République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Saad Dahleb Blida

Faculté des sciences

Département informatique

Mémoire de fin d'étude

Pour l'obtention du diplôme de master en informatique

Spécialité : Réseaux et système informatiques

Thème :

ALGORITHMES PARALLÉLES POUR LA RECHERCHE PAR DICTIONNAIRE

Mémoire présenté par :

- **Meftahi Oualid**
- **Messaoud Mohamed Amine**

Promotrice : Mme Oukid Saliha

Encadreur : Mr Belazzougui Djamal

Date de la Soutenance : **16 juillet 2019**

2018/2019

Dédicace :

Je voudrais remercier dieu pour toute l'énergie qu'il m'a donné durant ces cinq années, nous croyons au destin, nous pouvons traverser les moments difficiles en regardant toujours le bon côté de la chose, hamdoulillah.

Je voudrais dédier mon travail à toutes les personnes qui m'ont aidé durant mon cursus universitaire.

Je voudrais principalement remercier mes chers parents, mes chers frères ainsi que toute ma famille sans qui je n'aurais jamais pu faire ce travail.

ملخص

في هذا العمل، نحن مهتمون **بمشكلة البحث النصي بالقاموس**، وخاصة خوارزميات **الأنماط متعددة** Rabin karp و Aho-Corasick.

نظرًا لنمو البيانات النصية ومقدار المعلومات المطلوب معالجتها، هناك حاجة قوية إلى خوارزمية فعالة من حيث المساحة بالإضافة إلى وقت المعالجة السريعة لهذه البيانات. لهذا السبب يجب أن يفي الخوارزميات بهذين المعيارين:

- بالنسبة إلى الخوارزمية Aho-Corasick ، فإننا ننفذ **تطبيقًا موجز** بخاصية **التوازي** معه من أجل تحسين تكلفة الذاكرة وتسريع عمليات حساب هذا التطبيق.

- بالنسبة إلى خوارزمية Rabin karp، ننفذ **جدول تجزئة** وفقًا لطريقة التحقيق الخطية من خلال تطبيق **التوازي** لتحسين حساب **دالة التجزئة**.

نقوم بإجراء اختبارات على هذه الخوارزميات لدراسة كفاءة هذه التطبيقات. لقد أظهرت هذه الاختبارات أن خوارزمياتنا فعالة في الممارسة من حيث وقت الحوسبة ومساحة الذاكرة المشغولة.

الكلمات الدالة: البحث النصي بالقاموس، الأنماط متعددة، تطبيقًا موجز، لتوازي، جدول تجزئة، دالة التجزئة

Résumé

Dans ce travail, nous nous sommes intéressés au problème de **la recherche par dictionnaire**, en particulier aux algorithmes **multi-motifs** d'Aho-Corasick et de Rabin Karp. En raison de la croissance des données textuelle et de la quantité d'information à traiter, alors il existe un fort besoin d'algorithme efficace en termes d'espace aussi bien que de temps pour un traitement rapide et non couteux de ces données. C'est la raison pour laquelle l'implémentation des deux algorithmes doit répondre à ces deux critères :

- Pour l'algorithme d'Aho-Corasick, nous effectuons une implémentation **succincte** que nous **parallélisons** afin d'optimiser le coût mémoire et d'accélérer les calculs.
- Pour l'algorithme de Rabin-Karp, nous effectuons une implémentation d'une **table de hachage** par la méthode de **linear probing** (essai linéaire) en appliquant le **parallélisme** pour améliorer le calcul de **la fonction de hachage**.

Nous avons effectué des tests sur ces algorithmes pour étudier l'efficacité de ces implémentations. Ces tests ont montré que nos algorithmes sont efficaces en pratique en termes de temps de calcul et d'espace.

Mots-clés : la recherche par dictionnaire, multi-motifs, succincte, parallélisme, table de hachage, linear probing, fonction de hachage.

Abstract

In this work, we are interested in **dictionary matching** problem, in particular, in the **multi pattern** algorithms Aho-Corasick and Rabin Karp. Due to the growth of textual data and of the amount of information to be handled, there is a strong need for an efficient algorithm in terms of space as well as time for a fast and optimal data processing. This is why the implementation of both algorithms must meet these criteria:

- For Aho-Corasick algorithm, we implement a **succinct** version that will be **parallelize** in order to optimize memory cost and speedup the process.
- For Rabin Karp algorithm, we use **linear probing** method to implement the **hash table** that will be **parallelize** to improve the calculation time of the **hash function.**

We perform a set of tests on this algorithms to study the efficiency of this implementation. The results show that the algorithms are efficient in practice in terms of computing time and memory space.

Keywords: dictionary matching, multi pattern, succinct, parallelize, linear probing, hash table, hash function.

Remerciement

On dédie ce mémoire,

à tous ceux et toutes celles

qui m'ont accompagné et soutenu

durant cette année de formation

Pour commencer, on veut adresser nos remerciements à notre promotrice Mme. Oukid Saliha et notre encadreur Mr Belazzougui Djamel pour leurs grandes disponibilités et leurs encouragements tout au long de la rédaction de cette étude et ce mémoire.

On remercie également nos parents ainsi nos frères et sœur pour leur aide précieuse et leur encouragement dans toutes épreuves.

Enfin, on adresse nos remerciements à nos amis, nos camarades, ainsi toute personne qui a contribué à la réalisation de ce travail.

TABLE DES MATIÈRES

Liste des figures

Liste des tableaux

Liste des algorithmes

Introduction générale

La recherche par dictionnaire (ou bien dictionary matching en anglais) est un problème fondamental en informatique dont la tâche consiste à identifier l'occurrence d'un ensemble de motifs dans un dictionnaire au sein d'un texte donné. Les applications de ce problème incluent la recherche des phrases spécifiques dans un livre, l'analyse d'un fichier pour la recherche des signatures des virus et la détection d'intrusions sur le réseau. Egalement, dictionary matching est appliqué dans le domaine des sciences biologiques (bio-informatique) tel que la recherche dans une séquence d'ADN.

Pour la recherche par dictionnaire, plusieurs algorithmes ont été développés dans le passé comme l'algorithme naïf (Brute-Force en anglais), l'algorithme de Boyer-Moore [17], Knuth-

Morris-Pratt [15] etc… Ces algorithmes de recherche de motifs peuvent être divisés principalement en deux sous-catégories: les algorithmes de recherche de motif simples et les algorithmes de recherche de multiples motifs. Dans notre recherche nous nous intéressons aux algorithmes de recherche multiples : Aho-Corasick [28] et Rabin Karp [18]

Ces dernières années, il y a eu une prolifération massive de la quantité d'informations textuelles due à la croissance des donnée (Big Data, base de donné etc…), de telle façon que le processus de traitement de texte est devenu couteux en terme de complexité spatiale. L'augmentation de l'efficacité de l'algorithme de la recherche par dictionnaire, augmentera donc considérablement l'efficacité de toutes ces applications.

L'algorithme d'Aho-Corasick est un algorithme de correspondance multi pattern qui localise toutes les occurrences d'un ensemble de motifs dans un texte, il est représenté par un automate dont le temps pour répondre à une requête en utilisant cet automate est optimal. Mais son implémentation traditionnelle est couteuse en espace mémoire, de plus avec un dictionnaire de grand taille, l'espace mémoire utilisé sera couteux et pose de problème de stockage.

L'algorithme de Rabin Karp est considéré comme l'un des algorithmes les plus efficaces dans les domaines de la recherche de motif. Cet algorithme utilise la technique du hachage qui permet un gain d'espace.

De ce qui précède, nous déduisons qu'il existe un fort besoin d'une implémentation efficace des algorithmes exploitant les architectures parallèles ainsi que des techniques permettant de réduire l'utilisation de l'espace mémoire et le temps d'exécution.

Dans cette thèse, nous apportons de nouvelles solutions au problème de la recherche par dictionnaire, toutes fondées sur l'idée d'associer une implémentation pratique avec des structures permettant d'optimiser l'espace mémoire utilisé et des architectures parallèles pour la réduction du temps d'exécution.

Une implémentation efficace en terme d'espace mémoire repose sur l'utilisation des structures des données succincte qui n'occupent pas plus d'espace que les données utiles sur l'automate classique d'Aho-Corasick pour le but de réduire sa complexité spatiale d'une façon que la représentation sera plus optimiser par rapport à l'implémentation traditionnelle.

Une implémentation efficace en termes de temps de calcul doit exploiter les architectures parallèles (multi-cœurs).

Organisation de la thèse

Le reste de cette thèse est organisé comme suit :

Dans le chapitre qui suit cette introduction, nous donnons quelques préliminaires et définitions nécessaires à la compréhension du domaine et des chapitres qui suivent. Par la suite, nous donnons un état de l'art où nous expliquons les principes généraux des algorithmes de recherche par dictionnaire et nous introduisons quelques méthodes qui sont liées à notre travail.

Dans le chapitre qui suit nous présentons les deux algorithmes de la recherche par dictionnaire en utilisant les structures des données succinctes pour représenter l'automate d'Aho-Corasick et l'architecture parallèles pour les deux algorithmes

Dans le dernier chapitre, une évaluation pour les méthodes optimisées que nous avons proposée et présentée en évaluant les performances de la version implémentée. Enfin. Nous concluons cette thèse et donne quelques perspectives de recherche.

I. Erreur ! Il n'y a pas de texte répondant à ce style dans ce document.PRELIMINAIRES

I.1. Introduction

Dans ce chapitre, nous présentons des définitions de base de l'algorithmique du texte, des notations et des notions clés utilisées dans la suite de ce mémoire. Nous introduisons également des structures de données utilisées pour la représentation des dictionnaires

I.2. Définition et notation

Alphabet : Un alphabet Σ est un ensemble fini non vide d'éléments appelés symboles ou lettres. La cardinalité de cet ensemble est notée par$|\Sigma| = \sigma$.

Motif : Un motif v défini sur un alphabet Σ est une concaténation d'éléments de $\Sigma, v = e_1 e_2 \dots e_m$. La longueur d'un motif est le nombre de lettres qui le composent, et nous la notons par $|v| = m$. Exemple : $\Sigma_2 = \{0,1\}, v = 01101001$ est défini sur $\Sigma_2$et$|v| = 8$. Le mot vide (sa longueur vaut zéro) est noté par ε. L'ensemble de tous les motifs définis sur l'alphabet Σ est noté par $\Sigma^*, \Sigma + = \Sigma^* - \{\varepsilon\}$. La concaténation de deux mots$u = aax$, et $v = bb$ définis sur l'alphabet Σ = {a, b, x} est le mot noté *uv* obtenu en mettant bout à bout u et v, $uv = aaxbb$.

Préfixe : Un mot u est préfixe d'un mot w s'il existe un mot v tel que $w = uv$. Soit w un mot dans $w = a_1 \dots a_n$, alors tout mot $u \in \{\varepsilon, a_1, a_1 a_2, \dots, a_1 \dots a_n\}$ est préfixe de w. On note par $Pref(w)$ l'ensemble {ε,a_1,$a_1 a_2$, ..., $a_1 \dots a_n$} de tous les préfixes de w. L'ensemble $Pref(w) - \{w\}$ est dit l'ensemble des préfixes propres de w.

Suffixe : Un mot u est un suffixe du mot w s'il existe un mot v tel que$w = vu$. Soit $w = a_1 \dots a_n$, un mot dans Σ*, alors tout mot $u \in \{\varepsilon, a_n, a_{n-1} a_n, \dots, a_1 \dots a_n\}$ est suffixe de$w$. On note par $Suff(w)$ l'ensemble de tous les suffixes de w. L'ensemble $Suff(w) -$ {w} est dit l'ensemble des suffixes propres de w.

Mot miroir : Le mot miroir du mot $w = a_1 \dots a_n$, est le mot $\overline{w} = a_n \dots a_1$.

Un texte T : est une suite de n caractères. Il peut être vu aussi comme étant un ensemble de d mots $T = \{w_1, w_2, \ldots, w_d\}$, où chaque mot peut avoir plusieurs occurrences.

I.3. Les structures de données pour dictionnaires

Un dictionnaire est un ensemble D = $\{w_1, w_2, ..., w_d\}$ de d mots, tels que $w_i \neq w_j$(i ≠ j). La taille du dictionnaire est $|D| = \sum_{i=1}^{i=d} |w_i| = n$.

Pour une recherche efficace dans le texte et/ou dans le dictionnaire, il est important d'organiser et de stocker les données en une structure qui en facilite l'exploitation (l'accès et éventuellement les modifications).

Généralement une structure de données occupe plus d'espace mémoire que les données sur lesquelles elle est construite, c'est pour cela qu'on utilise une structure de données succincte qui occupe un espace proche de la taille des données qu'elle doit représenter

I.3.1. Arbre de suffixes

Soit T un mot défini sur un alphabet Σ tel que $|T| = n$. L'arbre des suffixes [1] de T est un arbre enraciné à n feuilles tel que :

- Chaque feuille i représente un seul et unique suffixe, qui est le mot $T[i..n]$
- Ses branches sont étiquetées par des mots non vides.
- Ses nœuds internes sont de degrés (nombre de branches sortantes) > 1
- Les étiquettes des branches sortantes d'un nœud ne peuvent pas commencer par le même caractère.
- La concaténation des étiquettes sur le chemin qui commence à la racine R et qui se termine à une feuille i formes le suffixe $T[i..n]$.

Il existe différents algorithmes permettant la construction de l'arbre de suffixes dans un temps et espace linéaire [2].

I.3.2. Trie

Soit D un dictionnaire de d mots. Un Trie[3] de D est un arbre enraciné de préfixes communs et qui a d feuilles. Chaque nœud représente un préfixe commun des mots de D et a un ou plusieurs nœuds fils. Chaque arête est étiquetée par un caractère des mots de D. Deux arêtes qui sortent du même nœud ne peuvent pas être étiquetées par le même caractère. Chaque chemin de la racine à une feuille est un mot du dictionnaire. La construction d'un Trie peut se faire en temps $O(n)$ où n est la taille du dictionnaire (sommes des longueurs des mots du dictionnaire).

La recherche d'un mot P de longueur $|P| = m$ consiste à parcourir le Trie de la racine en suivant les lettres de P sur les arêtes et peut se faire en temps $O(m)$.

I.3.3. Trie inversé

On note par $\overline{w}$ le mot miroir de w et par $D = \{\overline{w_1}, \overline{w_2}, \dots, \overline{w_n}\}$ le dictionnaire des mots miroirs de D. Le Trie inversé de D est le Trie de D. Il permet de faire la recherche de droite à gauche.

I.3.4. Les automates

Un automate fini ou machines à états finis déterministe M est défini comme un quintuplé : $M = (Q; \Sigma; \delta; I; F)$, tel que:

- $\boldsymbol{Q}$ est un ensemble fini d'états;
- $\boldsymbol{\Sigma}$ est l'alphabet;
- $\boldsymbol{i \in Q}$ est l'état initial;
- $\boldsymbol{F \in Q}$ est l'ensemble des états terminaux;
- δ: Q* Σ→Q est la fonction transition permettant de passer d'un état à un autre par un caractère de l'alphabet.

Il y a plusieurs représentations possibles des machines à états, comme le montre la figure 1

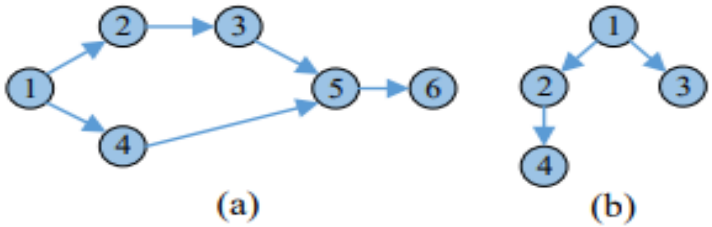


Figure 1 : *Exemples de machine à états orientée (a), et sous forme d'arbre orienté (b)*

Le graphe (a) représente un ensemble de nœuds ou $Q = \{1,2,3,4,5,6\}$, $i = 1$ est l'état initial, $F = 6$ et δ sont les transitions orientées entre l'ensemble des nœuds. De l'autre côté, le graphe (b) est une représentation sous la forme d'un arbre avec des branches orientées partant du haut, appelé « nœud racine », jusqu'en bas, appelé « feuilles ».

I.3.5. La table de hachage

Une table de hachage est une structure de données qui généralise la notion simple d'un tableau ordinaire pour implémenter les dictionnaires. On utilise une fonction de hachage h pour mapper l'ensemble des clés $U = \{cl_1, cl_2, \ldots, cl_n\}$ vers les positions $\{1,2, \ldots, n\}$ des cases d'une table de hachage table composée de n cases

Dans le cas idéal, lorsque la fonction de hachage h est bijective, elle associe à chaque clé une position unique et toute position est associée à une seule clé. Ce hachage est dit « parfait». Lorsque h n'est pas injective elle retourne la même valeur de position pour des clés différente. On dit qu'il y a « collision ». Il existe plusieurs solutions pour résoudre le problème de collision comme le hachage avec chaînage, l'adressage ouvert, le sondage linéaire, le double hachage, etc. [5]

I.3.6. Vecteur de bits

La structure de données vecteur de bits (bit-vector) est un tableau utilisé simplement pour stocker et retrouver les bits (1 ou 0) dans une position donnée. Généralement, chaque case du tableau a une taille d'un mot mémoire w. Dans les machines actuelles $w = 32$ ou 64 selon le système utilisé de 32 ou 64 bits respectivement. Dans une structure de dictionnaire, le vecteur de bits est augmenté avec des informations supplémentaires afin de supporter les opérations de rang et de sélection ($Rank/Select$) [6]. L'opération de rang ($Rank\ (1/0, i)$) calcule le nombre

de 1 (ou 0) depuis le début du tableau jusqu'à une position donnée i, et de même, l'opération de la sélection ($Select\ (1/0, i)$) permet de retourner la position de l'occurrence numéro i de l'élément 1 (ou 0)

I.3.7. Balanced parentheses

La représentation parenthèses équilibrées (balanced parentheses en anglais) est une structure de données utilisée pour une représentation succincte d'arbres de sorte que chaque nœud soit représenté par une paire de parenthèses correspondantes. Le principe est que, lorsque l'arbre est traversé en profondeur (depth-first preorder), une parenthèse ouverte est écrite lorsqu'un nœud est atteint pour la première fois et une parenthèse fermante est écrite lorsque ce nœud est à nouveau atteint. Il en résulte une séquence de $2n$ parenthèses équilibrées. La figure suivante représente la traversé en profondeur d'un arbre

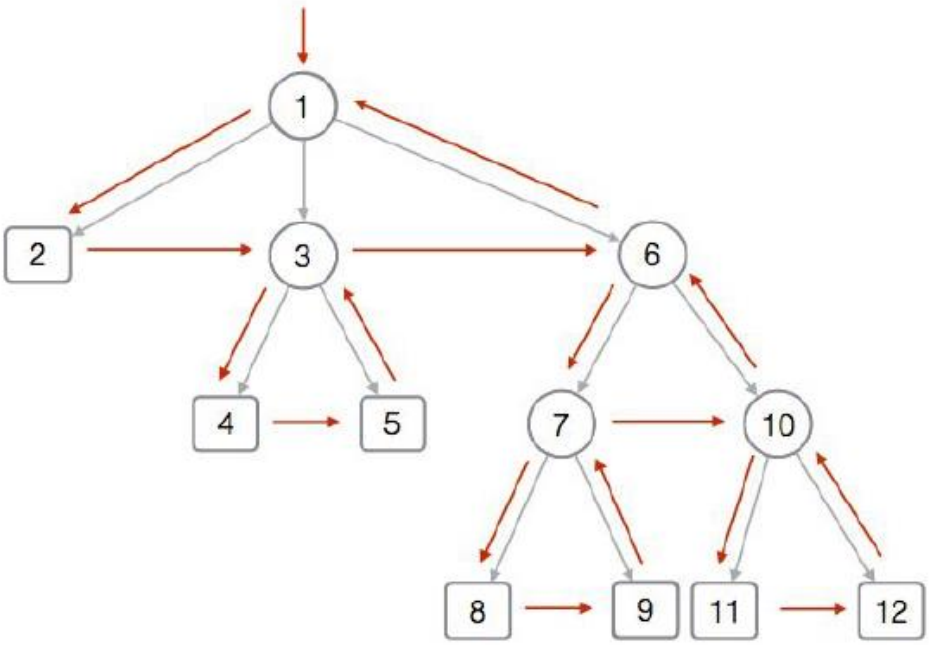
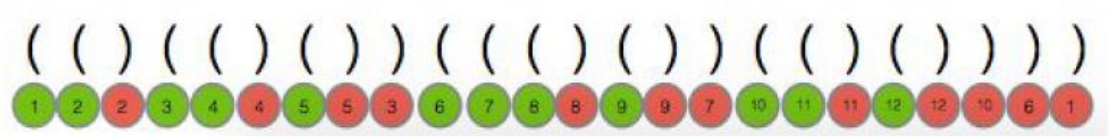


***Figure 2** : Exemple de une traversé en profondeur (depth-first perde) d'un arbre à 12 nœuds [40]*

Un nœud est identifié par ses parenthèses ouvrantes et un sous arbre est représenté par l'ensemble des parenthèses contenues entre les parenthèses ouvrantes et fermantes d'un nœud.



a

Figure 3: *La représentation de l'arbre de la figure 2 sous forme d'une représentation Balanced parentheses [40]*

I.4. Notion de complexité algorithmique

La complexité algorithmique [7] est une fonction qui permet d'évaluer la quantité de ressources (temps et espace mémoire) nécessaire pour le fonctionnement d'un algorithme donné. Pour calculer la complexité dans le pire cas, nous utilisons une évaluation asymptotique, dans laquelle le nombre des éléments n manipulés est assez grand (tend vers l'infini), et où nous ignorons les constantes.

