Rapport de stage de recherche de M1

Analyse d'algorithmes de pattern-matching avec prise en compte d'éléments d'architecture des ordinateurs

Victor Veillerette

Encadré par :
M. Cyril Nicaud
Mme Carine Pivoteau

Table des matières

Remerciements

Je tiens à remercier personnellement Monsieur Cyril Nicaud et Madame Carine Pivoteau pour, en premier lieu, m'avoir proposé ce stage de recherche sous leurs tutelles mais aussi pour m'avoir suivi avec attention pendant toute la durée de celui-ci.

Je les remercie également de m'avoir fait confiance en m'accordant une grande autonomie de travail et en m'aiguillant vers des pistes intéressantes quand cela s'avérait nécessaire.

Enfin, je les remercie de leurs aides pour la relecture de ce rapport ainsi que pour leurs commentaires sur ma présentation de soutenance.

Chapitre 1

Présentation du lieu d'accueil

1.1 L'université

Jeune université créée en 1991, l'**Université Marne-La-Vallée** est une université publique française pluridisciplinaire. Elle est située dans la commune de Champs-sur-Marne en Seine et Marne (77), principalement dans le campus Descartes.

1.2 Le laboratoire de recherche

Le Laboratoire d'Informatique Gaspard Monge (LIGM) est une Unité Mixte de Recherche spécialisée en informatique. Il existe depuis 1992 et a été créé par Maxime Crochemore. Bénéficiant du statut d'UMR CNRS depuis 2002, ce laboratoire est reconnu pour sa qualité de recherche.

Les principales activités du laboratoire sont bien définies et se concentrent sur l'informatique théorique – combinatoire, algorithmique, automates, géométrie et bio-informatique – sans oublier les domaines plus appliqués : images, traitement automatique des langues, logiciels avec notamment l'environnement Java et enfin les systèmes temps-réels et réseaux.

Le laboratoire possède quatre tutelles : le CNRS, l'ENPC, l'Ecole Supérieure d'Ingénieurs (ESIPE) ainsi que l'Université Marne-la-Vallée (UPEM) et est formé de cinq équipes, regroupant au total près de **150 personnes** incluant 80 chercheurs permanents, ce qui en fait, à l'échelle nationale, un laboratoire de recherche de taille *moyenne*.

Effectifs au LIGM	2013	2014	2015	2016	2017	2018
Chercheurs	16	17	18	19	21	20
Enseignants-Chercheurs	54	55	55	58	60	63
BIATS	4	5	6	5	5	6
Total	**74**	**77**	**79**	**82**	**86**	**89**

FIGURE 1.1 – Evolution des effectifs de recherche au LIGM

La laboratoire est structuré autour de *cinq équipes* de recherche :

— *Algorithmes, architectures, analyse et synthèse d'images* étudie les achitectures dédiées à l'imagerie et les mises en place en pratique ainsi qu'une partie théorique géométrique et mathématiques (28 chercheurs)

— *Combinatoire algébrique et calcul symbolique* étudie aussi bien les algèbres de Hopf et la combinatoire énumérative que les probabilités libres. Elle contribue au projet SAGE.

— *Logiciels, réseaux et temps réel* met en pratique des recherches en systèmes embarqués, objets connectés ainsi que dans la machine virtuelle Java.

— **Modèles et algorithmes** étudie notamment la bio-informatique, **l'algorithmique du texte**, les bases de données théoriques, **les automates** et logiques ainsi que **l'analyse en moyenne**.

— *Signal et communication* étudie les systèmes de transmissions de l'information et les problèmes qui en découlent.

1.3 Domaine de recherche

Le domaine de recherche de ce stage est assez varié, – comme vous pourrez le constater dans la suite de ce rapport –, il consiste tout d'abord en l'analyse et l'implémentation de plusieurs algorithmes classiques du Pattern-Matching, c'est-à-dire la recherche exacte d'un motif dans un texte ; en la mesure de divers statistiques les concernant, en la création de chaînes de MARKOV et enfin en analyse et compréhension de divers éléments des processeurs modernes comme les prédicteurs de branchement, les processeurs superscalaires ou encore les pipelines.

Ces thèmes ne sont généralement que *peu étudiés* et seuls deux chercheurs dans ce laboratoire, Cyril Nicaud et Carine Pivoteau, participent à la publication d'articles dans ce domaine.

1.4 Contexte et Sujet

Mes premiers contacts avec Monsieur Cyril Nicaud à propos d'un stage ont eu lieu dès septembre 2017 quand celui-ci m'a parlé d'un stage dans le domaine de l'architecture des ordinateurs, domaine alors peu étudié au laboratoire. L'objectif initial était d'étudier des algorithmes classiques en ayant à l'esprit que certains éléments hardware pouvait les influencer.

Plus tard, lors de la réunion de présentation du stage en octobre 2017, on me présente l'article *Good predictions are worth a few comparisons* [3] co-écrit avec Carine Pivoteau et Nicolas Auger à propos de l'amélioration notable que l'on peut apporter à des algorithmes standards en prenant en compte des éléments d'architecture, notamment la prédiction de branchements.

Dans cet article, on nous présente une version modifiée de l'algorithme classique de la recherche dichotomique : au lieu de diviser notre ensemble en deux portions égales, on choisit de le diviser en deux portions de respectivement $\frac{1}{3}$ et $\frac{2}{3}$. Evidemment, cela entraine une hausse du nombre de comparaisons à effectuer pour trouver l'élement que l'on cherche, mais cependant, en pratique, on observe une diminution du temps d'exécution :

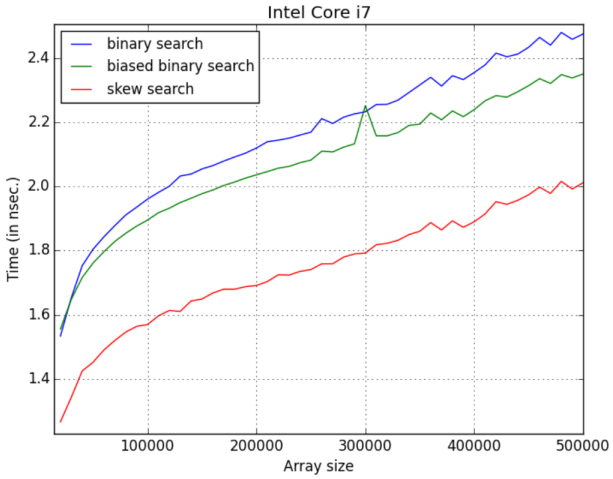


FIGURE 1.2 – L'algorithme *biased binary search* apparait être plus rapide que l'algorithme *binary search*. L'article explique alors que la raison est à trouver du côté des prédicteurs de branchements qui s'adaptent mieux au premier.

L'idée était alors de sélectionner des algorithmes de Pattern-Matching pour les étudier et de tenter de trouver de petites – *ou grandes* – modifications à leur apporter.

1.5 Organisation de travail

Le stage s'est déroulé au sein même du laboratoire de fin avril 2018 à août 2018 pour une durée totale de 3 mois sur la base de semaines de 35 heures.

J'ai disposé d'un bureau où je pouvais travailler avec ma machine personnelle et où un écran et des petites machines de test ont été installées à mon attention.

Mon travail s'est déroulé principalement en autonomie avec des phases de recherche sur les algorithmes ou des concepts et des phases purement techniques d'implémentation. Une réunion par semaine a été organisée avec mes tuteurs afin que je puisse présenter mon travail, échanger des idées et que mes tuteurs puissent m'aiguiller correctement sur la façon dont il fallait continuer et m'indiquer les pistes à suivre.

Chapitre 2

Découverte des algorithmes de Pattern-Matching

Le domaine du Pattern-Matching consiste, de manière générale, à trouver toutes les occurences d'un motif dans un ensemble séquentiels d'éléments du même type. En règle générale, on cherche à connaître le nombre d'occurrences ou les positions exactes de l'élément.
Dans le cadre de la recherche exacte de chaînes de caractères, on s'intéresse à trouver les occurrences d'une chaîne dans un texte, tous deux composés de caractères issus d'un alphabet : on parle de recherche de sous-chaînes.

Dans la suite du rapport, on notera :

— l'élément à rechercher : ***Motif*** ou $\boldsymbol{M}$, de taille m.
— le texte : ***Texte*** ou $\boldsymbol{T}$, de taille n.
— l'alphabet du texte : $\boldsymbol{A}$, de taille $|A|$.

2.1 Enjeux et applications

La recherche de sous-chaînes est un domaine de l'algorithmique qui a commencé à être étudié dès l'émergence de l'informatique. Aujourd'hui, elle intervient partout : la recherche d'un mot sur une page web (CTRL-F), la recherche de séquences ADN, les algorithmes de Deep-Learning ou encore le filtrage par motif en Caml.
De plus, elle s'applique à de grosses données d'entrée, particulièrement de gros textes de plusieurs milliards de caractères. Les premiers algorithmes naïfs ne suffisant rapidemment plus, de nouveaux algorithmes plus poussés ont vu le jour dès le milieu des années 70, notamment avec KMP et Boyer-Moore et encore aujourd'hui, de nouvelles méthodes voient le jour.

2.2 Mise en oeuvre

Les algorithmes sont généralement divisés en deux parties :

— une phase de *prétraitement* du motif ou du texte

— une phase de *recherche* du motif dans le texte.

Tout l'objectif ici est d'arriver à effectuer un prétraitement, si possible sur le motif plutôt que sur le texte, peu lourd et qui améliore fortement la vitesse d'exécution de la phase de recherche.

