Rapport de stage de recherche de M2

---

# Analyse réaliste d'algorithmes de recherche de motifs avec prise en compte d'éléments d'architecture des ordinateurs

---

Victor Veillerette

*Encadré par :*
M. Cyril Nicaud
Mme Carine Pivoteau

Avril - Août 2019

# Table des matières

I Introduction 2

1 Présentation du lieu d'accueil 2
- 1.1 L'université . . . 2
- 1.2 Le laboratoire de recherche . . . 2

2 Sujet 3
- 2.1 Domaine de recherche . . . 3
- 2.2 Contexte et sujet . . . 4
- 2.3 Organisation de travail . . . 4

II Modèles d'architectures des ordinateurs 5

1 Les pipelines 5
- 1.1 Le pipeline RISC . . . 5
- 1.2 Interdépendances . . . 7
- 1.3 Shortcuts . . . 8
- 1.4 Problématique des branchements conditionnels . . . 8

2 Les prédicteurs de branchements 9
- 2.1 Intérêts et coûts . . . 9
- 2.2 État du pipeline avec une misprédiction . . . 9
- 2.3 Prédicteur simple saturé . . . 10
- 2.4 Prédicteur Two-Level global . . . 10
- 2.5 Comparaison avec la réalité . . . 11

III Algorithmes de Pattern-Matching 12

1 Enjeux et applications 13

2 Mise en oeuvre 13

**3 L'algorithme bruteforce ou naïf** **13**
3.1 Phase de recherche . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2 Complexités . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.3 Implémentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**4 Introduction à KMP** **15**
4.1 Un bord . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
4.2 Une table de saut . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**5 L'algorithme Knuth-Morris-Pratt** **16**
5.1 Phase de recherche . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
5.2 Table de KMP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
5.3 Implémentation de la table des bords . . . . . . . . . . . . . . . . . . . . 17
5.4 Implémentation de la phase de recherche . . . . . . . . . . . . . . . . . . 17

**6 L'algorithme d'Horspool** **18**
6.1 Utiliser une table de saut . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
6.2 Implémentation de la phase de recherche . . . . . . . . . . . . . . . . . . 19

**7 Comparaisons expérimentales** **19**
7.1 Statistiques expérimentales . . . . . . . . . . . . . . . . . . . . . . . . . . 20
7.2 Algorithme naïf face à KMP . . . . . . . . . . . . . . . . . . . . . . . . . 21
7.3 La problématique de l'implémentation . . . . . . . . . . . . . . . . . . . . 21

**IV Simulateur de Processeur** **22**

**1 Intérêts et Objectifs** **22**

**2 Architecture interne** **23**
2.1 Fonctionnalités du langage assembleur . . . . . . . . . . . . . . . . . . . . 23
2.2 Simulateur de processeur . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.3 Interface graphique . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**V Introduction à l'analyse théorique de KMP** **26**

**1 Comparaisons du nombre de branches** **26**

**2 Introduction générale aux chaînes de Markov** **27**
2.1 Présentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.2 Calcul de la loi stationnaire . . . . . . . . . . . . . . . . . . . . . . . . . . 28

**VI Analyse du prédicteur global avec KMP** **30**

**1 Convention de nommage** **30**

**2 Construction de la chaîne** **31**
2.1 États déterministes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
2.2 État non déterministe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**3 Ajout de la prédiction de branchement** **32**
3.1 Rajout du prédicteur global . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.2 Calcul de la misprédiction . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

**4 Simplification du graphe** **34**
4.1 Minimisation calée sur les comparaisons . . . . . . . . . . . . . . . . . . . 34
4.2 Minimisation calée sur la découverte du texte . . . . . . . . . . . . . . . . 35
4.3 Tarjan et résultats de la minimisation . . . . . . . . . . . . . . . . . . . . 36

**5 Propagation inversée de la loi stationnaire** **37**
5.1 Principe de la propagation . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.2 Déroulé . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**6 Analyse fine d'un motif précis** **39**

**VII Bilan** **43**

**1 Synthèse des travaux** **43**

**2 Compétences acquises** **43**

**3 Conclusion et pistes futures** **44**

# Table des figures


1 Evolution des effectifs de recherche au LIGM . . . . . . . . . . . . . . . . 2
2 Une instruction commence lorsque la précédente est terminée . . . . . . . . . . 5
3 Division d'une instruction en 5 $\mu$-instructions (modèle RISC) . . . . . . . . . . 6
4 Parallélisation des instructions dans un pipeline . . . . . . . . . . . . . . . . . . . 6
5 Perte de deux cycles en cas d'interdépendance sémantique . . . . . . . . . . . . 7
6 Perte d'un cycle en cas d'interdépendance structurelle . . . . . . . . . . . . . . . 7
7 Illustration d'un shortcut . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
8 Illustration du problème des branchements conditionnels . . . . . . . . . . . . . 8
9 Annulation d'instructions dans le pipeline à cause d'une misprédiction . . . . . 9
10 Exemple avec un prédicteur saturé à 4 états . . . . . . . . . . . . . . . . . . . 10
11 Extrait de la table du prédicteur Two-Level avec $k = 5$ . . . . . . . . . . . . . 11
12 Exemple d'un indice de la table . . . . . . . . . . . . . . . . . . . . . . . . . . 11
13 Comparaison du prédicteur global avec la réalité. . . . . . . . . . . . . . . . . 12
14 Exemple de la recherche du motif BBABA . . . . . . . . . . . . . . . . . . . . 14
15 Exemple : table des bords de KMP pour le motif ABAA . . . . . . . . . . . . . 17
16 Exemple : Pour le motif BBABA on obtient alors la table ci-dessus . . . . . . . 19
17 Illustration du nombre de comparaisons des différents algorithmes présentés . . . 20
18 Temps d'execution des différents algorithmes . . . . . . . . . . . . . . . . . . . 20
19 Utilisation du simulateur avec ses entrées. . . . . . . . . . . . . . . . . . . . . . 22
20 Architecture interne du processeur. . . . . . . . . . . . . . . . . . . . . . . . . . 24
21 Interface graphique du simulateur réalisée avec tkinter. . . . . . . . . . . . . . 25
22 Accès à la table de décalage en fonction de l'alphabet . . . . . . . . . . . . . . 26
23 Un noeud de notre chaîne . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
24 Ajout du prédicteur global dans un noeud du graphe . . . . . . . . . . . . . . . 33
25 Taille du graphe en fonction de la taille du motif et de la taille de l'historique $l$. . 34
26 Taille du graphe en fonction du motif et pour une taille d'historique $l = 8$. . . . 37
27 Chaîne de MARKOV représentant le prédicteur simple du cas n°1 . . . . . 42

# 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 les pistes de travail 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.

Je remercie également Monsieur Stéphane Vialette, directeur du LIGM, de m'avoir accueilli au poste de stagiaire parmi ses équipes de recherche.

## Première partie

# Introduction

## 1 Présentation du lieu d'accueil

Dans le cadre de la deuxième année de mon Master d'informatique, j'effectue ce stage me permettant de valider mes acquis et d'approfondir certaines notions spécifiques peu étudiées.

### 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'ESIEE 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 – 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.

- *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.

# 2 Sujet

## 2.1 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. Pour mieux étudier ces différents éléments, j'ai réalisé la création d'un simulateur de processeur à but pédagogique.

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.

## 2.2 Contexte et sujet

L'année précédente, j'ai déjà effectué un stage avec mes tuteurs dans le cadre de la validation de ma première année de master. Étant donné que celui-ci s'est très bien passé, que le sujet m'intéressait toujours et que le domaine était prometteur, j'ai décidé de continuer sur ce sujet cette année pour ce stage plus long et pour me permettre d'aller plus loin.

Lors de mon stage précédent, j'ai pu voir que les différents éléments d'architectures d'un ordinateur pouvaient avoir de grands impacts sur les performances d'un algorithme donné. En effet, une des observations les plus significatives est le temps d'exécution inférieur d'un algorithme quadratique par rapport à un algorithme linéaire.

Une des premières étapes de mon stage était donc de comprendre plus en détail quelles en étaient les causes précises. Vu que l'outil classique de la complexité (dans le pire cas ou en moyenne) n'est pas représentatif ici, j'ai commencé par explorer les différents modèles possibles d'architecture en essayant d'en faire une synthèse simple et de les comparer expérimentalement avec la réalité matérielle étant donné que les infrastructures actuelles sont protégées par des brevets.

Dans ce stage, je me suis focalisé sur les algorithmes classiques de Pattern-Matching dont j'ai étudié le comportement de divers aspects d'architecture classique peu étudiés. Leurs fonctionnements simples et répétitifs sont alors grandement impactables par les questions d'architecture.

## 2.3 Organisation de travail

Le stage s'est déroulé au sein même du laboratoire de début avril 2019 à août 2019 pour une durée totale de 4 mois sur la base de semaines de 35 heures. Je travaillais dans une salle dédiée aux stagiaires dans laquelle je disposais d'un bureau où je pouvais travailler avec ma machine personnelle.
Les conditions de travail étaient bonnes et j'ai pu effectuer de nombreuses réunions 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. 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.

Deuxième partie

# Modèles d'architectures des ordinateurs

Étant donné que la majorité des modèles de processeurs actuels sont soumis aux brevets, il est difficile de connaître finement les différents composants réels de ceux-ci. En effet, depuis les années 2000, on a pu observer un ralentissement de la croissance des fréquences d'horloges ainsi que du nombre de coeurs. Pour continuer à améliorer les performances des processeurs, les fondeurs ont donc mis au point divers outils secondaires.
Parmi ceux qui sont incontournables de nos jours, on peut citer les mécanismes de cache [8] pour améliorer les performances mémoires, l'exécution à la volée dans le désorde [9] particulièrement étudiés mais aussi la parallélisation des instructions grâces aux pipelines et la prédiction de branchements que l'on présente ci-dessous.