I.4.1. Notation

Il existe plusieurs notations pour designer la complexité d'un algorithme donné. Parmi les notations les plus utilisées, nous avons les notations : $O(n), \Omega(n), \Theta(n)$[8]. Nous nous intéressons particulièrement à la notation $O(n)$.

Soient f et g deux fonctions mathématiques et n le nombre d'éléments manipulés, supposé suffisamment grand (tendant vers l'infini).

Grand O $f(n) = O(g(n))$, la fonction f est bornée asymptotiquement par la fonction g. $|f(n)| \leq k \times |(g(n))|$ avec k une constante strictement positive. Cela est équivalent à dire que$(f(n)\ )/(g(n)) \leq k$ quand n tend vers l'infini. La notation O décrit une borne supérieure asymptotique.

I.4.2. La notion de complexité moyenne

La complexité en moyenne [9] calcule la quantité des ressources (typiquement, le temps) nécessaire utilisée par un algorithme qui agit sur des données en entrée qui sont équiprobables.

La complexité en moyenne est nécessaire pour calculer la performance qui est proche de la réalité. Elle permet de distinguer l'algorithme le plus efficace en pratique entre l'ensemble de tous les autres algorithmes qui peuvent avoir la même complexité dans le pire des cas.

I.5. La recherche de motifs

La recherche de motifs est un problème fondamental en informatique et en traitement de texte. Il s'agit de détecter dans une structure la présence de sous-structures appelées motifs. Les structures considérées sont le plus souvent des mots (séquences finies de symboles), des arbres ou des graphes. Ici nous nous intéresserons aux mots ou textes et les motifs seront donnés sous la forme d'un ensemble de mots facilement représentable : un seul mot, un ensemble fini de mots, ou encore un ensemble rationnel de mots.

Tester l'apparition d'un motif dans un texte, compter les occurrences de ce motif, déterminer l'ensemble des positions de ces occurrences sont autant de questions centrales dans plusieurs branches de l'informatique notamment en bio-informatique, détection d'intrusion etc…

I.6. Notion du parallélisme

Le parallélisme consiste à mettre en œuvre des architectures permettant de traiter des informations de manière simultanée, ainsi que des algorithmes spécialisés pour celles-ci. Ces techniques ont pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible. En effet, la vitesse de traitement qui est liée à l'augmentation de la fréquence des processeurs connait des limites. La création de processeurs multi-cœurs, traitant plusieurs instructions en même temps au sein du même composant, résout ce dilemme. Certains types de calculs se prêtent particulièrement bien à la parallélisations, la fluidité du traitement de l'information et l'exploration de données, le traitement d'images ou la fabrication d'images de synthèse .C'est dans le domaine des supercalculateurs que le parallélisme a été utilisé pour la première fois, à des fins scientifiques. Pour bien comprendre la différence entre la programmation séquentielle et la programmation parallèle faudra appréhender les mesures (métriques) servant à l'évaluation de la performance.

I.7. Conclusion

Dans ce chapitre, nous avons introduit les définitions de bases et notation sur la recherche du motif, parallélisme et les algorithmes de la recherche par dictionnaire ainsi que les structures de données appliquées pour la représentation de nos deux algorithmes.

II. ETAT DE L'ART SUR LES ALGORITHME DE LA RECHERCHE PAR DICTIONNAIRE

II.1. Introduction

Dans ce chapitre, nous dressons un état de l'art sur la recherche par dictionnaire et nous introduisons les algorithmes de la recherche par dictionnaire dans un texte, ces algorithme sont présenté selon l'ordre chronologique. Nous discutons leur performance et nous présentons les notions des données succinctes et d'algorithme parallélisme.

II.2. Les algorithmes de recherche de motifs

La plupart des algorithmes de correspondance de chaînes fonctionnent généralement comme suit. Ils scannent le texte à l'aide d'une fenêtre du texte dont la taille est généralement égale à la taille du pattern m. Pour chaque fenêtre du texte, ils vérifient l'occurrence du motif (ce travail spécifique est appelé tentative) en comparant les caractères de la fenêtre avec les caractères du motif, ou en effectuant des transitions sur un type d'automate, ou en utilisant une méthode de filtrage. Après une correspondance du motif ou après une discordance, ils décalent la fenêtre vers la droite d'un certain nombre de positions. Ce mécanisme est généralement appelé mécanisme de fenêtre glissante (shift matching). Au début de la recherche, ils alignent les extrémités gauches de la fenêtre et du texte, puis répètent le mécanisme de la fenêtre glissante jusqu'à ce que l'extrémité droite de la fenêtre dépasse l'extrémité droite du texte. Nous associons chaque tentative à la position s dans le texte où la fenêtre est positionnée, c'est à dire $T\,[s..s\,+\,m-1]$. [13]

La correspondance efficace de motifs sur le texte T pour rechercher toutes les occurrences de motifs P signifie que le temps pour signaler toutes les occurrences d'occurrence est $O((|P| + occurrences)\ log\ (n))$. Dans ce qui suit nous décrivons quelques travaux importants sur la recherche de motifs

II.2.1. Algorithme naïf

- **Principe**

L'algorithme naïf est le plus simple et le plus anciens. Il procède de la manière suivante. Pour chaque position possible du motif P dans un texte T, on teste si cette position est une occurrence du motif. Ce test est effectué en comparant les caractères du motif avec les caractères du texte de gauche à droite. Si tous les caractères du motif sont égaux aux caractères du texte aux positions correspondantes, une occurrence a été trouvée et la position de cette occurrence est retournée. Sinon, la recherche se poursuit en passant à la position suivante[14]. Nous montrons par la suite dans la figure 4 le déroulement de l'algorithme naïf par un exemple, nous voulons chercher le motif 10100111 dans le texte suivant :

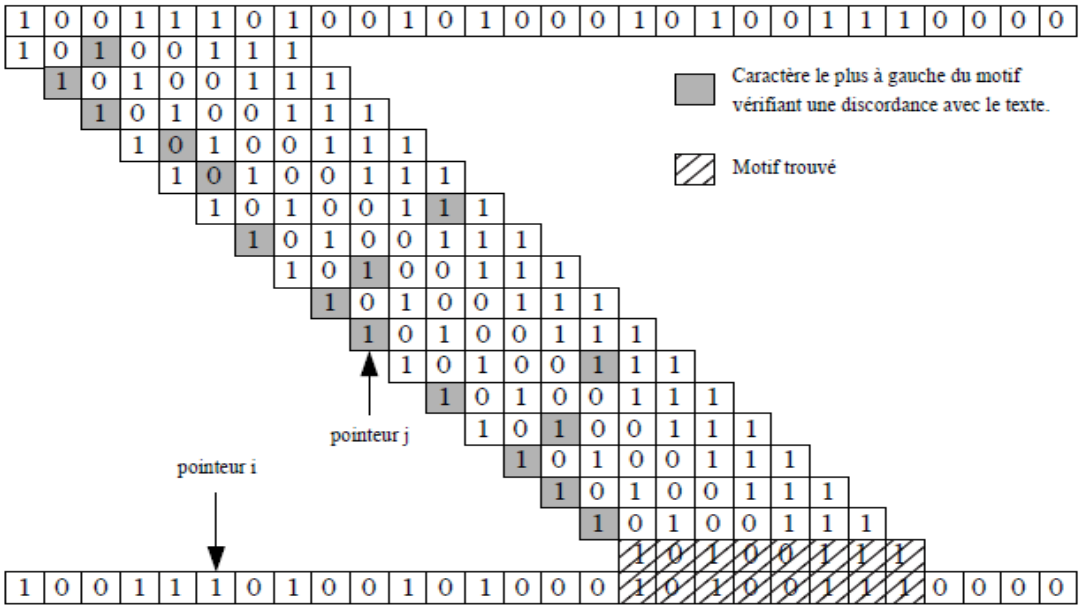


Figure 4 : *Exemple de déroulement de l'algorithme naïf [39]*

Le programme conserve un pointeur i dans le texte et un autre j dans le motif. Tant qu'ils font référence à des caractères concordants, les deux pointeurs sont incrémentés. Si i et j pointent sur des caractères discordants, j est remis à pointer sur le début du motif et i est repositionné de manière à faire avancer le motif d'un cran à droite en vue d'une nouvelle comparaison avec le texte.

- **Complexité**

La complexité temporelle de cette phase de recherche est $O\ (mn)$ (lors de la recherche d'un motif P de taille m dans un T de taille n). Le nombre attendu de comparaisons de caractères du texte est $2n$[14].

II.2.2. L'algorithme de Knuth et Morris Pratt

- **Principe**

L'algorithme a été développé par D.Knuth, J.Morris et V.Pratt en 1974[15], C'est une amélioration de l'algorithme de Morris-Pratt développé en 1970. Il est le premier algorithme pour lequel le temps d'exécution dans le pire cas est linéaire en la taille du texte et donc indépendant de la taille du motif. L'idée de base de cet algorithme est que chaque fois qu'une discordance est détectée, le "faux départ" se compose de caractères que nous avons déjà examinés. Nous pouvons profiter de cette information au lieu de répéter des comparaisons avec les caractères connus. L'algorithme KMP utilise la notion de fenêtre glissante. Il se décompose en deux phases principales:

- Une phase de prétraitement: au cours de laquelle on construit un tableau mémorisant les décalages à effectuer, à la fin de chaque itération sur une fenêtre balayant une séquence y d'entrée
- Une phase de recherche: au cours de laquelle on localise toutes les occurrences exactes d'un motif x dans la séquence y.

Pendant la phase de recherche, à chaque fois que l'on est dans une position i, on passe soit à i+1 ou à i - j (nous décalons de j positions lorsque une discordance se produit). La valeur de j est juste une fonction de i et ne dépend pas d'autres informations[16]. La figure 5 montre le fonctionnement de l'algorithme KMP:

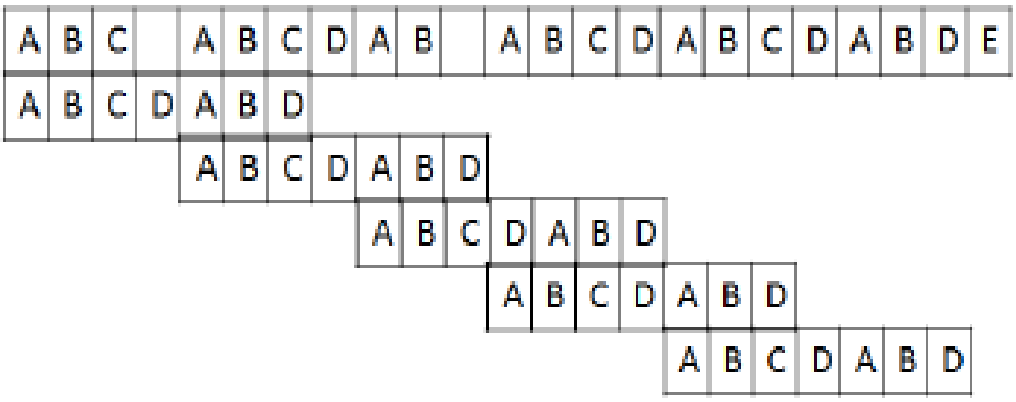


Figure 5 : *Le déroulement de l'algorithme KMP [39]*

La figure 5 est un exemple représentant le déroulement de l'algorithme KMP où nous avons la suite de caractère T= [$ABC\ ABCDAB\ ABCDABCDABDE$] et le motif P= [$ABCDABD$]. Pendant la phase de recherche on prendra une fenêtre dont la taille est égale à la longueur du motif. Les pointeurs pointent sur le début du texte et motif et compare les deux chaines caractère par caractère. Dès qu'une discordance est trouvée, le pointeur du motif i est décale vers la position j+1 du pointeur de texte comme il est montré dans l'exemple et nous recommençons le traitement par la comparaison des caractères du texte et de motif.

- **Complexité**

L'algorithme de KMP possède une complexité spatiale et temporelle de $O(m)$ dans la phase de prétraitement et une complexité du temps $O\ (n)$ en phase de recherche (indépendant de la taille de l'alphabet) [14].

II.2.1. L'algorithme d'Aho-Corasick

- **Principe**

Aho-Corasick (1975) [28] est un algorithme de correspondance multi pattern qui localise toutes les occurrences d'un ensemble de motifs dans un texte. Il crée d'abord des automates finis déterministes pour tous les motifs prédéfinis, puis utilise un automate pour traiter un texte en un seul passage. Il consiste à construire un automate d'états finis de correspondance de motifs

à partir des motifs, puis à utiliser les automates de correspondance de motifs pour traiter la chaîne de texte en un seul passage[14], [20] .

Dans le prétraitement, Aho-Corasick construit une machine à états (Trie) à partir des motifs. La machine à états commence par un nœud racine vide, qui correspond à l'état par défaut de non-correspondance. Chaque motif à faire correspondre ajoute des états à la machine, en partant de la racine pour aller à la fin du motif. La machine à états est ensuite traversée et des pointeurs d'échec sont ajoutés depuis chaque nœud au préfixe le plus long de ce nœud, ce qui conduit également à un nœud valide dans le Trie.

Nous montrons un exemple de l'automate de l'algorithme Aho-Corasick dans figure 6. Soit les motifs suivants : {WOMAN, MAN, MEAT, ANIMAL}

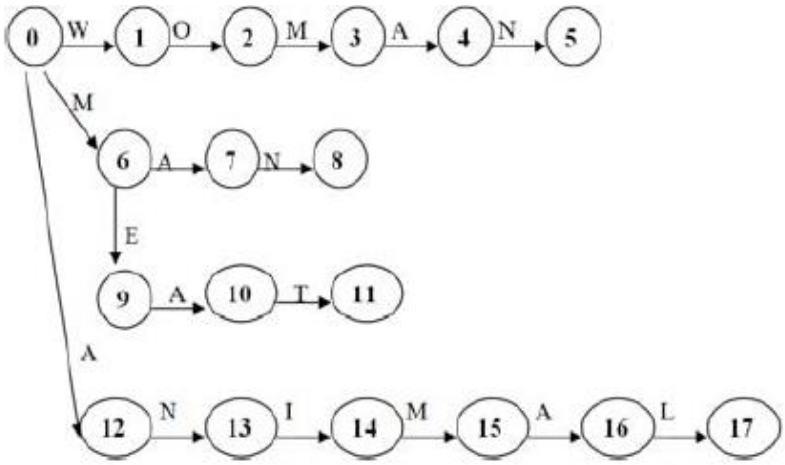


Figure 6 : *Automate d'Aho-Corasick*

Les états {5,8,11,17} sont des états finaux, une fois arrivé à ses états, nous dirons qu'on a trouvé une occurrence du motif pointé dans le texte.

Nous détaillerons la procédure de recherche, ainsi les fonctions échec (faillure fonction) et report (output function), dans le chapitre II.

Alors, plutôt de faire la recherche de motif de manière séquentielle, l'algorithme Aho-Corasick permet de la faire de manière parallèle, autrement dit, motifs multiples. Ceci dit, ce dernier peut être vu comme une généralisation de l'algorithme de Knuth-Morris-Pratt pour faire une recherche parallèle.

- **Complexité**

La durée d'exécution de l »algorithme Aho-Corasick est indépendante du nombre de motif. La complexité de l'algorithme Aho-Corasick est $O\ (n\ log\ n)$. Similaire à l'algorithme KMP, l'algorithme Aho-Corasick analyse le caractère dans le texte un par un sans saut [21].

II.2.2. L'algorithme de Boyer Moore

- **Principe**

L'algorithme de Boyer Moore (BM) a été développé par R.S. Boyer et J.C.Moore en 1977[17]. Il ressemble étrangement à l´algorithme naïf, sauf qu´il compare le motif P au texte T de droite à gauche, et il augmente une fenêtre S par un décalage d'une valeur qui n´est pas nécessairement égale à 1.

En cas de non concordance (ou de correspondance totale du motif), il utilise deux fonctions (heuristiques) pré-calculées pour décaler la fenêtre vers la droite. Ces deux heuristiques de décalage s'appellent l'heuristique de bon suffixe (également appelé décalage d'appariement) et l'heuristique de mauvais caractère (également appelé décalage d'occurrence). Il fonctionne en deux phases: phase de prétraitement et une phase d'adaptation.

Supposons qu'une inadéquation se produise entre le caractère $P\ [i]\ =\ b$ du motif et le caractère $[i\ +\ j]\ =\ a$ du texte lors d'une tentative dans la position j.
Alors $P\ [i\ +\ 1..m_1]\ =\ T\ [i\ +\ j\ +\ 1..j\ +\ m\ -\ 1]\ =\ u\ et\ P\ [i]\ \neq\ T\ [i\ +\ j]$.
L'heuristique du bon suffixe consiste à aligner le segment $T\ [i\ +\ j\ +\ 1\ldots j\ +\ m\ 1]\ =$ $P\ [i\ +\ 1\ldots m\ -\ 1]$ avec son occurrence la plus à droite dans P précédée d'un caractère différent de $P\ [i\ ]$[16]

- L'heuristique du bon suffixe: l'algorithme cherche la chaîne U dans P de droite à gauche. S'il existe un tel segment, il déplace P à droite pour obtenir une nouvelle

tentative pour la fenêtre. S'il n'y a pas un tel segment, le décalage consiste à aligner le suffixe le plus long V de $T\,[i\,+\,j\,+\,1\ldots j\,+\,m\,-\,1]$ avec un préfixe correspondant à P.