Il existe trois types d'algorithmes :

— basés sur *la comparaisons des caractères*, le plus utilisé historiquement.

— basés sur *un ou plusieurs automates*, où la phase de pré-traitement est généralement lourde, mais la recherche très rapide.

— basés sur *la parrallélisation de bits*, technique moderne qui repose plus sur le hardware que sur l'algorithmique.

On va s'intéresser ici uniquement à des algorithmes de références du pattern-matching, c'est-à-dire l'algorithme NAÏF, KMP ainsi que HORSPOOL et qui n'utilisent exclusivement que la comparaison de caractères.

2.3 L'algorithme bruteforce ou naïf

L'algorithme NAÏF, ou BRUTEFORCE [5] est l'algorithme de base puisqu'il ne possède pas de phase de pré-traitement et que sa phase de recherche consiste à tester toutes les possibilités. Cet algorithme est intéressant à étudier car d'un point de vu algorithmique, c'est le plus mauvais, donc il peut servir de valeur étalon pour évaluer la performance d'un autre algorithme.

2.3.1 Phase de recherche

La phase de recherche consiste à parcourir le texte de gauche vers la droite et de tester, pour chaque position, si le motif correspond.

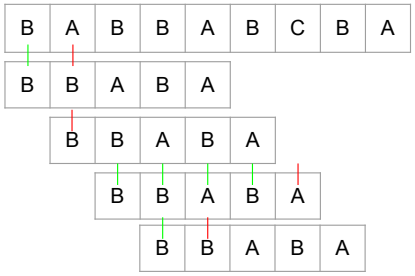


FIGURE 2.1 – **Exemple :** Recherche du motif BBABA

2.3.2 Complexités

— Dans le pire cas, l'algorithme doit, pour chaque indice de départ i potentiel, vérifier l'intégralité du motif. Pour une position donnée i, on peut avoir à vérifier tous les caractères du motif – *si il y a match à chaque fois* –, c'est-à-dire m comparaisons. Pour tout le texte on est donc en $\mathcal{O}(n \cdot m)$ dans le pire cas.

— En moyenne, on doit toujours vérifier chaque position i de départ du motif potentiel, mais en considérant un alphabet de taille $|A|$, on a une probabilité $\frac{|A|-1}{|A|}$ d'avoir un mis-match et donc, de ne pas vérifier la suite des lettres du motif. On est donc en $\mathcal{O}\left(n \cdot \frac{|A|}{|A|-1}\right)$. On observe que le nombre de comparaisons est fortement décroissant quand la taille de l'alphabet augmente et que donc l'algorithme tend à être linéaire sur de grands alphabets.

$$\lim_{|A| \to +\infty} \left(n \cdot \frac{|A|}{|A| - 1} \right) \to n$$

2.3.3 Implémentation

Algorithm 1 naive search.

```
1:  function NAIVE
2:      for i from 0 to n do                    ▷ Test all initial indices in the text
3:          for j from 0 to m do                ▷ Verify all pattern for a given i
4:              if pattern[j] ≠ text[i + j] then
5:                  break
6:              end if
7:          end for
8:          if j == m then
9:              find(i)                         ▷ Pattern is found at i
10:         end if
11:     end for
12: end function
```

2.4 Introduction à KMP

2.4.1 Un bord

Un bord est une *chaîne-portion* d'une autre chaîne de caractères qui est à la fois préfixe et suffixe de cette dernière. Autrement dit, un bord de taille k d'une chaîne S de taille n vérifie :

$$S[0 \dots k-1] = S[n-k \dots n-1]$$

Par convention, on définit que chaque chaîne possède au moins un bord de taille nulle : ϵ.

2.4.2 Une table de saut

A partir d'un indice i, une table de saut est un tableau d'indice qui permet d'accéder à sa prochaine valeur par un accès direct en $\mathcal{O}(1)$. En pratique, on l'utilise pour sauter certaines itérations d'indices dont on sait qu'elles ne seront pas utiles, et plus précisément dans le domaine du pattern-matching, dont on sait qu'elles ne pourront pas donner de concordance du motif.
Par exemple pour incrémenter un indice de deux en deux modulo 7 :

$table =$

i	0	1	2	3	4	5	6
$T[i]$	2	3	4	5	6	0	1

On fera la mise à jour avec $i = table[i]$. Utiliser une telle table permet, par exemple, de supprimer des sauts conditionnels – *comme un* IF – mais nécessite une plus grande quantité d'espace mémoire.

2.5 L'algorithme Knuth-Morris-Pratt

L'algorithme KNUTH–MORRIS–PRATT [8] [4], du nom de ses inventeurs, est un algorithme basé sur la comparaison de caractères créé en 1977. Il est le premier du genre à apporter une **réelle augmentation** de la performance – avec une **complexité linéaire** – dans le domaine du pattern-matching. Il procède en deux étapes :

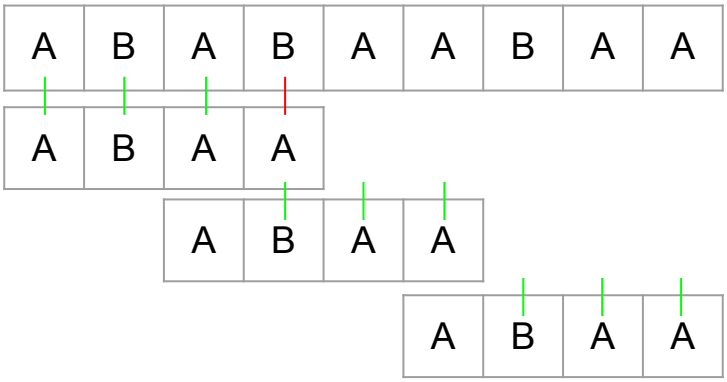
— la phase de pré-traitement du motif qui consiste en la création d'une table de saut.

— la phase de recherche qui utilise cette table pour sauter certaines décalages.

2.5.1 Phase de recherche

L'algorithme compare les caractères de gauche à droite et en cas de mismatch ou de détection du motif, recherche le plus grand bord non suivi de la lettre dont la comparaison a échoué pour continuer la recherche.

Exemple : Recherche du motif ABAA :



— ÉTAPE 1 : On commence à comparer les lettres de gauche à droite jusqu'à tomber sur un mismatch.

— ÉTAPE 2 : Vu qu'il y a mismatch en position 4 avec la lettre A, on recherche dans notre motif le bord le plus grand non suivi d'un A. On trouve le bord de taille 1. On décale donc notre motif en conséquence.

— ÉTAPE 3 : On recommence à comparer les lettres à partir de la position 1 du motif. On trouve le motif.

— ÉTAPE 4 : De la même manière qu'à l'étape 2, on recherche le plus grand bord non suivi d'un A. On replace notre motif et on continue.

2.5.2 Table de KMP

Pour permettre une recherche efficace, on doit pouvoir trouver le bord qui nous intéresse à partir d'une position dans le motif et ce, en temps constant.

Pour cela, on va utiliser une table de saut avec comme définition que $T[i]$ = le plus grand bord du motif non suivi de la lettre $pattern[i]$.
Pour simplifier le résultat, on note -1 si on ne trouve aucun bord et la taille de celui-ci dans le cas contraire.

-1	0	-1	1	1

FIGURE 2.2 – **Exemple :** table des bords de KMP pour le motif ABAA

2.5.3 Implémentation de la table des bords

Algorithm 2 KMP table.

```
1: function FILL_KMP_TABLE
2:     i = 0
3:     j = −1
4:     T[0] = −1
5:     while i < m do
6:         while j < −1 and text[i] ≠ text[j] do
7:             j = T[j]
8:         end while
9:         i++
10:        j++
11:        if text[i] == text[j] then
12:            T[i] = T[j]
13:        else
14:            T[i] = j
15:        end if
16:    end while
17: end function
```

2.5.4 Implémentation de la phase de recherche

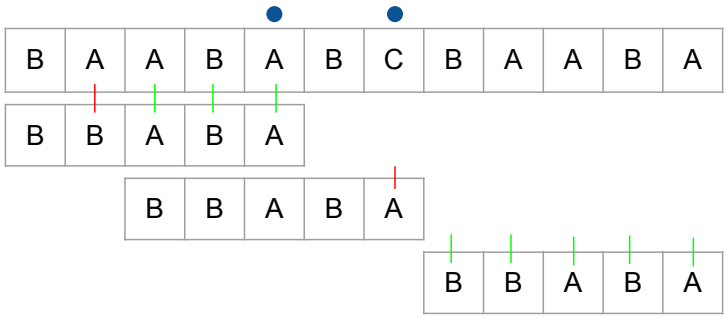
Algorithm 3 KMP search.

```
1:  function KMP
2:      s := i := j := 0
3:      fill_kmp_table(kmp_table)                    ▷ fill the jump table from pattern
4:      while j < n do
5:          while i > −1 and text[j] ≠ pattern[i] do   ▷ Test for all borders of the pattern
6:              i = kmp_table[i]
7:          end while
8:          i++
9:          j++
10:         if i == m then                            ▷ pattern is found at i − j
11:             find(i − j)
12:             s++
13:             i = kmp_table[i]
14:         end if
15:     end while
16: end function
```

2.6 L'algorithme d'Horspool

L'algorithme d'Horspool [7], du nom de son inventeur, est un algorithme à comparaison de caractères très simple et très rapide dans la grande majorité des cas. Il consiste à comparer les caractères du motif de **droite vers la gauche** et en cas d'échec, de décaler le motif pour l'aligner avec la première lettre du texte que l'on a comparée. Il permet, la plupart du temps, de **ne pas tester tous les caractères** du texte.

