UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ALGORITHMES VECTORIELS ET BIOINFORMATIQUE

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN MATHÉMATIQUES

PAR

SYLVIE HAMEL

SEPTEMBRE 2002

REMERCIEMENTS

Je voudrais commencer par dire un gros MERCI à ma directrice Anne Bergeron pour sa disponibilité, son enthousiasme, son humour, sa patience et bien entendu pour ses belles idées mathématiques toujours très intéressantes et joliement présentées. Merci pour tout le temps passé à lire et relire et corriger ma thèse et à envoyer des longs e-mails de Paris pour me donner ses opinions sur les nombreuses versions de certains chapitres de cette thèse. Merci à l'imprimante de Marne-la-Vallée pour avoir permis à Anne d'imprimer ma thèse de si nombreuses fois ☺

Merci à mes parents, Gisèle et Henri-Paul, à mon frère Steve et à ma soeur Josée pour m'avoir toujours appuyée dans mes études et parce qu'ils sont toujours là pour moi. Merci de m'aimer comme je suis, avec mes petites excentricités et ma tête de cochon!!

Merci à François pour son amour et sa grande patience envers mes angoisses de fin de doctorat. Merci d'être toujours là, à côté de moi. Merci à Cédrik, mon petit homme, pour sa bonne humeur constante et toutes les belles histoires qu'il me raconte. Gardez tous les deux cette capacité de me faire sourire, peu importe mon humeur. Je vous aime.

Merci à mes amis pour la vie, Anne-Marie, Stéphanie, Riccardo, Étienne, François S., François L., Marie-Isabelle, Carole et tous ceux et celles que j'oublie, pour leur soutien constant, leurs folies et toutes ces belles soirées passées ensemble.

Merci aux chercheurs du LACIM pour m'avoir permis de travailler avec eux pendant toutes ces années. Merci à André pour les nombreux conseils informatiques et le bon maintien du matériel informatique, sans quoi cette thèse n'aurait jamais vu le jour!

Merci à Manon Gauthier pour être toujours disponible pour nous guider avec le sourire, nous les pauvres étudiants, dans les dédales administratifs de l'université.

Finalement, merci au CRSNG, au FCAR et à l'ISM qui, grâce à leur support financier, m'ont permis de terminer cette thèse dans les délais prévus.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

Un algorithme *vectoriel* est un algorithme qui permet d'obtenir un vecteur de sortie en n'appliquant, sur un vecteur d'entrée, que des opérations vectorielles. Cet algorithme peut alors être implémenté en parallèle, en se servant des opérations sur les vecteurs de bits disponibles dans les processeurs, donnant lieu à des calculs très efficaces.

Nous nous sommes intéressées à savoir quels sont les automates pour lesquels il existe un algorithme vectoriel, fonctionnant en temps constant. On cherche donc à savoir quels sont les automates pour lesquels il est possible de trouver le vecteur de sortie $r_1 \dots r_m$, des états visités lors de la lecture d'un vecteur d'entrée $e_1 \dots e_m$, en n'utilisant qu'un nombre borné, indépendant de m, d'opérations vectorielles.

L'étude de cette question nous a menées, premièrement, à définir une classe d'automates, les automates *résolubles*, pour laquelle il est possible de construire des algorithmes vectoriels à partir des tables de transitions des automates. Nous avons ensuite démontré qu'un automate résoluble est au coeur de la résolution du problème de la recherche des occurrences approximatives d'un mot dans un texte, permettant ainsi de résoudre ce problème vectoriellement.

Dans le but de caractériser la classe d'automates pour lesquels il existe un algorithme vectoriel, nous nous sommes intéressées à la décomposition en cascade de Krohn-Rhodes (Krohn - Rhodes, 1965). L'étude de cette décomposition nous a permis de démontrer que le fait d'être *apériodique* pour un automate est une condition suffisante à l'existence d'un algorithme vectoriel pour cet automate.

INTRODUCTION

Le problème de la recherche des occurrences approximatives d'un mot P dans un texte T est important dans plusieurs domaines, dont la bioinformatique où il est à la base de plusieurs algorithmes de recherche de séquences génétiques (Gusfield, 1997). Le premier algorithme pour résoudre ce problème est dû à Sellers (Sellers, 1980) et a une complexité, en temps et en espace, de $\mathcal{O}(nm)$ où n est la longueur du texte et m, la longueur du mot cherché. Motivés par le fait que dans les applications, comme en biologie, mn est très grand, plusieurs travaux ont permis de développer de nouveaux algorithmes beaucoup plus rapides.

Un des moyens d'y parvenir est d'exploiter le parallélisme des opérations vectorielles ou, plus précisément, des opérations sur les vecteurs de bits. Plusieurs approches utilisant des vecteurs de bits ont donc été développées pour résoudre le problème de recherche approximative d'un mot dans un texte. La plupart de ces approches utilisent des vecteurs de bits pour coder l'ensemble des états d'un automate non-déterministe, (Wu - Manber, 1992; Baeza-Yates - Gonnet, 1996). Une approche alternative, développée par Myers (Myers, 1999), consiste à utiliser des vecteurs de bits pour coder les vecteurs d'entrée et de sortie d'un automate déterministe et calcule ensuite le vecteur de sortie en appliquant un nombre borné d'opérations vectorielles sur le vecteur d'entrée.

