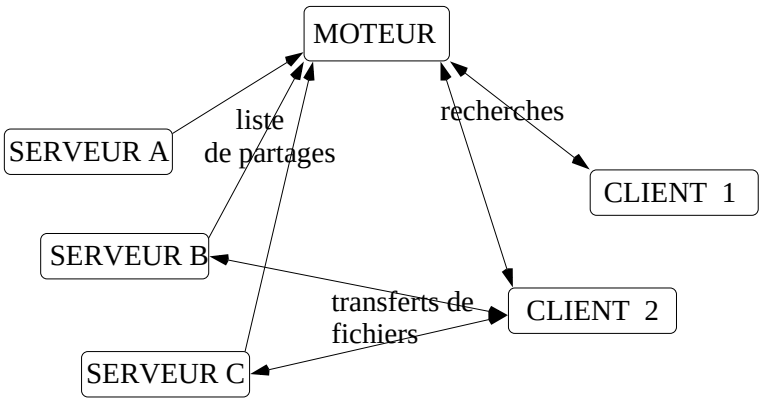


FIG. 6.1 – Interaction entre un moteur, des serveurs et des clients.

L'implémentation de FFSS se décompose en plusieurs applications :

- Serveur : Le programme donnant accès aux fichiers partagés sur la machine où il est exécuté. Il communique au moteur de recherche la liste des fichiers partagés et répond aux requêtes des clients.
- Gestionnaire : L'interface graphique du serveur sous windows, permettant d'ajouter, de retirer ou de modifier un partage.
- Client : Le programme permettant de parcourir les partages FFSS du réseau, de lancer des recherches et de transférer des fichiers.
- Moteur de recherche : Le programme maintenant la liste des machines du réseau local, l'index des fichiers partagés, qui répond aux requêtes de recherche et coordonne l'accès aux autres sous-réseaux.

Pour des réseaux de très grande taille, ou pour des réseaux séparés par un routeur, plusieurs moteurs peuvent être mis en place en parallèle. Les serveurs et clients sont alors répartis entre les différents moteurs. Quand un client effectue une requête auprès de son moteur, celui-ci y répond localement, et fait suivre la requête aux autres moteurs qui répondent directement au client.

Les serveur et les clients fonctionnent également sans moteur. En l'absence d'un moteur, le client envoie un message à toutes les machines de son

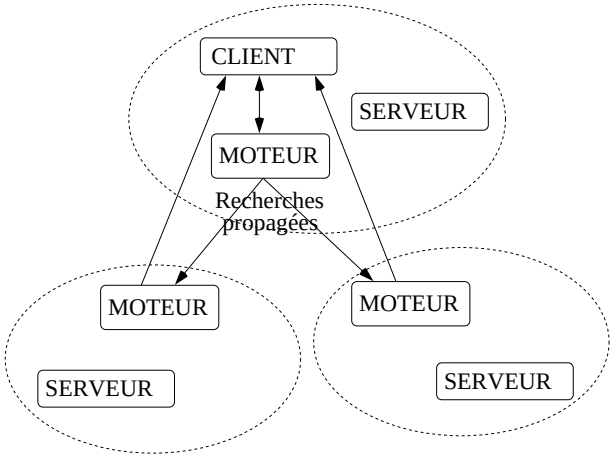


FIG. 6.2 – Interactions entre trois `moteurs` et un `client`.

réseau local pour déterminer les `serveurs` disponibles. Il peut ensuite se connecter à ces `serveurs`. Bien sur, le moteur de recherche n'est alors plus disponible.

6.3 Spécifications du moteur de recherche

Chaque fichier partagé est identifié par le serveur qui le partage, le nom du partage et un chemin d'accès dans ce partage. L'index du moteur de recherche décompose le chemin d'accès au fichier en mots-clés séparés par les caractères `{/_.}` et espaces. Les mots-clés de longueur supérieure à trois sont associés au fichier pour l'indexage, les mots-clés de longueur trois sont comparés à une liste d'extensions de fichiers, pour associer au fichier partagé un type, les mots-clés restants sont ignorés.

Chaque fichier partagé est référencé par le serveur qui le partage et par son numéro d'ordre dans le parcours préfixe de l'arborescence des fichiers partagés sur le serveur. Chaque répertoire est référencé par un intervalle parmi les numéros d'ordre dans ce parcours préfixe : la borne gauche de l'intervalle correspond au numéro d'ordre du répertoire, la borne droite au numéro d'ordre du dernier fichier contenu par le répertoire.

Une requête conjonctive sur les facteurs de mots-clés est un ensemble de mots. Tous les mots de longueur supérieure à trois doivent être contenus par un mot-clé associé au fichier. Les mots de longueur égale à trois correspondent à des restrictions supplémentaire sur les types de fichiers retournés par le moteur de recherche. Les mots de longueur inférieure à trois sont ignorés.

Pour chaque fichier ou répertoire de nom w, les trois premières lettres de w sont hachées dans un tableau comprenant un arbre de suffixe pour chaque combinaison de trois lettres possibles. La référence de ce fichier ou répertoire est insérée dans le noeud indiqué par le reste de w.

Chaque noeud d'un arbre de suffixes correspond ainsi à un mot w composé des trois premières lettres associées à l'arbre et de la suite de lettres étiquetant les arêtes de la racine à ce mot. Le noeud est associé à la liste des références des fichiers dont un mot-clé admet w comme *suffixe*. Le sous-arbre enraciné en ce noeud correspond à la liste des références des fichiers dont un mot-clé admet w comme *facteur*.

Cette liste est séparée par machine de manière à faciliter la mise à jour des fichiers par une machine particulière. Le noeud contient donc un tableau d'indices de machines partageant un fichier dont un mot-clé admet w comme suffixe, et pour chacune de ces machines la liste des références des fichiers correspondants.

Cette liste de références est donnée par un tableau d'intervalles correspondant aux répertoires et fichiers correspondant au mot w.

La réponse à une requête composée d'un seul mot w est calculée en parcourant le sous-arbre de l'arbre des suffixes enraciné dans le noeud correspondant à w. L'ensemble des références associées à w est l'ensemble des références associées aux noeuds de ce sous-arbre. C'est l'*union*, pour chaque noeud et pour chaque machine, des tableaux d'intervalles de références.

La réponse à une requête composée de plusieurs mots $w_1, \ldots, w_k$ de longueur supérieure à trois est l'intersection des ensembles de références correspondant à chaque mot. C'est l'*intersection* pour chaque mot de la requête de l'*union* pour chaque noeud et pour chaque machine des tableaux d'intervalles de références : c'est un problème d'*interunion*.

Si des mots de longueur trois sont présents dans la requête, alors seuls les références des fichiers de type correspondant à ces mots sont retournées au `client`. Les mots de la requête de longueur inférieure à trois sont ignorés.