Exemple : Recherche du motif BBABA :



— ÉTAPE 1 : On place le motif au début et on commence à comparer de droite à gauche jusqu'à trouver le motif, ou comme ici un mismatch à la position 2.

— ÉTAPE 2 : On décale alors le motif en alignant le A du texte à la position 5 avec le premier A du motif à partir de la droite, autre que le dernier, que l'on peut trouver. On recommence à comparer de droite à gauche et on tombe sur un mismatch dès la première comparaison.

— ÉTAPE 3 : On cherche un C à aligner dans le motif, il n'y en a pas. On décale le motif pour qu'il commence alors à la position 8. Enfin on a trouvé le motif en effectuant **10 comparaisons**.

2.6.1 Utiliser une table de saut

Pour rendre l'algorithme très efficace, il convient d'utiliser une table de saut pour trouver tout de suite comment décaler le motif dans la phase de recherche. Elle consiste simplement à calculer, pour chaque lettre de notre alphabet, la distance entre la première lettre trouvable et la fin du motif, tout en écartant la dernière lettre du motif. Si on ne peut pas la trouver, on lui assigne la taille du motif, vu que l'on veut alors décaler le motif entièrement.

Algorithm 4 HORSPOOL table.

```
1: function FILL_HORSPOOL_TABLE
2:     i = 0
3:     while i < 256 do
4:         T[i] = len(pattern)
5:         i++
6:     end while
7:     i = 0
8:     while i < len(pattern) − 1 do
9:         T[pattern[i]] = len(pattern) − 1 − i
10:        i++
11:    end while
12: end function
```

Lettre	Valeur
A	2
B	1
...	5

FIGURE 2.3 – **Exemple :** Pour le motif BBABA on obtient alors la table ci-dessus

2.6.2 Implémentation de la phase de recherche

L'implémentation se fait assez facilement en gardant deux indices : i pour la position dans le motif et sk pour la position dans le texte.

Algorithm 5 HORSPOOL search.

```
1: function HORSPOOL
2:     sk := s := 0
3:     fill_horspool_table(hor_table)          ▷ fill the jump table from the pattern
4:     while n − m ≥ sk do
5:         i = m − 1
6:         while text[sk + i] == pattern[i] do
7:             if i == 0 then
8:                 s++
9:             end if
10:            i−−
11:        end while
12:        sk+ = hor_table[texte[sk + m − 1]]
13:    end while
14: end function
```

2.7 Résultats expérimentaux

Chaque algorithme est testé à plusieurs reprises grâce à un programme codé en C et ce, en faisant varier la taille de l'alphabet $|A|$. Un texte est généré aléatoirement de manière équiprobable à partir de l'alphabet et une centaine de motifs *de tailles différentes* sont recherchés dans le texte. Le graphique 2.4 nous montre le nombre de comparaisons en fonction de la taille de l'alphabet. On peut observer que pour KMP et NAÏF, le nombre de comparaisons tendent vers n (ici 100000).

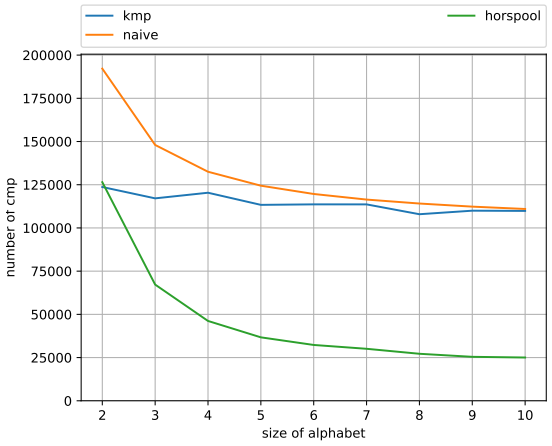


FIGURE 2.4 – Nombre de comparaisons des différents algorithmes présentés en fonction de la taille de l'alphabet du texte. La recherche est testée avec des motifs de différentes tailles et générés aléatoirement sur un texte de taille 100000.

Sur le graphique des temps d'exécution 2.5, cette fois, on obtient des **résultats surprenants**. En effet, on peut observer qu'en moyenne, l'algorithme NAÏF parait plus performant que l'algorithme KMP. Cela montre une réelle différence notable entre la complexité théorique, le nombre de comparaisons et le temps effectif d'exécution. On tentera d'élucider ce résultat étonnant dans le chapitre 5.

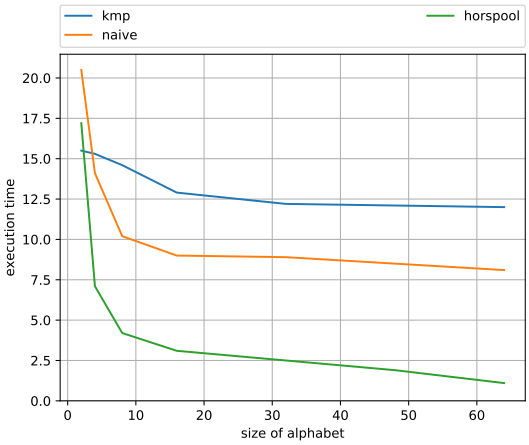


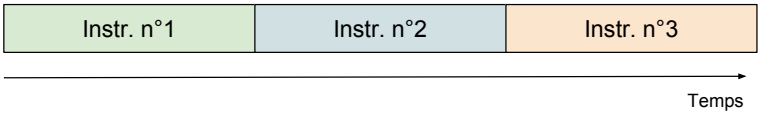
FIGURE 2.5 – Temps d'exécution des différents algorithmes en fonction de la taille de l'alphabet d'un texte de taille 100000.

Chapitre 3

Approche pratique : Prédicteurs de branchements

3.1 Contexte

Les processeurs modernes ont vu leurs performances fortement augmenter au fil des années et ce, grâce à de nombreuses améliorations : que ce soit simplement l'augmentation de leur cadence de calcul, ou alors une meilleure architecture qui permet d'améliorer la vitesse d'exécution.
Dans un processeur classique, celui-ci reçoit des instructions basiques qui doivent être exécutées pour qu'un résultat déterministe soit produit en retour. Une manière très humaine de faire cela en pratique serait simplement de traiter chaque instruction les unes à la suite des autres :



Cela fonctionne, cependant c'est très lent et cela n'est pas du tout adapté aux capacités d'un processeur. En effet, si deux instructions *sans rapport* et *faisant deux choses différentes* se suivent, – comme un accès mémoire suivi d'un calcul simple – pourquoi ne pourrait-on pas les exécuter en même temps ?

C'est pour répondre à cette question que les premiers PIPELINES sont apparus dans les années 60. L'idée du PIPELINE est de paralléliser les instructions « *du mieux qu'on peut* » pour accélérer l'exécution d'un programme. Cela est possible en divisant chaque instruction en micro-instructions qui ont chacune un rôle différent à jouer : accès à la mémoire, écriture dans un registre, calcul mathématique, etc :

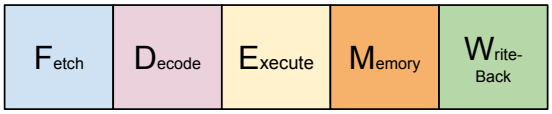


FIGURE 3.1 – Division d'une instruction en 5 micro-instructions (modèle RISC)

Si on suppose, par exemple, que notre processeur est capable d'exécuter 5 micro-instructions différentes à chaque unité de temps, on pourrait alors établir notre pipeline de la sorte :

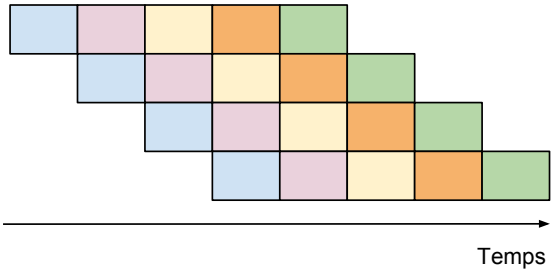


FIGURE 3.2 – Parallélisation des instructions dans un pipeline

Problème : Pour établir un tel pipeline, il faut *connaître* les instructions qui suivent et ce n'est pas toujours vrai en cas d'instructions à saut conditionnel. En effet, le résultat d'une instruction conditionnelle n'est connu qu'à la fin de son exécution et il faudrait donc retarder toute la suite du programme :

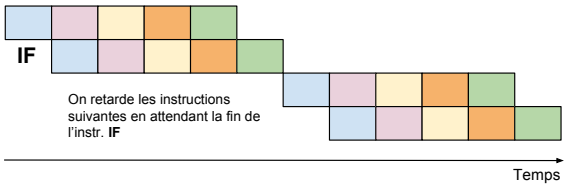


FIGURE 3.3 – Instructions décalées à cause d'un IF. Dans cet exemple, on perd 4 unités de temps.

C'est pour cela qu'on utilise aujourd'hui, dans les processeurs modernes, des **prédicteurs de branchement** qui vont se charger, lorsqu'une instruction conditionnelle se présente, de **prédire** quelles seront les instructions suivantes pour remplir correctement le pipeline.

3.2 Les prédicteurs de branchement

Un PRÉDICTEUR DE BRANCHEMENT est une composante hardware intégrée dans les pipelines des processeurs. Elle consiste en un circuit, plus ou moins complexe, qui va déterminer, lorsqu'une instruction conditionnelle entre dans le pipeline et avant même qu'elle ne commence à s'exécuter, si celle-ci sera prise (BRANCH TAKEN) ou non (BRANCH NOT TAKEN).

L'objectif du prédicteur est de fournir une prédiction la plus fiable possible, mais aucun modèle n'est parfait et il peut se tromper ; dans ce cas, on parle de MISPRÉDICTION.