Nous avons généralisé l'approche de Myers en démontrant l'existence d'un algorithme vectoriel efficace pour résoudre le problème de la recherche des occurrences approximatives d'un mot dans un texte où l'on permet l'emploi d'une distance plus générale (Bergeron - Hamel, 2002a). L'originalité de notre approche réside dans le fait qu'au coeur de notre algorithme se trouve un automate ne dépendant que de la distance utilisée et non du mot P cherché, comme c'était le cas dans les approches considérés auparavant.

L'étude de cet automate nous a amenées à étudier la question de savoir quels sont les automates pour lesquels il est possible de trouver le vecteur de sortie $r_1 \dots r_m$, des états visités lors de la lecture d'un vecteur d'entrée $e_1 \dots e_m$, en n'utilisant qu'un nombre borné, indépendant de m, d'opérations vectorielles. Nous avons défini une classe d'automates, les *automates résolubles*, satisfaisant cette propriété et pour lesquels la construction de l'algorithme vectoriel se fait à partir de la table de transition de l'automate (Bergeron - Hamel, 2001).

Finalement, dans le but de caractériser la classe d'automates pour lesquels il existe un algorithme vectoriel comportant un nombre borné d'opérations, nous nous sommes intéressées à la décomposition en cascade de Krohn-Rhodes (Krohn - Rhodes, 1965). L'étude de cette décomposition nous a permis de démontrer que la condition d'être *apériodique* pour un automate $\mathcal{A}$ est suffisante à l'existence d'un algorithme vectoriel pour $\mathcal{A}$ (Bergeron - Hamel, 2002b).

La structure de cette thèse est la suivante:

Le premier chapitre présente le problème de la recherche des occurrences approximatives d'un mot $P = p_1 \dots p_m$ dans un texte $T = t_1 \dots t_n$. Pour ce faire, nous commençons par introduire le problème, plus élémentaire, de l'alignement global de deux séquences puis nous généralisons aux occurrences approximatives. Nous présentons aussi dans ce chapitre le premier algorithme connu résolvant ce problème. Cet algorithme utilise la programmation dynamique pour construire une table de distances et a été présenté dans deux articles, à peu près à la même époque (Sellers, 1974; Wagner - Fischer, 1974).

Le deuxième chapitre présente différentes approches classiques pour résoudre le problème de la recherche approximative, que ce soit par un calcul partiel de la table des distances (Ukkonen, 1985), par un prétraitement du mot à chercher (Ukkonen, 1985), par un calcul de diagonales (Landau-Vishkin, 1988; Landau-Vishkin, 1989; Galil - Park, 1990; Ukkonen - Wood, 1993), par la méthode des 4 Russes (Masek - Paterson, 1980; Wu - Manber - Myers, 1996) ou par une méthode à la Boyer- Moore (Tarhio - Ukkonen,

1990). Des approches vectorielles au problème (Wu - Manber, 1992; Baeza-Yates - Gonnet, 1996; Myers, 1999) sont ensuite présentées dans le chapitre 3.

Les deux derniers chapitres, les chapitres 4 et 5, comprennent la partie originale de ce travail.

Dans le chapitre 4, nous introduisons une classe d'automates, les automates *résolubles*, pour lesquels nous démontrons l'existence d'un algorithme vectoriel permettant de trouver, étant donné une séquence d'entrée $\mathbf{e} = e_1 \dots e_m$, le vecteur $\mathbf{r} = r_1 \dots r_m$ des états visités par l'automate lors de la lecture de $\mathbf{e}$ en un nombre borné d'opérations vectoriels. Nous montrons ensuite qu'un automate résoluble est au coeur de la résolution du problème de la recherche des occurrences approximatives d'un mot dans un texte et nous en déduisons un algorithme sur les vecteurs de bits de complexité en temps $\mathcal{O}(nc)$, où n est la longueur du texte T et c le coût d'insertion ou de suppression d'un caractère du texte lors de l'alignement.

Le chapitre 5 étudie les liens entre la décomposition en cascade de Krohn-Rhodes (Krohn - Rhodes, 1965) et les algorithmes sur les vecteurs de bits. Cette étude permet de démontrer l'existence d'un algorithme vectoriel pour tout automate apériodique. Ces liens permettent également de donner un algorithme linéaire pour tout automate résoluble.

CHAPITRE I

RECHERCHE DES OCCURRENCES APPROXIMATIVES D'UN MOT DANS UN TEXTE

Dans ce chapitre, nous présentons le problème de recherche des occurrences approximatives d'un mot dans un texte. Pour ce faire, nous commençons par considérer le problème, plus élémentaire, de l'alignement de deux mots et nous présentons l'algorithme classique, (Sellers, 1974; Wagner - Fischer, 1974), pour le résoudre. Nous présentons ensuite la généralisation de cet algorithme au problème de la recherche approximative (Wagner - Fischer, 1974).