6.4 Bilan du projet

Les quatre résidences universitaires d'Orsay correspondent à quatre réseaux informatiques interconnectés par une passerelle. Les administrateurs de ces réseaux ont chacun installé un moteur de recherche FFSS sur une machine de l'association gérant le réseau. À la date du 26 juin 2002, 4 moteurs de recherche se répartissent les 124 serveurs de partage fichier FFSS disponibles sur le réseau. Il y a au moins autant de clients installés. Les moteurs de recherches sont également accessibles sans installation préalable par l'intermédiaire d'un client `cgi` installé sur le serveur `http` de la résidence universitaire "Les Jardins de Fleming".

Le système FFSS est supporté pour le système d'exploitation Windows de Microsoft et la plupart des distributions linux, et les sources distribués gratuitement peuvent être recompilés pour d'autres systèmes d'exploitations. Des captures d'écrans sont présentées dans l'annexe 6.4.

Le projet est encore en voie de développement et de test. L'index doit notamment être modifié de manière à indexer également les mots-clés de longueur inférieure ou égale à trois. Les calculs d'intersection doivent encore être optimisés, et remplacés par des calculs d'ensemble de multiplicité optimale. Les pilotes permettant de rendre le parcours des systèmes de partages de fichier transparent pour l'utilisateur sont en cours de développement.

Un autre projet est envisagé, nommé projet Slashome, qui intégrerait le moteur de recherche au système de fichiers local, de manière à mettre à jour l'index du moteur à chaque modification de l'arborescence des fichiers. Un système de fichiers Slashome pourrait être indexé plus efficacement par le moteur de recherche du projet FFSS.

Conclusion

Apport de la thèse

Les travaux de Demaine, Lopez-Ortiz et Munro sont à l'origine de cette thèse : l'exposé de Ian Munro à ALENEX [DLOM01] démontrait l'intérêt du problème d'intersection dans les moteurs de recherche indexés. Le problème posé était intéressant : l'algorithme `Parallèle` est optimal en théorie pour l'analyse dans le pire des cas par alternance, mais se révèle moins bon en pratique que l'algorithme `Séquentiel`, qui lui n'est *pas* optimal pour cette analyse.

Nous avons montré que l'algorithme `Séquentiel` est optimal dans le pire des cas pour les instances de signature et d'alternance fixées. L'alternance est une mesure de difficulté naturelle, et les résultats de l'analyse correspondent aux résultats pratiques.

Cette analyse diffère peu de l'analyse des algorithmes adaptatifs. La différence est que la mesure de difficulté des algorithmes adaptatifs est monotone et définie de l'ensemble des instances vers les réels, alors que l'analyse fine repose uniquement sur le regroupement des instances par classes d'équivalence, sans notion de monotonie. Comme pour les algorithmes adaptatifs, les analyses sont ordonnées selon un ordre partiel.

Nous avons défini le problème de calcul de l'ensemble de multiplicité optimale, analysé la complexité de son calcul et proposé un algorithme pour le calculer. L'analyse d'un tel algorithme sur des requêtes réelles doit se faire en temps réel, et après un temps d'adaptation des utilisateurs. Les requêtes qui peuvent être enregistrées sur un moteur de recherche effectuant un algorithme d'intersection correspondent principalement à des intersections non vides : les utilisateurs se limitent à un nombre réduit de mots-clés de manière à ne pas obtenir de réponse nulle. Sur un moteur de recherche calculant l'ensemble de multiplicité optimale, les utilisateurs pourraient utiliser plus de mots-clés, et ne jamais recevoir de réponse nulle.

Nous avons généralisé ces résultats dans le but de répondre à d'autre types de requêtes, comme les requêtes sur les *facteurs de mots-clés*, ou les requêtes sur des index de structure plus complexe qu'une liste de tableaux. Ces problèmes corres-

pondent à des problèmes rencontrés lors de la réalisation du moteur de recherche intégré au projet FFSS.

Perspectives

Algorithmes probabilistes et pathologie

Les pires instances considérées dans le théorème 3.2 pour le calcul de l'intersection sont pathologiques. Il est plus difficile de décider qu'un élément x n'est pas dans l'intersection si x est présent dans $k-1$ tableaux que s'il n'est présent que dans un seul tableau, ou dans la moitié des tableaux. Or la borne inférieure du théorème 3.2 repose sur δ éléments présents dans $k-1$ tableaux.

De manière similaire au cas du problème des pièces cachées (vu dans le chapitre 1), choisir au hasard le tableau dans lequel on cherche m permet de meilleurs résultats. Si t tableaux ne contiennent pas x, la probabilité de trouver parmi les $k-1$ tableaux restants un tableau ne contenant pas x du premier coup est $t/(k-1) = \alpha$. En cas d'échec, la probabilité de trouver un tableau ne contenant pas m ne peut que croître. Le nombre moyen de recherches nécessaires pour trouver un tableau ne contenant pas x est donc inférieur à $1/\alpha = (k-1)/t$. D'autre part, si x est présent dans k tableaux, $k-1$ recherches sont nécessaires pour vérifier que x est dans l'intersection.

Posons quelques définitions :

- l'*occurence* d'un intervalle I est le nombre de tableaux d'intersection vide avec I ($t(I) = \#\{i,\ A_i \cap I = \emptyset\}$),
- la *pathologie* d'un intervalle I est inverse à son occurrence ($p(I) = 1/t(I)$), et celle d'un singleton $\{x\}$ est 1,
- la *pathologie* d'une partition certificat $P = (I_j)_{1 \leq j \leq \delta}$ de l'intersection est la somme des pathologies des intervalles et singletons la composant ($p(P) = \sum_{j=1}^{\delta} p(I_j)$),
- la *pathologie* d'une instance $(A_1, \ldots, A_k)$ est la pathologie minimale d'une partition certificat de cette instance ($p(A_1, \ldots, A_k) = \min_P p(P)$).

Alors, la pathologie mesure la difficulté d'une instance, et la complexité probabiliste du problème est $\theta(p \sum_i \log(n_i/p))$ pour les instances de signature $(k, n_1, \ldots, n_k)$ et de pathologie p.

L'analyse du problème d'intersection basée sur ce principe est plus fine que l'analyse pour les instances de signature et d'alternance fixée : tout algorithme optimal dans le pire des cas pour les instances de signature et de pathologie fixée est optimal dans le pire des cas pour les instances de signature et d'alternance fixée.

Combinaisons de problèmes de décision

Dans le chapitre 5, les hypothèses sur les problèmes à combiner sont très contraignantes : elles amènent à ne considérer que les problèmes tels que la complexité probabiliste soit égale à la complexité déterministe à un facteur près. Entre autres, le théorème 5.1 (théorème de propagation) ne s'applique pas au problème d'intersection pour les instances de pathologie fixée, où la complexité probabiliste est inférieure à la complexité déterministe. Il est peut-être possible de relâcher l'hypothèse sur la borne inférieure de la complexité de différenciation *tronquée* : cette hypothèse n'est utilisée que vers la fin de la preuve, pour prouver les conditions initiales du raisonnement par récurrence.