- L'heuristique du mauvais caractère : le changement de caractères incorrects consiste à aligner le caractère du texte $T\,[i\,+\,j]$ avec son occurrence la plus à droite dans $P\,[0..m\,-\,2].Si\;T\,[i\,+\,j]$ n'est pas trouvé dans le motif P, aucune occurrence de P dans T ne peut inclure $T\,[i\,+\,j]$, et l'extrémité gauche de la fenêtre est alignée avec le caractère immédiatement après $T\,[i\,+\,j]$, à savoir$T\,[i\,+\,j\,+\,1]$

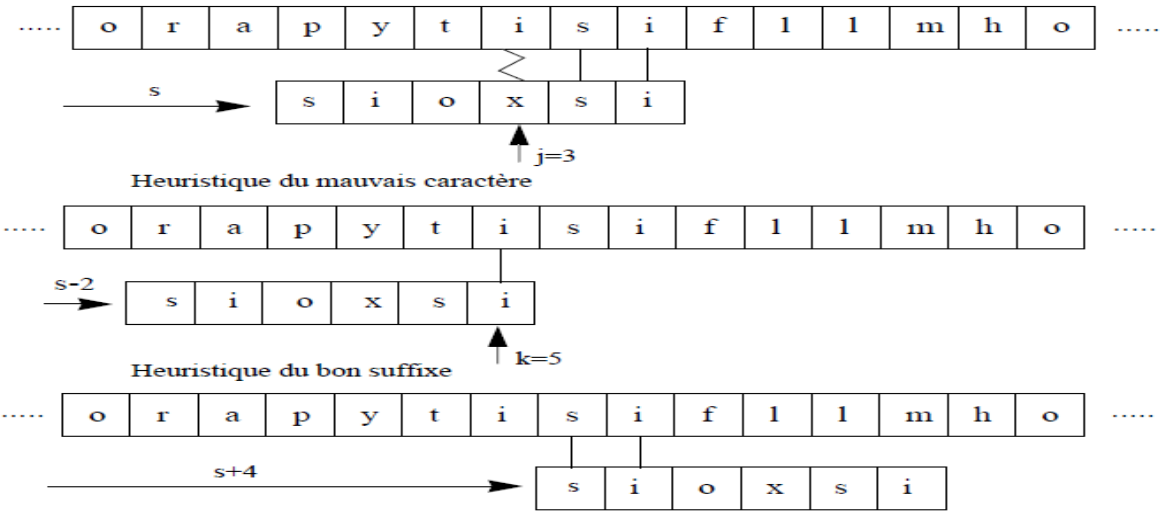


Figure 7 : *Exemple sur le déroulement de l'algorithme Boyer Moore [39]*

Dans cet exemple de la figure 7, nous trouvons une discordance à la position *j=3*.L´occurrence la plus à droite du mauvais caractère (i) dans le motif se trouve à la position *k=5*, l´heuristique du mauvais caractère propose un décalage de *j-k=-2* caractères.

Avec l´heuristique du bon suffixe, un déplacement du motif de 4 positions vers la droite permet de garantir que tous les caractères du motif, correspondant au bon suffixe *(si)* trouvé dans le texte, correspondront à ces caractères du suffixe.

Comme l´heuristique du mauvais caractère propose un déplacement vers la gauche de 2 positions, ce qui est inférieur à la proposition de l´heuristique du bon suffixe, l´algorithme de Boyer-Moore augmente le décalage de 4 positons.

Les expériences montrent que l'algorithme BM est rapide dans le cas d'un alphabet plus volumineux.

- **Complexité**

Pour phase de prétraitement de l'algorithme de BM, la complexité temporelle est égale à $O\ (m\ +\ |\sum)$. Pour la phase de recherche la complexité en temps est égale à $O\ (n\ /\ m)$ en moyenne [16]. Dans le pire cas, le temps d´exécution de recherche de l´algorithme de Boyer-Moore sera en $O((n-m+1)m+|S|)$ puisqu´il faudra un temps en O(m) pour valider chaque décalage [14].

II.2.1.L'algorithme de Karp Rabin

- **Principe**

L'algorithme de Rabin Karp [18](1987) est un algorithme de recherche de chaîne, qui prend en charge les capacités de correspondance de modèles uniques et multiples. Il est largement utilisé dans des applications telles que la détection de plagiat et la correspondance de séquence d'ADN Comme le hachage est en mesure de fournir une méthode simple pour éviter un nombre quadratique de comparaisons de caractères, Rabin Karp a utilisé cette notion pour trouver un motif P de taille m dans un texte T en utilisant une fonction de hachage de Rabin-Karp.

D'abord l'algorithme calcule le hache du motif de taille m, en appliquant la fonction du hache Rabin Karp. De la même manière, on calcule le hache d'une sous-chaine de taille m du texte. Si les deux valeurs de hache sont égales, l'algorithme fait une comparaison naïve caractère par caractère pour vérifie l'occurrence du motif dans la sous-chaine. Le hache de la prochaine fenêtre peut être calculé du hache du la fenêtre précédente. Il applique ce processus à plusieurs reprises pour tous les caractères de du texte. Pour trouver le motif correspondant, nous allons comparer le hachage du motif et le hachage de chaque sous-chaîne de texte. Si le hachage correspond à la fois à la sous-chaîne et au motif, nous comparerons les deux chaînes pour trouver une correspondance[[19]

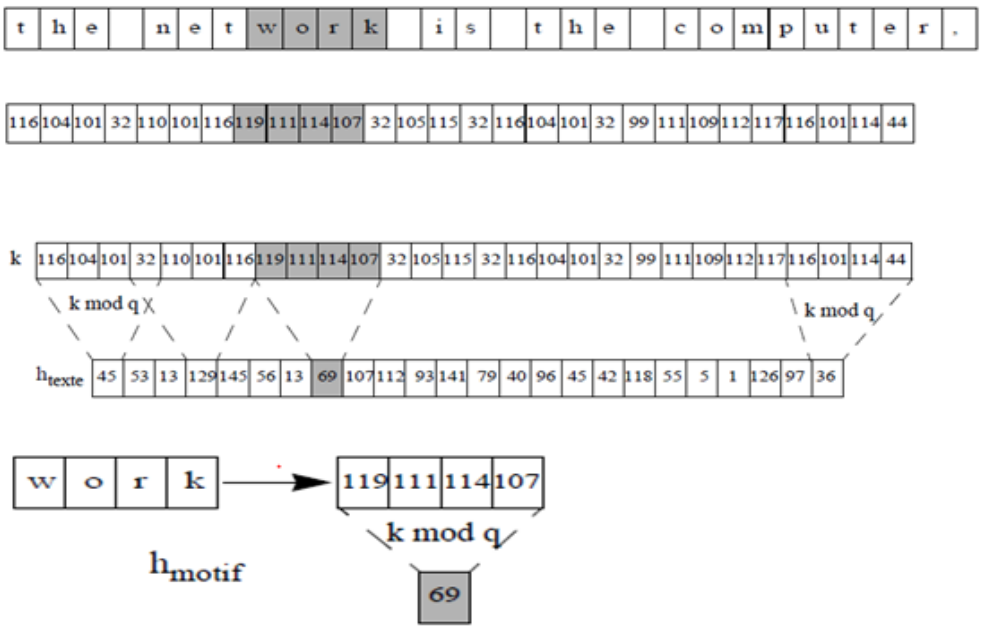


Figure 8 : *Exemple sur le déroulement de l'algorithme Rabin Krap [39]*

L'exemple sur la figure 8 montre le déroulement de l'algorithme Rabin Karp. On cherche le motif P=[*work]* de taille 4 dans le texte T= *[the network is the computer]* Le premier traitement, très simple en terme de calcul machine, consiste en la transformation des caractères de T et de P en leur valeur décimale sur la table ASCII. Ensuite on calcule la valeur de hachage du motif P et de tous les blocs de taille K=4 (taille du motif à rechercher) du texte à l'aide de la fonction *h*. L'existence du motif P dans le texte T est détectée par l'existence de son hachage dans le texte T (on compare la valeur de hachage de P avec celles des sous-blocs de T).

Nous détaillerons plus sur l'algorithme et sur la fonction du hachage dans les prochains chapitres.

- **Complexité**

Le plus gros avantage de cet algorithme est qu'il s'adapte facilement à la recherche de plusieurs motifs à la condition qu'ils soient de même taille. L'algorithme nécessite un prétraitement en $O(m)$ et fonctionne en temps $O(mn)$ dans le pire des cas. En moyenne il marche en temps $O(n + m \cdot (v + n/q))$ où v est le nombre d'occurrences du motif .Il est très facile à implémenter, plus efficace que l'algorithme naïf [14]

II.3. Etude comparative

Plusieurs algorithmes ont été présentés pour la recherche de motifs, sois pour résoudre le problème de recherche, de complexité ou problème de coût en espace mémoire.

Chaque algorithme a ses avantages et ses inconvénients, le tableau suivant récapitule les algorithmes vus en mentionnant une comparaison entre ces derniers dans une étude [22] menée en exécutant ces algorithmes :

Algorithme	Prétraitement	String matching	Type de recherche	Multiple motifs	Approche
Naïve	Pas de prétraitement	O(mn)	Préfixe	No	Recherche linéaire
KMP	O(m)	O(n + m)	Préfixe	No	Heuristiques
Boyer Moore	O(m + n)	O(mn)	Suffixe	No	Heuristiques
Rabin Karp	O(n)	O(mn)	Préfixe	No	Hachage
Aho-Corasick	O(m + n)	O(m + n)	Préfixe	Yes	Automate

Tableau 1 : *Tableau comparative des algorithmes de recherche de motifs*

Nous remarquons que l'algorithme naïve n'a pas de prétraitement ce qui devait être rentable en terme de temps d'exécution mais vu sa complexité en matière de recherche et que c'est algorithme qui recherche avec un seul motif à la fois nous déduisons que plus les données et le dictionnaire sont grand plus l'algorithme sera couteux en terme de temps. Du coup ce n'est pas évident d'utilise cet algorithme avec de grande données qui recherchent plusieurs motifs voir des millions.

De même pour l'algorithme Boyer Moore, de plus de sa ressemblance a l'algorithme naïve, il a aussi un prétraitent que sa complexité est considérable en matière de temps, mais il a su rattraper son retard grâce à sa technique utilisé en recherche de motifs (la fenêtre glissante).

Pour KMP et Aho-Corasick, nous remarquons que la complexité en recherche de motifs est la même, ils dépendent de nombre de motifs et la taille du texte. Par contre, dans le

prétraitement, KMP ne dépend que du nombre de motifs, alors que, Aho-Corasick dépend du nombre de motifs et taille du texte. Au finale, Aho-Corasick se distingue de cet algorithme avec sa recherche multiple motifs à la fois de.

Pour Rabin Karp, nous dirons que c'est un algorithme plutôt moyen en termes de complexité. De plus, c'est un algorithme de recherche d'un ou plusieurs motifs qui peut être faite si et seulement si les motifs sont de même taille.

Dans un exemple de systèmes de détection d'intrusion ou séquence ADN, l'automate d'Aho-Corasick est largement utilisé. Le temps pour répondre à la requête en utilisant cet automate est $O(n + occ)$ ou occ est le nombre d'occurrences trouvées. Cependant, la représentation de cet automate est couteuse et nécessite un grand espace mémoire : l'espace demandé est égal a $O(m \log m)$ bits ou m est la taille de motifs. Cet espace est prohibitif quand nous considérons que la représentation de l'ensemble de motifs demande seulement $m \log |\Sigma|$ bits. De ce fait pour une taille d'alphabet Σ constante, l'automate utilisera Ω (log m) plus d'espace que nécessaire.

D'autres parts l'algorithme de Rabin-Karp est l'une des solutions les plus populaires pour trouver le motif de manière plus efficace dans le problème de recherche par dictionnaire grâce à l'espace gagné par l'usage de la notion de hachage. L'accélération de ces deux algorithmes de recherche par dictionnaire est l'une des préoccupations majeures dans les domaines où l'algorithme de recherche de motifs est fortement utilisé. À l'ère du Big Data, l'accélération du traitement des textes est nécessaire pour assurer la rapidité du processus.

II.4. Les structures de données succinctes

Les structures de données sont utilisées pour organiser et stocker des informations afin d'interagir efficacement avec les données. Les arbres et les graphes, ils utilisent souvent des pointeurs pour représenter des liens entre les sommets ou les nœuds. Cette représentation n'est pas optimal car chaque pointeur doit adresser l'un des n emplacements de mémoire et donc nécessite $\log n$bits. Voyons d'abord la complexité spatiale des arbres sous forme de pointeur un arbre à n nœuds (n = 12) sous forme de pointeur. Une arborescence générale de n nœuds nécessite $O\ (nw)$ bits d'espace où $w \geq log2n$ est la longueur en bits d'un pointeur de la machine. Par conséquent, l'espace occupé est $O\ (n \log n)$. Examinons maintenant un arbre

étiqueté à n nœuds qui représente le dictionnaire suivant, D, où D contient des chaînes de l'alphabet $\Sigma : D = \{ab, bab, bca, cab, cac, cbac, cbba\}, \Sigma = \{a \dots z\}$

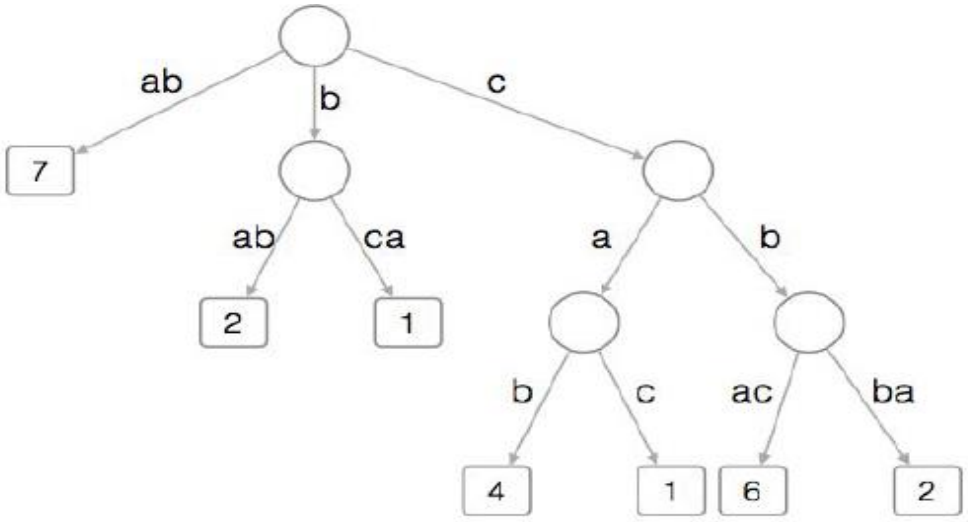


Figure 9 : *Arbre à 12 nœuds qui représente le dictionnaire D [40]*

Dans ce cas, chaque enfant d'un nœud est associé à une étiquette distincte dans une plage $[1, \Sigma]$. Dans ce cas, les chaînes de nœuds redondants (nœuds avec un seul enfant) sont jointes. Par exemple, le bord le plus à gauche, ab, correspond normalement à deux bords, a et b, joints en séquence. Les valeurs aux feuilles représentent le nombre de chaînes qui se terminent dans cette séquence. Par exemple, dans la liste de mots ci-dessus, le mot “cab” apparaît 4 fois.

Le traitement d'un ensemble de données volumineux présente le défi d'équilibrer le stockage, l’organisation et l’accessibilité à ces structures de données, d’où la nécessite une représentation succincte. Cette dernière, c'est une représentation des structures de données qui permet de réduire la taille M de la structure de données obtenue après prétraitement tout en maintenant un bon temps de réponse pour les requêtes. La structure de données est dite succincte si l'espace occupé est à un facteur constant près égal à l'espace occupé par les données sur lesquelles la structure est construite. La structure utilise un espace $\lambda+O(\lambda)$, ce qui implique que la structure à un surcoût en espace mémoire asymptotiquement négligeable comparé à l'espace λ occupé par les données initiales.

II.5. Parallélisme

Le traitement en parallèle est une technique d'information traitement qui met l'accent sur la concurrence manipulation de données appartenant à un ou plusieurs processus permettant de résoudre les problèmes. Cette technique est devenue populaire en raison du développement de la technologie de processeur, plus d'améliorations dirigé vers une architecture parallèle à la vitesse du processeur

La conception et l'analyse d'algorithmes parallèles est une pierre angulaire du domaine du parallélisme. Des algorithmes parallèles rapides sont nécessaires si l'on veut atteindre une réduction significative des temps de calcul dans la résolution de problèmes complexes sur des ordinateurs parallèles.

II.5.1. Définition d'un algorithme parallèle

La définition de la notion d'algorithme diffère sensiblement d'une référence à une autre. Cormen et al. [24] définissent un algorithme comme étant une procédure de calcul bien définie qui prend en entrée une valeur ou un ensemble de valeurs et qui produit en sortie une valeur ou un ensemble de valeurs. Selon ces auteurs, un algorithme est donc une séquence d'étapes de calcul permettant de passer de la valeur d'entrée à la valeur de sortie. On dit qu'un algorithme est correct s'il se termine avec une sortie correcte pour chaque instance d'entrée. L'algorithme résout alors le problème posé.

Cosnard et Trystram [25] définissent un algorithme séquentiel comme un ensemble d'opérations définies d'une façon rigoureuse et non ambiguë, de sorte que chacune des opérations puisse être effectuée par un ordinateur. Dans le cas des algorithmes parallèles, plusieurs opérations sont effectuées simultanément sur plusieurs processeurs. On peut définir un algorithme parallèle comme tout algorithme dans lequel les séquences de calcul peuvent être effectuées simultanément ou comme un algorithme dans lequel plusieurs opérations sont effectuées simultanément. Voici un petit exemple du calcule d’une opération d’addition exécuté en parallèle :

```
POUR i = 1 à n FAIRE EN PARALLÈLE
B(i) = A(i);
POUR h = 1 à log n FAIRE
      POUR i = 1 à n/2^h FAIRE EN PARALLÈLE
            B(i) = B(2i-1) + B(2i);
S = B(1);
```

Figure 10 : *Algorithme parallèle de calcul d'addition*

La Figure 10 présente un algorithme qui résout ce problème de calcul sur une machine avec n processeurs $\{P_1, P_2, \dots, P_n\}$. Le programme prend en entrée le tableau A. La Figure 11 illustre de façon arborescente le comportement de l'algorithme quand n est égal à 8

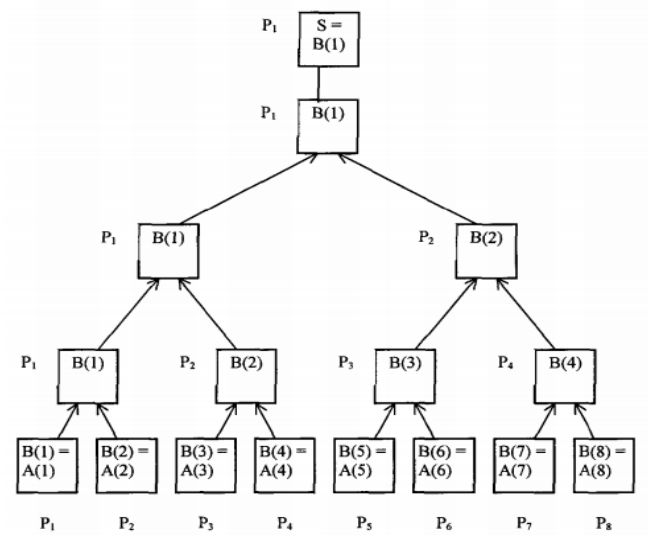


Figure 11 : *Schéma du calcul de la somme quand n = 8*

Chaque nœud interne Cet exemple très simple avait pour but de montrer de quelle façon il est possible d'exprimer le parallélisme d'un algorithme.