1.1 Introduction

Le problème de la recherche des occurrences approximatives d'un mot dans un texte consiste, étant donné un mot P et un nombre naturel t, à trouver toutes les positions, dans un texte T, où il y a une occurrence de P, ayant au plus t erreurs. Ce problème est important dans plusieurs domaines. Premièrement, il est omniprésent en informatique, plus particulièrement dans le domaine de la recherche dans des bases de données textuelles et dans celui de la recherche sur le dépistage de l'information. Une application intéressante dans ce dernier domaine est le correcteur orthographique. Le correcteur donne, étant donné un mot mal orthographié, une liste de mots possibles pour le remplacer (Kashyap - Oommen, 1984; Mateescu - Salomaa - Salomaa - Yu, 1995). La recherche approximative est aussi au coeur de la bioinformatique, que ce soit pour la recherche de molécules nouvellement séquencées dans les banques de données existantes (Altschul et al., 1990), ou pour la recherche de similitudes entre deux séquences biologiques (Smith - Waterman, 1981a; Needleman - Wunsch, 1970). Cette citation de

Gusfield nous fait comprendre la pleine mesure de ce que la recherche approximative apporte à la biologie.

> *In biomolecular sequences (DNA, RNA, or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity.* (Gusfield, 1997)

Deux protéines ayant des séquences similaires ont habituellement la même fonction ou la même structure. Ce fait nous permet donc un premier classement des protéines en familles.

La publication de la séquence quasi-complète du génome humain en février 2001 (Venter and al., 2001; Collins and al., 2001) est le point de départ d'une recherche intensive dont le but ultime est la connaissance de tous les gènes humains, évalués aujourd'hui à environ 30 000, ainsi que de toutes les protéines qu'ils encodent. L'une des techniques utilisées pour trouver des gènes inconnus dans un génome est de rechercher dans celui-ci des séquences similaires à des gènes provenant d'autres espèces. En effet, les gènes sont des structures qui ont été très conservées au cours de l'évolution:

> *Throughout the present work we see insights gained through our ability to look for sequence homologies by comparaison of the DNA of different species. Studies on yeast are remarkable predictors of the human system!* (Strauss, 1995)

Il ne fait aucun doute que la recherche d'occurrences approximatives jouera un rôle dans ces découvertes, et c'est pourquoi le développement d'algorithmes plus efficaces en temps et en espace pour ce problème est toujours d'actualité.

1.2 Alignement global de deux mots

Avant de considérer le problème de la recherche des occurrences approximatives d'un mot dans un texte, considérons le problème, plus élémentaire, de l'alignement global de deux mots. Pour ce faire, nous introduisons les concepts d'alignement de deux mots, de coût d'un alignement et de distance entre deux mots.

1.2.1 Qu'est-ce qu'un alignement?

Lorsque l'on considère le problème de l'alignement de deux mots, les mots à aligner sont composés d'éléments d'un certain *alphabet*.

Définition 1.1 Un *alphabet* $\mathcal{A}$ est un ensemble de symbole. On représente par $\mathcal{A}^*$ l'ensemble de tous les mots que l'on peut construire avec les éléments de $\mathcal{A}$. On note par λ le mot vide, c'est-à-dire le mot de longueur 0.

Les données biologiques apparaissent sous forme de longues séquences, composées soit de nucléotides, dans le cas de l'ADN (acide désoxyribonucléique) et de l'ARN (acide ribonucléique), soit d'acides aminés, dans le cas des protéines.

Exemple 1.1 (***Alphabets biologiques***)

1. $\mathcal{A}_{ADN} = \{A, C, G, T\}$ est l'alphabet représentant l'ADN. L'élément A de cet alphabet représente l'adénine, C, la cytosine, G, la guanine et T, la thymine.

2. $\mathcal{A}_{ARN} = \{A, C, G, U\}$ est l'alphabet représentant l'ARN. Cet alphabet est constitué des mêmes éléments que l'alphabet précédent, à une exception près. L'élément T a été remplacé ici par l'élément U, représentant le nucléotide uracile.

3. $\mathcal{A}_P = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$ est l'alphabet représentant les acides aminés. Les différentes lettres représentent les acides aminé suivants (Gonnick - Wheelis, 1991):

A: alanine	G: glycine	M: méthionine	S: la sérine
C: cystéine	H: histidine	N: asparagine	T: thréonine
D: acide aspartique	I: isoleucine	P: proline	V: valine
E: acide glutamique	K: lysine	Q: glutamine	W: tryptophane
F: phénylalanine	L: leucine	R: arginine	Y: tyrosine

Un mot de $\mathcal{A}^*_{ADN}$, de $\mathcal{A}^*_{ARN}$ ou de $\mathcal{A}^*_P$ est appelé une *séquence* d'ADN, d'ARN ou d'acides aminés, selon le cas.

Définition 1.2 Soit $\mathcal{A}$ un alphabet, et $a \in \mathcal{A}$. La projection $\Pi_a : \mathcal{A}^* \to \mathcal{A}^*$, est définie récursivement par

$$\begin{aligned} \Pi_a(\lambda) &= \lambda \\ \Pi_a(x\alpha) &= \begin{cases} \Pi_a(x) & \text{si } \alpha = a \\ \Pi_a(x)\alpha & \text{sinon} \end{cases} \end{aligned}$$

pour $\alpha \in \mathcal{A}$ et $x \in \mathcal{A}^*$. C'est donc la projection qui a un mot associe ce mot dans lequel toutes les occurrences du symbole a ont été effacées.