D'autre part, les combinaisons sont limitées aux fonctions à lecture unique. C'est un argument technique pour la généralisation des résultats sur les combinaisons par fonction de seuil. Une technique de preuve différente devrait permettre d'obtenir une borne inférieure de la complexité probabiliste dans le pire des cas du même ordre pour les combinaisons de problèmes de décision par fonction booléenne dépendant de toutes ses variables.

Borne inférieure quantique

Le problème d'intersection élémentaire est défini comme suit : étant donné un élément x, et k tableaux triés *chacun* de taille n, décider si x appartient à l'intersection des k tableaux. Dans le modèle classique sur les algorithmes probabilistes, la complexité dans le pire des cas pour les instances de signature $(k, n, \ldots, n)$ est $\Theta(k \log n)$.

Dans le modèle quantique, c'est une généralisation naturelle de deux problèmes largement étudiés :

- d'une part, le problème de la recherche d'un élément parmi k éléments non triés (c'est le problème de l'intersection élémentaire pour $n = 1$) peut être résolu de manière optimale en exactement $\sqrt{k}$ étapes (c'est un des deux exemples les plus fameux de l'accélération quantique),
- d'autre part, la recherche d'un élément parmi n éléments triés (c'est le problème de l'intersection élémentaire pour $k = 1$) peut être résolu de manière optimale en $\Theta(\log n)$ étapes (c'est un des exemples où la puissance quantique ne permet pas une accélération meilleure que par un facteur constant).

La réunion des bornes inférieures connues de ces deux problèmes ne donne qu'une borne inférieure de $\Omega(\log n + \sqrt{k})$ comparaisons, quand une borne supérieure de $\Omega(\sqrt{k}\log(n))$ comparaisons découle de l'analyse de l'algorithme de Grover. Chacune des deux bornes inférieures admet une preuve qui utilise la technique originale de Grover en majorant la différenciation que peut effectuer l'algorithme

à chaque étape entre les différentes instances possibles. Généraliser ces techniques pour résoudre ce problème pourrait mener à une preuve de la $\sqrt{k}$ optimalité de l'algorithme déterministe `Séquentiel` dans le modèle quantique.

Captures d’écran de FFSS

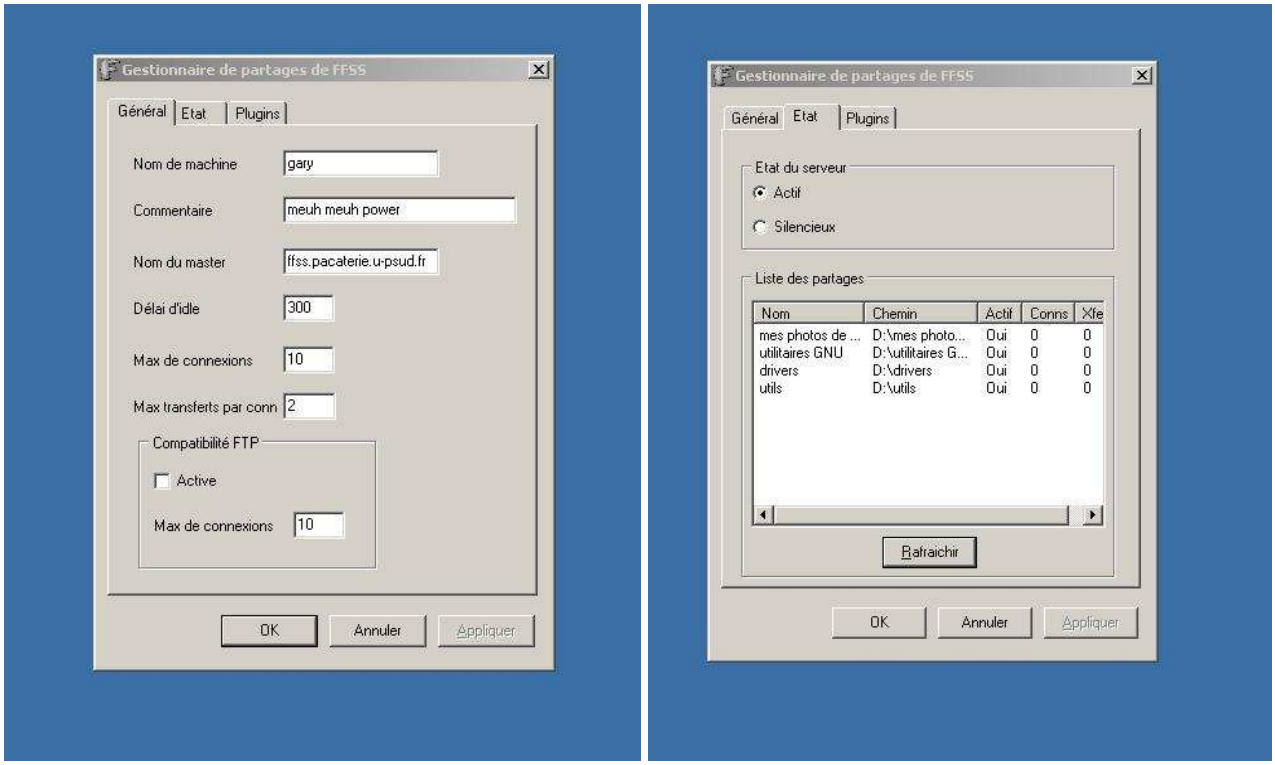


FIG. 3 – La configuration de l’identification et de l’état du `serveur` sous `windows`.