Une misprédiction peut coûter cher en temps (jusqu'à 18 cycles [1] sur les Intel Core i7) et en énergie ; c'est pourquoi il est intéressant d'étudier leurs impacts sur les algorithmes et quelques différents modèles simples existants.

3.3 Prédicteurs simples

La majorité des prédicteurs utilisent les résultats précédents d'un IF pour prédire son prochain saut. On utilise donc principalement des graphes et/ou des systèmes de table de cache pour se souvenir des informations associées à un IF.

3.3.1 Prédicteur saturé à $2k$ états

Ce prédicteur consiste en un graphe à $2k$ états, avec $k > 0$; chaque état étant associé la prochaine prédiction : TAKEN ou NOT TAKEN. A chaque IF complété, on se déplace dans le graphe. L'état de départ n'a pas de réelle importance, mais par convention, on se place sur l'état NOT TAKEN $(k-1)$ par défaut :

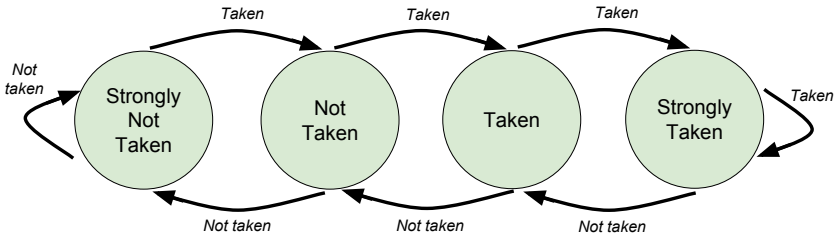


FIGURE 3.4 – Exemple avec un prédicteur saturé à 4 états

3.3.2 Prédicteur Two-Level à 2^k entrées

Ce prédicteur est une table qui permet d'associer un prédicteur saturé aux k derniers résultats de branchements représentés sous forme binaire. On accède

1. Un cycle est l'unité atomique de temps sur les processeurs. Dans un modèle théorique, on peut considérer que c'est le temps d'exécution d'une micro-instruction.

à une entrée de la table en donnant le nombre représentant les n derniers branchements et on utilise le prédicteur contenu dans l'entrée pour obtenir la prédiction.

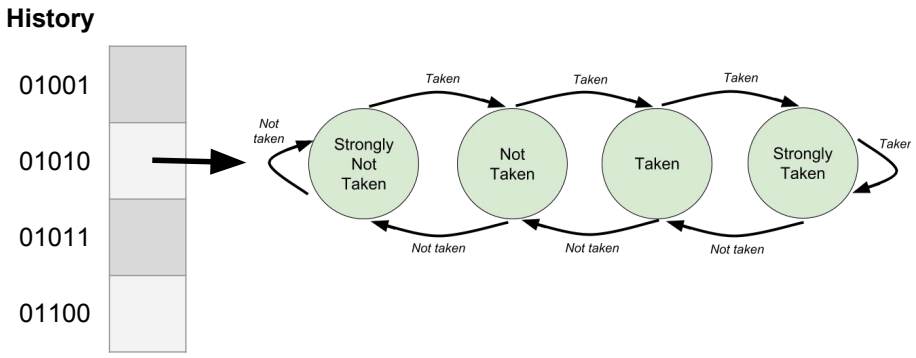


FIGURE 3.5 – Extrait de la table du prédicteur Two-Level avec $k = 5$

L'historique des résultats est un nombre binaire de k bits où 0 représente NOT TAKEN et où 1 équivaut à TAKEN. Ce prédicteur permet d'améliorer la prédiction quand un schéma court et répétitif est en place. Par exemple, si une branche alterne entre prise et non prise.

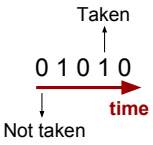
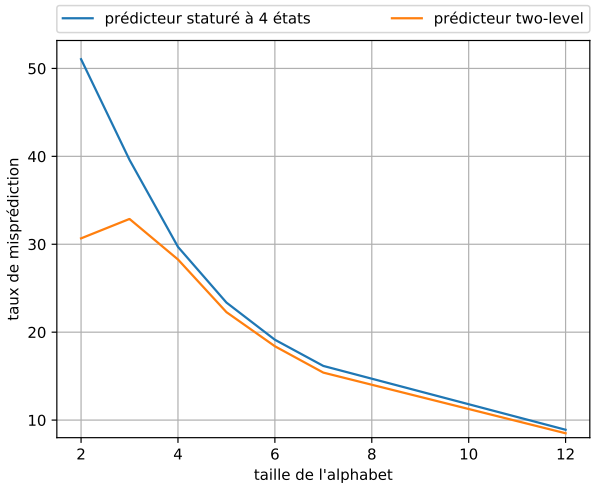


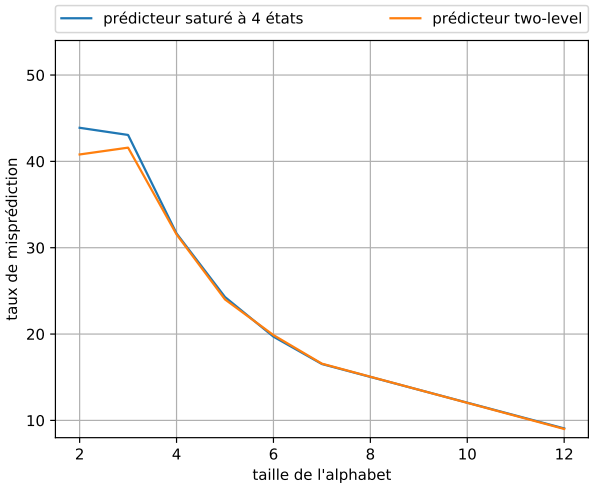
FIGURE 3.6 – Exemple d'un indice de la table

3.4 Implémentations et Résultats

Les deux prédicteurs ci-dessus ont été implantés en C et des macros ont été créées afin de pouvoir remplacer des IF ou des WHILE facilement dans les algorithmes. Dans chaque algorithme, il y a plusieurs branches mesurables mais seul le IF testant l'égalité entre deux caractères est pertinent :



Algorithme NAÏF



Algorithme KMP

Sur ces graphiques, on peut observer que plus la taille de l'alphabet augmente, moins il y a de misprédiction. En effet, la chance de comparer correctement une lettre du motif avec une lettre du texte est de $\frac{1}{|A|}$, donc plus il y a de lettres dans l'alphabet, plus il y a de chance de ne pas match et le prédicteur se trompera moins souvent.
On peut aussi remarquer que pour KMP, le prédicteur *Two-level* semble moins performant. Ceci est expliqué par la table des sauts qui bouleverse en quelque sorte le prédicteur. On essaiera de palier à ce problème dans la section 3.5.3.

Des mesures de prédictions de la réalité ont été réalisées avec la librairie PAPI [1] et donnent un résultat bien inférieur à ceux venant de prédicteurs simples. Malheureusement, les implémentations hardware des prédicteurs des processeurs actuels ne sont pas divulguées par les constructeurs.

3.5 Modifications des algorithmes existants

3.5.1 KMP avec deux IF

L'idée de séparer l'unique IF de comparaison en deux est proposée assez rapidemment par Mr Nicaud comme piste de départ d'amélioration. Elle consiste à se dire que lorsqu'il y a mismatch, on utilise la table des sauts qui nous envoie vers une nouvelle position à vérifier. Seulement on sait que cette position ne contient pas la meme lettre qu'à la dernière position du texte vérifié. On a donc une **probabilité différente** lors de la comparaison.
Il est alors possible que cette différence de probabilité perturbe les prédicteurs de branchement. Il est donc intéressant de voir si, en séparant le IF à $\frac{1}{|A|}$ du IF à $\frac{1}{|A|-1}$, il y a des changements.

En implémentant assez basiquement la modification, on trouve un algo-

rithme pouvant ressembler à celui-ci :

Algorithm 6 KMP2 search.

```
1: function KMP2
2:     s := i := k := 0
3:     fill_kmp_table(kmp_table)
4:     while j < n do
5:         if pat[k] == text[i] then                          ▷ First IF
6:             i++
7:             k++
8:             if k == m then
9:                 find(i − k)
10:                s++
11:                while pat[k] == text[i] do                 ▷ Second IF
12:                    k = kmp_table[k]
13:                end while
14:            end if
15:        else
16:            while pat[k] == text[i] do                     ▷ Second IF
17:                k = kmp_table[k]
18:            end while
19:            if k < 0 then
20:                i++
21:                k++
22:            end if
23:        end if
24:    end while
25: end function
```

Cependant, on obtient pas de différences significatives en terme de misprédiction. En revanche, la plupart des motifs qui ont des tables de bords intéressantes peuvent montrer l'intérêt d'utiliser cette deuxième version, étant donné que leur vitesse d'exécution sera légèrement accélérée (de l'ordre de 1 à 2%).

3.5.2 Comparer des caractères par blocs

Le problème majeur des algorithmes de pattern-matching dans le cadre des erreurs de misprédictions reste le fait qu'avec de **petits alphabets**, il est **impossible d'avoir une prédiction fiable**. En effet, avec un alphabet à 2 lettres et un prédicteur saturé à 4 états, le mieux qu'il puisse faire est d'avoir tort une fois sur deux.

Cependant, il y a un moyen de corriger ce problème : **agrandir virtuellement l'alphabet** en comparant les lettres deux par deux, ou quatre par quatre...
En effet, en comparant les lettres en **blocs**, on augmente la chance qu'une comparaison échoue. Par exemple avec un alphabet de 2 lettres, on a $\frac{1}{2}$ chance d'échouer, et en regroupant les lettres quatre par quatre, on aura $\frac{1}{16}$ chance d'échouer si le texte est homogène.