Exemple 1.2 Soit $ACGC \in \mathcal{A}_{ADN}$. Alors,

$$\Pi_C(ACGC) = AG.$$

Définition 1.3 Soit $\mathcal{A}$ un alphabet et soit $x = x_1x_2\ldots x_m$ et $y = y_1y_2\ldots y_n$ deux mots quelconques de $\mathcal{A}^*$. Supposons que le symbole "-" $\notin \mathcal{A}$ et définissons l'alphabet $\mathcal{A}' = \mathcal{A} \cup \{-\}$. Un *alignement* de x et y est donné par

$$x' = x_1{}'x_2{}'\ldots x_l{}'$$

$$y' = y_1{}'y_2{}'\ldots y_l{}',$$

où x' et $y' \in \mathcal{A}'$, $\Pi_-(x') = x$, $\Pi_-(y') = y$ et où chaque colonne est de la forme

$$\begin{bmatrix} x_i \\ y_j \end{bmatrix} \quad \text{ou} \quad \begin{bmatrix} - \\ y_j \end{bmatrix} \quad \text{ou} \quad \begin{bmatrix} x_i \\ - \end{bmatrix}.$$

Exemple 1.3 Soit $ACTGCT$ et $ACGTCG$ deux séquences d'ADN. La Figure 1.1 présente 3 alignements possibles de ces deux séquences:

$$\begin{array}{cccccccc} A & C & T & G & - & C & T \\ A & C & - & G & T & C & G \end{array}, \quad \begin{array}{cccccccc} A & C & - & T & G & C & T \\ A & C & G & T & - & C & G \end{array}, \quad \begin{array}{cccccc} A & C & T & G & C & T \\ A & C & G & T & C & G \end{array}$$

Figure 1.1 Différents alignements de deux séquences

Pour deux mots donnés, on a donc plusieurs alignements possibles et on aimerait pouvoir choisir les meilleurs. Pour ce faire, il est nécessaire de définir une *mesure*, associée à un alignement, qu'il sera possible d'optimiser. Cette mesure sera généralement dépendante du problème biologique ou informatique étudié.

Exemple 1.4 Considérons les deux alignements suivants:

1)
$$\begin{array}{cccccccccccccc} \mathbf{A} & \mathbf{C} & \mathbf{G} & \mathbf{T} & \mathbf{A} & A & T & A & \mathbf{T} & \mathbf{T} & \mathbf{G} & \mathbf{A} & - & - \\ \mathbf{A} & \mathbf{C} & \mathbf{G} & \mathbf{T} & \mathbf{A} & - & - & - & \mathbf{T} & \mathbf{T} & \mathbf{G} & \mathbf{A} & T & A \end{array}$$

2)
$$\begin{array}{cccccccccccc} \mathbf{A} & \mathbf{C} & \mathbf{G} & \mathbf{T} & \mathbf{A} & A & \mathbf{T} & A & T & \mathbf{T} & G & \mathbf{A} \\ \mathbf{A} & \mathbf{C} & \mathbf{G} & \mathbf{T} & \mathbf{A} & T & \mathbf{T} & G & A & \mathbf{T} & - & \mathbf{A} \end{array}$$

Le premier alignement est celui dans lequel le plus grand nombre de symboles a été conservé, alors que le deuxième alignement est celui dont le nombre d'erreurs d'alignement est minimal. Ces deux alignements peuvent donc être considérés comme étant *le meilleur* alignement des 2 séquences, dépendant de la mesure adoptée pour les comparer.

Dans ce qui suit, nous allons définir la notion de *coût* d'un alignement et cette notion nous servira de mesure pour comparer les différents alignements; un alignement sera considéré meilleur qu'un autre si son coût est moindre.

1.2.2 Coût d'édition d'un alignement

On peut associer plusieurs mesures à un alignement, mais celle qui est le plus souvent considérée par les informaticiens est le *coût d'édition*. L'idée est de pénaliser chaque erreur d'une unité, c'est-à-dire que chaque colonne d'un alignement qui contient des éléments différents coûte une unité. Plus formellement, on a la définition suivante.

Définition 1.4 Soit $\mathcal{A}$ un alphabet et $a, b \in \mathcal{A}$. Le *coût élémentaire* d'une opération est défini comme suit:

$$\begin{aligned} c(a,b) &= \begin{cases} 1 & \text{si } a \neq b \\ 0 & \text{sinon} \end{cases} \\ c(a,-) &= 1 \\ c(-,b) &= 1. \end{aligned}$$

On peut maintenant définir formellement le coût d'édition d'un alignement.

Définition 1.5 Soit $\mathcal{A}$ un alphabet et $x, y \in \mathcal{A}^*$. Alors, si $x' = x_1{}'x_2{}' \dots x_l{}'$ et $y' = y_1{}'y_2{}' \dots y_l{}'$ forment un alignement de x et y, le *coût d'édition* de cet alignement est

$$C(x', y') = \sum_{i=1}^{l} c(x_i{}', y_i{}').$$

Exemple 1.5 Le coût d'édition associé à l'alignement suivant est 3:

$$\begin{array}{cccccccc} A & C & T & G & - & C & T \\ A & C & - & G & T & C & G \end{array}$$

Notre but est, étant donné deux mots, de trouver un ou des alignements de ces deux mots de coût minimal. Dans la prochaine section, nous montrons que ce minimum définit une mesure de distance entre deux mots.