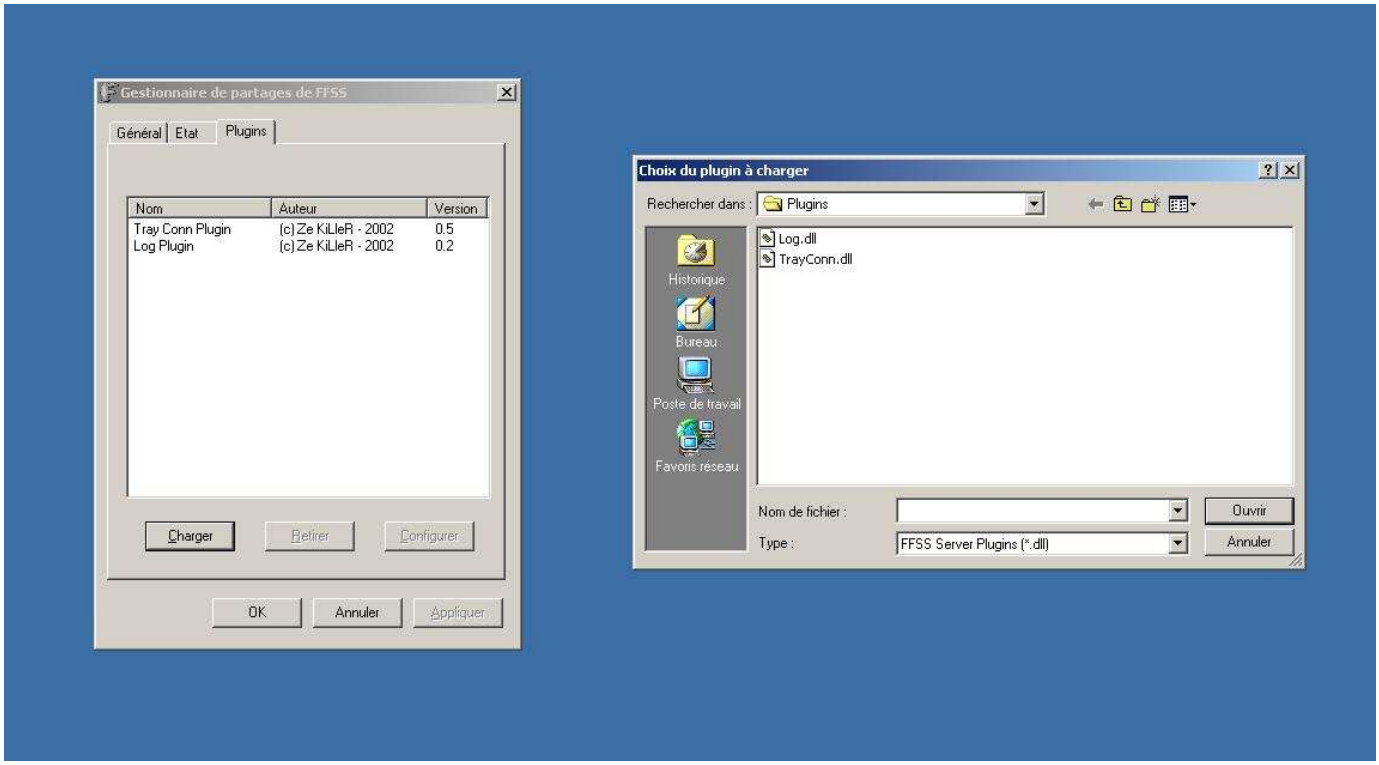


FIG. 4 – La configuration des modules du `serveur` sous `windows`.

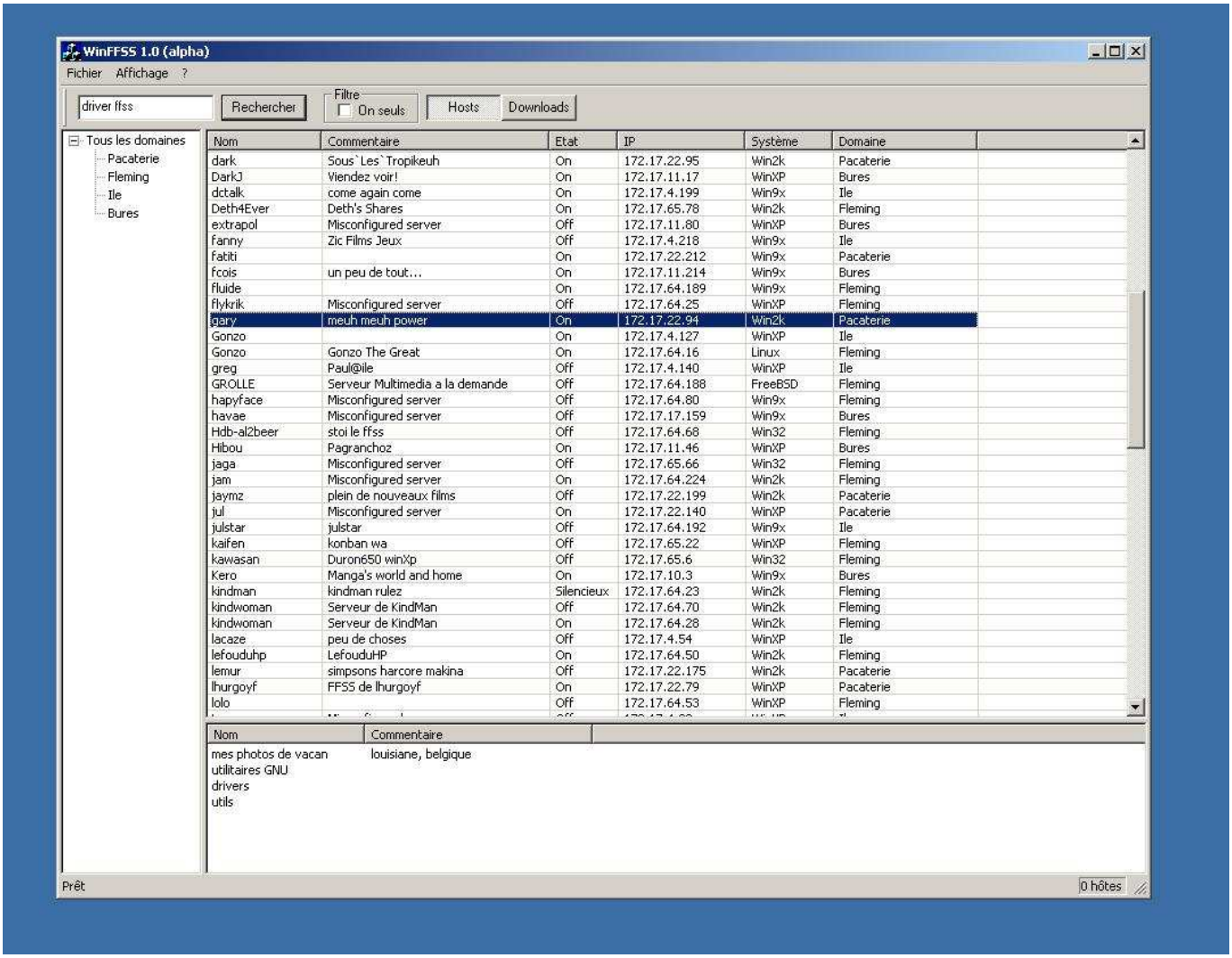


FIG. 5 – Le `client` de recherche, parcours et téléchargement sous `windows`.