Avoir moins de chance de réussir une comparaison est quelque chose de très positif pour les prédicteurs, et cela peut potentiellement baisser fortement leurs taux de mispréditions.

3.5.3 Horspool avec des blocs

Comparer en utilisant des blocs de lettres n'est pas possible sur tous les algorithmes, notamment avec KMP. En effet, l'algorithme doit connaître précisément à quelle lettre on a échoué pour utiliser sa table des bords ; cela annule totalement l'effet escompté des blocs qui ne sont pas applicables pour cet algorithme.

Cependant, il est tout-à-fait possible de l'implémenter avec HORSPOOL puisque quoi qu'il arrive, on décalera le motif toujours en fonction de la lettre positionnée au même endroit. Il faut alors faire attention de ne pas comparer plus de lettres qu'il n'en faut et faire quelques vérifications supplémentaires si la taille du motif n'est pas divisible par notre taille de bloc.

En C, on utilisera des pointeurs de type `short *` ou `int *` à la place de l'habituel `char *`. En faisant cela, on accélère également l'accès mémoire, mais on verra que c'est assez négligeable. L'implémentation est disponible en annexe B.1.

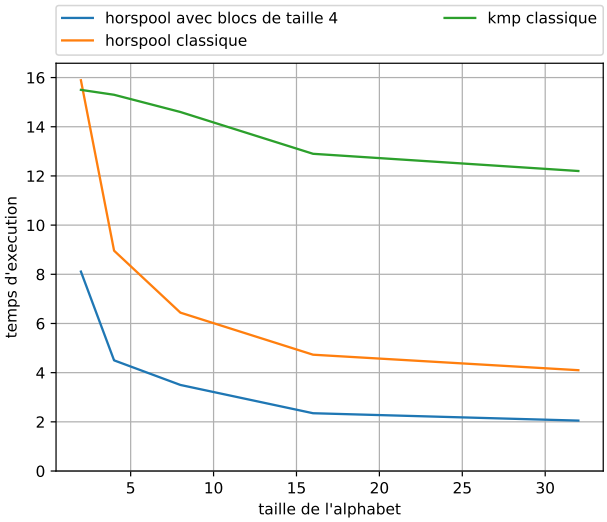


FIGURE 3.7 – Une fois l'implémentation effectuée, on obtient des résultats très intéressants. En moyenne, HORSPOOL AVEC BLOCS est **2 fois** plus rapide que HORSPOOL et jusqu'à **6 fois** plus rapide que KMP.

Chapitre 4

Approche théorique : les Chaînes de Markov

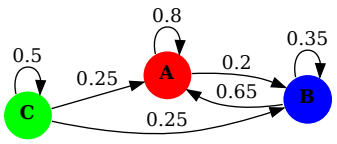
4.1 Chaînes de Markov

4.1.1 Introduction

Nous souhaitons maintenant étudier de manière théorique les deux algorithmes NAÏF et KMP. Pour ce faire, on va se donner un motif précis et essayer de simuler son exécution au travers de l'algorithme par un graphe de possibilités avec des probabilités dans le but de pouvoir calculer un taux de misprédiction.
Pour cela, on va utiliser des chaînes de MARKOV.

4.1.2 Présentation

Une chaîne de MARKOV [1] est un graphe orienté où l'on place sur chaque arc $\mathcal{A} \to \mathcal{B}$ la probabilité de passer de l'état $\mathcal{A}$ à l'état $\mathcal{B}$.
Par exemple, sur le graphe suivant, si l'on se trouve dans l'état $\mathcal{C}$, on a $\frac{1}{4}$ chance d'aller dans l'état $\mathcal{B}$:



L'idée principale des chaînes de MARKOV étant qu'à partir des données de l'état en cours, on sait calculer avec certitude les probabilités des prochains états potentiels. A partir de cela, on peut notamment calculer la probabilité stationnaire, probabilités d'être dans chaque état indépendemment de l'état initial et ce, après une marche aléatoire supposée grande.

1. Page wikipedia : `https://en.wikipedia.org/wiki/Markov_chain`

En pratique, on représente les chaînes de MARKOV par une matrice de transition où sont stockées les probabilités des arcs. Par exemple, voici la matrice de transition correspondant à la chaîne de MARKOV ci-dessus :

$$P = \begin{bmatrix} 0.8 & 0.2 & 0 \\ 0.65 & 0.35 & 0 \\ 0.25 & 0.25 & 0.5 \end{bmatrix}$$

Remarque La somme des probabilités de chaque ligne d'une matrice de transition donne 1. En effet, un parcours d'une chaîne de Markov ne s'arrête jamais, on a aucune chance de ne pas poursuivre dans un nouvel état :

$$\forall i \in [0 \dots n[, \sum_{j=0}^{m} T_{i,j} = 1.$$

4.1.3 Calcul de la loi stationnaire

En considérant ce système, on peut poser une hypothèse de départ $X^{(0)}$. Par exemple, si on part de l'état 0 :

$$X^{(0)} = \begin{bmatrix} 1 & 0 & 0 & \dots & \end{bmatrix}$$

On peut donc continuer pour trouver les prochain états :

$$\begin{array}{rcl} X^{(1)} & = & X^{(0)}P \\ X^{(2)} & = & X^{(1)}P \quad = \quad X^{(0)}P^2 \\ & & \dots \\ X^{(n)} & = & X^{(0)}P^n \end{array}$$

On pose q la loi de probabilité stationnaire d'une chaîne de Markov. Les propriétés des chaînes de Markov permettent de dire que celle-ci est indépendante de l'hypothèse de départ $X^{(0)}$ initiale. On a donc :

$$q = \lim_{n \to +\infty} X^{(n)}$$

Vu qu'il y a convergence, on peut donc écrire :

$$\begin{array}{rcl} qP & = & q\mathbb{I} \\ \iff q(\mathbb{I} - P) & = & 0 \\ \iff q(\mathbb{I} - P) & = & q\left(\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \end{pmatrix} - P\right) \\ & = & \begin{bmatrix} q_0 & \dots & q_n \end{bmatrix} \begin{pmatrix} (1 - P_{0,0}) & P_{0,1} & \dots & P_{0,m} \\ P_{1,0} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & P_{n-1,m} \\ P_{n,0} & \dots & P_{n,m-1} & (1 - P_{n,m}) \end{pmatrix} \\ & = & \begin{bmatrix} q_0 & \dots & q_n \end{bmatrix} P' = 0. \end{array}$$

Pour trouver $\begin{bmatrix} q_0 & \dots & q_n \end{bmatrix}$, il faut donc résoudre le système linéaire ci-dessous :

$$\begin{cases} (1-P_{0,0})q_0 & + & P_{1,0}q_1 & + & \dots & + & P_{n,0}q_n & = & 0 \\ (P_{0,1})q_0 & + & (1-P_{1,1})q_1 & + & \dots & + & P_{n,1}q_n & = & 0 \\ & & & \dots & & & & & \\ (P_{0,m})q_0 & + & P_{1,m}q_1 & + & \dots & + & (1-P_{n,m})q_n & = & 0 \end{cases}$$

Vu que q représente une loi de probabilité, la somme de ses composantes vaut forcément 1. Il faut donc rajouter la contrainte suivante :

$$q_0 + q_1 + \cdots + q_n = 1$$

Pour pouvoir automatiser la résolution de ce système, on va transformer la matrice P' afin d'obtenir une matrice résolvable par l'algorithme du pivot de Gauss-Jordan :

— ÉTAPE 1 : Transposer la matrice P'.

— ÉTAPE 2 : Agrandir la matrice obtenue d'une ligne et une colonne, puis placer des 0 sur la dernière colonne et des 1 sur la dernière ligne de sorte que la matrice obtenue soit de la forme :

$$F = \begin{pmatrix} * & \dots & * & 0 \\ \vdots & tr(P') & \vdots & \vdots \\ * & \dots & * & 0 \\ 1 & \dots & & 1 \end{pmatrix}$$

— ÉTAPE 3 : Effectuer un pivot de Gauss-Jordan [2] standard pour résoudre le système.

On obtient alors une matrice de la forme suivante avec les composantes de la loi stationnaire sur la dernière colonne :

$$\begin{pmatrix} 1 & \dots & 0 & q_0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 1 & q_n \\ 0 & \dots & & 0 \end{pmatrix}$$

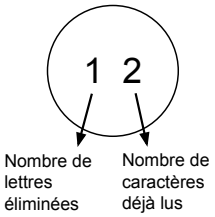
2. pivot de Gauss-Jordan : `https://en.wikipedia.org/wiki/Gaussian_elimination`

4.2 Graphe KMP simple

Lors de l'exécution de l'algorithme, on tient deux indices : l'indice du texte et l'indice du motif. Vu qu'ici il n'est pas question de texte, on va conserver celui du motif – pour savoir où on en est dans le motif et dans la table des bords–.

De plus, il nous faut simuler la seule information supplémentaire que l'on connait avec KMP, c'est-à-dire que lorsque l'on parcourt la table des bords, on élimine des lettres possibles du texte et donc on change les probabilités de match.

Pour ce graphe, on va alors considérer des états constitués de deux informations :



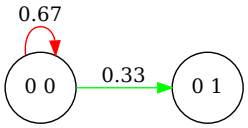
Chaque état va alors avoir deux arcs sortants, l'un correspondant à la probabilité de réussir la prochaine comparaison, et l'autre correspondant à l'inverse.