## 1 Les pipelines

Les PIPELINES [8] introduits pour la première fois en 1961 par l'IBM 7030 [2] permettent la parallélisation automatique des instructions.
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, voir figure 1. Cette méthode fonctionne, cependant

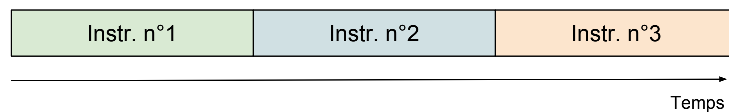



FIGURE 2 – Une instruction commence lorsque la précédente est terminée

les capacités automatiques d'un processeur sont largement inutilisées, étant donné l'hétérogénéité des ensembles d'instructions : il y a des instructions qui intéragissent avec la mémoire, qui effectuent des calculs ou qui effectuent des sauts. De plus, si deux instructions se suivent et n'intéragissent pas entre elles, on a envie de les exécuter au même instant.

### 1.1 Le pipeline RISC

C'est pour répondre à ce problème que les PIPELINES [8][9] furent introduits. 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. L'un des modèles les plus standards est le modèle RISC créé par David Patterson et toujours utilisé de nos jours. [1] Il consiste à diviser chaque instruction en **5 $\mu$-instructions**, voir figure 3 :

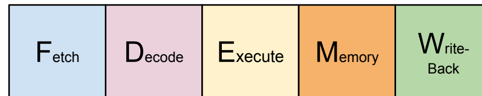



FIGURE 3 – Division d'une instruction en 5 $\mu$-instructions (modèle RISC)

comprenant :

- FETCH : Incrémente le compteur d'instruction et récupère l'instruction.
- DECODE : Décode l'instruction et lit au besoin les registres.
- EXECUTE : Effectue les calculs arithmétiques.
- MEMORY : Réalise les mouvements mémoires en commençant par l'écriture puis la lecture.
- WRITE BACK : Enregistre le résultat dans les registres correspondants.

L'interêt de ce découpage est de pouvoir exécuter chaque type de $\mu$-instruction en parallèle. On peut alors obtenir une séquence d'exécution de la sorte :

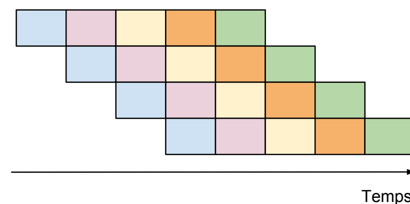



FIGURE 4 – Parallélisation des instructions dans un pipeline

En pratique, on considère qu'une $\mu$-instruction est l'étape atomique du processeur et sa durée est $d(\mu) = 1$ *(cycle)*.
Si on a $n$ instructions totalement indépendantes, on a alors $d_{seq}(n) = 5 \times n$ et $d_{\text{RISC}}(n) = 5 + n$, alors $\forall n \geq 1, d_{seq}(n) > d_{\text{RISC}}(n)$.

1. Son utilisation a été annoncée par de grands groupes tels que Nvidia ou Qualcomm dans le cadre de l'initiative RISCV : `https://riscv.org/`

Le pipeline est donc un excellent outil ; cependant, il existe une multitude de problèmes qui peuvent être pour la plupart résolus en introduisant de la latence, c'est-à-dire en décalant une (ou plusieurs) ligne(s) de notre pipeline pour permettre une exécution possible. On appellera cela une **bulle** ou « *cycle perdu* ». Les différentes causes engendrant ces latences sont décrites ci-dessous.

### 1.2 Interdépendances

Un des premiers problèmes qui se pose est le problème **d'interdépendances** entre les instructions qui se suivent. Il en existe deux types principaux :

— Vu que l'exécution des instructions se chevauchent par parallélisation, il est possible de vouloir lire une valeur alors que l'instruction précédente qui l'écrit n'est pas encore terminée. C'est ce que l'on appelle l'**interdépendance sémantique**. Pour rappel, on lit les registres des opérades à l'étape DECODE. Voir l'exemple figure 5.

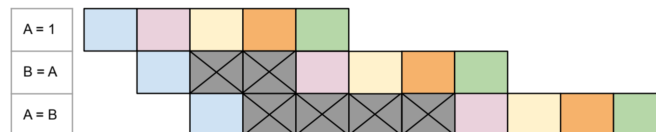



FIGURE 5 – Par exemple, on perd deux cycles à l'instruction n°2 car on doit attendre que $A$ soit écrit.

— Dans notre pipeline, il est également possible d'avoir une $\mu$-instruction FETCH et MEMORY à exécuter en même temps et cela pose problème. En effet, on s'apprête à lire en mémoire à la fois une instruction et une opérade, ce qui n'est pas possible. C'est ce que l'on appelle une **interdépendance structurelle**. Toutefois, si la première est obligatoirement effectuée pour toute instruction, la deuxième ne l'est que pour les instructions liées à la mémoire. Voir l'exemple figure 6.

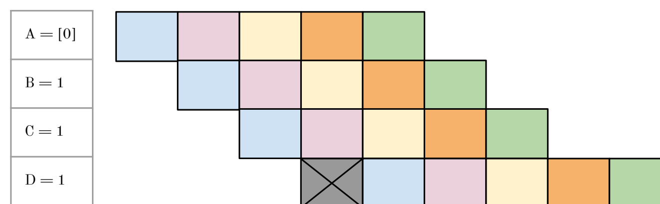



FIGURE 6 – Par exemple, on perd un cycle à la dernière instruction car on doit effectuer l'opération mémoire de la première instruction.

## 1.3 Shortcuts

Les processeurs actuels intègrent toute une série d'optimisations liées au pipeline. Une des plus anciennes utilisées est la technique des **shortcuts**. Cette technique améliore la latence liée à l'interdépendance sémantique et permet d'avoir accès à une valeur avant qu'elle ne soit écrite dans un registre.

Deux cas se présentent :

— La valeur demande la lecture de registres et/ou un calcul arithmétique et est disponible directement après l'opération EXECUTE.

— La valeur demande une écriture mémoire et on doit attendre la fin du MEMORY

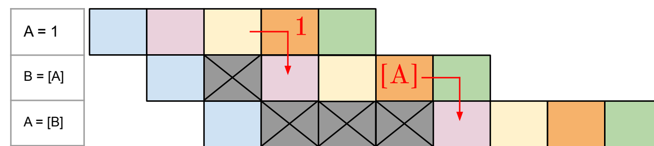



FIGURE 7 – Reprise de la figure 5 avec des shortcuts. Dans le premier cas, la valeur est directement accessible

## 1.4 Problématique des branchements conditionnels

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 l'exécution de sa valeur et il faudrait donc retarder toute la suite du programme pour savoir si on prends la branche ou non et donc connaître la prochaine instruction :

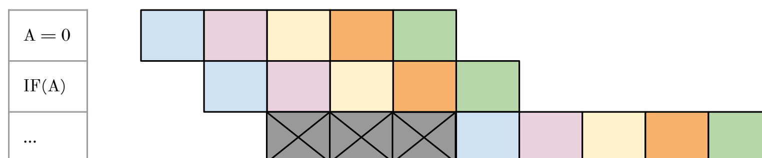



FIGURE 8 – On est obligé d'attendre la valeur de la première instruction pour rajouter la nouvelle instruction au pipeline.

À noter qu'ici, le coût de la bulle introduite est variable et dépend directement du coût de la dernière instruction située avant le branchement.

## 2 Les prédicteurs de branchements

Pour résoudre le problème des branchements conditionnels, on a introduit les PRÉDICTEURS DE BRANCHEMENTS qui permettent de prévoir la prochaine instruction grâce à un circuit hardware plus ou moins complexe en temps constant.

### 2.1 Intérêts et coûts

Plus précisément, le prédicteur va estimer, sur la base des données des précédentes branches, si la prochaine branche 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[2] 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 notables.

### 2.2 État du pipeline avec une misprédiction

Quoi qu'il arrive, on décide de remplir sans latence le pipeline avec la branche résultant de notre prédiction. Cependant en cas de mis-prédiction, on est obligé d'annuler les instructions déjà entamées dans le pipeline :

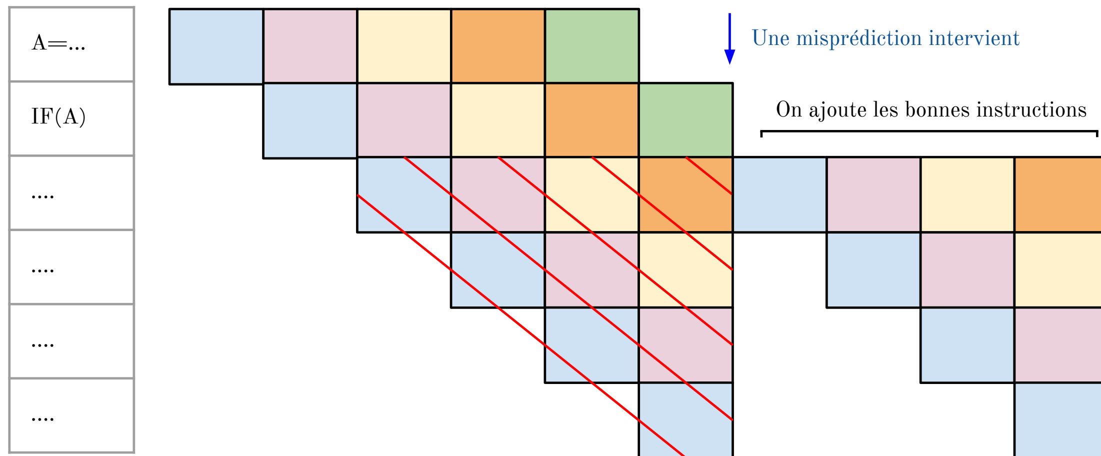



FIGURE 9 – Annulation d'instructions dans le pipeline à cause d'une misprédiction

Avec ce modèle, une misprédiction coûte donc $|pipeline| - 1 = 4\ cycles$. Sur les processeurs actuels, étant donné que le pipeline est beaucoup plus grand, le coût d'une

2. Un cycle est l'unité atomique de temps sur les processeurs. Dans notre modèle théorique, on peut considérer que c'est le temps d'exécution d'une $\mu$-instruction.

misprédiction est de l'ordre de 15 à 20 cycles.

### 2.3 Prédicteur simple saturé

Ce prédicteur consiste en un graphe à $2^k$ 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, voir figure 10.