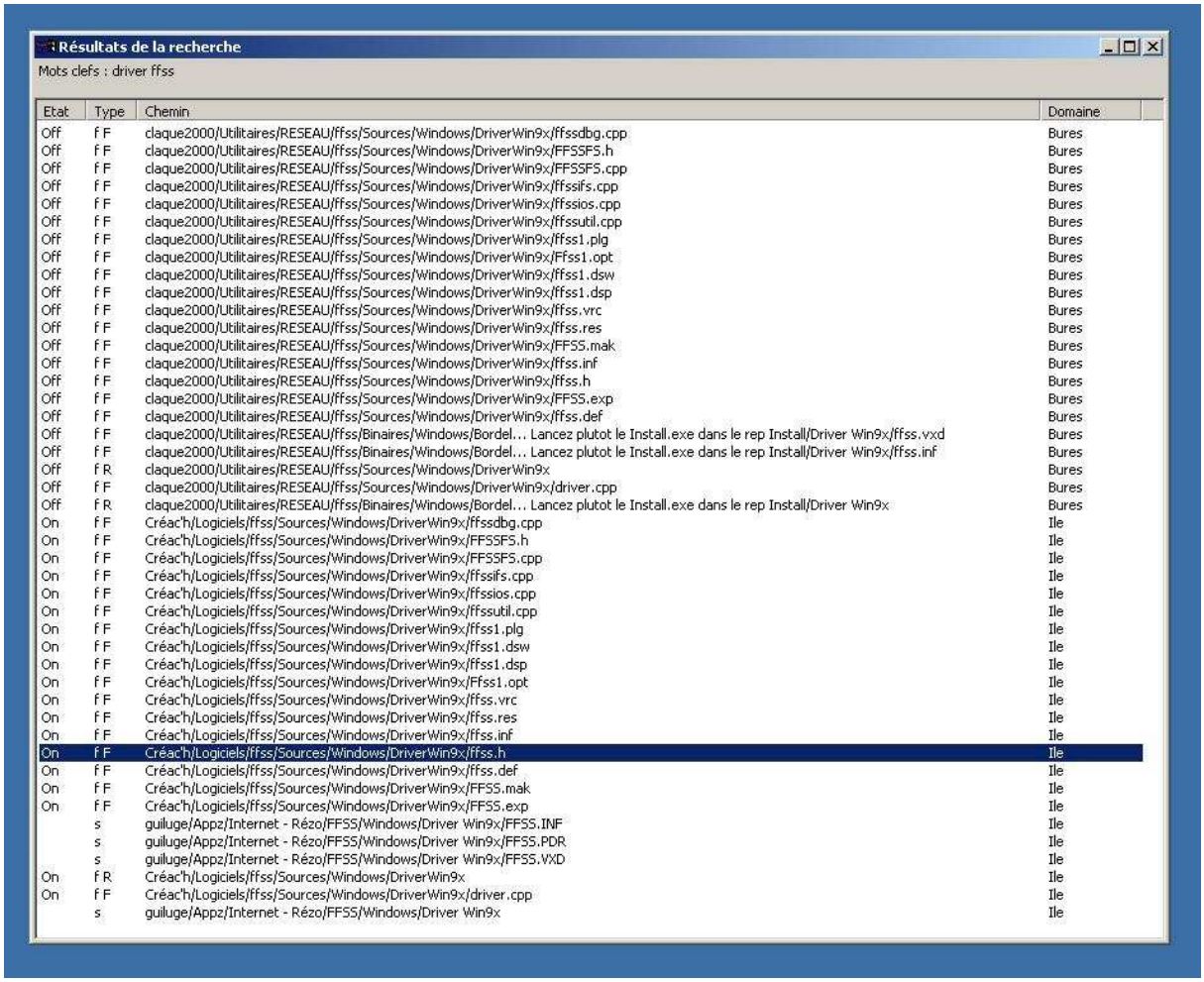


FIG. 6 – Le résultat d'une recherche sous `windows`.

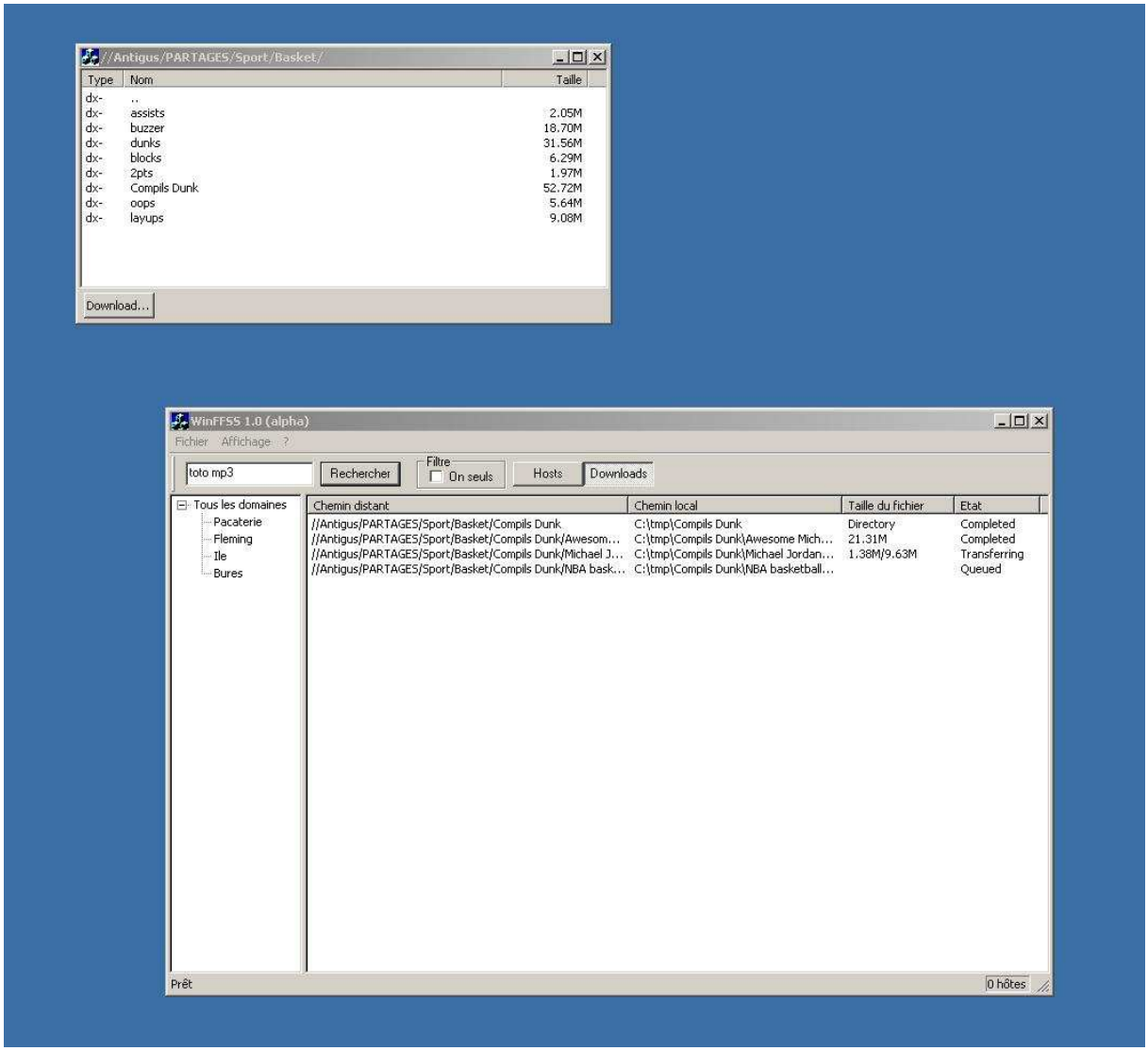


FIG. 7 – Le téléchargement de fichiers sous `windows`.