Exemple : Prenons le motif "ACAAA" sur l'alphabet ABC.
On calcule la table des bords associée :

-1	0	-1	1	1	1	1

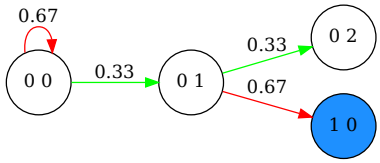
— Au début, on n'a rien lu, on part donc de l'état $\begin{pmatrix} 0 & 0 \end{pmatrix}$:

— On sait que la prochaine lettre du motif à comparer est A, par contre on a aucune information sur celle du texte. On a donc $\frac{1}{3}$ de réussir la comparaison et d'avancer. En cas d'échec, on regarde la table des bords. On a -1 donc on décale le motif et on revient au début de celui-ci :

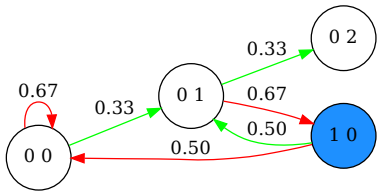


— On remarque que l'arc qui correspond à la réussite de la comparaison sera similaire tout le temps sauf quand on arrive au bout. En cas d'échec, on trouve 0 cette fois dans la table, on doit donc effectuer une comparaison

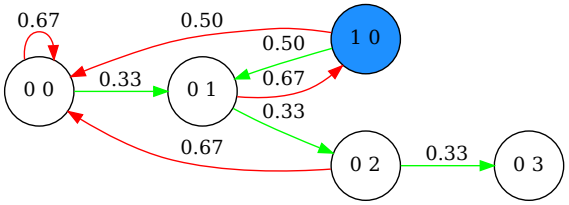
supplémentaire à l'indice 0 du motif et l'on sait que l'on ne le comparera pas à un C



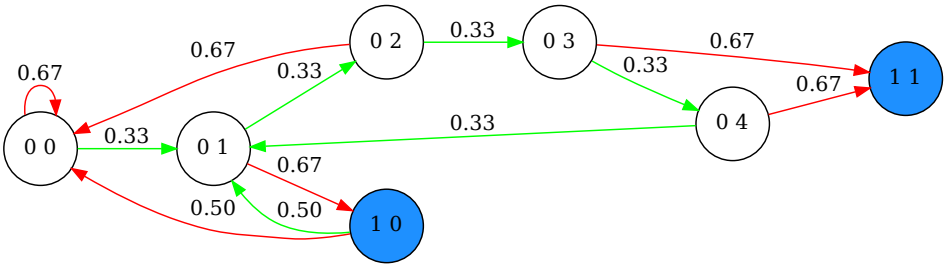
À partir de l'état $\begin{pmatrix}1 & 0\end{pmatrix}$, il ne reste que deux lettres possibles (A et B) et donc $\frac{1}{2}$ chance de faire matcher le A :



— L'état $\begin{pmatrix}0 & 2\end{pmatrix}$ est facile à traiter puisqu'il y a -1 dans la table des bords, donc on avance en cas de réussite ou on revient au début en cas d'échec :

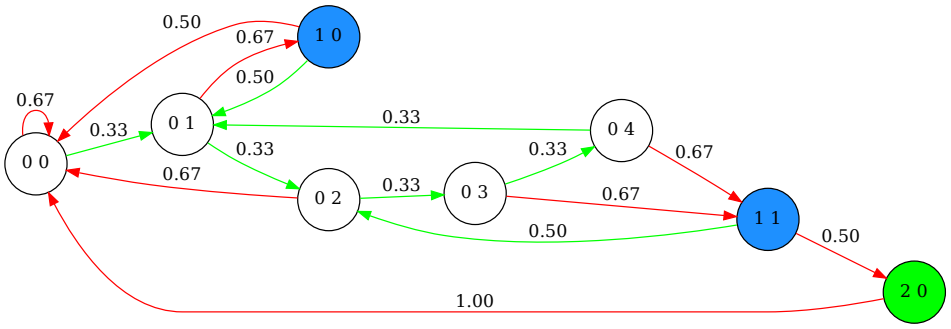


— Les états $\begin{pmatrix}0 & 3\end{pmatrix}$ et $\begin{pmatrix}0 & 4\end{pmatrix}$ sont assez similaires, sauf que le deuxième représente la fin du motif (Une fois celui-ci traité il ne reste plus qu'une lettre à vérifier, donc il est inutile d'ajouter un état supplémentaire). En cas d'échec, la table des bords indiquant 1, on ne retourne pas au début du motif ce qui équivaut à dire qu'on a déjà lu une lettre :



— Pour traiter $\begin{pmatrix}1 & 1\end{pmatrix}$, on a éliminé la lettre A, donc on a une probabilité de $\frac{1}{2}$ d'avancer et en cas d'échec on va dans $\begin{pmatrix}2 & 0\end{pmatrix}$ car on vient également d'éliminer la lettre C. Pour le dernier état, on sait que la comparaison va

échouer car on attend un A. On revient alors au début car il y a -1 dans la table des bords :



Une fois le graphe créé, on peut facilement calculer la probabilité de se trouver dans chaque état grâce à la méthode 4.1.3 fournie plus haut :

Noeud	(0 0)	(0 1)	(0 2)	(0 3)	(0 4)	(1 0)	(1 1)	(2 0)
q	0.448	0.229	0.090	0.029	0.010	0.153	0.027	0.013

4.3 Rajouter la misprédiction

4.3.1 Modifier le graphe existant

Maintenant, on souhaite rajouter sur chaque noeud une information concernant l'état d'un prédicteur théorique et ce, pour au final, calculer le taux de misprédiction qu'engendrera l'algorithme avec un certain motif donné.
On va alors utiliser un prédicteur saturé à 4 états et retenir dans chaque noeud de notre graphe l'état de celui-ci. Le but est de créer trois nouveaux graphes dérivés de celui ci-dessus :

— un global pour calculer le taux de misprédiction de KMP avec un seul IF. Ici on met à jour l'état de misprédiction à chaque fois.

— un que l'on ne met à jour que lorsqu'on se trouve dans un état de **couleur blanche** (premier IF)

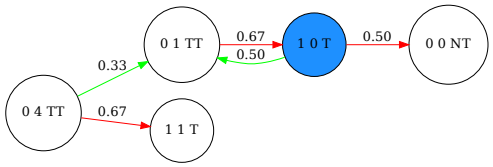
— un que l'on ne ne met à jour que lorsqu'on se trouve dans un **état coloré** (second IF)

Exemple : Reprenons le motif "ACAAA" sur l'alphabet ABC :

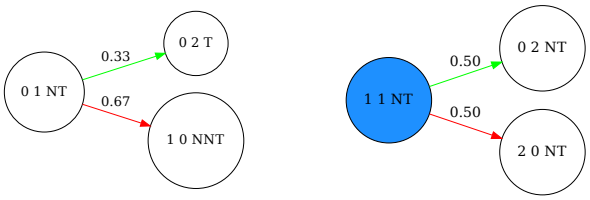
— Dans tous les cas, on repart donc du même état $\begin{pmatrix} 0 & 0 & NT \end{pmatrix}$ avec l'information NT, correspondant à NOT TAKEN de notre prédicteur :

— La construction du graphe se déroule de la même manière que précédemment. Sauf qu'ici on met à jour notre état de prédiction à chaque nouveau noeud. En cas d'erreur de comparaison, le prochain état sera décalé vers les NOT TAKEN et sinon, vers les STRONGLY TAKEN.

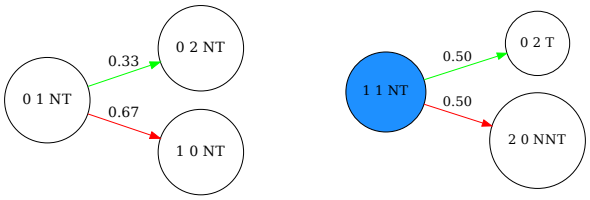
— **Graphe global :** On met à jour tout le temps



— **Graphe du premier if** : On ne met à jour que depuis un état non coloré



— **Graphe du second if** : On ne met à jour que depuis un état coloré



4.3.2 Calcul de la misprédiction

On peut rapidemment calculer le taux de misprédiction du IF que l'on souhaite en utilisant le bon graphe ainsi qu'en sélectionnant les bons noeuds $\mathbb{U}$.
Ensuite pour chaque noeud, il faut ensuite déterminer la probabilité d'effectuer une misprédiction.

Par exemple si l'état d'un noeud u est à NT, alors sa probabilité de misprédiction est la probabilité de réussite de la prochaine comparaison, c'est-à-dire $p(u)$. Enfin, on effectue une moyenne pondérée sur la loi stationnaire de chaque noeud.

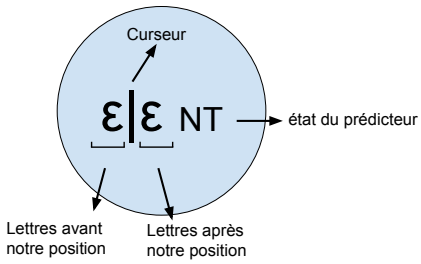
La formule suivante permet de résumer le calcul :

$$\frac{1}{\sum_{u}^{\mathbb{U}} q(u)} \times \sum_{u}^{\mathbb{U}} \begin{cases} q(u) \times p(u) & si\ u\ est\ NT\ ou\ NNT \\ q(u) \times (1 - p(u)) & sinon \end{cases}$$

4.4 Graphe de l'algorithme naïf

Le graphe de l'algorithme NAÏF se construit un peu de la même manière que celui pour KMP. Il faut cependant garder plus d'information à l'intérieur de chaque noeud.