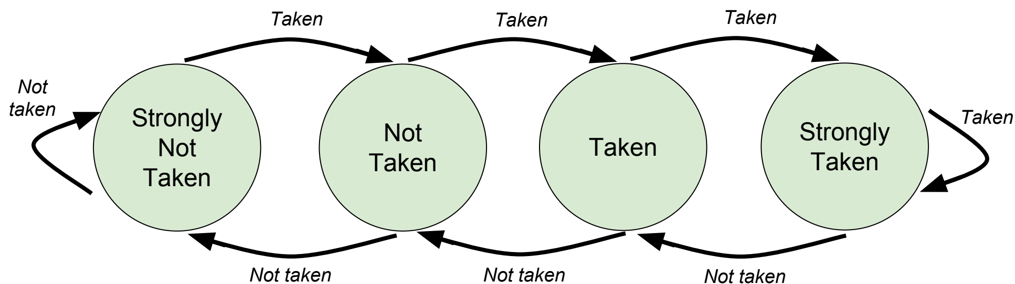



FIGURE 10 – Exemple avec un prédicteur saturé à 4 états

Si la probabilité $p$ d'effectuer un TAKEN est inchangée et indépendante, alors le papier *Good predictions are worth a few comparisons* [1] indique que le taux de misprédiction d'un tel prédicteur est donné par :

$$\mu_{k=1}(p) = 2 \times p \times (1-p) \tag{1}$$

$$\mu_{k=2}(p) = \frac{p \times (1-p)}{1 - 2 \times p \times (1-p)} \tag{2}$$

### 2.4 Prédicteur Two-Level global

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

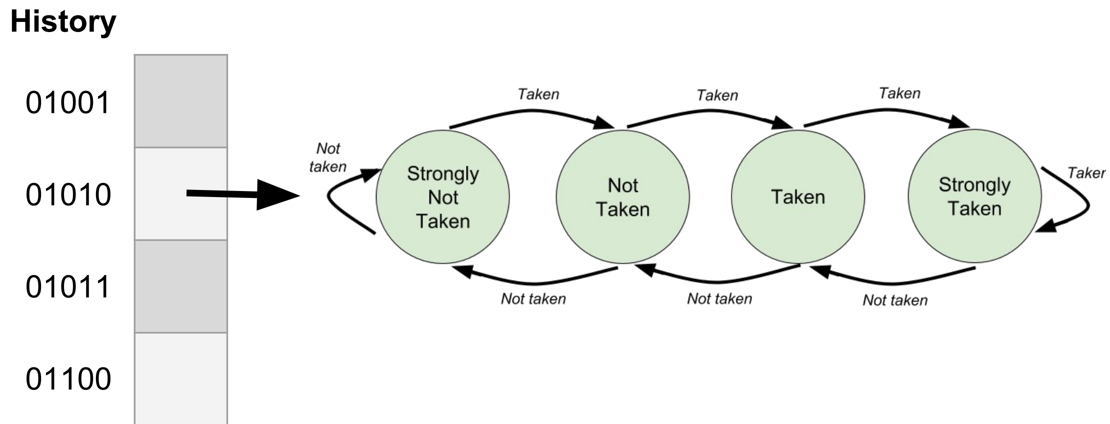



FIGURE 11 – Extrait de la table du prédicteur Two-Level avec $k = 5$

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 les branches alternent entre prise et non prise.

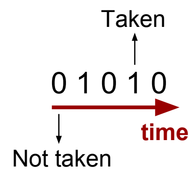



FIGURE 12 – Exemple d'un indice de la table

Des répétitions de taille $\geq k$ peuvent généralement être bien prédites lorsqu'on évite les cas pathologiques. Donner le nombre de misprédictions d'un tel prédicteur est beaucoup plus complexe.

## 2.5 Comparaison avec la réalité

Le graphique suivant est le résultat obtenu après mesure d'un benchmark travaillant sur divers tableaux randomisés. Ce benchmark inclue des branchements imbriqués et simples et une recherche naïve d'une valeur dans un ensemble. Le prédicteur réel inconnu est celui d'un Intel Core i5 de 2018.

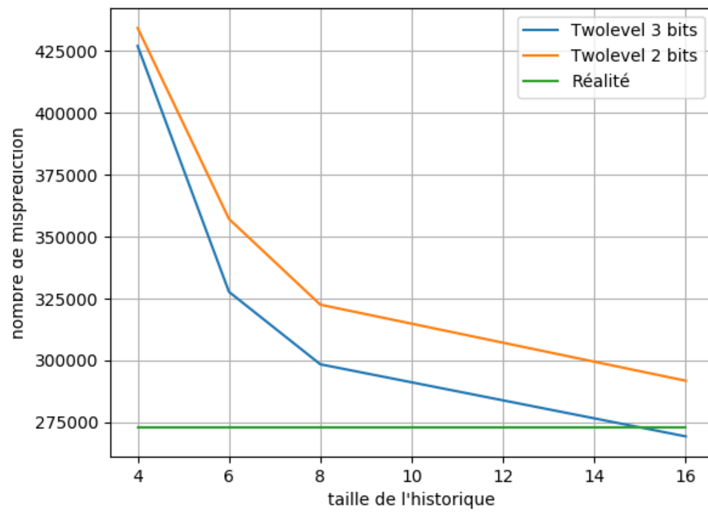



FIGURE 13 – Comparaison du prédicteur global avec la réalité.

Comme on peut l'observer sur la figure 13, le prédicteur TwoLevel de taille 16 muni de prédicteurs 3 bits est meilleur que le prédicteur réel. Cela s'explique par le fait qu'un tel prédicteur demande une grande place mémoire puisqu'il requiert $2^{16} \times 3 bits = 65536 \times 3 = 196608\ bits$ ce qui est difficile à mettre en place en pratique.

## Troisième partie

# 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 $k$.

## 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 rapidement 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 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 parallélisation 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.

## 3 L'algorithme bruteforce ou naïf

L'algorithme NAÏF, ou BRUTEFORCE [4] 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.

### 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.

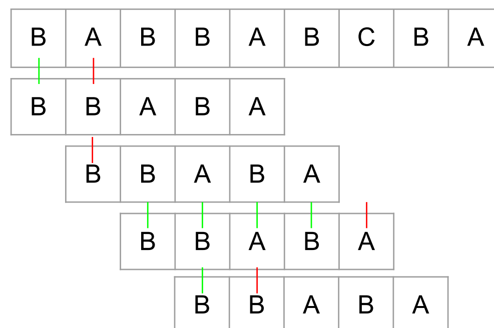



FIGURE 14 – Exemple de la recherche du motif BBABA

### 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.

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

### 3.3 Implémentation

**Algorithm 1** Recherche naïve.

```
 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
```

## 4 Introduction à KMP

### 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 :

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

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

### 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.

## 5 L'algorithme Knuth-Morris-Pratt

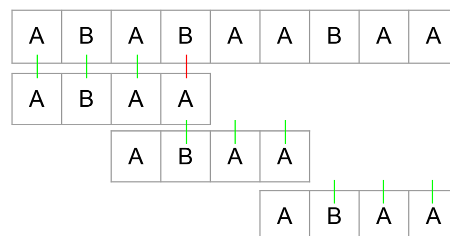
L'algorithme Knuth–Morris–Pratt [7] [3], du nom de ses inventeurs, est un algorithme basé sur la comparaison de caractères créé en 1977. Il est le premier du genre à apporter une **réelle augmentation** de la performance – avec une **complexité linéaire** – dans le domaine du pattern-matching. Il procède 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.

### 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.

### 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 15 – Exemple : table des bords de KMP pour le motif ABAA

### 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
```

### 5.4 Implémentation de la phase de recherche

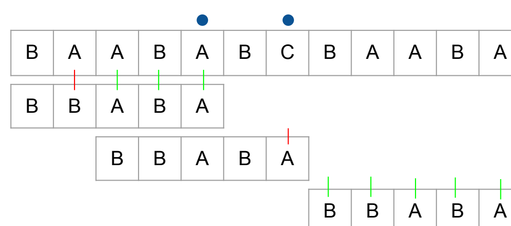
**Algorithm 3** KMP search.

```
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
```

## 6 L'algorithme d'Horspool

L'algorithme d'Horspool [6], 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**.

## 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 16 – Exemple : Pour le motif BBABA on obtient alors la table ci-dessus

## 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
```

# 7 Comparaisons expérimentales

## 7.1 Statistiques expérimentales

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 17 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). L'algorithme HORSPOOL, plus réaliste de nos jours, permet de situer la performance des deux premiers.

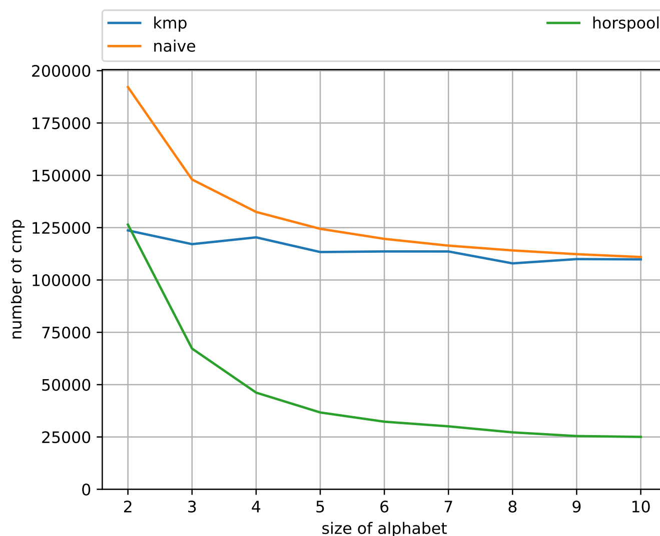



FIGURE 17 – Nombre de comparaisons des différents algorithmes présentés en fonction de la taille de l'alphabet du texte. La recherche est testée avec des motifs de différentes tailles et générés aléatoirement sur un texte de taille 100000.