1.2.3 Distance d'édition ou de Levenshtein

La *distance d'édition* entre deux mots, aussi appelée la distance de Levenshtein(Levenshtein, 1966), est le coût d'édition minimal de tous les alignements possibles de ces deux mots.

Définition 1.6 La *distance d'édition* entre deux mots x et y est définie par

$$D(x, y) = \operatorname*{Min}_{(x', y')} C(x', y'),$$

où le minimum se calcule sur tous les alignements (x', y') possibles de x et y et où C représente le coût d'édition.

Proposition 1.1 *La distance d'édition D possède les trois propriétés suivantes:*

1. $D(x, x) = 0$,
2. $D(x, y) = D(y, x)$,
3. $D(x, z) \leq D(x, y) + D(y, z)$.

C'est donc une distance, au sens mathématiques du terme. Nous allons maintenant définir la distance d'édition à l'aide de *transcrits d'édition* et ensuite nous servir de cette nouvelle définition pour démontrer la proposition 1.1.

Définition 1.7 Un *transcrit d'édition* est une suite des opérations suivantes, qu'on applique à un mot x pour obtenir un mot y:
- M, l'identité
- S_α, la suppression du symbole α
- I_α, l'insertion du symbole α
- $R_{\alpha,\beta}$, le remplacement du symbole α par le symbole β

Exemple 1.6 Le transcrit d'édition correspondant à l'alignement suivant:

$$\begin{array}{cccccccc} x' & = & A & C & T & G & - & C & T \\ y' & = & A & C & - & G & T & C & G \end{array}$$

est $t = MMS_TMI_TMR_{TG}$, c'est-à-dire qu'on obtient la séquence $y = ACGTCG$ de la séquence $x = ACTGCT$ avec le transcrit t:

$$\mathbf{A}CTGCT \overset{M}{\to} A\mathbf{C}TGCT \overset{M}{\to} AC\mathbf{T}GCT \overset{S_T}{\to} AC\mathbf{G}CT \overset{M}{\to}$$
$$ACG\mathbf{C}T \overset{I_T}{\to} ACGT\mathbf{C}T \overset{M}{\to} ACGTC\mathbf{T} \overset{R_{TG}}{\to} ACGTCG$$

Figure 1.2 Passage d'une séquence x à une séquence y avec un transcrit d'édition

Proposition 1.2 *La distance d'édition entre deux mots x et y, est donnée par le minimum d'opérations d'insertions, de suppressions et de remplacements permettant de passer de x à y.*

Preuve: La distance d'édition est le nombre de colonnes, dans un alignement de coût minimal, où les symboles diffèrent. Ces colonnes représentent l'insertion, la suppression ou le remplacement d'un symbole. Le transcrit correspondant à un alignement de coût minimal contient donc un nombre minimal d'opérations I, S et R. ■

Preuve de la proposition 1.1 : Le fait que D est symétrique et que $D(x, x) = 0$ est immédiat. Nous voulons montrer que

$$D(x, z) \leq D(x, y) + D(y, z).$$

Supposons que t est un transcrit de x à y, tel que le nombre d'opérations d'insertions, de suppressions et de remplacements est égal à $D(x, y)$. On a donc ici un nombre d'opérations d'édition minimal.

De la même façon, supposons que t' est un transcrit de y à z, tel que le nombre d'opérations d'insertions, de suppressions et de remplacements est égal à $D(y, z)$. Alors, le transcrit tt' est un transcrit de x à z de coût $D(x, y) + D(y, z)$. Comme $D(x, z)$ est le coût minimal d'un transcrit de x à z, on a nécessairement que $D(x, z) \leq D(x, y) + D(y, z)$. ■

La distance d'édition est une distance simple mais mal adaptée aux impératifs biologiques. La prochaine section propose une généralisation de la distance d'édition.

1.2.4 Distance d'édition généralisée

En biologie, il arrive que certains remplacements soient beaucoup plus fréquents que d'autres. Par exemple, le remplacement d'un nucléotide A par un nucléotide G est beaucoup plus plausible que le remplacement de A par C ou T. Ce phénomène s'explique tout simplement par les propriétés chimiques de ces nucléotides. En fait, les nucléotides

se divisent en deux classes d'éléments ayant des propriétés chimiques similaires:

$$\text{les purines} = \{A, G\}$$
$$\text{et les pyrimidines} = \{C, T\}.$$

Remplacer une purine par une purine est beaucoup plus probable, et donc beaucoup moins coûteux, que de remplacer une purine par une pyrimidine. (Li - Graur, 1991)

Le même phénomène existe lorsqu'on compare deux protéines. Les vingt acides aminés formant les protéines se regroupent en différentes classes ayant les mêmes propriétés chimiques et/ou électriques, et les remplacements au sein de ces classes sont beaucoup plus probables que les autres remplacements. Il existe plusieurs matrices de substitutions entre acides aminés, les plus connues étant les matrices PAM (Dayhoff, 1978) et BLOSUM (Hennikoff - Hennikoff, 1992).