Etant donné qu'il n'y a pas de table associée, on doit se rappeler quelles lettres on a déjà pu lire précédemment. De plus, vu que notre curseur de lecture va revenir en arrière, on peut lire potentiellement plusieurs fois une même lettre du texte.
Enfin, on garde l'information d'un prédicteur de branchement théorique, ici saturé à 4 états, comme précédemment :

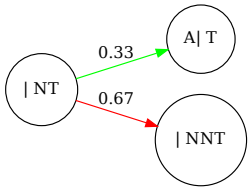


Pour une meilleure lecture visuelle, les noeuds correspondant à des décalages du motif (en cas de mismatch notamment) seront affichés sous forme de rectangles.

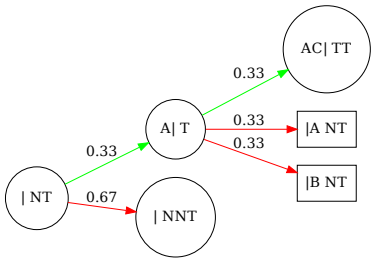
Exemple avec le motif ACAA et l'alphabet ABC :

— On part de l'état suivant, où on n'a encore rien lu et où l'état du prédicteur est placé à NT :

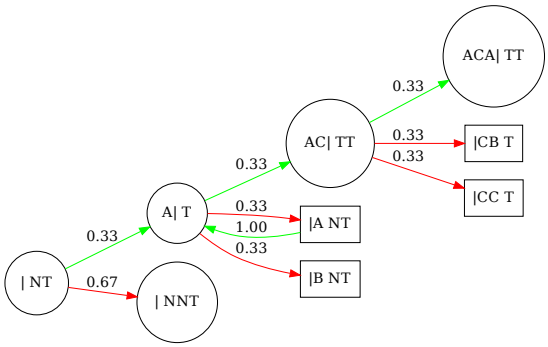
— La première lettre du motif est A, donc soit on lit un A dans le texte et on avance d'une lettre, soit on lit une autre lettre et dans ce cas il y a échec de la comparaison et on décale le motif vers la droite où il n'y a que des lettres inconnues :



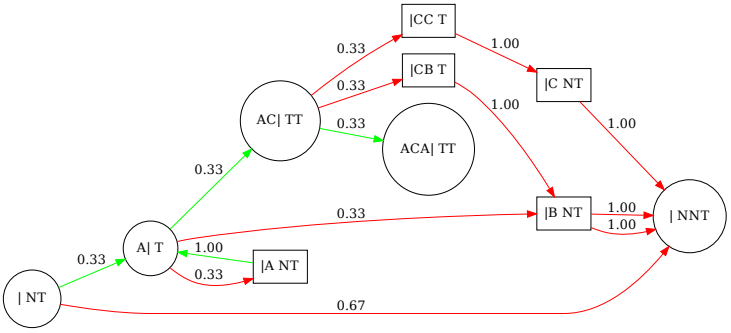
— Depuis `A| T`, on a toujours $\frac{1}{3}$ de chance de lire un C. En cas d'échec, on décale le motif de un et on revient au début de celui-ci, donc on sait que la prochaine lettre à lire est soit un A soit un B :



— Depuis `|A NT`, on sait que la prochaine lettre du texte à lire est un A, comme la prochaine lettre du motif. On va forcément réussir la prochaine comparaison et revenir à l'état `A| T` puisqu'il y aura maintenant un A dernière nous. De la même manière que précédemment, depuis `AC| TT`, on peut lire la bonne lettre A ou bien se tromper en lisant B ou C :



— Pour traiter les noeuds à rectangle, il suffit de les suivre en éliminant une lettre par une lettre. Vu que l'on sait exactement les prochaines lettres à lire, il n'y a que des probabilités de 1 :



— En continuant de la même manière, on arrive à un graphe bien plus gros de 30 états. On peut maintenant calculer, comme on l'avait fait avec le graphe du KMP global (Voir section 4.3.1), le taux de misprédiction.

Chapitre 5

Version naïve meilleure que kmp ?

5.1 Observation

Comme énoncé dans la section 2.7, on observe que l'algorithme NAÏF quadratique est plus performant en pratique que KMP.
Le premier réflexe que l'on a en observant ce surprenant résultat est de suspecter une implémentation erronée ou des mesures mal faites. J'ai donc été chercher d'autres versions des algorithmes ainsi qu'une application externe de mesure des performances.

Les tests ont été effectués avec trois versions de KMP :

— ma propre version avec un et deux IF [1]

— la version wikipedia anglaise [2] (la version française est en réalité l'algorithme MORRIS-PRATT)

— la version de Mr Lecroq [3], chercheur spécialiste de l'algorithmique du texte

De l'autre côté, la version naïve est conforme avec la version de Mr Lecroq [4]. De plus, plusieurs forums ont déjà remarqué cet état de fait [5] mais aucune réponse satisfaisante n'est fournie.

Au sujet de l'implémentation, j'ai utilisé deux méthodes :

1. KMP avec deux IF : Voir section 3.5.1
2. KMP version wikipedia : `http://fr.wikipedia.org/wiki/Algorithme_de_Knuth-Morris-Pratt`
3. KMP version Lecroq : `http://www-igm.univ-mlv.fr/~lecroq/string/node8.html`
4. NAÏF version lecroq : `http://www-igm.univ-mlv.fr/~lecroq/string/node3.html`
5. par exemple sur StackOverflow : `http://stackoverflow.com/questions/20016092/why-is-naive-string-search-algorithm-faster`

— mon propre code C qui génère des motifs et textes aléatoires et mesure plusieurs fois chaque recherche. *(Voir le dossier* `timeperf/`*)*

— l'application SMART[6] qui permet d'ajouter ses propres algorithmes et de mesurer les temps de recherche sur divers sources : aléatoire avec alphabet défini ou corpus (littérature et ADN)

De plus, plusieurs machines ont été utilisées pour obtenir des résultats :

— mon ordinateur portable équipé d'un *Intel Core i3 M370* ainsi que mon ordinateur fixe équipé d'un *Intel Core i7 4790K*

— le serveur MONGE du laboratoire équipé d'un *Intel Xeon E5-2643*

— plusieurs Raspberry Pi d'architecture ARM

Autre point important, le **nombre d'instructions** de l'algorithme NAÏF est toujours plus grand que celui ce KMP. Cependant, on peut observer que le nombre de cycles réels tourne à l'avantage du premier. On a donc un rapport *instruction/cycle* bien plus avantageux pour le NAÏF et qui permet, au final, d'obtenir des temps d'exécution inférieur au KMP.

Conclusion : Tous ces algorithmes, ces méthodes et ces machines donnant les mêmes résultats, cela convainc de leurs exactitudes. Il est donc intéressant d'essayer de comprendre pourquoi c'est ainsi.

5.2 Causes éliminées

— On observe que le **nombre de comparaisons**[7] des algorithmes, correspondant aux attentes de complexités, ne permet pas d'expliquer le phénomène.

— De plus, **le cache** a peu d'influence puisqu'on a pu observer expérimentalement que les différences du nombre de cache-miss sont négligeables sur les deux algorithmes

— Enfin, le **taux de misprédiction** est lui aussi dans la même moyenne pour les deux algorithmes.

Comme expliqué dans la section précédente, le rapport *instruction/cycle* du NAÏF est plus élevé et peut même dépasser le ratio de **1**.

Cela peut paraître surprenant mais c'est une chose commune sur les processeurs modernes qui contiennent plusieurs unités de calculs par coeur mais

6. SMART : `https://smart-tool.github.io/smart/`
7. Graphique des nombres de comparaison : Voir section 2.7

cela ne change rien à notre problème.

Pour tenter d'expliquer plus en détails ces conditions, il faut se repencher sur le pipeline, comme par exemple le modèle RISC.

5.3 Le pipeline RISC plus en détails

Dans le monde des microprocesseurs modernes, le pipeline classique RISC [2][9][10] est la modélisation théorique de référence créée dans les années 1970 par David Patterson.
Ce modèle permet notamment la parralélisation des instructions tout en veillant à ce que cela reste implémentable.

Pour ce faire, ce modèle décompose, comme nous avons déjà pu le faire, l'instruction en 5 micro-instructions :

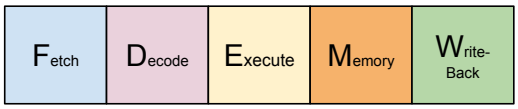


FIGURE 5.1 – Division d'une instruction en 5 micro-instructions (modèle RISC)

Ce qui est important à noter est que chaque micro-instruction a un RÔLE précis :

— FETCH : Récupère l'instruction en mémoire et incrémente le Program Counter.

— DECODE : Décode l'instruction et lit les registres associés aux opérandes si besoin.

— EXECUTE : Cette micro-instruction agit différemment en fonction des types d'opérandes :

- une des opérandes est un accès mémoire : effectue un calcul d'adresse virtuelle.
- les deux opérandes sont des registres : effectue le calcul associé à l'instruction
- une des opérandes est un instantané (un nombre) : effectue le calcul associé directement.

— MEMORY : Si l'instruction est de type LOAD, la micro-instruction effectue une écriture mémoire et si c'est un STORE, elle effectue une lecture mémoire. Sinon elle ne fait rien.

— WRITE-BACK : Ecrit le résultat final de l'instruction dans le registre de destination.

Comme on peut rapidemment l'observer, il peut y avoir des problèmes d'interdépendances entre les micro-instructions DECODE et WRITE-BACK pour les registres et la micro-instruction MEMORY pour les accès mémoire.