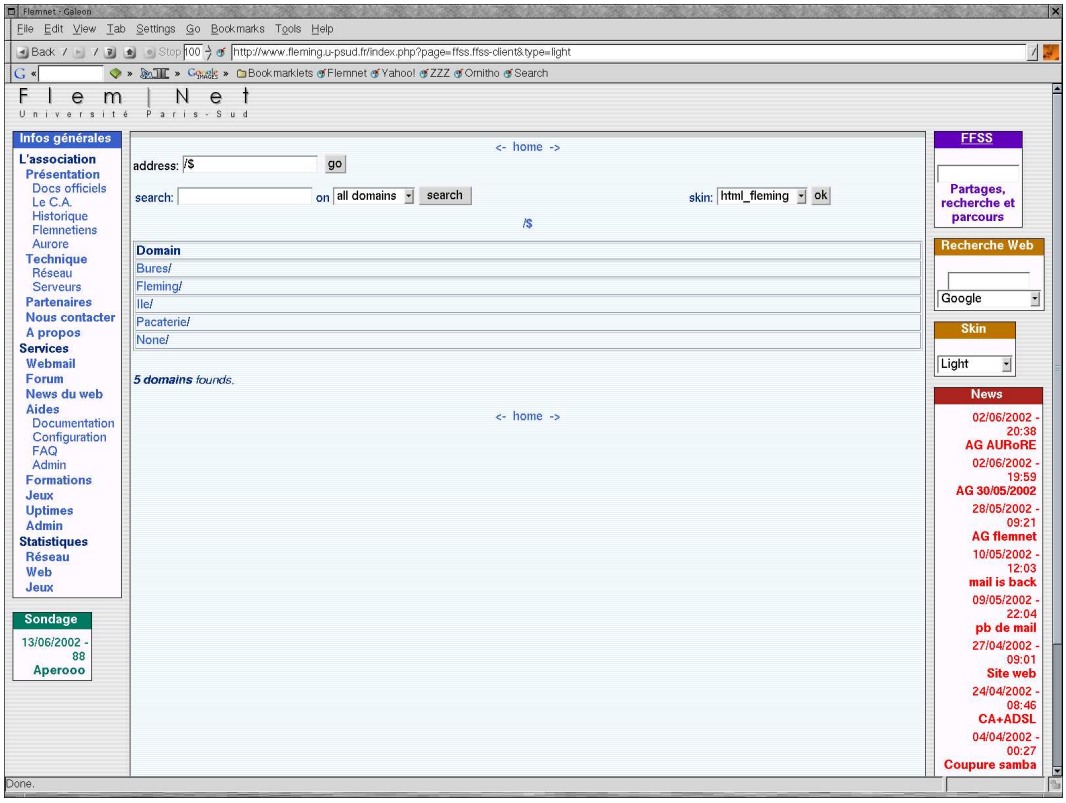


FIG. 8 – Le client `cgi` installé sur le site `http` du réseau de la résidence universitaire "Les Jardins de Fleming".

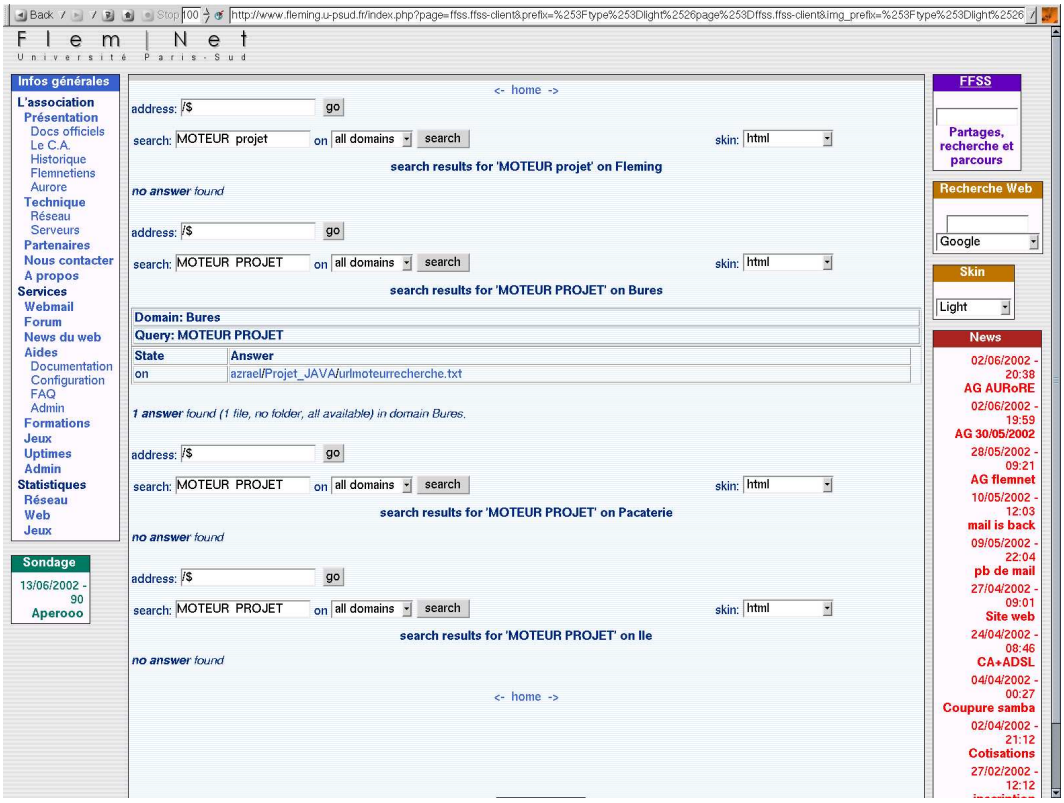


FIG. 9 – Une recherche à l'aide du `client cgi`.