II.6. Conclusion

Dans ce chapitre, nous avons dressé un état de l'art sur la recherche du motif et les algorithmes de la recherche par dictionnaire dans un texte. Ces algorithmes sont présentés selon l'ordre chronologique.

Dans notre cas, nous avons choisi les algorithmes Aho-Corasick et Rabin Karp pour les étudier et les implémenter car ils sont les solutions les plus populaires pour trouver le motif de manière plus efficace dans le problème de recherche par dictionnaire.

La conclusion que nous pouvant tirer que la représentation d'algorithme Aho-Corasick est couteuse en terme de complexité spatiale. En raison de la grande utilisation des algorithmes Aho-Corasick et Rabin Karp dans plusieurs domaines, les accéléré est surtout nécessaire dans le domaine de la recherche des motifs.

En conclusion amélioré ces deux algorithmes, Nous proposons l'application des structure des données succinctes pour diminuer l'usage de mémoire pour l'algorithme Aho-Corasick et nous appliquons le parallélisme par l'utilisation de multithreading pour accélérer le processus de recherche de motifs par les deux algorithmes concernés.

III. RECHERCHE PAR DITCTIONNAIRE : APPROCHE SUCCINCTE ET PARALLÉLE.

III.1. Introduction

Dans ce travail, nous présentons deux algorithmes de recherche par dictionnaire, qui présente une recherche multiple de motifs. Le principe consiste à parcourir le texte une seule fois dans de gauche à droite. Dans la première section de ce chapitre nous présentons l'implémentation ainsi que la recherche (matching) de motifs d'Aho-Corasick. Dans la seconde, nous présentons l'algorithme de Rabin Karp ainsi que sa méthode de recherche.

III.2. Implémentation d'algorithme d'Aho-Corasick (version amélioré)

Pour l'implémentation de l'algorithme d'Aho-Corasick, nous nous sommes inspirés de la représentation mentionnée dans [23]. Dans l'automate d'Aho-Corasick, nous avons m états avec trois types de transitions: les transitions *next, fail et report*. Nous définissons l'ensemble P comme l'ensemble de tous les préfixes des chaînes de l'ensemble S, y compris la chaîne vide et toutes les chaînes de S, où S est l'ensemble de nos motifs. Pour chaque élément de P, nous avons un état correspondant dans l'automate et inversement. On a donc $|P| = m < n + 1$ états dans l'automate (où « n » est le nombre de caractère de S). Les états qui correspondent aux chaînes dans S sont appelés états terminaux. Nous définissons maintenant les trois types de transitions:

- Pour chaque état v_p correspondant à un préfixe p, nous avons une transition *next* (v_p, c) marquée du caractère c de l'état v_p à chaque état v_{pc} correspondant à un préfixe pc, pour chaque préfixe pc ∈ P. Nous pouvons donc avoir jusqu'à σ transitions *next* par état, où σ est la taille de l'alphabet.

- Pour chaque état v_p, nous avons une transition *fail* qui connecte v_p à l'état v_q correspondant au suffixe q de p le plus long tel que q ∈ P et p ≠ q.

- De plus, pour chaque état v_p, nous pouvons avoir une transition *report* de l'état v_p à l'état correspondant au suffixe le plus long q de p tel que q ∈ S et p ≠ q si un tel q existe. Si pour un état donné v_p, aucune chaîne de ce type n'existe, nous n'avons pas de transition *report* à partir de l'état v_p.

La figure 12 représente l'automate d'Aho-Corasick avec ses différentes transitions

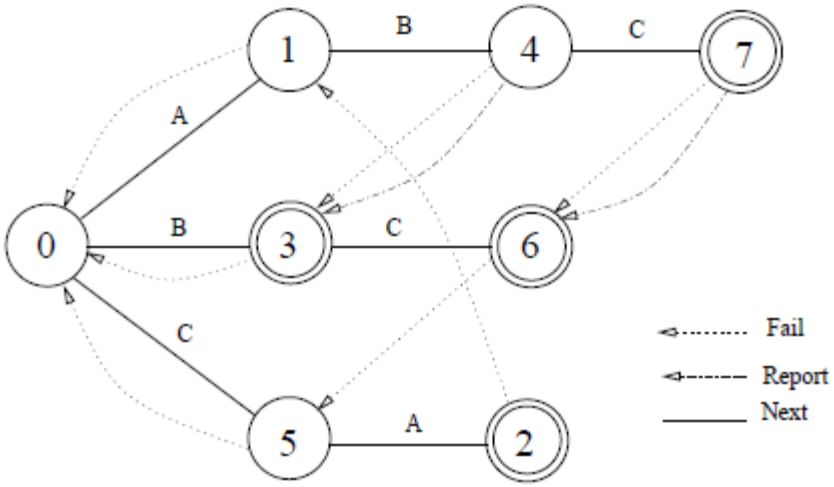


Figure 12 : *Automate d'Aho-Corasick de l'ensemble {"ABC","B","BC","CA"}*

Ainsi, représenter l'automate, il faudrait passer par deux étapes. La première étape s'intéresse à en la représentation des états, alors que la seconde s'intéresse à la représentation de ses trois ensembles de transitions : *next, fail et report.*

III.2.1. La représentation des états

Nous décrivons maintenant la représentation des états et la correspondance entre états et chaînes. Les états de notre représentation d'automate Aho-Corasick sont définis de la manière suivante : Soit P l'ensemble des préfixes des chaînes de S, et m = |P|. Nous identifions les états par des entiers dans l'intervalle [0; m-1], donnés par une fonction state définie sur l'ensemble P et telle que state(p) = $rang_p$ (p) est le rang de la chaîne p dans P suivant l'ordre lexicographique miroir.

Pour l'exemple de la figure 12, nous avons l'ensemble S = {"ABC","B","BC","CA"} et l'ensemble P = {"","A","AB","ABC","B","BC","C", "CA"}. Nous mettons P dans l'ordre lexicographie miroir et ce qui nous donne l'ordre suivant : P = {"","A","CA","B", "AB", "C", "BC", "ABC"}. L'ordre lexicographique miroir est défini de la même manière que l'ordre lexicographique standard, sauf que les caractères des chaînes sont comparés dans un ordre allant de droite à gauche plutôt que dans un ordre allant de gauche à droite. Afin de distinguer les états finaux des autres états, nous notons simplement que nous avons exactement « d » états terminaux correspondant aux « d » éléments de S. Comme indiqué dans la définition, chacun

des m états est identifié de manière unique par un nombre compris dans l'intervalle [0; m - 1]. Par conséquent, afin de distinguer les états terminaux des états non-terminaux, nous utilisons un vecteur *succinct* qui ne comporte que des bits (0 et 1) et dans lequel, une position i égale à 1 si et seulement si l'état i est final. La taille de ce vecteurs correspond au nombre d'états et le nombre de 1 dans le vecteur correspond au nombre de motifs.

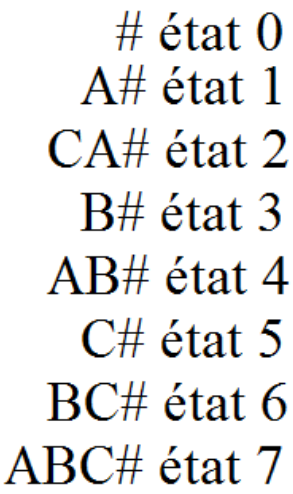


Figure 13 : *Représentation des états selon l'ordre lexicographie miroir*

III.2.2. La représentation des transitions *next*

Nous décrivons maintenant comment les transitions *next* sont représentées. Il est facile de voir que l'ensemble des états dotés des transitions *next* est précisément un Trie construit sur l'ensemble S, les états et les transitions *next* de l'automate correspondant aux nœuds et aux arêtes du trie. Tout d'abord, nous notons qu'une transition va toujours d'un état correspondant à un préfixe p ∈ P à un état correspondant à un préfixe pc ∈ P. Par conséquent, afin de trouver la transition étiquetée avec le caractère c et qui passe de l'état correspondant à la chaîne p (si une telle transition existe), nous devons trouver deux informations : existe-t-il un état correspondant au préfixe pc et le cas échéant quel est le numéro correspondant à cet état. En d'autres termes, étant donné state(p) (exemple de la figure 13 state(AB)=4) et le caractère c, il faudra déterminer s'il existe un état correspondant à pc, et si c'est le cas, détermine son identifiant. Par exemple nous avons p=AB, state(AB) = 4, nous devons trouver tous les préfixes AB s'ils existent dans les états, et on trouve ABC, state(ABC)=7, alors nous notons que nous avons une transition de

état 4 a l'état 7 avec le caractère C etc…Nous ferons cette opération jusqu'à ce que nous obtenions l'arbre de transition *next* :

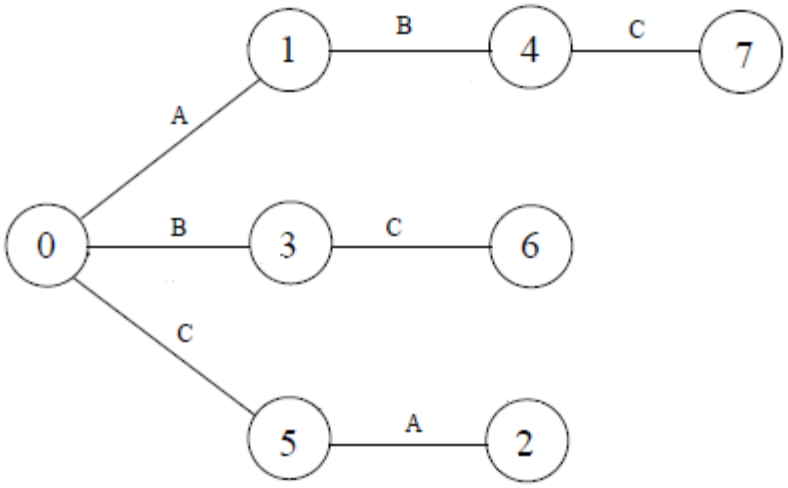


Figure 14 : *Représentation de l'arbre de transition next*

III.2.3. La représentation des transitions *fail*

Nous décrivons maintenant comment les transitions *fail* (échec) sont représentées. Une transition *fail* connecte un état représentant un préfixe p à l'état représentant un préfixe q, où q est le plus long suffixe de p tel que q ∈ P et q ≠ p. L'ensemble des transitions *fail* peut être représenté avec un arbre appelé arbre d'échec.

Chaque nœud dans un arbre d'échec représente un élément de P et chaque élément de P a un nœud correspondant dans l'arbre. L'arbre d'échec est simplement défini de la manière suivante:

- Le nœud représentant une chaîne p est un descendant d'un nœud représentant la chaîne q si et seulement si q ≠ p et q est suffixe de p.
- Les fils de n'importe quel nœud sont classés en fonction de l'ordre lexicographique miroir des chaînes qu'ils représentent.

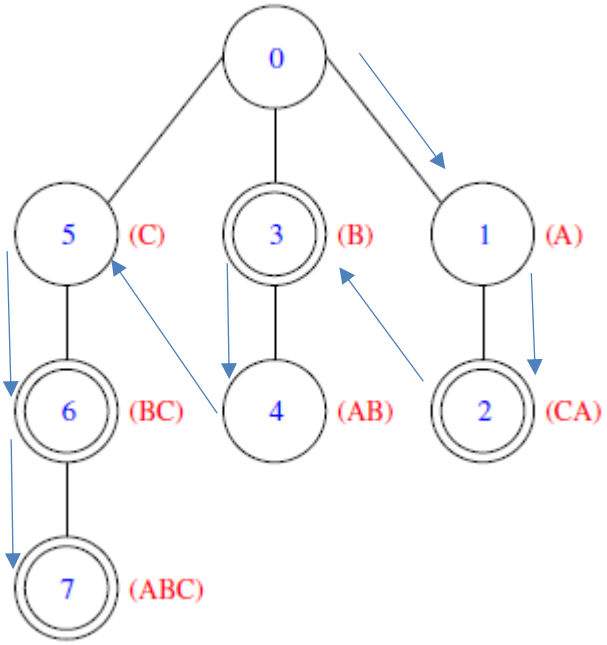


Figure 15 : *Représentation de fail transitions*

Maintenant, une observation importante sur l'arbre que nous venons de décrire, c'est qu'une première traversée en profondeur de l'arbre énumérera tous les éléments de P dans un ordre lexicographique miroir. C'est le pré-ordre des nœuds dans l'arbre qui correspond au suffixe de l'ordre lexicographique des chaînes de P. Il est clair dans la description ci-dessus que la recherche de la transition *fail* qui connecte un état (p) à un état (q) (où q est l'élément le plus long de P tel que q est un suffixe de p et q ≠ p) correspond à la recherche du parent dans arbre d'échec du nœud représentant l'élément q.(par exemple :transition *fail* de ABC de l'état 7 est le suffixe BC qui correspond à l'état 6 tel que BC est le plus long suffixe de ABC, etc…).

III.2.4. La représentation des transitions *report*

La représentation des transitions *report* est similaire à celle des transitions *fail*. La seule différence avec arbre d'échec est que, à l'exception de la racine, chaque nœud interne doit représenter un élément de S (un nœud interne, tous nœud possédant un fils). Nous remarquons que les transitions *report* forment une forêt d'arbres, qui peut être transformée en un arbre en connectant toutes les racines de la forêt (nœuds n'ayant pas de transition *report*) comme enfants de l'état 0 (qui devient ainsi la racine de l'arbre).

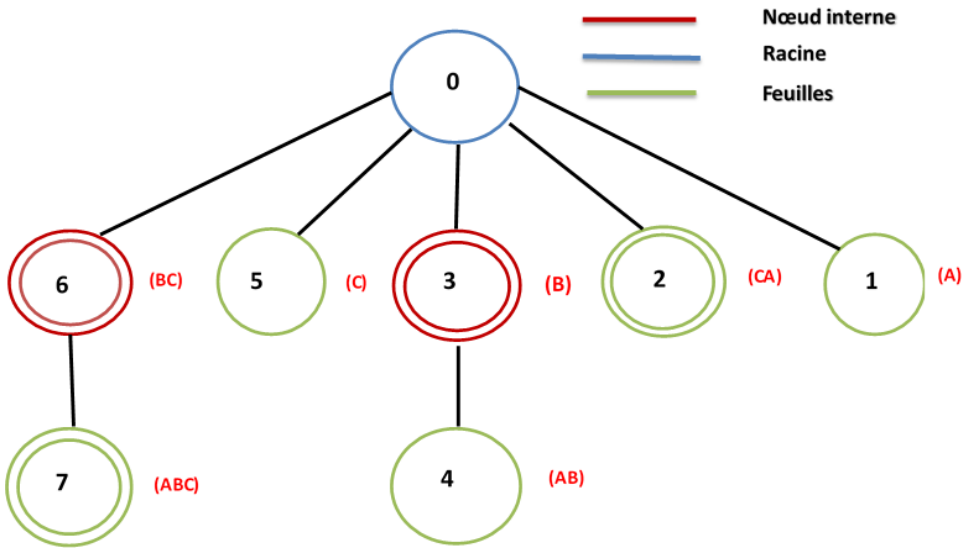


Figure 16 : *Représentation de l'arbre de transition report*

Pour chaque nœud interne tel que ni ∈ S. Exemple : *report* de l'état 7 est le nœud interne 6 (état 6).

L'arbre *report* est l'unique arbre construit sur les éléments de P qui satisfait :

- chaque nœud de l'arbre représente un élément distinct de P.
- Tous les nœuds sont des descendants de la racine (représentant l'état 0) qui représente la chaîne vide.
- Le nœud représentant une chaîne p est un descendant d'un nœud représentant une chaîne non vide s si et seulement si s ∈ S, s ≠ p et s est un suffixe de p.
- Tous les enfants d'un nœud donné sont classés en fonction de l'ordre lexicographique miroir des chaînes qu'ils représentent.

III.2.5. Représentation d'occurrence des motifs trouvés

Après la lecture du texte et la construction de l'automate, le texte se parcours caractère par caractère. Tous d'abords, nous ferons la correspondance à partir de l'état 0. Une fois une transition trouvée à partir d'une position du texte, nous testons pour chaque état différent de 0 si c'est un état final à partir du vecteur de bits mentionné au-dessus (1 état final, 0 non final). Et pour chaque état différent de 0 nous testons si ce dernier a un *report* transitions pour reporter

tous les motifs. Si par la suite nous ne trouvons pas une transition *next*, nous allons vers une transition *fail* et ainsi de suite. Du coup, le texte est parcouru une seul fois et la complexité du parcours est de O(n).

Notre automate Aho-Corasick correspondra aux chaînes de S qui sont des suffixes de préfixes du texte T. Cela signifie que l'automate Aho-Corasick affichera les positions finales des occurrences. Toutefois, l'utilisateur peut également avoir besoin d'indiquer la position de départ des occurrences. Pour cela, nous avons choisi de signaler les occurrences sous forme de triplets (occ_start_pos; occ_end_pos; string_id), où string id ∈ [0 ; d-1] est l'identifiant de la chaîne correspondante (pour chaque états final est un id du motif) et occ_start_pos (occ_end) est la position de début (fin) de l'occurrence dans le texte. Pour cela, nous devons connaître la longueur des chaînes correspondantes. C'est pour cela, nous ajoutons un tableau pour stocker ces information, pour chaque position du tableau correspond à l'état final du motif (par exemple état 7 correspond au motif "ABC"), alors dans la position 7 du tableau on stock 3 pour dire que le motif est de longueur 3.

Le principe d'algorithme Aho-Corasick est montré dans l'algorithmique ci-dessus

Algorithm 1 principe $Aho - Corasick$

Require: $dict, T$;
Require: $src = 0, dist$;
Require: $n, m, i,$:Entier;
$i \leftarrow 0$;
$n \leftarrow taille(T)$;
$Gen - transition(next, report, fail, dist, taille(dist))$;
while $i < n$ **do**
  $dist \leftarrow next(etat, T[i])$;
  $i \leftarrow i++$;
  while $dist \neq 0$ **do**
    $etat \leftarrow dist$;
    while $report(etat) \neq 0$ **do**
      $etat \leftarrow report(etat)$;
      print "Motif trouvé a la positions i"
    end while
    if $is - final - state(dist) = 1$ **then**
      print "Motif trouvé "
      $etat \leftarrow dist$;
    end if
    $src \leftarrow dist$
    $dist \leftarrow next(src, T[i])$;
    $i \leftarrow i++$;
    if $dist = 0$ **then**
      while $fail(src, T[i]) \neq 0$ **do**
        $dist \leftarrow fail(src, T[i])$
        $src \leftarrow dist$
      end while
    end if
  end while
end while

Algorithme 1 : *Principe d'algorithme d'Aho-Corasick*

III.2.6. Coût mémoire

Récapitulation de l'utilisation de l'espace utilisée en pratique pour la représentation de l'automate Aho-Corasick :

- Pour le vecteur qui indique si c'est un état final, occupe m bits (ou m est le nombre d'état).
- Pour la transition *next*, nous avons trois type de donné, l'état source, état suivante, et le caractère de transition, tous cela occupe (1 + 4 + 4) × m octet, où 1 octet pour

représenter le caractère de transition, 4 octet pour représenter état source et 4 octet pour représenter état de destination.