Définition 1.8 Soit $\mathcal{A}$ un alphabet et $\mathcal{A}' = \mathcal{A} \cup \{-\}$. On définit une *fonction de substitution* $f : \mathcal{A}' \times \mathcal{A}' \to \mathbb{R}$, qui à chaque paire de symboles de $\mathcal{A}'$ associe le coût de remplacement du premier symbole par le deuxième.

Ici, $f(a, -)$ représente le coût de suppression du symbole a, alors que $f(-, a)$ représente le coût d'insertion de ce même symbole. Dans ce qui suit, nous allons considérer ces deux coûts comme étant égaux, et ce, pour tous les symboles de l'alphabet $\mathcal{A}$. De plus, pour empêcher l'alignement d'un symbole $-$ du premier mot avec un symbole $-$ du deuxième mot, on va poser $f(-, -) = \Omega$, où Ω est un très grand nombre entier. Finalement, pour favoriser l'opération de remplacement d'un symbole a par un symbole b plutôt que la suite d'opérations 'suppression de a' puis 'insertion de b', on pose $f(a, b) < 2f(a, -)$. (Cette dernière inégalité est essentielle en biologie étant donné que la mutation d'un nucléotide en un autre nucléotide est beaucoup plus plausible que la perte d'un nucléotide en un endroit du génome suivi par l'insertion d'un nouveau nucléotide à ce même endroit.)

Exemple 1.7 Soit l'alphabet $\mathcal{A}_{ADN}$. Tenant compte de la discussion précédente, on

peut définir la fonction de substitution f suivante sur cet alphabet:

f	A	C	G	T	-
A	0	1	1/3	1	2
C	1	0	1	1/3	2
G	1/3	1	0	1	2
T	1	1/3	1	0	2
-	2	2	2	2	Ω

Figure 1.3 Une fonction de substitution

Définition 1.9 Soit $\mathcal{A}$ un alphabet et $x, y \in \mathcal{A}^*$. Alors, si $x' = x_1{}'x_2{}' \dots x_l{}'$ et $y' = y_1{}'y_2{}' \dots y_l{}'$ forment un alignement de x et y, le *coût de substitution* de cet alignement est

$$C_f(x', y') \quad = \quad \sum_{i=1}^{l} f(x_i{}', y_i{}').$$

Exemple 1.8 Si on utilise la fonction de substitution f de la figure 1.3, le coût de substitution associé à l'alignement suivant est de 16/3:

$$\begin{array}{cccccccc} A & C & T & G & - & C & T \\ A & C & - & A & T & C & G \end{array}$$

Définition 1.10 Soit $\mathcal{A}$ un alphabet et f une fonction de substitution sur $\mathcal{A}' = \mathcal{A} \cup \{-\}$. La *distance d'édition généralisée* entre 2 mots x et y est donnée par

$$D_f(x, y) = \underset{(x', y')}{\text{Min}} C_f(x', y'),$$

où le minimum est calculé sur tous les alignements (x', y') possibles de x et y.

On a maintenant deux notions de distance en main. Dans ce qui suit, on va s'intéresser aux méthodes de calcul de $D_f(x, y)$, étant donné deux mots x et y, ainsi qu'aux méthodes de construction des alignements réalisant cette distance.

1.3 Calcul de la distance d'édition entre deux mots

Même si les notions de distance d'édition et de distance d'édition généralisée sont simples à définir, les calculer explicitement peut demander beaucoup de travail. En effet, le nombre d'alignements entre deux mots est exponentiel par rapport à la taille des mots à aligner. On peut donc considérer qu'il est impossible de trouver $D_f(x, y)$ en énumérant tous les alignements possibles et en comparant leur coût.

Lemme 1.1 *Si $f(n)$ est le nombre d'alignements différents entre deux mots de longueur n alors la série génératrice $f(x)$ du nombre d'alignements, donnée par $f(x) = \sum_{n\geq 0} f(n)x^n$, est égale à*

$$\frac{1}{\sqrt{1-6x+x^2}},$$

et $f(n) \sim (3+2\sqrt{2})^n$, lorsque n est grand.

Preuve: Premièrement, on a que

$$f(n) = \sum_{k=0}^{n} \binom{n+k}{k}\binom{n}{k}.$$

Ce résultat vient du fait que pour aligner deux mots, on peut commencer par placer k trous dans le premier mot. On a $\binom{n+k}{k}$ façons différentes de placer ces trous. Ensuite, pour que notre alignement soit valide, on doit placer les trous du deuxième mot sous une lettre du premier mot. Comme les mots sont de longueur n, on a $\binom{n}{k}$ choix possibles pour placer ces nouveaux trous. Il ne nous reste plus qu'à faire varier le nombre de trous de 0 à n, et on a le résultat mentionné.