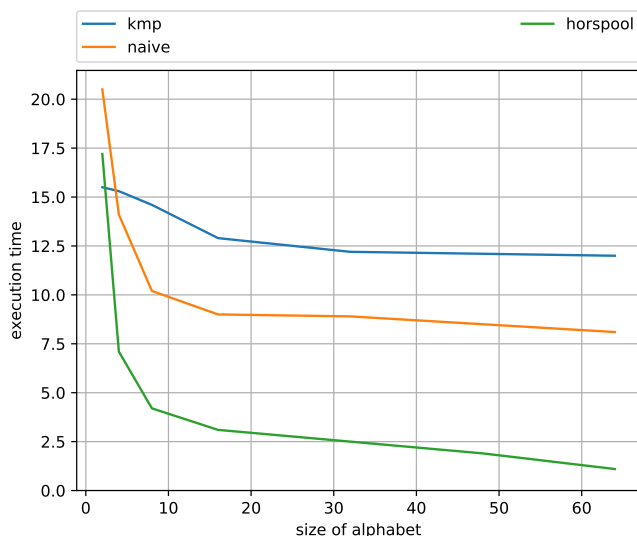



FIGURE 18 – Temps d'exécution des différents algorithmes en fonction de la taille de l'alphabet et d'un texte de taille 100000.

Sur le graphique des temps d'exécution 18, 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.

## 7.2 Algorithme naïf face à KMP

Pour rappel, on compare l'algorithme naïf de complexité quadratique dans le pire cas $\mathcal{O}(n \cdot m)$ et l'algorithme KMP de complexité linéaire $\mathcal{O}(n)$. Expérimentalement le nombre de comparaisons corresponds bien à cette complexité.

Comment expliquer cette différence de performances en temps sachant que le modèle de processeur n'a aucune influence ?

Plusieurs causes possibles sont rapidemment abandonnées :

— Cache mémoire

— Nombre de misprédictions du prédicteur de branchement

Lors de la fin de mon précédent stage, une de mes pistes était alors que l'algorithme KMP entrainait une plus forte latence dans le pipeline de part les interdépendances sémantiques ou structurelles. L'idée était alors de construire un simulateur de pipeline (qui s'est transformé en simulateur de processeur) pour essayer de comprendre si cela confirmait notre hypothèse et savoir plus précisément quelles instructions étaient visées.

## 7.3 La problématique de l'implémentation

Établir un algorithme, même simple, est une chose ; mais la méthode d'implémentation a une importance considérable. Dans notre cas par exemple, comparer naïvement

les caractères par blocs de 8 octets entraine une accélération considérable sans pour autant modifier le déroulé de l'algorithme. On se retrouve alors avec un algorithme plus *mécanique* qu'intelligent. Un algorithme comme le NAÏF écrit à l'envers – c'est-à-dire comparant les caractères de droite à gauche – améliore les performances sans pour autant qu'une explication fiable puisse être avancée.

De nos jours, les algorithmes les plus performants comme TWO-WAY [5] – version de référence de la libC – sont algorithmiquement proche du NAÏF mais profitent d'une large accélération liée à la mémoire notamment par des comparaisons de gros blocs de textes. Les systèmes liés à la mémoire comme les caches étant largement étudiés dans la littérature, nous nous intéresserons plutôt à d'autres aspects moins connus.

Quatrième partie

# Simulateur de Processeur

De part l'impact des différentes architectures sur la performance que l'on a pu observer sur l'exécution des algorithmes, il est intéressant de se poser la question du comportement pour un modèle d'architecture des processeurs.
Pour faciliter leur compréhension ainsi que pour visualiser l'impact sur un algorithme, on a décidé la création d'un simulateur de processeur générique. En effet, il existe une multitude d'architectures différentes et connaître le fonctionnement interne exact d'un modèle de processeur est aujourd'hui impossible.

## 1 Intérêts et Objectifs

L'objectif initial du simulateur était de comparer le comportement du pipeline avec nos deux algorithmes NAÏF et KMP pour savoir si leur différence de performances pouvait provenir de cet outil.
Par la suite, il s'est peu à peu transformé en outil pédagogique de compréhension des différents éléments de l'architecture des ordinateurs et en outil de statistiques réelles sur l'exécution d'un programme.

Par mesure de simplification, un programme est une fonction écrite en assembleur, comme par exemple la fonction de recherche du NAÏF ou une dichotomie. Cette fonction travaille sur deux sources : un texte et un motif *randomisés* de taille variable ainsi qu'un motif fixé par l'utilisateur (par exemple une constante ou une chaîne de caractère), voir figure 19.

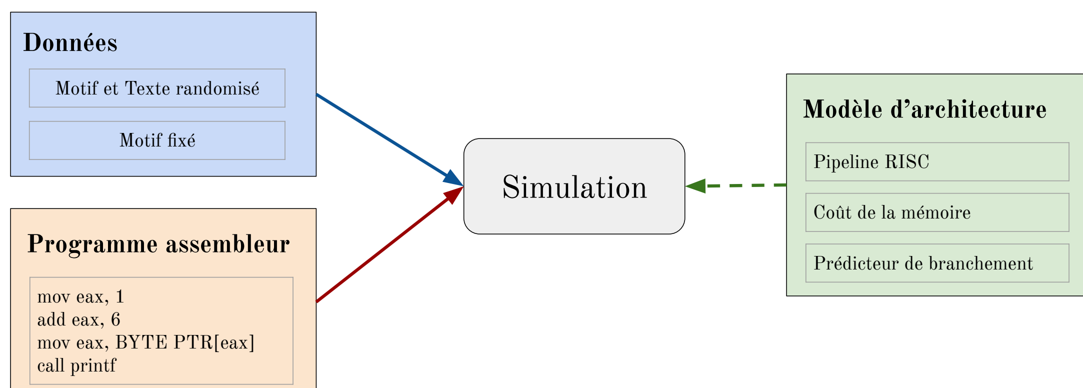



FIGURE 19 – Utilisation du simulateur avec ses entrées.

Au niveau de la personnalisation de l'architecture de processeur, le pipeline RISC utilisé est celui présenté précédemment **avec les shortcuts**. Le coût mémoire est de 1 cycle par défaut mais il est possible de rajouter du temps pour simuler une vraie mémoire RAM. Toutefois, en pratique aujourd'hui, le coût d'un accès cache est proche d'un coût registre. Le prédicteur de branchement peut être facilement changé d'un saturé simple à un global et il est possible de coder soit même son propre prédicteur.

Comme le simulateur est avant tout à but pédagogique, plusieurs exemples de programmes sources sont fournis pour bien comprendre les fonctionnalités assembleurs disponibles.

# 2 Architecture interne

Le simulateur est codé en *Python 3* ce qui le rend multiplaforme et performant avec des outils comme PyPy [3].

Le simulateur est divisé en divers modules :

— Parseur d'assembleur

— Simulateur de processeur

— Interface graphique grâce à tkinter [4]

## 2.1 Fonctionnalités du langage assembleur

Le langage assembleur utilisé pour la réalisation des programmes sources est une simplification du language NASM [5].

Les **Instructions supportées** dans le code assembleur sont les suivantes :

| mov | sub | add | mul | jmp | jne | je | jl | js | cmp | jcmp | call |
|---|---|---|---|---|---|---|---|---|---|---|---|

Le langage supporte les fonctionnalités suivantes :

— Les lignes commençant par '.' sont ignorées, excepté les headers

— Commentaires avec le mot clé #

— Chaînes constantes pour faciliter l'appel à print

— Labels utilisés pour les sauts

— Registres invariants en taille

3. PyPy est un JIT pour Python donc les performances sont proches du C : `http://pypy.org/`
4. Interface graphique de référence de Python : `http://tkinter.fdex.eu/`
5. Site officiel NASM : `https://www.nasm.us/`

## 2.2 Simulateur de processeur

Le simulateur permet l'exécution réelle d'une série d'instructions. Plusieurs composants sont configurables, les données affichées sur la figure 21 sont les valeurs par défaut.

Il est possible de paramétrer :

— la *quantité de mémoire* du tas

— le *nombre de variables de pile*

Quant aux registres, en plus de ceux de bases déjà présents en NASM, il est à noter qu'il en existe une infinité et que donc on peut accéder aux registres de la forme $\forall n \in \mathbb{N}^+, r_n$.

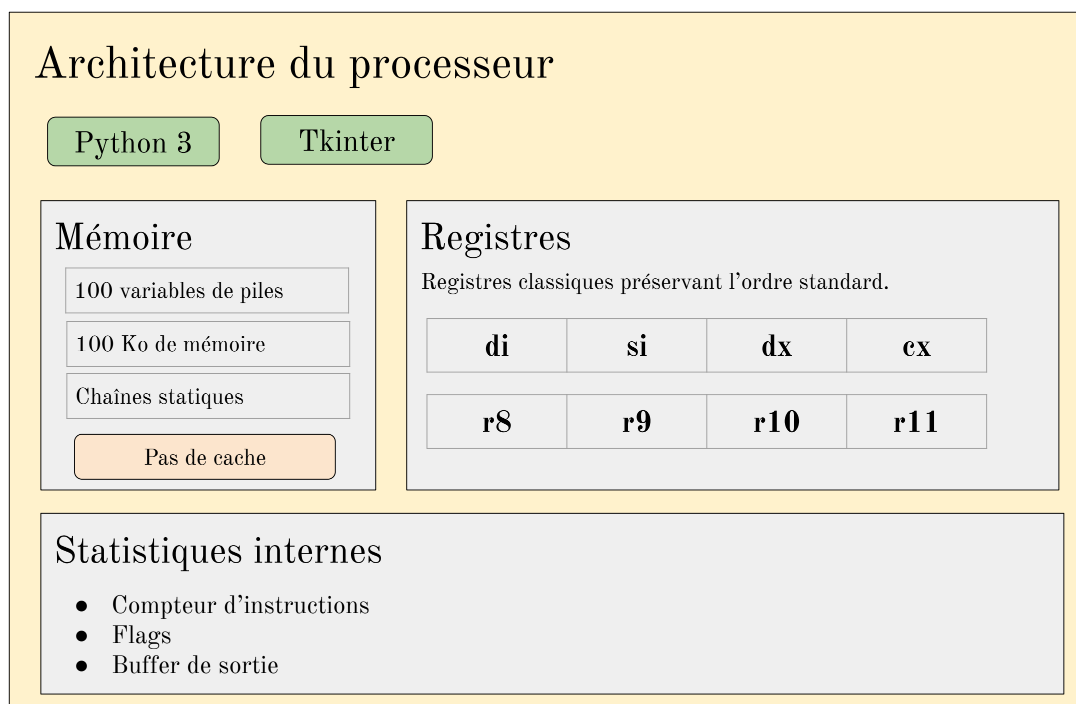



FIGURE 20 – Architecture interne du processeur.