- Pour la transition *fail* occupe 4 × m octet, l'indice de la transition représente l'état source, alors nous avons besoin de 4 octet pour représenter l'état destination.
- Pour la transition *report* occupe 4 × m octet, l'indice de la transition représente l'état source, alors nous avons besoin de 4 octet pour représenter l'état destination.

Où *m* est : $m \leq \sum card(motifs)$

Nous remarquons que la représentation de l'automate en intégralité est couteuse en terme espace. Dans un exemple de recherche de séquence ADN avec pour chaque séquence, nous avons 100 caractère, de plus nous cherchons plus de 1 milliard de séquence ADN, ça devient très couteux en espace mémoire et sa dépasser la mémoire de la RAM que nous pouvons allouer. Pour cela nous avons pu optimiser l'automate et la rendre moins couteuse avec les structure de donnés succinctes en utilisé les vecteurs de bits pour représenter la transition *next*, et balanced parentheses pour représenter les transitions *fail et report*.

III.3. Représentation succincte des transitions d'algorithme Aho Corasick

Pour l'utilisation des vecteurs à bits et de balanced parentheses, nous utilisons une librairie **SDSL LITE** qui définit les vecteurs de bits de manière optimisée, et qui inclut des fonctions pour balanced parentheses que nous avons utilisé pour représenter les transitions *fail et report*. Nous avons utilisé trois de ces fonctions : *select ( ), rank ( ) et enclose ( ).* On a deux type de *select ( )* une pour vecteurs à bits et l'autre pour balanced parentheses, et deux type de *rank ( )* une pour les vecteurs à bits et l'autre pour balanced parentheses.

Select ( ) : Renvoie la i-ème position de occurrence de 1 dans le vecteur de bits.

Select ( ) : Renvoie l'index de la i-ème parenthèse ouvrante.

Rank ( ) : Retourne le nombre de parenthèses -1 ouvrantes jusqu'à et y compris l'index i.

Rank ( ) : renvoie ((le nombre de 1) -1) dans le préfixe [0..m-1] dans le bit vecteur pris en charge.

Enclose ( ) : return Index de la parenthèse ouvrante qui correspondant au parent de la parenthèse courante, s'il existe son parent.

III.3.1. La représentation succincte de transitions next

L'idée est simple, nous construisons un vecteur de bits $T_b$de taille (nombre d'état × taille d'alphabet) qui sera initialisé à 0 en premier temps (d'où le nombre d'état dans ce vecteur représente l'état source de transition). Une transition est définie comme suit : pour chaque transition, nous indiquons l'état source et l'état de destination et le caractère de transition, d'où l'état source, c'est la position de l'état par rapport au nombre d'état dans ce vecteur, et le caractère de transition, c'est la position du caractère par rapport à la taille de l'alphabet. Alors, pour chaque transition avec un caractère, nous indiquons l'état source et la position du caractère qui correspond au a sa position dans l'alphabet (nous multiplions cette position par le nombre d'état pour faire la rotation dans vecteur de bits voir figure 17) et nous mettons cette position à 1. Cette position dans le vecteur de bit est donnée par la formule suivante :

$$\boldsymbol{T_b[\,(nombre\ d'état \times position\ du\ caractère) + état\ source] = 1}$$

Exemple : nous avons une transition de l'état 0 vers l'état 1 avec le caractère 'A' du coup, on met cette position à 1, cette position est donné avec la formule suivante :

Etat source = 0, position du caractère 'A' = 0, nombre d'état = 7

$$\boldsymbol{T_b[\,(7 \times 0) + 1] = 1}$$

Le tableau suivant correspond à un exemple de représentation de transition *next* avec les bits vecteur. Nous prenons comme taille de l'alphabet = 3 {'A', 'B', 'C'}

Les états	*0*	*1*	*2*	*3*	*4*	*5*	*6*
A	*1*	*0*	*0*	*0*	*0*	*1*	*0*
B	*1*	*1*	*0*	*0*	*0*	*0*	*0*
C	*1*	*0*	*0*	*1*	*1*	*0*	*0*

Tableau 2 : *Représentation succincte simplifié de transition next.*

Ce tableau sera étendu sur les 256 caractères du code ascii, afin d'entreprendre tous les caractères. Alors, nous généralisons la formule précédente par la formule suivante :

$$\boldsymbol{T_b\left[\left(nombre\ d'état \times code\ ASCII\ (c)\right) + état\ source\right] = 1}$$

Où « c » est le caractère de la transition.

Le parcours de cette représentation est simple, nous avons une transition de l'état 0 vers l'état 1 avec le caractère A. l'état source est donné par la position sur le vecteur, par contre pour savoir l'état de destination, nous calculons le nombre 1 de la position 0 vers la position courante incluant la position courante, et c'est le rôle de la fonction *rank ( ).*

Pour savoir si nous avons une transition, nous testons avec la formule suivante :

$$\textbf{Si}\quad \boldsymbol{T_b\left[\left(nombre\ d'état \times code\ ASCII\ (c)\right) + état\ source\right] == 1}$$

Où« c » est le caractère de transition.

Nous dirons que nous avons une transition de l'état source avec le caractère A si est egal a 1. Exemple : Nous avons une transition de l'état 5 avec le caractère A vers l'état 2, état 2 est obtenu par le nombre 1 qui précède la position courante. (Voir figure 17)

Nous avons une transition de l'état 3 avec le caractère C vers l'état 6, état 6 est obtenu par nombre de 1 qui précède la position courante (Voir figure 17). L'état3 du tableau 2 c'est la position 17 et c'est pour cela que nous multiplions le code ascii de la transition par le nombre d'état pour faire la correspondance et la rotation des états de 0 à 6.

En réalité le vecteur à bit est de la forme suivante :

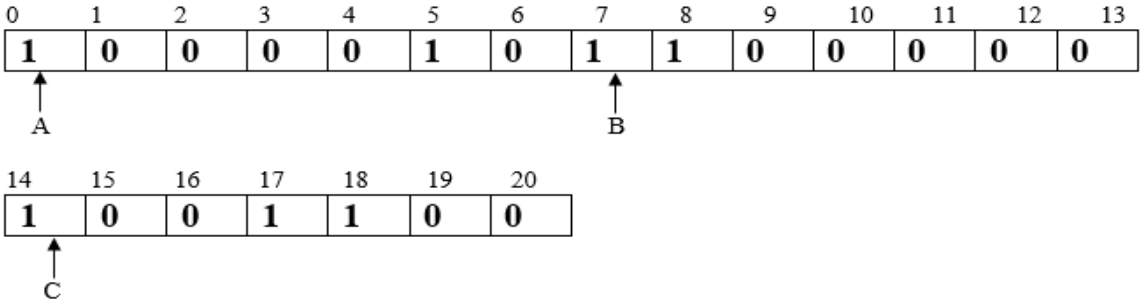


Figure 17 : *Représentation de transition next avec le vecteur de bit*

L'état de destination est donné avec la formule suivante :

Etat destination = *rank (nombre* d'état × ASCII (c)+ état source + 1*)*

Où « c » est le caractère de la transition de l'arbre.

III.3.2. La représentation succincte de transitions *fail*

L'idée de la représentation de transition *fail* est d'utilisé la notion de parent avec parenthèses équilibrés (balanced parentheses) sur un vecteur de bitF_b. La taille de ce tableau sera deux fois le nombre d'état (2× m), car une parenthèse ouvrante pour dire que nous sommes dans cet état, et une parenthèse fermante pour dire que nous sommes sortis de cet état. Nous représentons une parenthèse ouvrante avec 1 et une parenthèse fermante avec un 0, voir la figure 18.

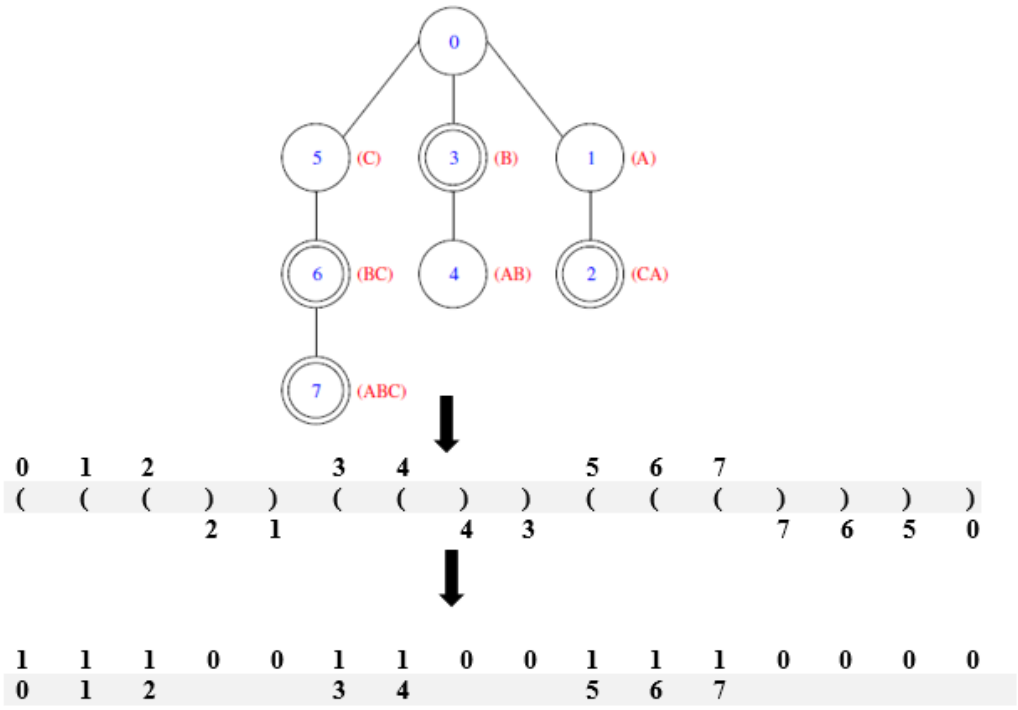


Figure 18 : *Représentation de fail transition avec vecteur de bits*

L'idée de construction est simple, tout d'abord, le vecteur de bitF_b est initialisé qu'avec la parenthèse de la racine de l'arbre de *fail* transition (voir figure 19), par la suite, nous parcourons notre arbre et pour chaque état nous cherchons la parenthèse fermante (nous parcourons le bit vecteur de gauche à droite) du parent de l'état actuel (voir figure 19)

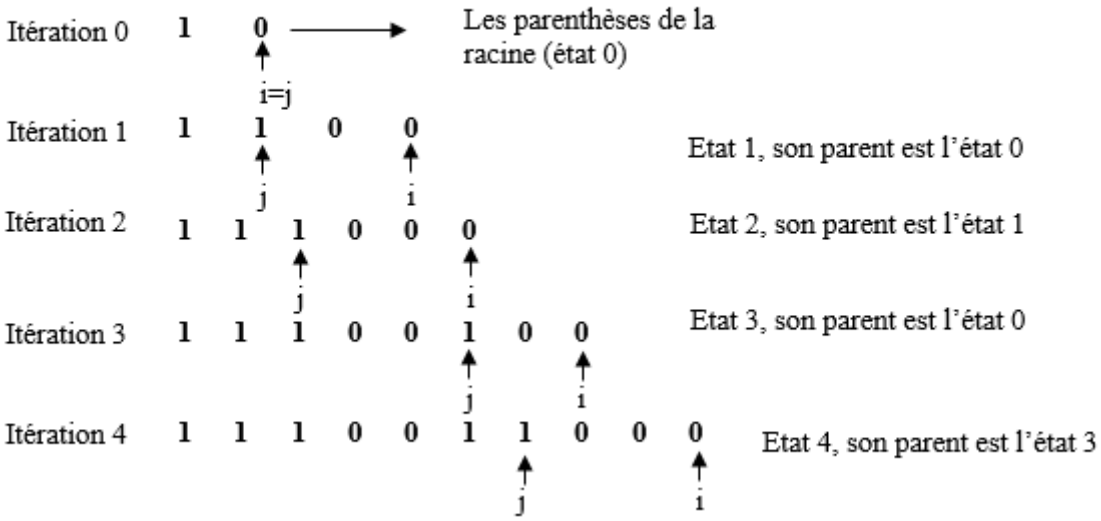


Figure 19 : *Construction de transitions fail par balanced parantheses*

La figure 19, présente la construction de parenthèses équilibrées (balanced parentheses). Dans l'itération 1, nous somme dans l'état 1de l'arbre de transition *fail* et son parent est l'état 0, alors, nous cherchons la parenthèse fermante de l'état 0 qui est dans la position j à partir de la position i (ici i=j) et nous insérons les parenthèses de l'état 1 (1= parenthèses ouvrante, 0= parenthèses fermante). Dans l'itération 2, nous sommes dans l'état 2 de l'arbre de *fail* transition et son parent est l'état 1, alors, nous cherchons la parenthèse fermante de l'état 1 qui est dans la position j à partir de la position i et nous insérons les parenthèses de l'état 2. (1= parenthèses ouvrante, 0= parenthèses fermante).

Nous ferons le même travail pour les autres états jusqu'à ce que nous arrivions au résultat de la figure 18.

Pour avoir une transition *fail* d'un état, nous utilisons trois fonction, *select (i), enclose (i) et rank (i).* La fonction *select* (i) nous donne la position de la i-éme parenthèse ouvrante d'un état ce qui signifie la position de notre état dans le vecteur à bit (exemple : *select* (état = 4 + 1) nous renvoie la position du 5éme bit a 1, car nous avons l'état 0), la fonction *enclose (i)* nous donne la position de la parenthèse ouvrante du parent de l'état courant dans le vecteur (exemple : *enclose (*état = 4*)* nous renvoie la position de la parenthèse ouvrante de l'état 3, car l'état 3 est le parent de l'état 4). Et la fonction *rank(i)* nous calcule le nombre de bit à 1 inferieur à la

position courante incluant le bit de la position courante (exemple : *rank (*position état = 3*)* nous renvoie 4, car il inclut l'état 0, alors, ça devient *rank ( )-1*.

La formule globale pour obtenir une transition *fail* d'un état est comme suit :

$$\boldsymbol{F_b = rank\ (\ enclose\ \big(\ select\ \ (état\ source + 1)\big) - 1)}$$

III.3.3. La représentation succincte de transitions *report* :

Pour la transition *report* R_bc'est le même principe de *fail*, nous travaillons avec l'arbre de *report* mentionné dans la figure 16. Nous la représentons avec les *balanced parentheses*. L'idée est la même que la transition *fail*. La formule générale pour trouver une transition *report* est comme suit :

$$\boldsymbol{R_b = rank\ (\ enclose\ \big(\ select\ \ (état\ source + 1)\big) - 1)}$$

D'où la fonction de construction de $R_b$est la même que celle de $failF_b$

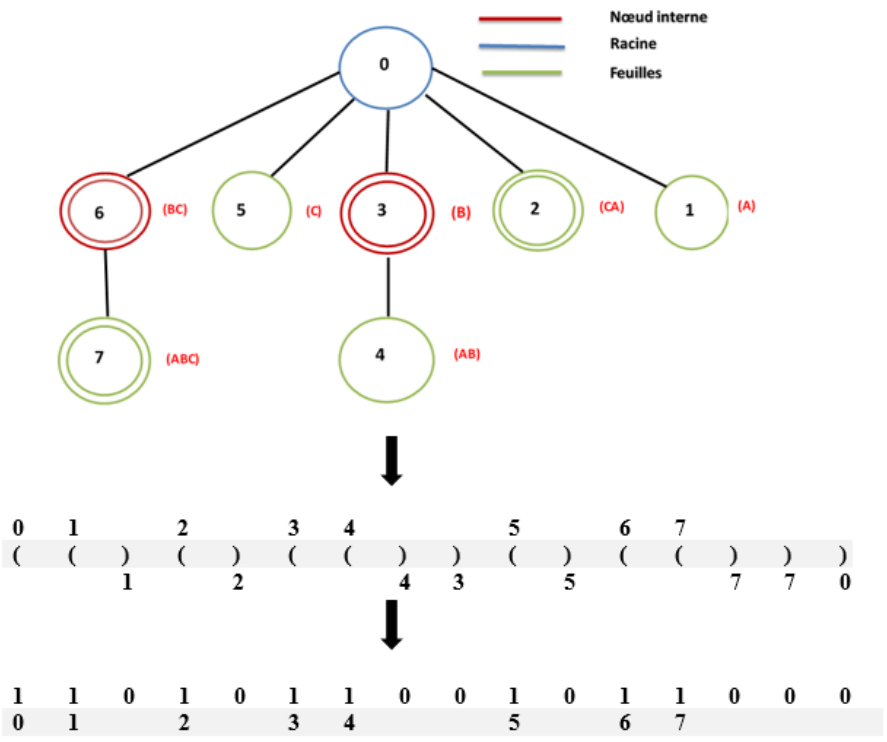


Figure 20 : *Représentation de transition report avec vecteur de bit*

III.3.4. Coût mémoire de la représentation succincte

Pour cette représentation succincte de l'automate Aho-Corasick, cout en espace en théorie est donné comme suit :

- Pour le vecteur à bit, qui stocke les états terminaux (chaque indice de ce vecteur indique l'état de l'automate, si cette état est terminal la position est égale a 1 sinon 0), occupe $m.\log\frac{n}{m}$ bits [27] où n est la longueur du vecteur (nombre d'état) et m est les bits définis (les bits égaux a 1).

- Pour les transitions *next* : nous utilisons la représentation sd_vector (sparse bits vector en englais) [27] qui utilise $m.\log\frac{n}{m}$ + o(m) pour représenter un vecteur de n bits dont m bits sont égaux à 1. Dans notre cas nous avons :

  $m.\log\frac{n}{m} + \text{o(m)} \leq \text{m.log}(\sigma) + 2\text{m+o(m)}$ bits où $n = \text{m} \times \sigma$ est la longueur du vecteur (nombre d'état multiplié par la taille de l'alphabet) et m est le nombre de transitions (égal au nombre d'états).

- Pour les transitions *fail et report* : nous utilisons des vecteurs de balanced parenthenses qui occupe $2 \times m + O(\text{m} \times \log\frac{m}{\log m})$ bits [26].

Mais le problème avec cette représentation succincte, c'est que les requêtes vers ces structures son complexe, nous perdons en terme temps de recherche de motif (ou le parcours du texte avec l'automate). C'est pour cela, une parallélisations de recherche de motifs saura cacher ce problème.

III.4. Implémentation de l'algorithme Rabin Karp multi-motifs

Nous implémentons dans notre étude l'algorithme Rabin Karp à multiple motifs de taille m, pour un dictionnaire de k motif $k > 1$. Notre travail est inspiré de l'étude [19]. L'algorithme Rabin Karp utilise la technique de hachage pour trouver les occurrences des motifs d'un dictionnaire contenant un ensemble de motifs P dans un texte T. L'algorithme fait la correspondance en comparant les motifs et les sous chaines de texte à travers une fonction de hachage.

Pour implémenter l'algorithme, il faut passer par deux étapes : la phase des prés-traitement, où nous faisons la représentation du dictionnaire et le calcul du hachage, et la phase de recherche des motifs dans le texte.

III.4.1. La phase de prétraitement

Nous décrivons maintenant la représentation du dictionnaire. Nous définissons un ensemble D de motifs p de taille $k > 1$ ou chaque motif est de taille m. Nous mettons l'ensemble D dans un tableau de taille k.