Maintenant, considérons comme acquis les deux identités combinatoires suivantes (pour plus d'informations sur ces identités voir (Stanley, 1997, chapitre 1)):

$$(1-x)^{-n} = \sum_{k\geq 0} \binom{n+k-1}{k} x^k \tag{1.1}$$

$$(1-4x)^{-1/2} = \sum_{k\geq 0} \binom{2k}{k} x^k \tag{1.2}$$

On veut calculer $\sum_{n\geq 0} f(n)x^n = \sum_{n\geq 0}(\sum_k \binom{n+k}{k}\binom{n}{k})x^n$. Comme $\binom{n+k}{k}\binom{n}{k} = \binom{n+k}{2k}\binom{2k}{k}$, on va plutôt calculer:

$$\sum_{n\geq 0}\sum_k \binom{n+k}{2k}\binom{2k}{k}x^n \quad = \quad \sum_k \binom{2k}{k}\sum_{n\geq 0}\binom{n+k}{2k}x^n \tag{1.3}$$

Commençons par calculer $\sum_{n\geq 0}\binom{n+k}{2k}x^n$. On a les identités suivantes:

$$\sum_{n\geq 0}\binom{n+k}{2k}x^n = x^k\sum_{n\geq 0}\binom{n+k}{n-k}x^{n-k} = x^k\sum_{n+k\geq 0}\binom{n+2k}{n}x^n = x^k\sum_{n\geq 0}\binom{n+2k}{n}x^n.$$

L'identité 1.1 nous donne alors que

$$x^k\sum_{n\geq 0}\binom{n+2k}{n}x^n = x^k(1-x)^{-(2k+1)}$$

On peut donc réécrire 1.3 comme:

$$\sum_k \binom{2k}{k}\frac{x^k}{(1-x)^{2k+1}} = \frac{1}{1-x}\sum_k \binom{2k}{k}\left(\frac{x}{(1-x)^2}\right)^k.$$

Finalement, en utilisant l'identité 1.2, on obtient:

$$\frac{1}{1-x}\sum_k \binom{2k}{k}\left(\frac{x}{(1-x)^2}\right)^k = \frac{1}{1-x}\left(1-4\left(\frac{x}{(1-x)^2}\right)\right)^{-1/2} = \frac{1}{\sqrt{1-6x+x^2}}.$$

Maintenant, puisque la plus petite racine de $1-6x+x^2$ est $3-2\sqrt{2}$, on a que $f(n)$ est asymptotiquement équivalent (Bergeron - Labelle - Leroux, 1997) à

$$\frac{1}{(3-2\sqrt{2})^n} \quad = \quad (3+2\sqrt{2})^n. \qquad \blacksquare$$

Exemple 1.9 Les protéines ont une longueur moyenne d'environ 300 acides aminés. Le nombre d'alignements différents entre 2 protéines est donc d'environ

$$f(300,300) \sim 10^{230}.$$

L'énumération de tous les alignements de deux mots étant impossible, l'emploi d'autres techniques est requis pour calculer la distance d'édition entre deux mots. Nous présentons maintenant l'algorithme de programmation dynamique permettant de trouver la distance d'édition entre deux mots et les alignements correspondants à cette distance. Nous présentons ensuite la généralisation de cet algorithme pour le calcul des occurrences approximatives d'un mot dans un texte, qui est le problème auquel nous nous intéresserons exclusivement par la suite.

1.3.1 Le problème

Le principe de la programmation dynamique est d'utiliser l'information connue sur des petits problèmes, pour en résoudre de plus gros. Dans ce qui suit, nous étudions l'approche dynamique du calcul de la distance d'édition entre deux mots. Nous indiquons aussi les modifications à apporter pour que les résultats s'appliquent au calcul de la distance d'édition généralisée.

Considérons le problème consistant à calculer la distance d'édition entre deux mots $x = x_1 \dots x_m$ et $y = y_1 \dots y_n$, sur un alphabet $\mathcal{A}$.

Définition 1.11 On définit

$$D(i, j) = D(x_1 \dots x_i, y_1 \dots y_j)$$

comme étant la distance d'édition entre le préfixe $x_1 \dots x_i$ de longueur i de x et le préfixe $y_1 \dots y_j$ de longueur j de y.

Avec cette définition, calculer la distance d'édition entre $x = x_1 \dots x_m$ et $y = y_1 \dots y_n$ revient à calculer $D(m, n)$.

Remarque: Si la distance utilisée est une distance d'édition généralisée, on a qu'à poser $D_f(i, j) = D_f(x_1 \dots x_i, y_1 \dots y_j)$ comme étant la distance d'édition généralisée entre le préfixe $x_1 \dots x_i$ de x et le préfixe $y_1 \dots y_j$ de y. Alors, $D_f(m, n)$ est la distance d'édition généralisée entre les mots x et y.

1.3.2 La relation de récurrence

On veut maintenant établir une relation de récurrence entre $D(i, j)$ et des valeurs de D d'index plus petits que i et j. Pour ce faire, commençons par établir les conditions initiales de la récurrence, c'est-à-dire les valeurs $D(i, j)$ lorsque i ou $j = 0$.

$D(i,0)$ est la distance d'édition entre $x_1 \dots x_i$ et le mot vide λ et, symétriquement, $D(0,j)$ est la distance d'édition entre λ et $y_1 \dots y_j$. On a donc, $D(i,0) = i$ et $D(0,j) = j$.

Proposition 1.3 *Lorsque i et j sont strictement positif, on a*

$$D(i,j) = \min \begin{cases} D(i-1,j) & + & 1 \\ D(i,j-1) & + & 1 \\ D(i-1,j-1) & + & \delta(i,j) \end{cases},$$

où $\delta(i,j) = 0$ si $x_i = y_j$ et 1, sinon.