## 2.3 Interface graphique

L'interface graphique est divisée en deux, d'une part le code assembleur actuellement chargé où on peut facilement voir :

— la prochaine instruction à être exécutée

— l'état du pipeline

— la quantité de latence introduite pour chaque instruction

Le menu à droite permet de lancer ou stopper la simulation à plusieurs niveaux de vitesse. Toutes les statistiques du simulateur de processeur sont disponibles et il est possible d'activer ou désactiver :

— l'interdépendance structurelle

— l'interdépendance sémantique

— la latence mémoire

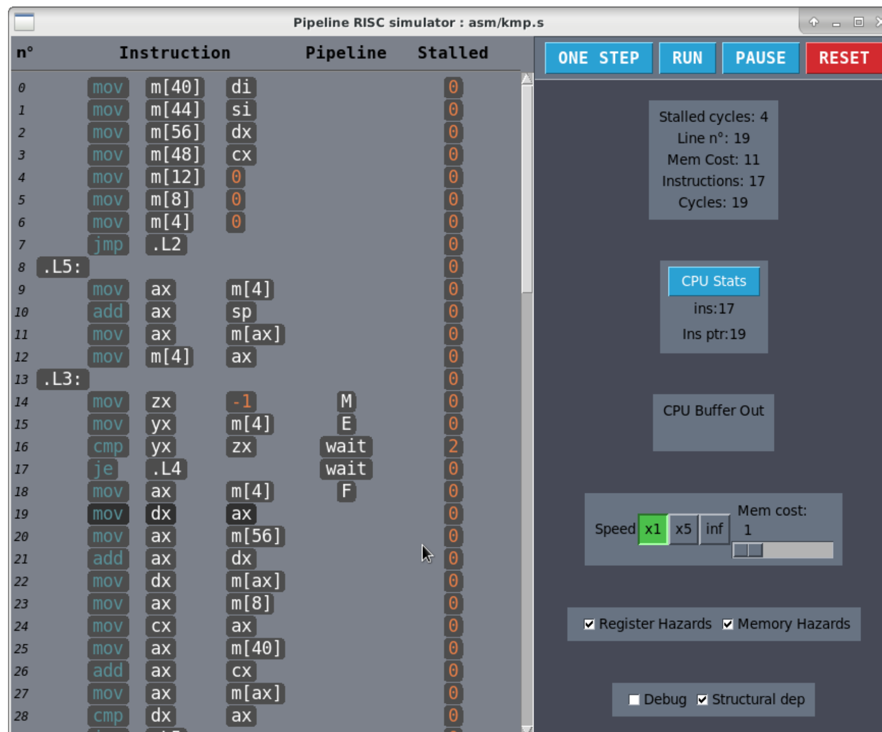



FIGURE 21 – Interface graphique du simulateur réalisée avec tkinter.

## Cinquième partie

# Introduction à l'analyse théorique de KMP

## 1 Comparaisons du nombre de branches

Pour rappel, on essaie d'expliquer pourquoi les temps d'exécution de KMP sont largements plus importants que ceux du NAÏF lorsque les alphabets sont importants.
Le simulateur de processeur mis en place, nous avons pu observer quelles instructions étaient impactées par des bulles. Le résultat est sans équivoque, les instructions de KMP qui suivent l'accès à la table des bords sont systématiquement ralenties.

L'idée est que le nombre de prises de branchements de KMP est beaucoup plus important. En effet, les deux algorithmes sont écrits dans le sens contraire, le NAÏF avance lorsque la comparaison rate alors que KMP accède à la table des bords dans ce cas. Le nombre de branches prises étant directement liées à la réussite d'une comparaison, il est tout naturel d'accéder à la table des bords avec une probabilité $\frac{k-1}{k}$ alors que la branche parallèle dans le NAÏF n'est prise qu'avec une probabilité $\frac{1}{k}$.

Ceci est largement confirmé par les résultats expérimentaux de comptage de branches et d'accès à la table des bords, voir figure 22 :

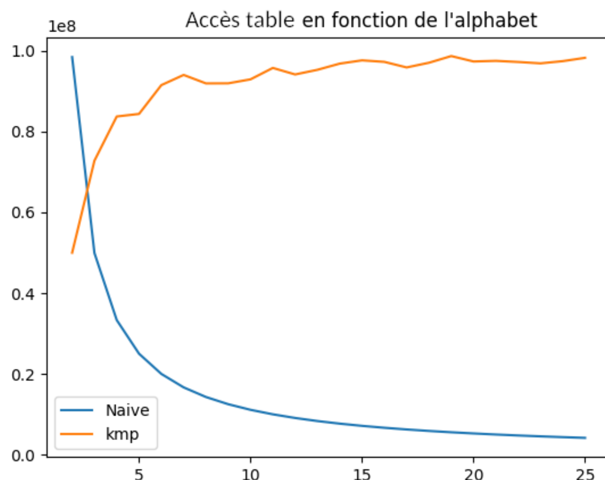
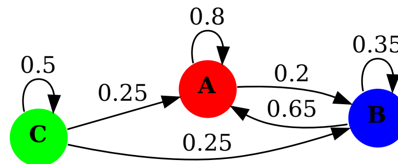



FIGURE 22 – Accès à la table de décalage en fonction de l'alphabet

## 2 Introduction générale aux chaînes de Markov

### 2.1 Présentation

Une chaîne de MARKOV [6] 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.

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 :

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

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

## 2.2 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 :

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

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 :

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

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 :

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

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 [7] – en $\mathcal{O}(n^3)$ – pour résoudre le système.

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

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

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

## Sixième partie

# Analyse du prédicteur global avec KMP

L'objectif de cette partie est d'analyser le comportement du prédicteur global présenté section 2.4, étant le prédicteur le plus proche de la réalité pour permettre d'obtenir un algorithme non simulant donnant le nombre de misprédictions à partir d'un motif. Le déroulé de cette partie est le suivant :

1. Construire une chaîne de Markov représentant le comportement de l'algorithme
2. Introduire le comportement du prédicteur global
3. Simplifier algorithmiquement la chaîne
4. Calculer des statistiques : misprédiction, nombre de branches...

L'objectif est donc de construire la chaîne en fonction d'un motif, d'un alphabet ainsi que la taille d'historique et de prédicteur.

## 1 Convention de nommage

Étant donné que l'on va créer des chaînes complexes, on définit un noeud comme suit :

FIGURE 23 – Un noeud de notre chaîne

noté $\begin{pmatrix} color & suffix & nif & predicteur \end{pmatrix}$ et où on a :

- **color** : Nombre de lettres éliminées dans la cadre d'une recherche de bord
- **suffix** : Suffixe du motif déjà lu
- **nIf** : Numéro du branchement en cours
- **predicteur** : Etat du prédicteur courant, voir section 3. Non utilisé dans un premier temps.

Pour rappel les branchements sont les suivants :

- n°0 : `while(j < len(text))`
- n°1 : `if(i > -1)`
- n°2 : `if(pat[i] != text[j])`
- n°3 : `if(i == len(pat))`

## 2 Construction de la chaîne

La chaîne sera donc constituée d'une multitude de noeuds reliés par des arcs représentant des probabilités.
Il est à remarquer immédiatement deux types de noeuds :

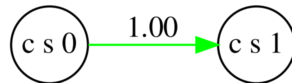
- les états déterministes qui correspondent aux IF n°0, n°1 et n°3. Puisque ces branches n'ont aucune intéraction avec les lettres du texte et qu'on connait le motif, le prochain état est donc connu avec certitude (probabilité de 1).
- les états de comparaisons qui correspondent au IF n°2. On gère les deux cas possibles : match du caractère avec une probabilité $p = \frac{1}{k}$ ou mismatch avec une probabilité $p = \frac{k-1}{k}$.

**Par exemple**, Prenons le motif ABAA sur l'alphabet ABC. Sa table des bords de KMP est la suivante :

| -1 | 0 | -1 | 1 | 1 |
|---|---|---|---|---|

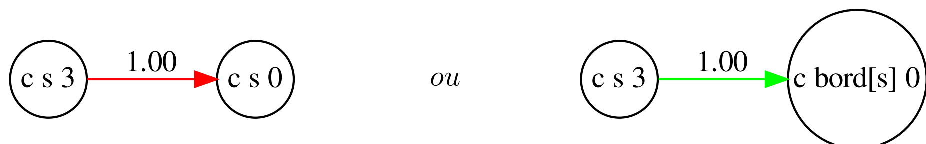
### 2.1 États déterministes

Le numéro $nIf_{next}$ d'un noeud déterministe est donc connu, il vaut 1 pour le premier branchement $nIf = 0$ :




- Pour $nIf = 1$, il vaut $nIf_{next} = 2$ et le branchement est vrai si $bord[s] > -1$, sinon $nIf_{next} = 3$.

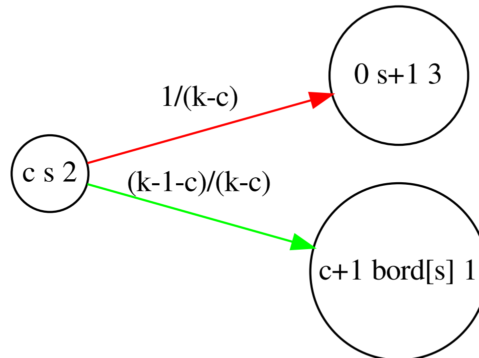
- Pour $nIf = 3$ il vaut toujours $nIf_{next} = 0$ mais le branchement est vrai ou faux selon que le motif est entièrement lu. De plus, si le motif est trouvé, on a $suffix_{next} = bord[suffix]$ et reste inchangé dans le cas contraire.