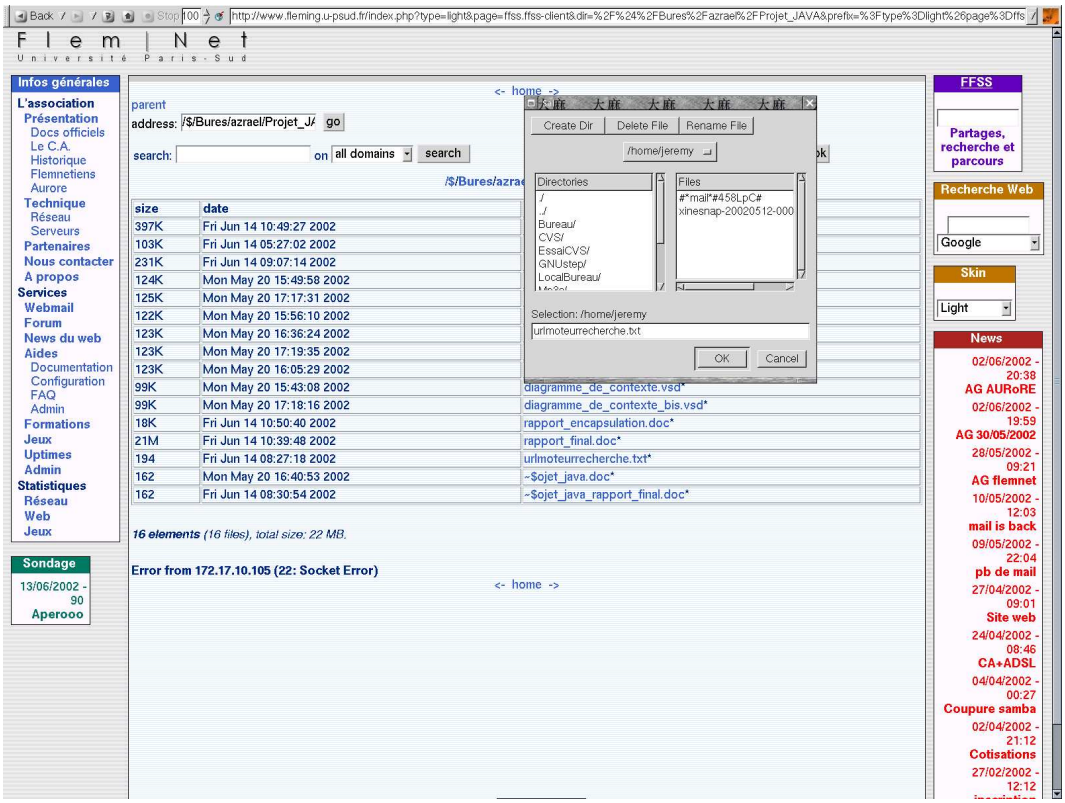


FIG. 10 – Un téléchargement à partir du `client cgi`.

Liste des Algorithmes

1 Énoncé de l'algorithme parcourant les cartes de gauche à droite. . 6
2 `Intersection naïve`$(A_1, \ldots, A_k)$ 25
3 `Recherche doublante`(x, A, p) 27
4 `Recherche dichotomique`(x, A, g, d) 27
5 `Intersection simple`(A, B) 28
6 `Deux-à-deux`$(A_1, \ldots, A_k)$ [DLOM01] 29
7 `Parallèle`$(A_1, \ldots, A_k)$ 34
8 `Séquentiel`$(A_1, \ldots, A_k)$[DLOM01] 36
9 `Alternance`$(A_1, \ldots, A_k)$ 41
10 `Multiplicité`$(t, A_1, \ldots, A_k)$ 60
11 `Incrémental`$(t, A_1, \ldots, A_k, p_1, \ldots, p_k, S, M)$ 64
12 `Multiplicité optimale`$(A_1, \ldots, A_k)$ 65
13 `Interunion`$(l_1, \ldots, l_k, A_1^1, \ldots, A_1^{l_1}, \ldots, A_k^1, \ldots A_k^{l_k})$ 79

Table des figures

1.1 Une instance du problème des pièces cachées : une pièce d'argent parmi 4 pièces de cuivre dissimulées par des cartes. 5
1.2 L'algorithme parcourant les cartes de gauche à droite : en trait pointillé les exécutions possibles, en trait plein l'exécution sur cette instance. La réponse de l'algorithme est positive car la rangée contient une pièce d'argent. 5
1.3 Un arbre de décision pour le problème des pièces cachées, quand il y a 5 pièces. 8
1.4 Décomposition de l'enveloppe convexe de taille 9 d'un ensemble de 16 points. 16
1.5 Décomposition d'une union en blocs 18

3.1 Une instance, un certificat et une partition certificat. Chaque axe représente un tableau, où chaque élément x est noté par un point d'abscisse x. 42
3.2 Une instance selon $\mathcal{D}$. 45
3.3 L'arbre de décision de A restreint aux instances telles que $(p_1, \ldots, p_k)$ soit fixé. 46
3.4 Ordre partiel sur les analyses. Une flèche d'une analyse A vers une analyse B signifie que tout algorithme optimal pour l'analyse A est optimal pour l'analyse B. 53

4.1 Chaque intervalle ne peut intersecter plus de $t-1$ tableaux. Chaque instance est de t-alternance au plus $2\lceil k/(k-t+1)\rceil + 2$. 57
4.2 Si $t \geq k/2$, on considère les instances telles que les $k-t$ plus petits tableaux ne contiennent pas x. 58
4.3 Si $t < k/2$, on considère les instances telles que les $t-1$ plus petits tableaux contiennent x. 58

5.1 Chemins communs dans l'arbre de décision. 73

6.1 Interaction entre un `moteur`, des `serveurs` et des `clients`. . 82
6.2 Interactions entre trois `moteurs` et un `client`. 83

3 La configuration de l'identification et de l'état du `serveur` sous `windows`. 91
4 La configuration des modules du `serveur` sous `windows`. . . . 92
5 Le `client` de recherche, parcours et téléchargement sous `windows`. 93
6 Le résultat d'une recherche sous `windows`. 94
7 Le téléchargement de fichiers sous `windows`. 95
8 Le client `cgi` installé sur le site `http` du réseau de la résidence universitaire "Les Jardins de Fleming". 96
9 Une recherche à l'aide du `client cgi`. 97
10 Un téléchargement à partir du `client cgi`. 98

Bibliographie

[BEY98] BORODIN (A.) et EL-YANIV (R.), *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[BK02] BARBAY (J.) et KENYON (C.), « Adaptive intersection and t-threshold problems », dans *Proceedings of the* 13^{th} *ACM-SIAM Symposium On Discrete Algorithms (SODA)*, p. 390–399. ACM-SIAM, ACM, January 2002.

[BO83] BEN-OR (M.), « Lower bounds for algebraic computation trees », dans *Proc. 15 th ACM STOC*, p. 80 – 86, 1983.

[BR02] BARBAY (J.) et REMPE (L.). « Randomized lower bounds for combinations of decision problems ». Submitted, May 2002.

[Bur58] BURGE (W.), « Sorting, trees, and measure of order », *Information and Contro*, vol. 1, nº 3, 1958, p. 181–197.