Pour commencer la recherche des motifs dans un texte quelconque, il faut d'abord calculer les valeurs de hachages de l'ensemble des motifs. Nous construisons une structure de donnée contenant ces valeurs de hachages.

III.4.1.a. La représentation du dictionnaire

Nous décrivons maintenant comment construire notre structure de hachage. Pour représenter le dictionnaire, nous construisons une table de hachage tab_hash qui contient les haches des motifs. Nous appliquons la méthode de essai linéaire (linear probing en anglais)[38] qui est défini dans notre implémentation comme suit : Soit tab_hash de $taille = (k \times 8/6)$, la multiplication par $8/6$ est pour avoir des cases vides et plus d'espace que la taille de dictionnaire (la table est remplie à 75%). En premier lieu, tab_hash est initialisé a -1 (la valeur -1 représente la case est vide), pour tout $i = \{0,1, \dots, taille\}$. Pour remplir la table, Nous calculons le hachage du motif p_i en employant la fonction de hachage. Nous mettons le hachage du motif à la position $p \ mod \ taille$ (où taille représente la taille de la table de hachage). Pour chaque motif, nous calculons son hachage et nous le mettons dans la table à la position donnée par la formule :

$$\boldsymbol{pos_{h_p} = h_p \ mod \ taille}$$

Si la position $\boldsymbol{pos(h_p)}$ n'est pas vide, c'est-à-dire $\boldsymbol{tab\_hash[\ pos(h_p)] \neq (-1)}$, dans ce cas nous allons vérifier la position suivante d'une manière circulaire (si nous atteindrons la fin de table, la prochaine position à choisi est 0) jusqu'à l'obtention d'un case vide (-1) et nous mettons le hachage dans cette position. Ce processus est répété jusqu'à l'obtention k haches des motifs dans tab_hash. La figure 21 montre un exemple d'une table de hachage en appliquons

linear probing. Nous avons $D = \{26,35,7,52,102,56,124,45,96\}$ alors $k = 9$. Pour calculer la position du hachage d'un motif, nous utilisons la formule $h(x) = x\ mod\ taille$ pour $taille = {}^{9*8}/_{6}$.

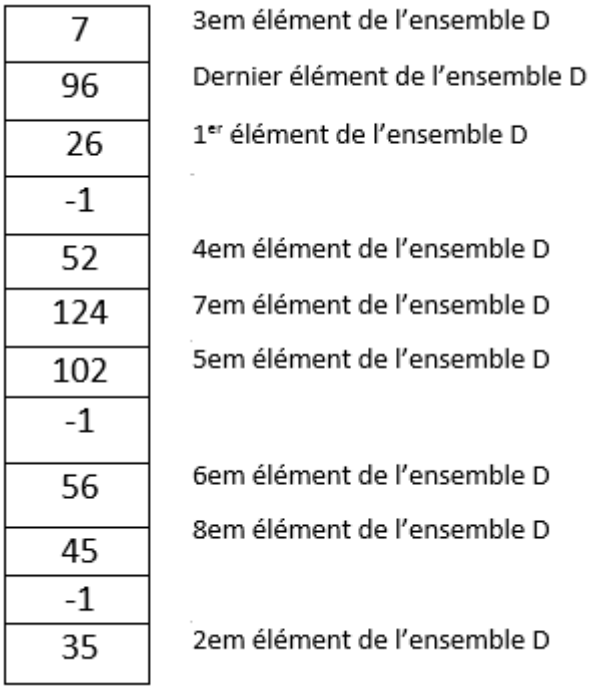


Figure 21 : *Exemple sur une représentation de la table de hachage*

Nous calculons le hache de chaque élément de l'ensemble D, nous commençons par $p = 26$ donc nous avons $h(26) = 26 mod 12 = 2$, Cela implique que le 1er élément sera stocké dans l'emplacement 3 du tableau, pour le 2em élément de D, nous avons $p = 35$ de la même manière $h(35) = 35\ mod\ 12 = 11$. Ce calcule est répété pour chaque élément de D pour avoir une table de hachage.

III.4.1.b. La fonction de hachage

Pour la construction de la table de hachage, une bonne fonction de hachage est demandée, c'est ce qui détermine l'efficacité de l'algorithme. Nous implémentons l'algorithme en appliquant le hachage de roulement de Rabin Karp (Rolling hash en anglais) [29]. Cette méthode n'est pas différente de celle de la recherche simple, sauf que le calcul de la valeur de hache de la position i est basé sur la valeur de hachage de la position i-1.

Le hache h dans algorithme Rabin Karp est définit en appliquant la règle d'Horner [31], il calcule le hache du motif en temps linéaire O(m), avec la formule qui suit :

$$\boldsymbol{h = (P[m] + d\,P[m-1] + d^2P[m-2] + \cdots + d^{m-1}P[2] + d^mP[1]) mod\ q}$$

Où q est un nombre premier très grand (par exemple q=1000000007) et d est un nombre quelconque dans l'intervalle [0..q-1]. De la même maniéré, nous calculons le hachage de la première fenêtre du texte de taille m. Cependant, la seule difficulté avec cette procédure est que h peut-être trop grand pour pouvoir fonctionner convenablement (si nous avons la taille de pattern très grande, un débordement de la variable allouer).

Nous définissons la fonction de hachage de la manière suivante : Soit D l'ensemble des motifs P de taille m qui contient k motifs, et nous avons un texte T de taille n. Nous commençons par hacher le motif P et une fenêtre s qui contient une sous chaine du texte de taille m. Nous remplaçons les caractères par leurs valeur sur la table d'ASCII. Pour surmonter le problème de la taille de hachage, nous multiplions la valeur ASCII par un nombre premier que nous appelons une base d à la puissance du la position du caractère c dans la chaines. De plus, nous prenons le reste de division du hachage h calculé par un grand nombre premier q (le choix de q est important pour éviter le problème de débordement des variables allouer). Cette procédure est appliquée pour tous les caractères de motif ou bien sur les fenêtres du texte.

Donc le hachage des motifs et de la première sous chaine du texte sera calculé par la formule suivante :

$$\boldsymbol{h = \sum_{i=0}^{m-1}\left(ACSII(c_i) * d^{m-i}\right) mod\ q\ .}$$

La figure 22 montre un exemple de calcul de hachage pour une chaine T= {ABC} pour une base d et un nombre premier q

T=

A	B	C

$$\boldsymbol{h(t)_1 = \left((65 \times d^3) + (66 \times d^2) + (67 \times d^1)\right) mod\ q}$$

Figure 22 : *Exemple de calcul d'un hachage pour la première fenétre*

Nous appliquons le hachage à roulement, pour calculer le hachage du la prochaine fenêtre du texte. Au début, nous avons la première fenêtre de taille m, nous décalons à la droite en supprimant le premier caractère de la fenêtre c_0 et nous ajoutons le caractère c_m. Nous remarquons que le hachage de la nouvelle fenêtre du texte h_{t_new} peut-être calculer à partir de hachage de l'ancienne fenêtre h_{t_old}

Alors, h_{t_new} est calculé en soustrayant le hachage du premier caractère c_0 et en ajoutant le hachage du nouveau caractère c_m.

$$\boldsymbol{h_{t\_new} = (d \times (h_{t_{old}} - h_{c_i}) + d \times h_{c_{i+m}}) mod\ q}$$

Ou (c_i) est le premier élément de la fenêtre précédente et (c_{i+m}) est le dernier caractère de la fenêtre courante

La figure 23 montre un exemple de calcul de hachage pour une chaine T= {ABCD} pour une base d et un nombre premier q.

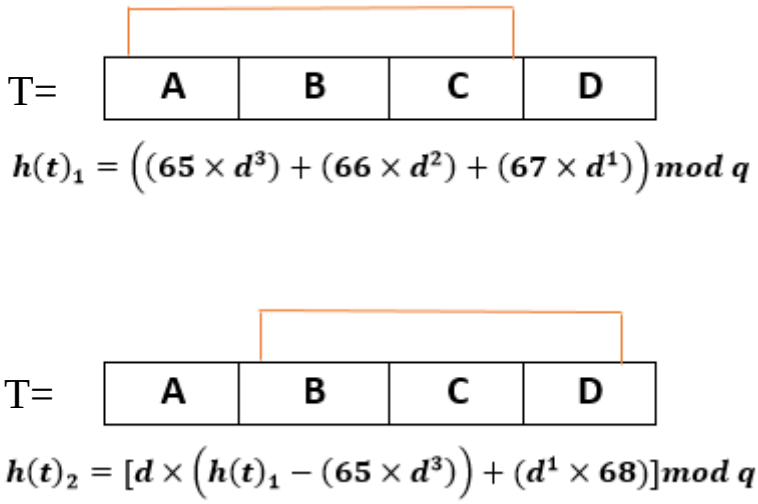


Figure 23 : *Exemple de calcul un hachage roulement pour un texte T*

III.4.2. La phase de recherche

III.4.2.a. La comparaison des valeurs hachage

Après la construction de la table de hachage, nous vérifions l'existence des éléments (motifs) du dictionnaire *D* dans le texte *T*. Le principe est simple, nous comparons la valeur de hachages

de des fenêtres du texte avec la table tab_hash. Pour chaque hachage d'une fenêtre du texte, nous vérifions la position de ce hachage dans la table de hachage avec formule suivante :

$$\boldsymbol{pos(fenêtre) = hash(fenêtre)\ mod\ taille}$$

A partir de la position obtenu, nous comparons la valeur du hachage de la fenêtre du texte avec les hachages jusqu'à obtention d'une correspondance ou bien arrivé à une case vide (-1). La méthode de linear probing assure la rapidité de recherche en évitant de parcourir toute la table de hachage.

III.4.2.b. Comparaison naïve

Malgré le fait de trouvé une bonne fonction de hachage, la probabilité d'une collision est toujours présente. Pour cela, une fois une correspondance d'un hachage est trouvé, Nous ferons une comparaison naïve, c'est-à-dire caractère par caractère, cette comparaison permet d'éviter les cas de collisions (une collision est d'avoir deux sous chaine différentes qui ont la même valeur de hachages)

Si les deux sous chaines correspondent, nous retournons l'occurrence de l'élément i de l'ensemble des motifs et sa position d'occurrence j dans le texte. Ce processus de recherche est répète jusqu'à atteindre la fin du texte T. L'algorithme suivant montre le principe algorithme de Rabin Karp.

Algorithm 1 Calculate *RabinKarp*

Require: $q, d, hash_p, hash_t, h$: Entier;
Require: n, m, i, j:Entier;
  $hash_p \leftarrow 0$;
  $hash_t \leftarrow 0$;
  $pat[] \leftarrow pattern$;
  $m \leftarrow strlen(pat)$;
  $txt[] \leftarrow pattern$;
  $n \leftarrow strlen(txt)$;
  $h \leftarrow d^{m-1} modq$;
  for $i \leftarrow 0$ to m **do**
    $hash_p \leftarrow (hash_p * d + pat[i]) modq$;
    $hash_t \leftarrow (hash_t * d + txt[i]) modq$;
  end for
  for $i \leftarrow 0$ to $n - m$ **do**
    if $hash_p = hash_t$ **then**
      for $j \leftarrow 0$ to m **do**
        if $pat[j] = txt[i + j]$ **then**
          print "occurence a la positions i"
        end if
      end for
    else
      $hash_t \leftarrow (d * (hash_t - txt[i] * h) + txt[i + m]) modq$;
    end if
  end for

Algorithme 2 : *Principe d'algorithme de Rabin Karp*

En ignorant les collisions, les collisions, qui ont une probabilité «faible», il existe n - m + 1 de chaînes de longueur appropriée, l'algorithme prendra le temps $O\ (n\ -\ m\ +\ 1\ )$ [30] pour calculer $h_p$est de $O\ (p)$ et pour le prétraitement $O(1)$[31]. En résultat nous aurons une complexité de O (m + n). Dans le pire des cas, est $O(knm)$, puisque dans le pire des cas, on devra faire une comparaison de chacune des n-m+1 sous-chaînes du texte de tailles m avec tous les k motifs stockés dans la table de hachage.

.

III.4.3. Parallélisations de la recherche de motifs

Pour une recherche optimiser en espace mémoire, et de recherche de motif optimisé, la parallélisations de recherche saura cacher le problème complexe des requêtes de cette représentation succincte.

L'idée est simple, c'est de diviser le texte selon le nombre de thread envoyé. Mais un problème surviendra, c'est que, à la position de la division, nous pourrons perdre des motifs qui sont dans cette position.

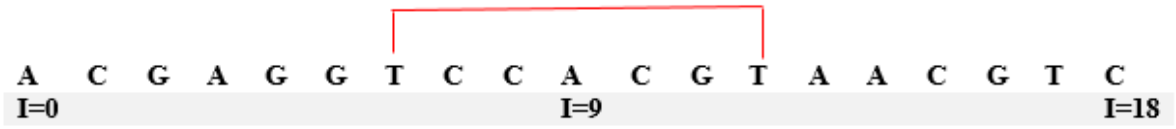


Figure 24 : *La perte de motif dans la position 9*

Dans la figure 24, nous avons un texte de taille 19, et nous lançons deux thread pour la recherche de motif, le premier de la position i = 0 à i = 9, et le deuxième de i = 9 à i = 18.

Dans cette exemple nous prenons {ACGA, TCCACGT, AAC, CCAC}. Si nous remarquons bien, les deux motifs TCCACGT et CCAC sont au milieu de la division, du coup les deux motifs ne seront pas trouver.

Pour résoudre ce problème, nous ferons un chevauchement à la position de la division du texte. Le premier thread sera dont dans la position $9 + P_{mx} - 1$ où P_{mx} est la taille du plus grand motif. Et le seconde thread, ne reportera pas les motifs qui sont entre la position du premier thread et sa position actuelle pour éviter de reporter deux fois le même motif.

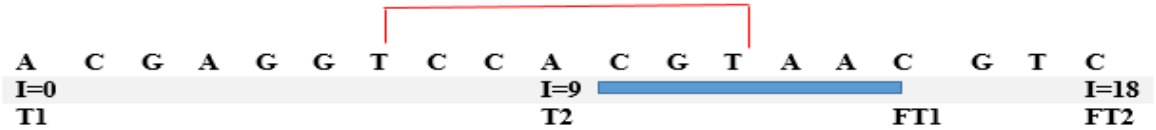


Figure 25 : *Chevauchement de parallélisations*

Tous les motifs qui sont au-dessous du trait bleu seront trouver par le T1, et ignorer par le T2, pour éviter la redondance.

III.4.4. Conclusion

L'implémentation des deux algorithmes présentée dans ce chapitre vise à réduire au maximum l'utilisation mémoire surtout pour l'automate d'Aho-Corasick. Elle vise aussi à réduire le temps de traitement des textes (détections des motifs) en utilisant le multithreading.

En d'autres termes la représentation succincte ainsi que l'utilisation du parallélisme ont permis de rendre les algorithmes plus adaptatifs a tous type machines.

IV. TESTS ET RESULTATS

IV.1. Introduction

Cette phase a pour objet de mettre en œuvre notre implémentation proposée. En premier lieu, nous présenterons l'environnement d'implémentation qui illustre les équipements matériels et logiciels pour la construction de notre system de teste. Ensuite, dans la deux partie section 1, nous ferons des tests sur les deux algorithmes, par la suite dans la section deux, nous calculons efficacité de ces implémentations proposé dans le chapitre précèdent.

IV.2. Environnent implémentation

IV.2.1. Equipement logiciel (Système d'exploitation)

L'environnement Linux (Ubuntu) (version 18.04) a été choisi pour l'implémentation et la validation de notre implémentation**.** Ubuntu [33] est une des nombreuses distributions du système d'exploitation GNU/Linux, souvent abrégée en Linux. Elle est inspirée d'une autre distribution (l'une des plus célèbres, nommée Debian) et est sponsorisée par la société Canonical Ltd. Ubuntu a été créé avec pour caractéristiques principales d'être :

- Conviviale : Cette distribution est donc l'une des plus faciles à l'utilisation.
- Simple : un seul outil pour chaque tâche qui fait ce qu'on demande.
- Libre : Comme un logiciel libre est le plus souvent gratuit et le code ouvert, l'avantage est que chacun peut participer à son développement.
- Gratuite : La gratuité fait partie intégrante de la philosophie d'Ubuntu. Tout le monde doit pouvoir avoir accès à un système d'exploitation et à des applications performantes sans être obligé de payer [33].

De plus, Ubuntu est considéré comme une distribution stable et sécurisée pour le développement, et supporte le parallélisme, c'est pourquoi nous avons décidé de l'utiliser dans notre travail.

IV.2.1.a. Librairie SDSL LITE

Il s'agit d'une librairie réalisée par SIMON GOG [34], qui contient l'implémentation de beaucoup de structures de données succinctes. Nous l'avons utilisé pour l'implémentation de la version succincte de l'automate d'Aho-Corasick.

Cette librairie permet de représenter un objet (tel qu'un vecteur de bits ou une arborescence etc..) dans un espace proche de la limite inférieure de l'objet en théorie tout en prenant en charge efficacement les opérations sur l'objet d'origine. La complexité temporelle théorique d'une opération effectuée sur la structure de données classique et la structure de données succincte sont (la plupart du temps) identique.

SDSL-LITE comporte des structure de données de vecteur de bits, notre choix s'est porté sur sd_vector (sparse bit vector) [27] pour sa capacité de stockage très réduite par rapport à bit_vector (bit vector) et de son support efficace des deux requêtes *rank ( ) et select ( )*. La figure 26 montre l'avantage de sd_vector par rapport à bit_vector en termes d'espace de stockage.

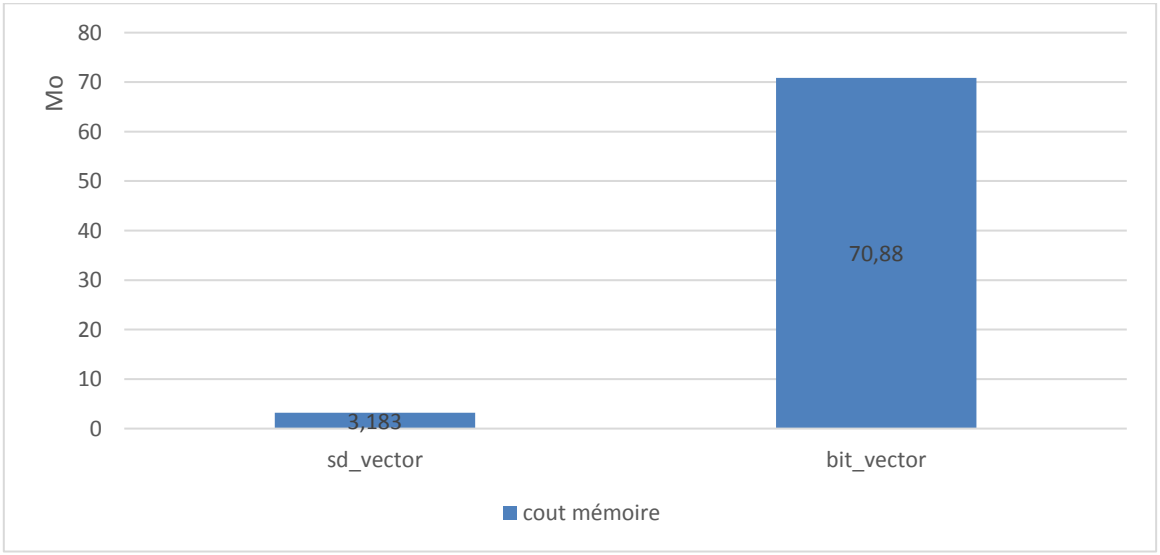


Figure 26 : *Histogramme représentant le coût mémoire pour le stockage du dictionnaire des URL*

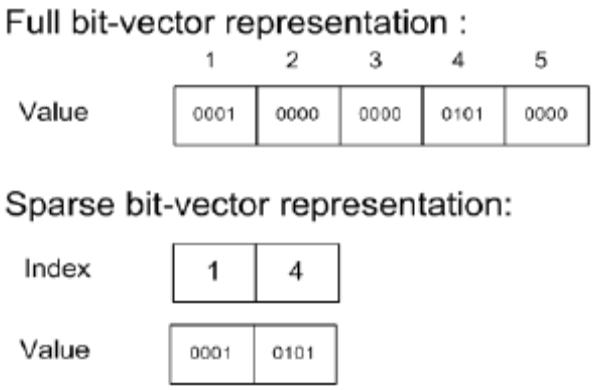


Figure 27: *Technique de représentation de sparse bit vector et bit vector [27]*

IV.2.2. Equipement matériel (machine)

La machine dans laquelle nous ferons les tests est équipe d'un processeur INTEL i7-4510u comportant deux cœurs cadencé à 2 GHz et pouvant exécuter chacun deux threads en parallèle. La machine est également équipée de 8 Go de RAM.