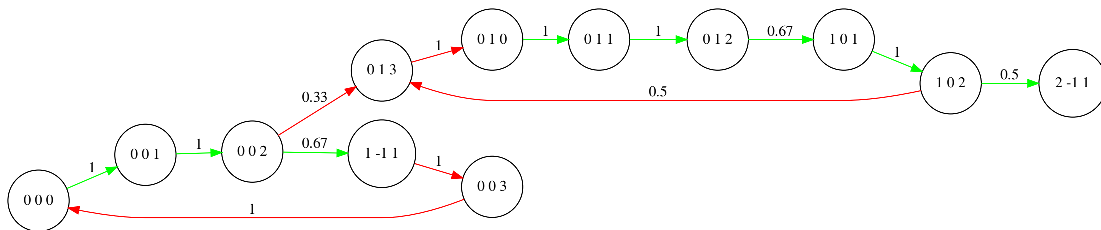

### 2.2 État non déterministe

Le seul état non déterministe est l'état de la forme $\begin{pmatrix} c & s & 2 \end{pmatrix}$ et correspond au branchement de comparaison de caractère. C'est le seul moment non prévisible dans notre chaîne.

Le branchement est alors pris si il y a mismatch, avec une probabilité de $p = \frac{k-1-c}{k-c}$. Dans ce cas, on retourne dans le branchement n°1 avec une couleur additionnelle étant donné qu'on cherche un bord non suivi de la même lettre (table de KMP. En cas de match, on sort de la boucle et on passe au branchement n°3.




Grâce à ces règles, on peut alors construire le graphe complet. **Par exemple**, le début du graphe correspondant au motif ABAA sur l'alphabet ABC est :




En calculant la loi stationnaire de ce graphe, on peut alors obtenir quelques statistiques sur l'exécution de notre algorithme avec ce motif :

— Nombre de branchements pris et non-pris

— Nombre de comparaisons

— Nombre d'accès à la table des bords par comparaison

## 3 Ajout de la prédiction de branchement

Dans cette section, on s'intéresse à rajouter les informations liées au prédicteur global présenté précédemment. Pour ce faire, on doit rajouter, dans chaque noeud, l'état du prédicteur (c'est-à-dire sa table d'historique ainsi que les états de tous les prédicteurs saturés).

Dans la suite de cette section, nous travaillerons sur une taille d'historique de 2 et des prédicteurs saturés de 1 bit [8].

## 3.1 Rajout du prédicteur global

Dans un noeud il faut encoder l'état de chaque prédicteur saturé, que l'on notera 0 pour NOT-TAKEN et 1 pour TAKEN. C'est identique pour la table d'historique, voir figure 24 :

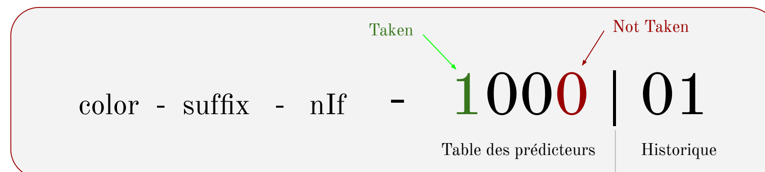
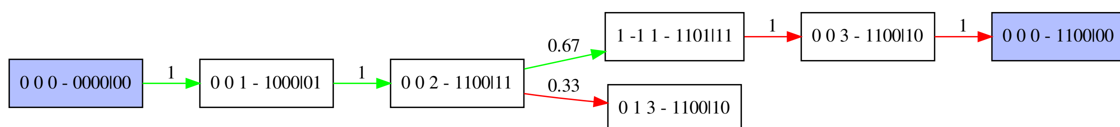



FIGURE 24 – Ajout du prédicteur global dans un noeud du graphe

Par convention, on initialise notre premier noeud avec des prédicteurs à NOT-TAKEN et un historique de valeur 0. On met ensuite à jour le prédicteur à chaque changement d'état.
Si on reprend l'exemple précédent (avec des rectangles pour plus de lisibilité) :




Comme on peut l'observer immédiatement en bleu, le noeud $\begin{pmatrix}0 & 0 & 0\end{pmatrix}$, auparavant unique, est maintenant divisé en deux noeuds à cause de l'état des prédicteurs : $\begin{pmatrix}0 & 0 & 0 & 0000|00\end{pmatrix}$ et $\begin{pmatrix}0 & 0 & 0 & 1100|00\end{pmatrix}$.

## 3.2 Calcul de la misprédiction

Dans un premier temps il faut effectuer le calcul de la loi stationnaire $\pi$ tel qu'expliqué section 2.2. Pour rappel, un arc émet une misprédiction si la prédiction liée à son noeud parent est différente de la prise du branchement relatif à l'arc, comme par exemple si la prédiction est NOT-TAKEN et qu'on est sur un arc vert (TAKEN).

8. En pratique, ces valeurs ne sont pas réalistes, mais cela permet de rester clair pour expliquer avec de petits graphes

La formule suivante permet de résumer le calcul. $p(u)$ est la probabilité liée à l'arc vert qui sort de l'état $u$ (0 s'il n'existe pas).

$$\frac{1}{\sum_u^{\mathbb{U}} \pi_u} \times \sum_u^{\mathbb{U}} \begin{cases} \pi_u \times p(u) & si\, pred(u)\ est\ \text{NT}\ ou\ \text{NNT} \\ \pi_u \times (1 - p(u)) & sinon \end{cases}$$

# 4 Simplification du graphe

Le problème immédiat est la taille du graphe. En effet, avec des prédicteurs 1 bit on obtient des tailles extrêmement grandes, voir figure 25 et, étant donné que le calcul de la loi stationnaire demande une résolution de système linéaire en $\mathcal{O}(n^3)$, cela devient alors irréalisable :

| | | Taille du motif | | | |
|---|---|---|---|---|---|
| | | 6 | 12 | 20 | 40 |
| taille de l'historique | 6 | 3000 | 5000 | 7000 | 12000 |
| | 8 | 120000 | 163000 | 149000 | 273000 |

Figure 25 – Taille du graphe en fonction de la taille du motif et de la taille de l'historique $l$.

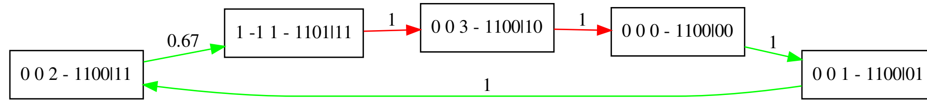
On cherche alors à minimiser la taille du graphe. Deux possibilités s'offrent à nous.

## 4.1 Minimisation calée sur les comparaisons

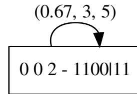
Une des premières évidences est la suppression pure et simple des états déterministes. On souhaite donc obtenir uniquement des états liés au $if\ n°2$ qui resteront avec une ou deux sorties.
Il faut cependant faire attention à bien conserver les informations nécessaires au calcul futur de la misprédiction, celle-ci dépendant directement du nombre de branches. On va alors devoir rajouter deux informations supplémentaires à chaque arc : le nombre de misprédictions effectuées pendant le parcours des états déterministes et le nombre total de branches parcourues.
Les nouveaux arcs sont donc de la forme $(p \quad mis \quad branch)$ avec $R$ l'ensemble des noeuds déterministes rencontrés :

- la probabilité de l'arc vaut $p = \prod_u^R p_u$
- le nombre de misprédictions est la somme des misprédictions pour chaque noeud
- le nombre de branches est $branch = |R|$.

Par exemple la boucle suivante :




devient :




En applicant cette méthode pour chaque noeud de ce type, on divise la taille du graphe par 4 en moyenne.

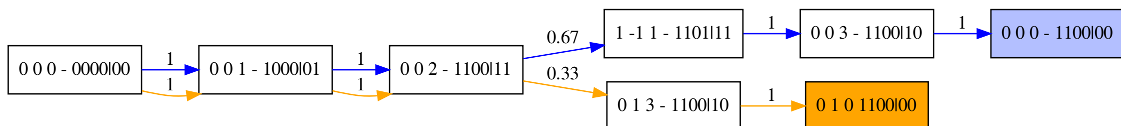
## 4.2 Minimisation calée sur la découverte du texte

La minimisation précédente pose le problème suivant : on obtient des statistiques en fonction du nombre de comparaisons. Par exemple, le nombre de misprédictions par comparaison. Ce qui peut vite freiner l'analyse. L'objectif de cette nouvelle simplification est de caler chaque état sur la découverte d'une nouvelle lettre du texte, c'est-à-dire le branchement n°0.

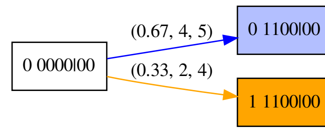
L'idée est de partir d'un état $nIf = 0$ et d'effectuer un parcours en largeur et de relier directement à ce premier état tous les autres noeuds $nIf = 0$ que l'on trouvera. Les noeuds pourront alors avoir plus de deux arcs sortants. À noter que l'on trouve forcément au moins un noeud lors du parcours, étant donné que l'on avance inévitablement dans le texte – c'est-à-dire qu'on ne tourne pas en rond – avec KMP.

Toujours de la même manière, lors de la simplification, il faut stocker sur les nouveaux arcs les informations que l'on souhaite conserver. Par exemple, comme précédemment : $(p \quad mis \quad branch)$. Étant donné que sur les noeuds correspondant au $nIf = 0$, les valeurs $nIf$ et $color$ sont toujours égales à 0, nous faisons disparaitre ces informations.

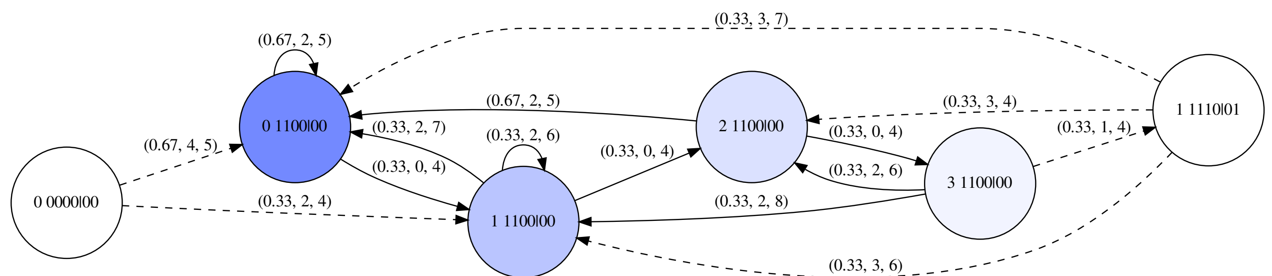
Reprenons une partie du graphe de départ correspondant au motif ABAA. Ci-dessous, on a tracé les deux chemins trouvés par le parcours, respectivement en bleu et en orange.