[CK80] COOK (C.) et KIM (D.), « Best sorting algorithm for nearly sorted lists », *Communication of ACM*, vol. 23, 1980, p. 620–624.

[Dij82] DIJKSTRA (E. W.), « Smoothsort, an alternative for sorting in situ », *Science of Computer Programming*, vol. 1, nº 3, May 1982, p. 223–233.

[DLOM00] DEMAINE (E. D.), LÓPEZ-ORTIZ (A.) et MUNRO (J. I.), « Adaptive set intersections, unions, and differences », dans *Proceedings of the* 11^{th} *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, p. 743–752, 2000.

[DLOM01] DEMAINE (E. D.), LÓPEZ-ORTIZ (A.) et MUNRO (J. I.), « Experiments on adaptive set intersections for text retrieval systems », dans *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments, Lecture Notes in Computer Science*, p. 5–6, Washington DC, January 2001.

[ECW92] ESTIVILL-CASTRO (V.) et WOOD (D.), « A survey of adaptive sorting algorithms ». ACM Computing Surveys, 1992. 24(4) :441–476.

[GMPR77] GUIBAS (L.), MCCREIGHT (E.), PLASS (M.) et ROBERTS (J.), « A new representation of linear lists », dans *Proc. 9th Annu. ACM Symp. Theory Comput.*, p. 49–60, 1977.

[Knu73] KNUTH (D. E.), *The Art of Computer Programming, Vol 3*, chap. Sorting and Searching, Section 5.3. Addison-Wesley, 1973.

[KS86] KIRKPATRICK (D. G.) et SEIDEL (R.), « The ultimate planar convex hull algorithm ? », *SIAM J. Comput.*, 1986. 15(1) :287–299.

[Man85] MANNILA (H.), « Measures of presortedness and optimal sorting algorithms. », dans *IEEE Trans. Comput.*, vol. 34, p. 318–325, 1985.

[Meh79] MEHLHORN (K.), « Sorting presorted files », dans SPRINGER, éditeur, *Proceedings of the 4th GI-Conference on Theoretical Computer Science*, vol. 67 (coll. *Lecture Notes in Computer Science*), p. 199–212, 1979.

[Meh84] MEHLHORN (K.), *Data Structures and Algorithms 1 : Sorting and Searching*, chap. 4.2 Nearly Optimal Binary Search Tree, p. 184–185. Springer-Verlag, 1984.

[M.L75] M.L. (F.), « Two applications of a probabilistic search technique : Sorting x+y and building balanced search trees », dans 7*th STOC*, p. 240–244, 1975.

[PM95] PETERSSON (O.) et MOFFAT (A.), « A framework for adaptive sorting », *Discrete Applied Mathematics*, vol. 59, 1995, p. 153–179.

[Sch81] SCHWARTZ (L.), *Cours D'analyse*. Hermann, Hermann 293 rue Lecourbe, 75015 Paris, 1981.

[Yao77] YAO (A. C.), « Probabilistic computations : Toward a unified measure of complexity », dans *Proc. 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, p. 222–227, 1977.

Index

algorithme, 4
algorithme probabiliste, 8
alternance, 40, 56, 62, 77
arbre de comparaison, 7
arbre de décision, 7

certificat, 30
citation d'Alice, 4
client, 82
coût, 31
complexité, 5, 69, 70

dispersion, 31

ensemble de multiplicité, 56, 62
enveloppe convexe, 16
exemple, 4–8, 14, 15, 17, 18, 20, 24, 40, 56, 62

gestionnaire, 82

index, 21
intersection, 23, 24, 40, 68
inversions, 17

mesure de désordre, 17
mesure de difficulté, 15
moteur de recherche, 82
mots-clés, 21
multiplicité, 56, 62

partition certificat, 40, 56
problème de recherche, 77
problème des pièces cachées, 4–8, 14, 15, 20, 88

questionnaire, 7

requêtes conjonctives, 21

serveur, 82
signature, 18, 23, 68, 76

taille, 18, 23

union, 18

Analyse fine : bornes inférieures et algorithmes de calculs d'intersection pour moteurs de recherche

La résolution de requêtes conjonctives dans les moteurs de recherche indexés met en œuvre l'intersection de tableaux triés. L'analyse classique dans le pire des cas ne permet pas de distinguer les algorithmes résolvant ce problème. Faisant suite aux travaux de Demaine, Lopez-Ortiz et Munro [DLOM00] nous proposons une analyse plus fine [BK02] de la complexité probabiliste de ce problème : idéalement un algorithme efficace devrait résoudre des instances "faciles" plus rapidement. Cette analyse est basée sur la complexité non-déterministe du problème et permet de montrer l'optimalité de l'algorithme de Demaine, Lopez-Ortiz et Munro [DLOM01].

Nous généralisons l'intersection à l'*ensemble de multiplicité t* et à l'*ensemble de multiplicité optimale*, et montrons des résultats similaires à ceux obtenus sur l'intersection. Nous étudions la généralisation de ces travaux à d'autres requêtes, sous la forme de *combinaisons de problèmes de décision* [BR02]. Cela nous permet de généraliser les résultats précédents au calcul de l'*InterUnion* (intersection d'unions de tableaux triés). Ces problèmes de calcul apparaissent dans le moteur de recherche intégré à FFSS, un système de partage de fichiers visant à remplacer le protocole de partage de fichier SMB, en cours de développement. Nous décrivons le concept de moteur de recherche intégré, les spécifications du protocole, et l'implémentation de divers aspects du moteur de recherche.
Mots-clés : algorithme adaptatif, analyse fine, intersection.

Fine Analysis : Lower Bounds and Algorithms of Intersection Computation for Search Engines

Indexed search engines solving conjunctive queries need to compute the intersection of sorted sets. The usual worst case analysis is too crude to compare the algorithms for this problem. Ideally, an efficient algorithm should solve "easy" instances more quickly. This problem was already studied by Demaine, Lopez-Ortiz and Munro [DLOM00]. We define a model of finer analysis [BK02] with a new measure of "difficulty" based on the non-deterministic complexity of the problem. We prove the optimality of the algorithm of Demaine, Lopez-Ortiz and Munro [DLOM01] in this model.

We define the *t-threshold set* problem, a generalization of the problems of finding the intersection or the union of sorted sets, and we prove similar results on this problem. We define the *opt-threshold set*, the minimal non-empty threshold set, and we give an almost optimal algorithm to compute this set. We study further generalization to *combinations of decision problems* [BR02]. This generalizes previous results to the computation of the *interunion* (computing the intersection of a union of sorted sets), a problem motivated by search engines indexed on *factors* of keywords. As an application, we present the search engnine integrated to FFSS, a file sharing system.
Keywords : adaptive algorithm, fine analysis, intersection.