IV.2.3. Langage de programmation et compilation

Nous avons implémenté les deux algorithmes en langage de programmation C/C++, car ils offrent une marge de contrôle importante sur la machine (notamment sur la gestion mémoire).

Pour la compilation de l'algorithme Aho-Corasick, nous avons utilisé la commande

*g++ -std=c++11 -O3 -I ~/include -L ~/lib *.cpp *.c -o program -lsdsl -ldivsufsort -ldivsufsort64g++ -fopenmp -std=c++11 -O3 -DNDEBUG -I ~/include -L ~/lib *.cpp *.c -o program -lsdsl -ldivsufsort -ldivsufsort64.*

Le g++ c'est compilateur pour les programmes écrit en C++. Nous avons utilisé le -O3 [37] pour une meilleure optimisation pour plus de performance pour nos algorithmes. Le reste de la commande est pour la compilation de la librairie SDSL-LITE [34].

Pour la compilation de l'algorithme Rabin Karb, nous avons utilisé la commande suivante :

g++ Rabin_Karp.cpp –o Rk_exe –O3

IV.3. Les Mesures de performance

La programmation séquentielle est une suite d'instructions qui s'exécutent les unes après les autres. A contrario, La programmation parallèle (multithread) permet d'exécuter plusieurs instructions en même temps.

Les mesures de performance consistent en un ensemble de métriques pouvant être utilisées pour quantifier la qualité d'un algorithme. Pour les algorithmes séquentiels, les métriques temps et espace sont suffisantes. Pour les algorithmes parallèles, le scénario est un peu plus compliqué. Outre le temps et l'espace, des métriques telles que l'accélération et l'efficacité sont nécessaires pour étudier la qualité d'algorithme parallèle. De plus, lorsqu'un algorithme ne peut pas être complètement parallélisé, il est utile d'avoir une estimation théorique de l'accélération maximale possible. Dans ces cas, les lois d'Amdahl et de Gustafson deviennent utiles pour une telle analyse

- **L'accélération (Speed up)**

Elle consiste à mesurer à quel point un algorithme parallèle est plus rapide que le meilleur séquentiel. Pour un problème de taille n, l'expression d'accélération est donnée par:

$$Sp = \frac{Ts(n,1)}{T(n,p)}$$

Où $Ts\ (n,1)$ est le temps du meilleur algorithme séquentiel (c'est-à-dire,$Ts\ (n,1) \leq T\ (n,1)$) et $T\ (n,p)$ est le temps de l'algorithme parallèle avec p processeurs, résolvant tous les deux le même problème.

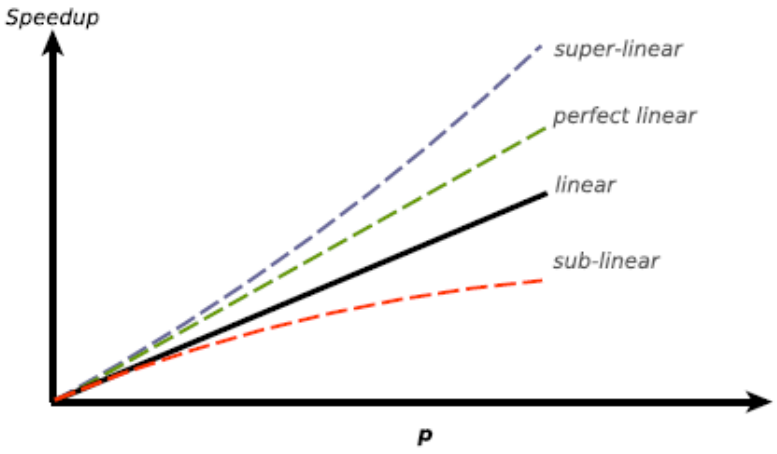


Figure 28: *Les quatre courbes possibles pour l'accélération [10]*

La figure 3 montre les courbes possibles de l'accélération en fonction du nombre de processeurs. Si l'accélération augmente linéairement en fonction du nombre de processeur p, nous parlons alors d'accélération linéaire. L'accélération linéaire signifie que la surcharge de l'algorithme est toujours dans la même proportion que son temps d'exécution, pour tout processeur. L'accélération idéale (idéal speed up en anglais) ou l'accélération linéaire parfaite (perfect liner speed up en anglais) est la valeur théorique maximale d'accélération qu'un algorithme parallèle peut atteindre lorsque n est fixé. Il est difficile de réaliser une accélération linéaire et encore moins qu'une accélération linéaire parfaite, à cause des goulots d'étranglement de la mémoire (memory bottlenecks en anglais) et de la surcharge qui augmente en fonction $de\ p$ [10]. Ce que nous constatons dans la pratique, c'est que la plupart des programmes atteignent des performances d'accélération sous-linéaires, c'est-à-dire $T\ (n, p)\ \geq\ Ts\ (n, 1)\ /$ p où le n est la taille du problème.

Si un problème ne peut pas être complètement mis en parallèle (une des causes de l'accélération sous-linéaire), une expression d'accélération partielle est nécessaire. Amdahl et Gustafson ont proposé à chacun une expression permettant de calculer l'accélération partielle. Ils sont connus comme les lois de l'accélération.

- **La loi d'Amdhal**

Soit c la fraction d'un programme parallèle, $(1 - c)$ la fraction d'exécution séquentielle et p le nombre de processeurs. La loi d'Amdahl [11]stipule que pour un problème de taille fixe, l'accélération globale attendue est donnée par :

$$S(p) = \frac{1}{1 - c + \frac{c}{p}}$$

Si $p \approx \infty$ alors :

$$S = S(p) = 1/(1 - c)$$

La loi d'Amdahl est utile pour les algorithmes qui doivent adapter leurs performances à une fonction du nombre de processeurs, en fixant la taille du problème.

- **La loi de Gustafson**

La loi de Gustafson [12]est une autre mesure utile pour l'analyse théorique de la performance, elle utilise le modèle à temps fixe où le travail par processeur est maintenu constant lors de l'augmentation de p et n. Dans la loi de Gustafson, le temps d'un programme parallèle est composé d'une partie séquentielle et une partie parallèle c exécutée par p processeurs.

$$T(p) = s + c$$

Si le temps séquentiel pour tout le calcul est $s + cp$, alors l'accélération est:

$$S(p) = \frac{s}{s + c} + \frac{cp}{s + c}$$

Définissant a comme fraction du calcul en série $a = s / (s + c)$, la fraction parallèle est égale à $1 - a = \frac{c}{s+c}$:

$$S\,(p) = \; a \; + \; p\,(1 - a) = \; p - a\,(p - 1).$$

- **L'Efficacité**

E_p est l'efficacité d'un algorithme utilisant un nombre p de processeurs, elle indique à quel point les processeurs sont utilisés.$E_p \; = \; 1$ est l'efficacité maximale et signifie une utilisation optimale des ressources. Une efficacité maximale est difficile à atteindre dans une solution mise en œuvre (c'est une conséquence de la réalisation difficile d'une accélération linéaire parfaite). Aujourd'hui, l'efficacité est devenue aussi importante que l'accélération, sinon plus, car elle mesure la qualité du matériel utilisé et elle indique quelles mises en œuvre doivent avoir la priorité lors de la compétition pour des ressources limitées :

$$Ep \; = \frac{Sp}{p} = \frac{Ts(n,1)}{pT(n,p)} \leq \; 1$$

IV.4. Etude du comportement des algorithmes

Pour étudier l'efficacité de nos deux algorithmes, nous avons pris des jeux de données [35] [36] dans trois domaine diffèrent : traitement du texte, bio-informatique et wikis. Pour le traitement du texte, nous avons pris un dictionnaire anglais qui comporte 213 000 mots, et comme texte, la bible de KING JAMES qui comporte plus de 100 000 lignes. Pour la bio-informatique, nous avons utilisé une base de séquence d'ADN d'une taille 100 caractères chacune. Cette base, qui représente notre dictionnaire, comporte 99 995 séquences ADN que nous devons rechercher dans une séquence (texte) qui se compose de 11 million de caractères.

```
GTGTTCCTCAAAAACCTATGCTAAAAAAAAATTTTCCCTTATAACCATATTTAAGTATACGGTTCTGAAGCATTAAGTACATTTACATTGTTGTGCAACC
TACATTTACATTAAAAAAAAAAAAAAAAAAAGCCCCCGAAATGCACCTCCTATGCAAACTTGGACCATGCTTAAAGTGTGTTCCTTCATTTTAGTAATTGCA
```

Figure 29 : *Représentation de deux séquences ADN du dictionnaire ADN*

IV.4.1. Etudier la construction de l'automate Aho-Corasick dans le temps et l'espace

Pour étudier le comportement de construction de l'automate Aho-Corasick, nous avons effectué des tests de construction de l'automate avec un dictionnaire de 1 million de site populaire [35]. A chaque test, nous augmenterons le nombre de motifs (taille du dictionnaire) et nous calculons le temps de construction de l'automate.

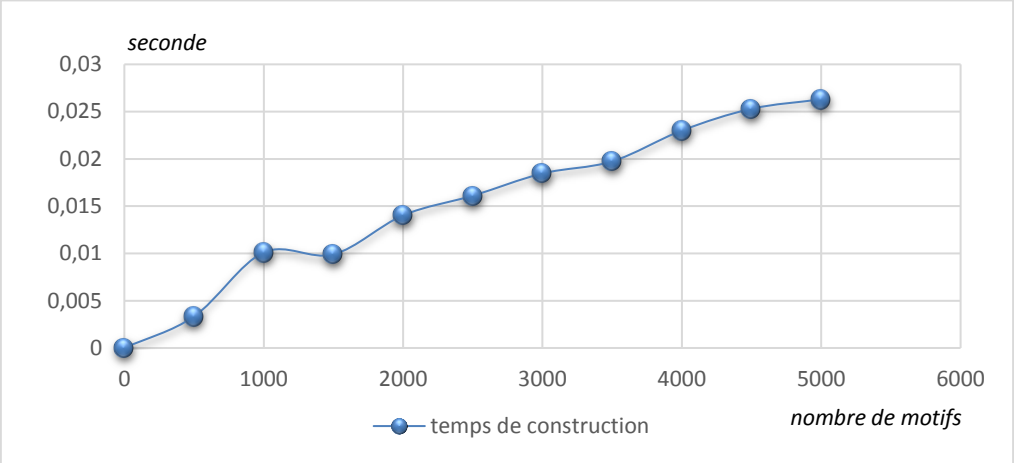


Figure 30 : *Graphe représentant le temps de construction de l'automate d'Aho-Corasick*

Nous remarquons que le temps de construction de l'automate d'Aho-Corassick est quasiment linéaire par rapport à la taille du dictionnaire.

IV.4.1.a. Etudier le temps de recherche dans le texte

Pour effectuer ce test, nous avons utilisé les jeux de données mentionnés en dessous. Nous calculons le temps de réponse des deux algorithmes. En premier lieu, nous utilisons les deux représentations d'Aho Corasick décrites dans le chapitre 3. Les occurrences des motifs du dictionnaire seront générées sous forme d'un fichier texte. La figure 30 montre le résultat obtenu pour l'automate classique amélioré.

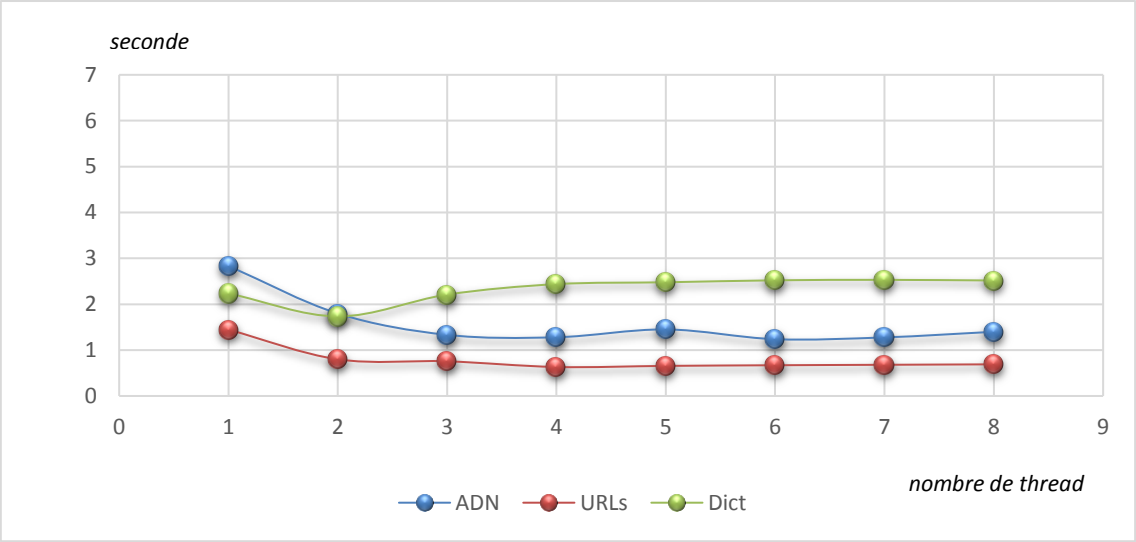


Figure 31 : *Graphe représentant les résultats de temps de recherche de motifs par l'automate selon le nombre de thread lancés*

Nous remarquons que le temps d'exécution diminue à mesure que le nombre de threads augmente jusqu'à ce que la machine atteigne son maximum de cœur logiques (4) et se stabilise ensuite.

Pour la représentation succincte nous avons les résultats suivant :

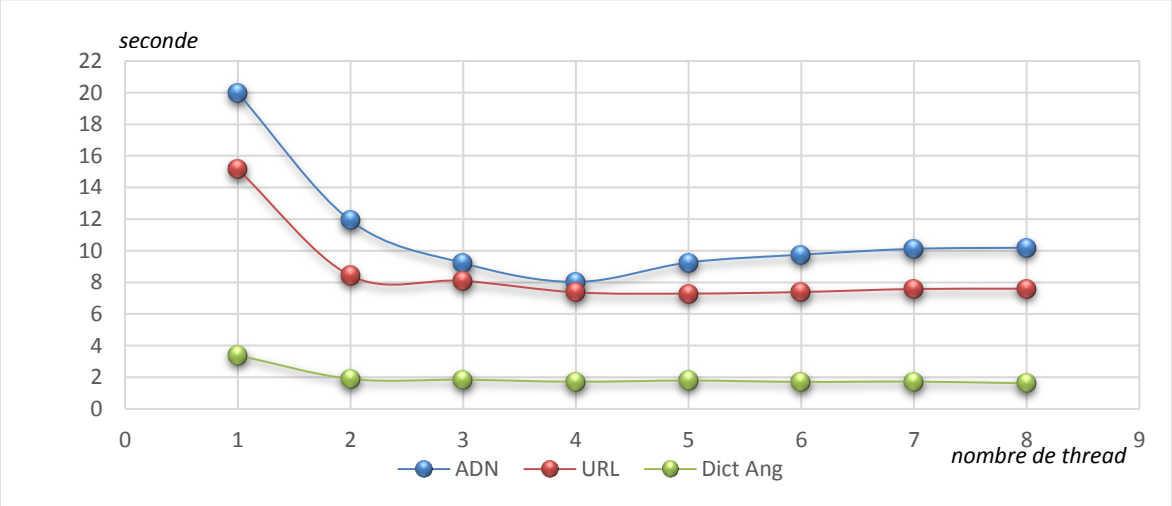


Figure 32: *Graphe représentant le temps de recherche par la construction succincte selon le nombre de thread lancé*

Nous remarquons que la recherche avec la représentation succincte a consommé plus de temps que la version automate, mais le parallélisme a permis de réduire ce temps ce en le rapprochant du temps d'exécution séquentiel de la recherche utilisant l'automate classique.

IV.4.2. Calcule du speed up

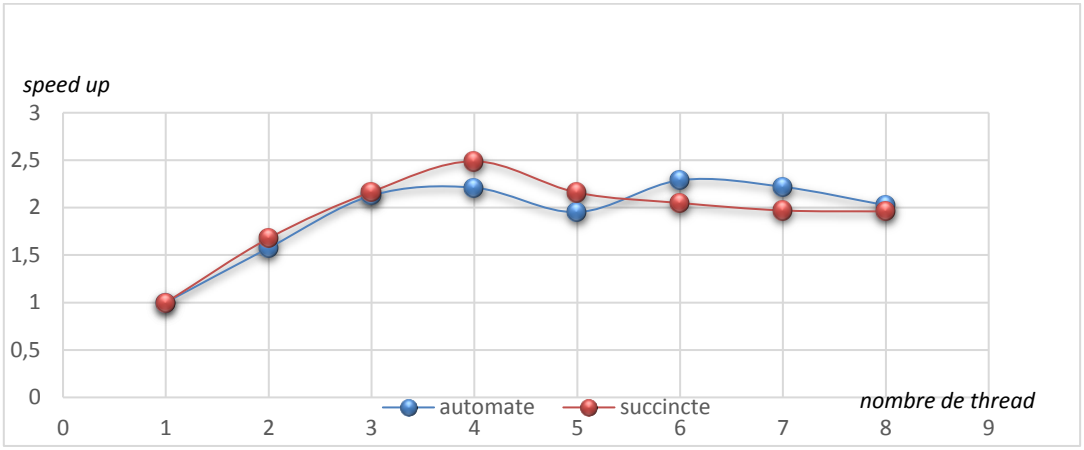


Figure 33 : *graphe représentent le calcul du speed up de test ADN des deux représentations de l'automate d'Aho-Corasick*

Dans le graphe de la figure 32, nous avons choisi d'utiliser la séquence d'ADN pour le calcul du speed-up. Nous remarquons que le speed-up est linéaire jusqu'au nombre de thread est égale au nombre de cœurs logiques de la machine. Ensuite, le speed-up devient sous-linéaire, car le nombre de threads dépasse la capacité de la machine.