Après contraction des arcs et accumulations des informations de misprédiction et du nombre de branches, on obtient :




Le **graphe entier**, pour le motif ABAA sur l'alphabet ABC avec un prédicteur global TwoLevel de taille d'historique 2 muni de prédicteurs saturés 1 bit, est alors le suivant :




On peut observer notamment la force de la loi stationnaire en bleu : le noeud $\begin{pmatrix}0 & 1100|0\end{pmatrix}$ est le plus emprunté ; on peut également voir les moments où il y a des modifications dans la table des prédicteurs, représenté par des pointillés. Excepté le premier état qui n'est emprunté qu'une seule fois, on peut donc voir que ces modifications n'interviennent qu'à certains moments particuliers : quand on lit le motif ou quand on itère sur les bords.

## 4.3 Tarjan et résultats de la minimisation

Les méthodes précédentes permettent de diviser par 4 la taille de notre graphe ; elles s'appliquent alors sur le fonctionnement de l'algorithme. Il est alors également intéressant de vouloir appliquer en dernier une méthode générique. Étant donné que la loi stationnaire équivaut à effectuer une marche aléatoire infinie dans notre graphe, une situation où un seul état ou un groupe d'états ne peuvent plus être revisités revient à dire que ces noeuds auront une loi stationnaire nulle.

Dans notre cas, ces noeuds sont alors inutiles et peuvent être retirés avec l'algorithme de Tarjan [10] qui permet de récupérer la composante connexe principale.

|  | Motif $m$ | | | |
|---|---|---|---|---|
|  | ABAA | $\|m\| = 10$ | $\|m\| = 20$ | $\|m\| = 80$ |
| Graphe de base | 72480 | 105647 | 273000 | 733209 |
| Minimisation 4.1 | 16014 | 24208 | 63526 | 180758 |
| Minimisation 4.2 | 14787 | 22109 | 58883 | 174115 |
| Tarjan sur minimisation 4.2 | 1009 | 1429 | 3286 | 9830 |
| Temps minimal de calcul | 1.5sec | 3.5sec | 53sec | 15min |

FIGURE 26 – Taille du graphe en fonction du motif et pour une taille d'historique $l = 8$.

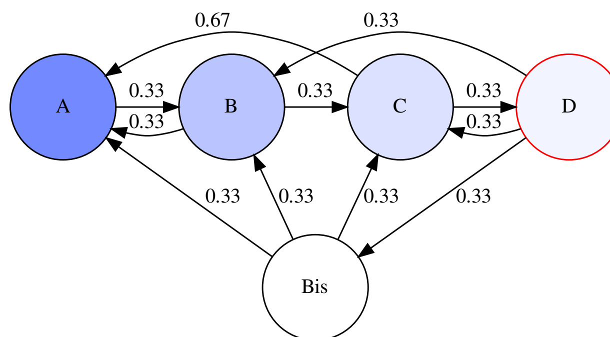
L'algorithme de Tarjan est très efficace ici car notre graphe est composé de plusieurs grandes composantes ; les états du prédicteurs sont directement à mettre en cause et une sorte de stabilité se met en place, ce qui fait que l'on finit par tourner dans un ensemble restreint d'états de prédicteurs.

# 5 Propagation inversée de la loi stationnaire

Même fortement minimisés, les graphes atteignent plusieurs milliers d'états et le calcul de la loi stationnaire est en $\mathcal{O}(n^3)$. De manière générale, il n'est pas possible de faire mieux ; cependant nos chaînes sont très particulières et ont une forme bien spécifique. On est alors capable de dire des choses sur l'algorithme.

## 5.1 Principe de la propagation

Prenons l'exemple d'un graphe simplifié de la forme qui nous intéresse :




Il y a plusieurs observations à faire :

— Le noeud $\mathcal{D}$ correspond à la lecture d'un motif moins une lettre, on connait donc sa loi stationnaire : $\pi_{\mathcal{D}} = \frac{1}{k^{m-1}}$ qui est la probabilité de trouver le motif sans la dernière lettre.

— $\forall u \in G, u$ a un seul arc entrant (comme $\mathcal{D}$ ou $\mathcal{B}is$) OU l'ensemble de ses arcs entrants sauf un sont déductibles par propagation.

## 5.2 Déroulé

La propagation se déroule de la manière suivante pour déduire dans l'ordre : $\pi_{\mathcal{D}} \to \pi_{\mathcal{B}is} \to \pi_{\mathcal{C}} \to \pi_{\mathcal{B}} \to \pi_{\mathcal{A}}$ :

1. On connait $\pi_{\mathcal{D}}$, donc on peut en déduire immédiatement $\pi_{\mathcal{B}is} = \frac{1}{3}\pi_{\mathcal{D}}$.
2. On peut aussi en déduire $\pi_{\mathcal{C}}$ puisque $\pi_{\mathcal{D}} = \frac{1}{3}\pi_{\mathcal{C}}$ alors $\pi_{\mathcal{C}} = 3\pi_{\mathcal{D}}$.
3. Pour $\pi_{\mathcal{B}}$, on suit simplement le même principe :

$$\pi_{\mathcal{C}} = \frac{1}{3}\left(\pi_{\mathcal{D}} + \pi_{\mathcal{B}is} + \pi_{\mathcal{B}}\right) \tag{3}$$

$$\Leftrightarrow \qquad \pi_{\mathcal{B}} = 9\pi_{\mathcal{D}} - \pi_{\mathcal{D}} - \frac{1}{3}\pi_{\mathcal{D}} \tag{4}$$

$$= \frac{23}{3}\pi_{\mathcal{D}} \tag{5}$$

4. Enfin pour le dernier état :

$$\pi_{\mathcal{A}} = \frac{2}{3}\pi_{\mathcal{C}} + \frac{1}{3}\pi_{\mathcal{B}} + \frac{1}{3}\pi_{\mathcal{B}is} \tag{6}$$

$$= \left(2 + \frac{23}{3} + \frac{1}{9}\right)\pi_{\mathcal{D}} \tag{7}$$

$$= \frac{88}{9}\pi_{\mathcal{D}} \tag{8}$$

Toutes les lois stationnaires sont donc exprimables en fonction de $\pi_{\mathcal{D}} = \frac{1}{k^{m-1}}$ et sont donc calculables linéairement de la taille du graphe si les conditions citées au dessus sont valides.

**Problème :** dans notre cas, cela marche très bien tant que l'on ne rajoute pas dans le graphe les états de prédicteurs : en effet, ceux-ci vont dupliquer certains noeuds et on va alors se retrouver avec plusieurs noeuds correspondant à notre $\mathcal{D}$ précédent. On ne connaîtra alors plus $\pi_{\mathcal{D}}$.

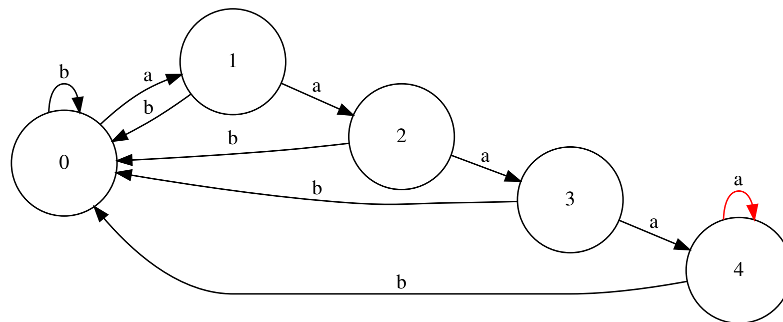
## 6 Analyse fine d'un motif précis

Dans la suite de cette section, nous étudierons le motif AAAAA (5) de table KMP

| -1 | -1 | -1 | -1 | -1 | 4 |
|---|---|---|---|---|---|

sur un prédicteur global 2 bits d'historique **8**. Pour simplifier l'explication, on dit que lire un $a$ équivaut à lire une lettre du motif et que lire un $b$ revient à lire tout autre lettre de l'alphabet de taille $k$.

L'idée dans un premier temps est de construire l'automate déterministe de reconnaissance du motif. L'arc rouge correspond à la lecture du motif.

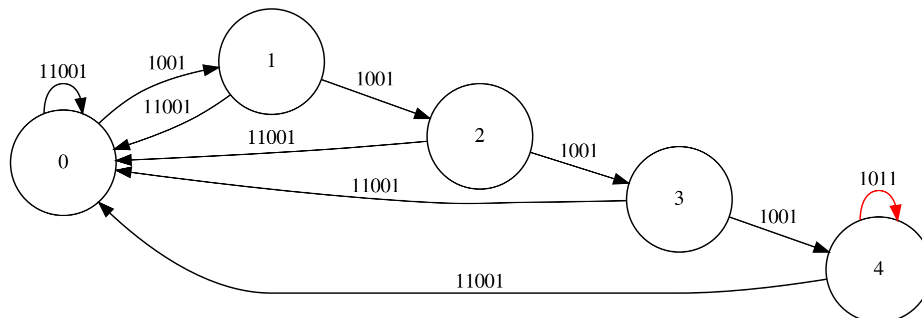



Ensuite, on va placer pour chaque arête les valeurs des branchements pris : 1 pour un branchement pris et 0 sinon. Il y a trois cas possibles :

— On lit un B : `11001`

— On lit un A mais on ne lit pas le motif : `1001`

— On lit le motif : `1011`




On va ensuite lister toutes les mots de taille au minimum 9 qu'il est possible de former. On note $\bar{a}$ lorsque l'on lit le motif.

$bbb$ : `110011100111001`
$bba$ : `11001110011001`
$bab$ : `11001100111001`
$baa$ : `1100110011001`
$abb$ : `10011100111001`
$aba$ : `1001110011001`
$aab$ : `1001100111001`
$aaa$ : `100110011001`

$aa\bar{a}$ : `100110011011`
$a\bar{a}\bar{a}$ : `100110111011`
$\bar{a}\bar{a}\bar{a}$ : `101110111011`
$a\bar{a}b$ : `1001101111001`
$\bar{a}\bar{a}b$ : `1011101111001`
$\bar{a}bb$ : `10111100111001`
$\bar{a}ba$ : `1011110011001`

À partir de celà, on va pouvoir récupérer toutes les formes d'historique qu'il est possible de croiser lors de la recherche de ce motif et étudier si le branchement suivant est pris ou non, puisque c'est justement l'objectif du prédicteur. On peut faire quelques observations immédiates :

— Les `0` apparaissent seuls ou par deux.
— Un `1` n'apparait jamais seul : `010`.
— Il n'y a jamais cinq `1` de suite.