En cas d'interdépendance, on introduit donc une latence, c'est-à-dire un temps d'attente sur l'une des deux instructions qui veut accéder au même registre ou à la même mémoire pour permettre d'obtenir un résultat final correct, ce qui reste la priorité absolue malgré tout.

Dans ce modèle, une micro-instruction prend une unité atomique de temps pour s'exécuter. Sur les processeurs actuels, l'unité atomique de temps est le **cycle**. La latence que l'on cherche à déterminer s'exprime donc en nombre de cycles perdus, ou mis en attente, appelés **stalled cycles**.

Maintenant qu'on a une conception un peu plus fine du pipeline RISC, on veut essayer de compter avec un script ces **stalled cycles** pour mesurer le taux d'attente d'un algorithme.

5.4 Script Python et Résultats

En ayant tout cela à l'esprit, j'ai créé un script Python permettant de prendre en entrée des instructions au format assembleur NASM [8] et construisant un pipeline RISC.

Le script simule alors l'exécution naturelle du programme en rajoutant les instructions au fur et à mesure dans le pipeline et en déterminant les interdépendances de registres et de mémoire.

Il simule également un accès mémoire plus réaliste en introduisant des temps de latence aléatoires plus au moins grands. La réussite de comparaison des caractères est paramétrable en changeant la probabilité de réussir une comparaison entre une lettre du motif et du texte.

Enfin, le script compte le nombre de stalled cycles résultant de sa simulation et j'ai pu dresser un graphique comparatif de l'algorithme NAÏF et KMP :

8. Assembleur NASM : `https://en.wikipedia.org/wiki/Netwide_Assembler`

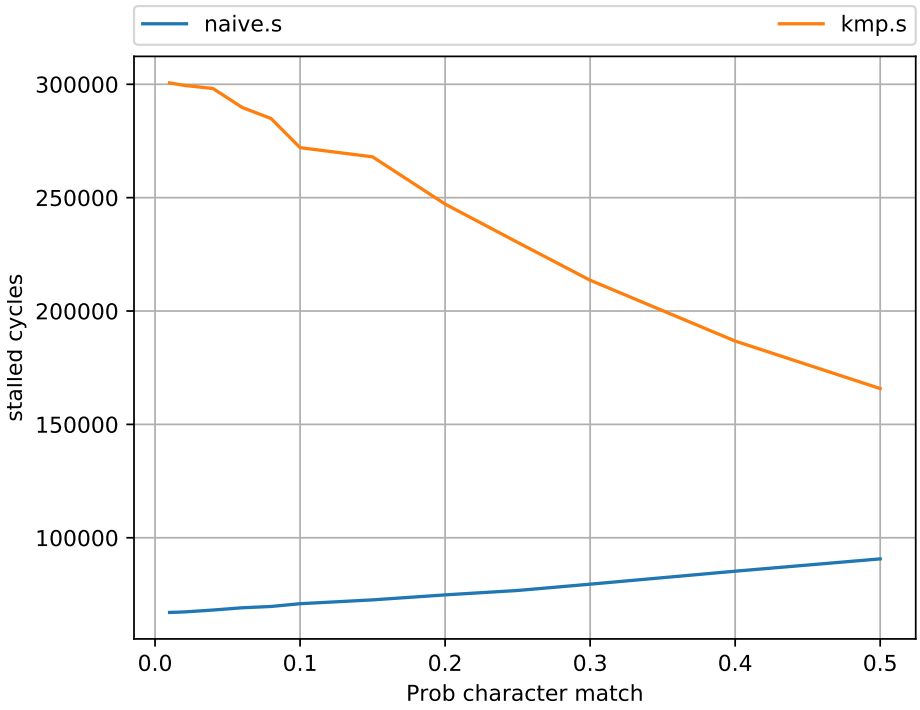


FIGURE 5.2 – Nombre de cycles stalled en fonction de la probabilité de réussite d'une comparaison. On observe que KMP est bien moins performant ce qui explique un fort ralentissement de son exécution.

Ce graphique montre la différence notable entre les deux algorithmes où on peut voir que KMP possède beaucoup plus de latence que le NAÏF, ce qui peut expliquer au moins une grande partie de la lenteur d'exécution de KMP.
Je n'ai cependant pas pu montrer quelles instructions étaient principalement mis en attente car je n'ai pas pu observer facilement de schéma caractéristique où certaines instructions étaient favorisées par rapport à d'autres.

Chapitre 6

Bilan

6.1 Implémentations effectuées

Lors de cet stage, j'ai pu effectuer de nombreuses implémentations que ce soit en C ou en Python, totalisant 7600 lignes de code au total.

Les différentes implémentations concernent :

— 5 algorithmes classiques de Pattern-Matching ainsi que leurs variantes : NAÏF, KMP, BOYER-MOORE, HORSPOOL et SKIPSEARCH ainsi que plusieurs améliorations dont HORSPOOL avec des blocs ou KMP avec deux if.

— Deux prédicteurs théoriques notamment le prédicteur saturé et sa variante le prédicteur two-level.

— la création des chaînes de Markov c'est-à-dire la création des graphes à partir d'un motif. De plus j'ai également codé moi-même un outil de gestion de matrices afin effectuer le calcul des lois stationnaires et de la misprédiction.

— plusieurs scripts en Python permettant de manipuler plus facilement les programmes précédents ainsi que le script qui interprète les instructions assembleurs, qui simule un pipeline et permettant de calculer le nombre de stalled cycles.

L'intégralité de mes sources sont disponibles en ligne sur mon site web : `http://veillerette.me/stage-M1/`

6.2 Compétences acquises

Au cours de ce stage, j'estime avoir acquis plusieurs compétences techniques, telles que :

— l'approfondissement de mes connaissances sur les pipelines, les prédicteurs de branchements, ainsi que sur les spécificités liés aux processeurs superscalaires.

— la manière dont on peut réfléchir à des algorithmes en prenant en compte des éléments externes, comme ceux associés à l'architecture des ordinateurs

— la découverte des chaînes de MARKOV incluant leurs créations et calculs associés ; et la façon dont on peut s'en servir pour analyser des algorithmes de manière probabiliste.

De plus, ce stage m'a aussi permis de découvrir le monde de la recherche qui est basé sur l'autonomie, la discussion d'idée, le doute et l'investigation personnelle, quel que soit le domaine.

Bibliographie

[1] Papi library and documentation. https://icl.cs.utk.edu/projects/papi/wiki/Main_Page.

[2] Wikipedia : Pipeline risc. https://en.wikipedia.org/wiki/Classic_RISC_pipeline.

[3] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Good predictions are worth a few comparisons. *Leibniz International Proceedings in Informatic*, 2016. https://hal-upec-upem.archives-ouvertes.fr/hal-01212840/document.

[4] Christian Charras and Thierry Lecroq. KMP algorithm. http://www-igm.univ-mlv.fr/~lecroq/string/node8.html.

[5] Christian Charras and Thierry Lecroq. NAÏVE algorithm. http://www-igm.univ-mlv.fr/~lecroq/string/node3.html#SECTION0030.

[6] Simone Faro and Thierry Lecroq. The exact string matching problem : a comprehensive experimental evaluation. https://arxiv.org/abs/1012.2547, 2010.

[7] Nigel Horspool. Practival fast searching in strings. http://www.cin.br/~paguso/courses/if767/bib/Horspool_1980.pdf, 1980.

[8] Donald Knuth, James Morris, and Vaughan Pratt. Fast pattern matching in strings. https://pdfs.semanticscholar.org/4479/9559a1067e06b5a6bf052f8f10637707928f.pdf, 1977.

[9] David Patterson and John Hennessy. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, 1989.

[10] David Patterson and John Hennessy. *Computer Organization and Design*. Morgan Kaufmann, 1994.

Annexe A

Résultats expérimentaux

A.1 Mesure de taux de misprédiction

Les mesures sont en pourcentage de branches testées sur plusieurs textes/motif générés aléatoirement.

A.1.1 Algorithme Naïf

taille de l'alphabet	2	3	4	5	6	7	12
prédicteur saturé à 4 états	51.05	39.60	29.70	23.37	19.14	16.16	8.9
prédicteur two-level k=5	30.67	32.88	28.27	22.28	18.4	15.4	8.5
avec PAPI	5.1	5.3	4.1	3.3	2.6	2.2	0.9

A.1.2 Algorithme KMP, un seul IF

taille de l'alphabet	2	3	4	5	6	7	12
prédicteur saturé à 4 états	43.89	43.06	31.65	24.29	19.70	16.53	9.07
prédicteur two-level k=5	40.8	41.59	31.54	24.01	19.88	16.57	9.01
avec PAPI	10.8	9.5	6.5	4.4	3.6	3.0	1.6

Annexe B

Algorithmes supplémentaires

B.1 Horspool par blocs de 4

```
void horspool_block_int(char *source, int n, char *pat, int m)
{
        int T[256];
        int i;
        int s = 0;
        unsigned char *text = (unsigned char *)source;
        unsigned char *end = text + n - m;
        fill_horspool_first_table(T, pat, m);
        while(text < end)
        {
                i = m-1;
                while(i-3>= 0 &&
                *((int *)(text+i-3)) == *((int *)(pat+i-3)))
                        i -= 4;
                if (i<=3 && ((i<=-1)
                || (i>=0 && i <= 2 && pat[i] == text[i]
                     && (i==0 || pat[i-1] == text[i-1])
                     && (i<2  || pat[i-2] == text[i-2])
                  )))
                        s++; /* found an occur at text - source */
                text += T[(int)text[m-1]];
        }
}
```