Nous remarquons que le speed up de la représentation succincte est légèrement supérieure au speed-up de l'automate Aho-Corasick.

$$\boldsymbol{Efficacité_{succincte}} = \frac{\boldsymbol{Speedup(p)}}{\boldsymbol{p}} = \frac{\boldsymbol{1,677}}{\boldsymbol{2}} = \boldsymbol{83.8\%}$$

$$\boldsymbol{Efficacité_{automate}} = \frac{\boldsymbol{Speedup(p)}}{\boldsymbol{p}} = \frac{\boldsymbol{1,576}}{\boldsymbol{2}} = \boldsymbol{78.8\%}$$

Nous remarquons que l'efficacité de la représentation succincte est supérieure à l'efficacité de la représentation classique, pour le teste de séquence ADN. Nous dirons que l'utilisation des processeurs par la représentation succincte est légèrement supérieure.

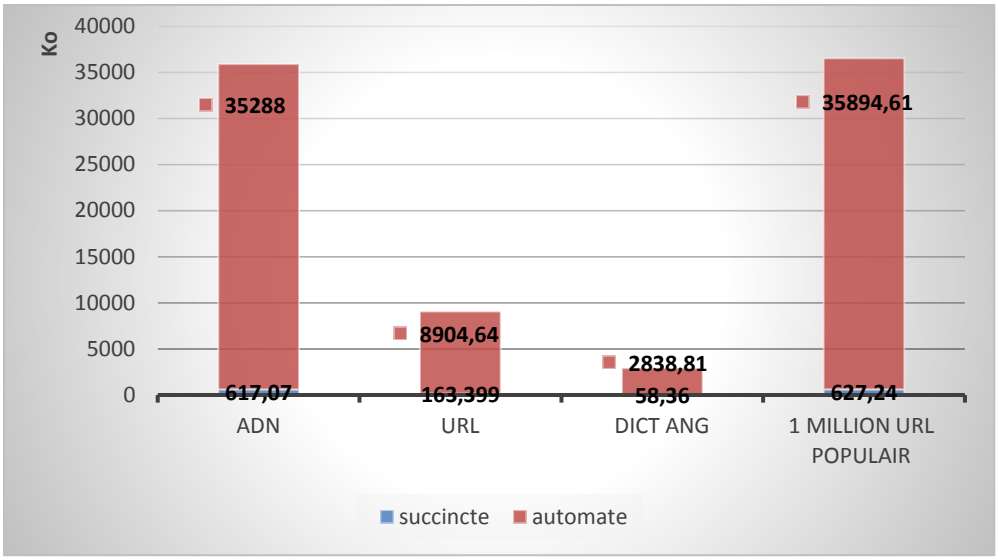


Figure 34 : *Histogramme représentant le coût mémoire des représentations de transitions fail et report d'Aho Corasick*

Dans la figure 33, nous remarquons une forte consomation mémoire de la représentation classique de l'automate d'Aho-Corasick en comparaison de la représentation succincte qui occupe beaucoup moins d'espace permettant ainsi d'utiliser des dictionnaires beaucoup plus grands.

Le meilleur speedup (et dont l'efficacité) obtenu par la représentation succincte pourrait être dû au fait que l'algortihme passe plus de temps dans les calculs (e.g. éxecution des fonctions *rank* et *select*) plutôt que dans les accés mémoire, alors que l'algorithme sur la représentation classique utilise très peu de calculs et passe plus de temps dans des accés mémoires potentiellement couteux.

Il est bien connu que les architectures multicoeurs sont beaucoup plus efficaces pour accélérer les calculs plutôt que les accès memoires. D'autre part, la représentation succincte étant beaucoup plus petite, elle à beaucoup plus de chance de tenir dans les niveaux bas de la hiérarchie mémoire (caches mémoires), et ainsi permettre une meilleure efficacité dans une architecture multi-cœurs.

IV.4.3. Etude de l'algorithme parallèle de Rabin Karp

Dans cette partie, nous calculons le temps de la repense de l'algorithme parallèle de Rabin Karp. Pour effectuer ce teste, nous utilisons les jeux de données des séquences ADN mentionnées en dessous.

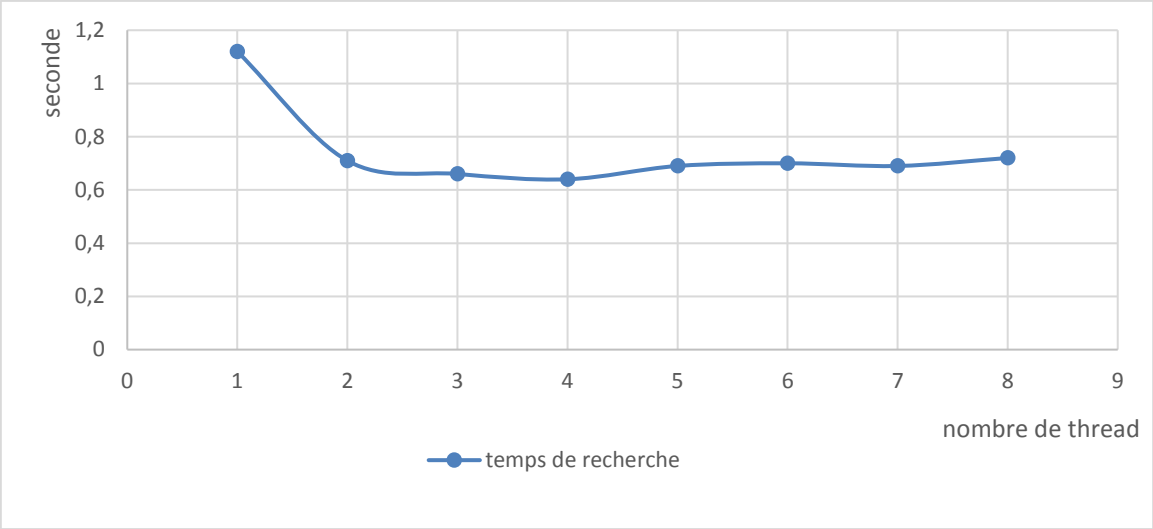


Figure 35 : *Graphe représentant les résultats de temps de recherche de motifs par l'algorithme parallèle de Rabin Karp selon le nombre de thread lancé*

Le résultat de cette figure illustre clairement ce que nous avions affirmé dans le figure 30, que le temps d'exécution diminue à mesure que le nombre de thread augmente jusqu'à atteindre le nombre de cœur logiques de la machine (4).

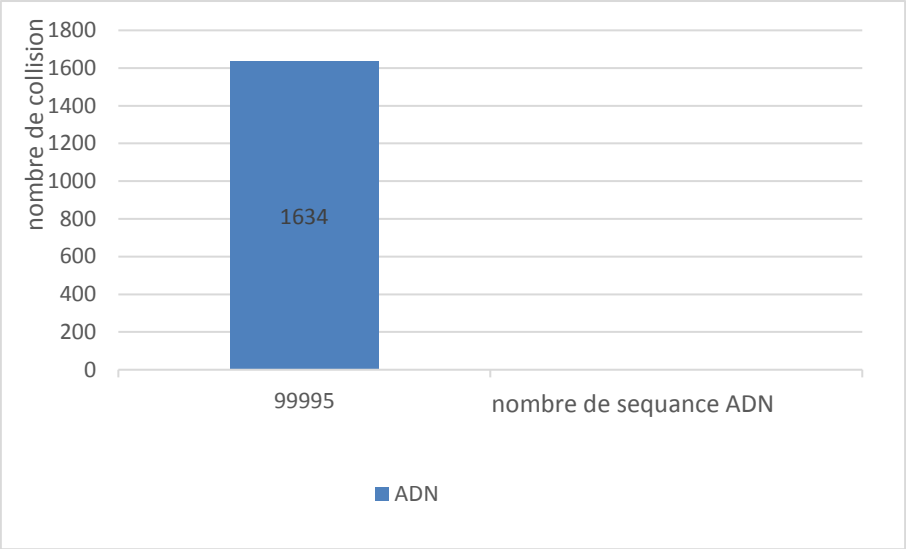


Figure 36 : *Histogramme représentant le nombre de collision dans l'algorithme Rabin Karp par rapport au nombre de séquence ADN*

Dans la figure 35, nous remarquons que le nombre de collisions est petit par rapport au nombre de séquence d'ADN/ Ce résultat permet de a montré l'efficacité de la fonction de hachage utilisé

IV.4.4. Calcul du speed up :

Pour le teste nous utilisons la séquence d'ADN pour le calcul du speed-up, et les résultats sont dans la figure ci-dessus

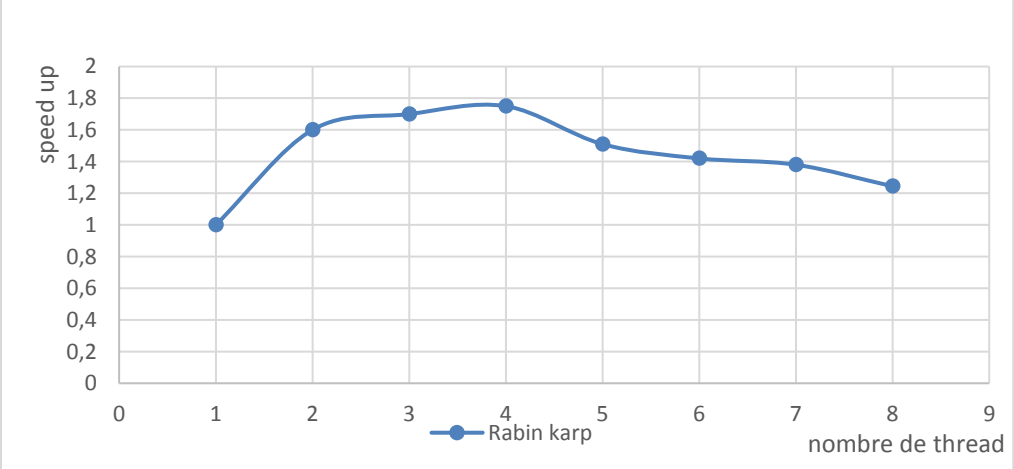


Figure 37 : *Graphe représentent le calcul du speed up de test ADN pour l'algorithme Rabin Karp*

Dans le graphe de la figure 36, Nous remarquons que le speed-up augmente d'une manière linéaire jusqu'à ce que le nombre de threads soit égale au nombre de cœurs logiques de la machine. Ensuite, le speed-up devient sous dégrade et devient sous-linéaire, à cause du nombre de threads qui a dépassé la capacité de la machine.

$$\boldsymbol{Efficacité_{RK} = \frac{Speedup(p)}{p} = \frac{1{,}6}{2} = 80\%.}$$

Où p est le nombre des cœur physique de la machine

Il apparait clairement que le parallélisme utilise efficacement les processeurs, car l'algorithme Rabin Karp utilise beaucoup de calcul (calcul de hachage) ce qui a permisd'utiliser éfficacement les deux cœurs physiques de la machine(80%).

IV.5. Conclusion

Dans ce chapitre nous avons fait une étude sur les algorithmes d'Aho-Corasick et Rabin Karp sur des jeux de données. Ceci nous permet d'évaluer l'avantage de l'utilisation des structures des données Succinctes et le parallélisme.

Les résultats obtenus montrent que les structures de données succinctes permettent de réduire largement le coût mémoire occupé par l'automate d'Aho-Corasick. D'autre part l'application de parallélisme sur l'algorithme Rabin Krap et Aho-Corasick a permis d'accéléré le processus de la recherche par dictionnaire, en raison des calculs faites par le hachage ainsi que les structures succincte (*rank et select*). Les résultats trouvé montrent que les implémentations nous utilisons sont efficaces pour la recherche de motifs. L'inconvénient avec Rabin Karp est que la taille des motifs doit être la même, mais le bon choix de représentation de la table de hachage, a permis à Rabin Karp de réduire le nombre de collision, et qui peut-être ignorer par rapport au nombre de motifs utiliser.

Conclusion général

Dans ce travail, nous nous sommes intéressés au problème de recherche par dictionnaire. Dans ce problème nous avons en entrées un dictionnaire D constitué de k mots et d'un texte T, et nous devons rechercher dans T l'ensemble des occurrences des mots de D. Dans ce cadre, nous avons implémenté deux algorithmes de recherche multi-motifs.

Le premier algorithme est celui d'Aho-Corasick. Ce dernier a été optimisé en coût mémoire grâce aux structures de données succinctes. En effet, ces structures de données ont permis un large gain en espace mémoire Cependant, l'utilisation de cette représentation succincte eu comme conséquence une augmentation du temps de recherche de motifs, car l'algorithme passe plus de temps dans des calculs (e.g : opérations *rank et select* sur les structures succinctes)

Le deuxième algorithme est celui de Rabin-Karp qui est basé sur le hachage. Cet algorithme a été optimisé pour la recherche multi-motifs grâce à l'utilisation d'une table de hachage performante qui permet de réduire le nombre de collisions. La méthode de construction de la table de hachage, essai linéaire (linear probing), a permis à l'algorithme Rabin Karp d'être rapide en termes de recherche de motifs. L'efficacité de cet algorithme est due à l'utilisation d'une fonction de hachage à roulement qui permet d'éviter de recalculer pour le hachage de chaque fenêtre du texte

Il est bien connu que les architectures multicoeurs sont beaucoup plus efficaces pour accélérer les calculs plutôt que les accès mémoires. C'est la raison pour laquelle nous avons realisé un parallelisme sur la recherche multi-motifs avec les structures succinctes et la recheche multi-motifs avec hachage à roulement.

D'aprés les testes que nous avons effectué sur les différentes jeux de données, les deux algorithmes ont donné des resultats satisfaisants pour la recherche par dictionnaire. Nous pouvons conclure que nos deux algorithmes proposés sont efficaces, le seul inconvénient étant que l'algorithme de Rabin-Karp requière que les motifs doivent être tous de la même longueur.

Comme perspective à notre travail, nous pouvons suggérons de:

- Ameliorer l'algorithme Rabin Karp pour tenir compte de la taille de motifs variante.
- Réduire la complexité temporelle des structures de données (surtout les structures succinctes) pour avoir de meilleures résultats.

REFERENCES BIBLIOGRAPHIQUE

[1] M. Crochemore, “Encyclopedia of Database Systems,” *Encycl. Database Syst.*, no. July 2014, pp. 0–5, 2009.

[2] E. Ukkonen, “On-line construction of suffix trees,” *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[3] M. Crochemore and T. Lecroq, “Encyclopedia of Database Systems,” *Encycl. Database Syst.*, no. September 2015, pp. 0–6, 2009.

[5] Cormen, T.H.: Introduction to algorithms. MIT press (2009).

[6] R. Raman, V. Raman, and S. R. Satti, “Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets,” vol. 3, no. 4, pp. 1–25, 2007.

[7] Knuth, D.: The Art of Computer Programming: Fundamental algorithms. Number vol. 1 in Addison-Wesley series in computer science and information processing. Addison-Wesley (1997).

[8] D. E. Knuth, “Big Omicron and big Omega and big Theta,” *ACM SIGACT News*, vol. 8, no. 2, pp. 18–24, 1976.

[9] Y. Gurevich, “Average case complexity,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 510 LNCS, pp. 615–628, 1991.

[10] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, “A survey on parallel computing and its applications in data-parallel problems using GPU architectures,” *Commun. Comput. Phys.*, vol. 15, no. 2, pp. 285–329, 2014.

[11] C. A. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Mauroux, “Benchmarking OLTP/web databases in the cloud,” *Proc. fourth Int. Work. Cloud data Manag. - CloudDB ’12*, p. 17, 2012.

[12] J. L. Gustafson, “Reevaluating amdahl’s law,” vol. 31, no. 5, pp. 532–533, 1988.

[13] J. Á. Fernández Sacasas, “El principio rector de la Educación Médica cubana Un reconocimiento a la doctrina pedagógica planteada por el profesor Fidel Ilizástigui Dupuy,” *Educ. Médica Super.*, vol. 27, no. 2, pp. 239–248, 2013.

[14] A. V. Aho, “Pattern Matching in Strings,” *Form. Lang. Theory*, pp. 325–347, 2014.

[15] D. E. Knuth, J. H. Morris, and V. R. Pratt, “Fast patter matching in strings*,” *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977.

[16] A. Rasool, A. Tiwari, G. Singla, and N. Khare, “String Matching Methodologies: A Comparative Analysis,” *Int. J. Comput. Sci. Inf. Technol.*, vol. 3, no. 2, pp. 3394–3397, 2012.

[17] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[18] M. O. Rabin and R. M. Karp, “Efficient randomized pattern-matching algorithms,” *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.

[19] P. Shah and R. Oza, “Improved parallel Rabin-Karp algorithm using compute unified device architecture,” *Smart Innov. Syst. Technol.*, vol. 84, pp. 236–244, 2018.

[20] T. Lecroq, “Handbook of Exact String-Matching Algorithms Christian Charras,” no. May, 2014.

[21] X. Zha and S. Sahni, “Multipattern string matching on a GPU,” *Proc. - IEEE Symp. Comput. Commun.*, no. October, pp. 277–282, 2011.

[22] J. N. Associate and T. Mahalakshmi, “Survey of Exact String Matching Algorithm for Detecting Patterns in Protein Sequence,” vol. 10, no. 8, pp. 2707–2720, 2017.

[23] D. Belazzougui, “Succinct Dictionary Matching With No Slowdown,” pp. 1–23, 2010.

[24] T. Cormen, C. Leiserson and R. Rivest, Introduction à l'algorithmique, Dunod, 1994.

[25] M. Cosnard and D.Trystram, Parallel Algorithms and Architectures, International Thomson Computer Press, 1995.

[26] G. Navarro and K. Sadakane, “Fully-Functional Static and Dynamic Succinct Trees,” pp. 1–39, 2009.

[27] A. Fiat and S. Shporer, “AIM: Another Itemset Miner.,” *Fimi*, no. March, 2003.

[28] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[29] A. Lecture and T. Karp-rabin, “Karp Rabin Fingerprint,” vol. 1976, pp. 2–4, 2014.

[30] L. Chi and X. Zhu, “Hashing Techniques: A Survey and Taxonomy,” *ACM Comput. Surv.*, vol. 50, no. 1, p. Article 11, 2017.

[31] A. Recitation, “Rolling Hash (Rabin-Karp Algorithm),” no. L, 2011.

[32] F. Cajori, “Horner’s method of approximation anticipated by Ruffini,” *Bull. Am. Math. Soc.*, vol. 17, no. 8, pp. 409–415, 1911.

[33] c.d.Vikidia,"Ubuntu,"7Avril2017.[enligne].Availabale: https://fr.vikidia.org/w/index.php?title=Ubuntu&oldid=1243126.

[34] S. a. B. T. a. M. A. a. P. M. Gog, «From Theory to Practice: Plug and Play with Succinct Data Structures,» *13th International Symposium on Experimental Algorithms,* pp. 326-337, 2014.

[35] N. S. Dmitry Kosolobov, «bitbucke,» 31 mars 2019. [En ligne]. Available: https://bitbucket.org/umqra/multiple-pattern-matching/src/master/LICENSE.

[36] N. S. Dmitry Kosolobov, «Compressed Multiple Pattern Matching,» *30th Annual Symposium on Combinatorial Pattern Matchin,* n° 110, 31 mars 2019.

[37] «GCC, the GNU Compiler Collection,» Free Software Foundation, Inc., last modified 20 juin 2019. [En ligne]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[38] Archaya, A., 2012. Input Segmented Universal Hashing Collusion in a Hash Table”, International Journal of Scientific & Engineering Research, 2014.

[39] I. D. E. S. Resultats, “RECHERCHE DE MOTIFS,” pp. 0–22, 2000.

[40] S. Heilbron, “Succinct Data Structures Exploring succinct trees in theory and practice Problem Background,” 2017.