Ce qui implique pour commencer, les régles suivantes :

— `------00` $\rightarrow$ 1.
— `------01` $\rightarrow$ 1.
— `----1111` $\rightarrow$ 0.

Ces cas ne génèrement pas de misprédiction car ils sont parfaitemement prédits par un prédicteur, leurs branchements étant toujours identiques. Il nous reste trois cas à traiter : les historiques se terminant par 10, 011 ou 0111.

Les autres cas se résument facilement dans le tableau suivant :

<table>
<tr><th>Motif précédent</th><th>Case mise à jour</th><th>Branchement suivant</th><th>Cas</th></tr>
<tr><td>$a\bar{a}$ ou $aa$</td><td><code>01100110</code></td><td>$\begin{cases} 0 & si\ \boldsymbol{a} \\ 1 & si\ \boldsymbol{\bar{a}} \end{cases}$</td><td>n°1</td></tr>
<tr><td>$a$</td><td><code>00110011</code></td><td>$\begin{cases} 0 & si\ \boldsymbol{a}\ ou\ \boldsymbol{\bar{a}} \\ 1 & si\ \boldsymbol{b} \end{cases}$</td><td rowspan="3">n°2</td></tr>
<tr><td>$a\bar{a}$</td><td><code>00110111</code></td><td rowspan="2">$\begin{cases} 0 & si\ \boldsymbol{\bar{a}} \\ 1 & si\ \boldsymbol{b} \end{cases}$</td></tr>
<tr><td>$\bar{a}\bar{a}$</td><td><code>01110111</code></td></tr>
<tr><td>$\bar{a}b$</td><td><code>11110011</code></td><td rowspan="2">$\begin{cases} 0 & si\ \mathbf{a} \\ 1 & si\ \boldsymbol{b} \end{cases}$</td><td rowspan="2">n°3</td></tr>
<tr><td>$ab$ ou $bb$</td><td><code>01110011</code></td></tr>
</table>

— le cas n°**2** équivaut à un prédicteur simple saturé où le TAKE est la lecture d'un $b$ et le NOT-TAKE est la lecture d'un $a$ sachant qu'on a lu un $a$ précédemment. La probabilité d'un NOT-TAKE est alors $\frac{1}{k}$ et ce prédicteur est appelé lorsque l'on lit un $a$ donc avec une probabilité de $\frac{1}{k}$. Le taux de misprédiction[9] de ce cas est alors

$$mis_2 = n \times \frac{1}{k} \times \mu_2(\frac{1}{k})$$

— le cas n°**3** équivaut au même prédicteur à la différence que celui-ci est appelé plus souvent (on lit un $b$) donc :

$$mis_3 = n \times \frac{k-1}{k} \times \mu_2(\frac{1}{k})$$

On peut observer que :

$$mis_2 + mis_3 = n \times \mu_2(\frac{1}{k})$$

— le cas n°**1** est plus complexe. On effectue un TAKE lorsque qu'on lit le motif et un NOT-TAKE lorsque l'on lit $aa$ mais qu'on ne trouve pas le motif. En règle générale, notre prédicteur sera bloqué en NOT TAKED mais il est possible le saturer dans l'autre sens si on trouve beaucoup d'occurences du motif.
Notre recherche du motif équivaut à reconnaître le langage suivant :

$$\left[\left(\epsilon + a + aa + aaaa + aaaa\,(\bar{a})^{+}\right) b\right]^{*}$$

Pour calculer le taux de misprédiction associée au cas n°**1**, on va construire une petite chaîne de MARKOV représentant les états de notre prédicteur. Chaque arc contiendra la probabilité de lire $n$ caractères $a$ de notre motif, noté $P_i = \frac{1}{k^i}$ et on a $P_0 = \frac{k-1}{k}$. On définit également :

$$\alpha_j = \sum_{i=j}^{m-1} P_i \qquad (9)$$

$$\beta = \sum_{i \geq m+2} P_i \qquad (10)$$

Dans ce graphe, on écrira la probabilité $\alpha_j$ ou $\beta$ pour plus de lisibilité.

9. On utilise la formule de la section 2.3 du deuxième chapitre

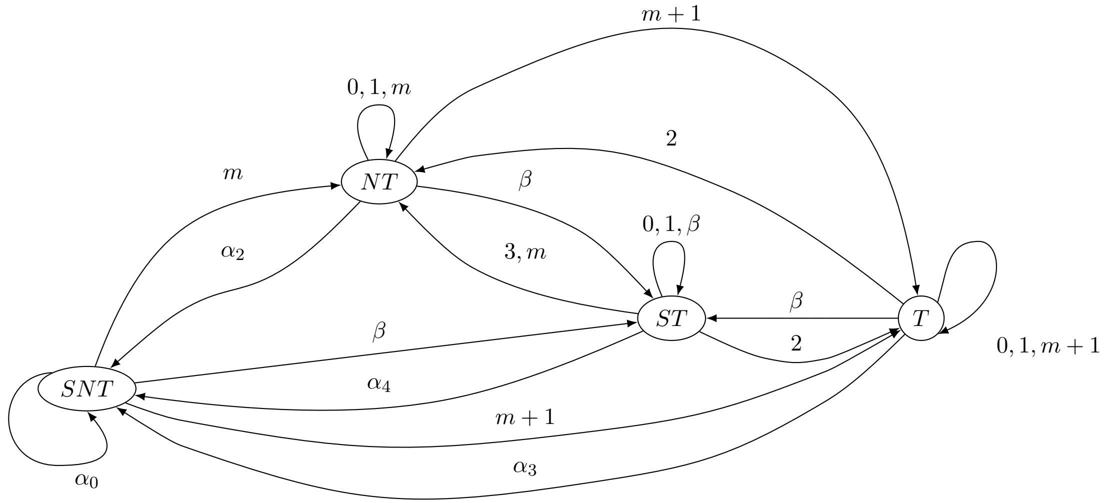



Figure 27 – Chaîne de markov représentant le prédicteur simple du cas n°1

Pour chaque arc, on peut alors associer un nombre de misprédictions et effectuer le calcul de la loi stationnaire sur ce graphe. Cependant, on ne peut pas calculer directement le taux de misprédiction pour ce graphe, en effet, chaque arc ne correspond pas à une lettre découverte dans le texte. On rajoute comme on a pu le faire en section 4.1 deux informations à nos arcs : le nombre de misprédiction *mis* et le nombre de lettres lues *number*. On obtient alors les mêmes types d'arcs : $\big(p \quad mis \quad number\big)$. Le calcul final du taux de misprédiction est le même que dans la section précédente donnant le taux moyen par lettre du texte.

En pratique, sauf si notre alphabet est de taille $k = 1$, ce taux de misprédiction est négligable et on peut donc estimer pour ce motif :

$$mis \approx n \times \mu_2(\frac{1}{k})$$

Cette méthodologie est tout à fait réutilisable pour n'importe quel motif et n'importe quel taille d'alphabet. On pourra obtenir plus de cas d'historiques possibles mais le calcul de la loi stationnaire final s'appliquera toujours sur un graphe de taille 4 pour des prédicteurs 2 bits. Utiliser des prédicteurs 3 bits et une taille d'historique $l \in [12, 16]$ donnera un taux de misprédiction très proche de la réalité.

**Septième partie**

# Bilan

## 1 Synthèse des travaux

Ce stage de M2 a été l'occasion pour moi d'établir un modèle théorique simplifié de processeur – pipeline et prédicteur de branchements – permettant une étude plus fine du comportement d'algorithmes.

En pratique, j'ai réalisé l'implémentation de ce modèle théorique à travers la création d'un émulateur de processeur en Python simulant un algorithme et permettant d'observer précisément de multiples statistiques d'ordinaire non mesurables, ainsi que de faire varier différentes composantes intéressantes.

J'ai également pu fournir une explication simple valable du problème de performance NAÏF-KMP ammenant de ce fait l'analyse plus complète de KMP avec des prédicteurs de branchements globaux. Cette grande partie mène à la découverte et la création de chaînes de MARKOV, d'un travail de minimisation ainsi qu'à l'établissement au final d'une formule calculant le taux de misprédiction grâce à l'étude plus précise d'un motif particulier.

## 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 plusieurs spécificités liées à l'architecture des ordinateurs.

— 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, diverses opérations comme leurs minimisations et leurs calculs associés ; et la façon dont on peut s'en servir pour analyser des algorithmes de manière probabiliste.

— l'implémentation en C d'un créateur de graphes, de ses dépendances et de divers outils mathématiques d'algèbre.

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.

## 3 Conclusion et pistes futures

Ce stage de M2 vient conclure ma scolarité et fut l'occasion pour moi d'avoir fourni un travail aussi bien théorique que pratique. Cette grande diversité fut quelque chose qui m'a beaucoup plu et appris. Cela m'a notamment permis de mettre en pratique mon autonomie et ma curiosité qui furent des qualités importantes durant ce stage.

Les observations effectuées mènent à une remise en question du réalisme des outils classiques de complexités et de certains aspects de l'algorithmie. L'impact de l'architecture actuelle étant de plus en plus important, il serait intéressant de pouvoir fournir des méthodes :

- d'analyse de la performance réelle d'un algorithme : composants d'architecture et temps réel
- d'implémentations en pratique en tenant compte de ces aspects

Les tendances actuelles montrent une libération très lente des architectures utilisées, mais des initiatives comme RISCV poussent les fabricants à promettre une publication d'architectures libres dans les prochaines années.

## Références


[1] 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.

[2] Richard S. Ballance. *Planning a Computer System.* McGraw-Hill Book Company, 1962.

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

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

[5] Maxime Crochemore Dominique Perrin. Two-way string-matching. https://hal.archives-ouvertes.fr/hal-00619582, 1991.

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

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

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

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

[10] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1972. vol. 1, p. 146–160.