Preuve: Supposons qu'on ait un alignement A de coût minimal entre $x_1 \dots x_i$ et $y_1 \dots y_j$. Alors, A peut prendre l'une des trois formes suivantes:

1. A se termine en alignant x_i et y_j et on a $\begin{array}{|c|}\hline A' \\ \hline \end{array} \begin{matrix} x_i \\ y_j \end{matrix}$

2. A se termine en supprimant x_i et on a $\begin{array}{|c|}\hline A' \\ \hline \end{array} \begin{matrix} x_i \\ - \end{matrix}$

3. A se termine en insérant y_j et on a $\begin{array}{|c|}\hline A' \\ \hline \end{array} \begin{matrix} - \\ y_j \end{matrix}$

Il faut voir que dans chacun de ces cas, les alignements A' sont minimaux, et donc leurs coûts égalent respectivement $D(i-1,j-1)$, $D(i-1,j)$ et $D(i,j-1)$.

Pour ce faire, supposons le contraire, c'est-à-dire qu'il existe un alignement A'' de coût moindre que l'alignement A'. Alors, l'alignement composé de A'' suivi d'une insertion, d'une suppression ou d'un remplacement (ou identité) serait de coût inférieur à notre alignement A de départ, qui était considéré minimal. Cela nous amène à une contradiction et on a bien le résultat voulu. ■

Remarque: Lorsqu'on emploi une distance d'édition généralisée, on a un coût $f(a,-) = f(-,a) = k$ pour chaque insertion ou suppression et donc les conditions initiales se

modifient comme suit: $D_f(i,0) = k * i$ et $D_f(0,j) = k * j$. Pour les i et j strictement positifs, on a alors que

$$D_f(i,j) = \min \begin{cases} D_f(i-1,j) & + \quad k \\ D_f(i,j-1) & + \quad k \\ D_f(i-1,j-1) & + \quad f(x_i, y_j) \end{cases}$$

1.3.3 Calculer D(m, n) à l'aide d'une table

Le premier algorithme présenté pour calculer la distance d'édition entre deux mots est l'algorithme évident: calculer $D(i,j)$ pour des valeurs croissantes de i et de j, en se servant des valeurs déjà calculées. Pour ce faire, on construit une table des distances D qui contient $D(i,j)$ en position (i,j). Cet algorithme a été présenté dans deux articles différents, à peu près à la même époque (Sellers, 1974; Wagner - Fischer, 1974).

Exemple 1.10 Pour trouver la distance d'édition entre les séquences $AGGAGGA$ et $AAGCTAAG$, on commence par initialiser la table des distances D de cette façon:

	λ	A	A	G	C	T	A	A	G
λ	0	1	2	3	4	5	6	7	8
A	1								
G	2								
G	3								
A	4								
G	5								
G	6								
A	7								

Figure 1.4 Initialisation de la table des distances D entre deux mots

Ensuite on calcule, ligne par ligne ou colonne par colonne, la valeur des autres cellules (i,j) de la table, en se servant des valeurs calculées précédemment. On obtient que la

distance d'édition entre $AAGCTAAG$ et $AGGAGGA$ est de 5 (Voir figure 1.5). Pour trouver cette distance on doit calculer chacune des cases du tableau. La complexité de cet algorithme en temps et en espace est donc de $\mathcal{O}(nm)$, où n et m représentent les longueurs des mots à aligner.

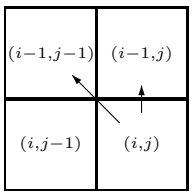
	λ	A	A	G	C	T	A	A	G
λ	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
G	2	1	1	1	2	3	4	5	6
G	3	2	2	1	2	3	4	5	5
A	4	3	2	2	2	3	3	4	5
G	5	4	3	2	3	3	4	4	4
G	6	5	4	3	3	4	4	5	4
A	7	6	5	4	4	4	4	4	**5**

Figure 1.5 Table des distances entre les séquences $AAGCTAAG$ et $AGGAGGA$.

1.3.4 Obtenir les alignements optimaux

On connaît maintenant la distance d'édition entre deux mots et on veut trouver un ou des alignements réalisant cette distance. Pour se faire, il s'agit de garder des pointeurs, lors du calcul de la table, qui indiquent la cellule d'où provient la valeur $D(i,j)$. Comme $D(i,j) = \min\{D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+\delta\}$, la valeur de la cellule (i,j) provient soit de la cellule $(i-1,j)$, soit de la cellule $(i,j-1)$, soit de la cellule $(i-1,j-1)$ ou de plusieurs de ces cellules.

Exemple 1.11 Si $D(i,j) = D(i-1,j)+1 = D(i-1,j-1)+\delta$ alors on met les pointeurs suivants:



Maintenant, pour trouver les alignements optimaux entre deux mots, on a la proposition suivante:

Proposition 1.4 *Il y a bijection entre les chemins de* (m, n) *à* $(0, 0)$ *suivant les pointeurs dans la matrice* D *et les alignements optimaux, c'est-à-dire de distance minimale, de deux mots.* ■

En fait, on a la bijection suivante entre les flèches d'un chemin et les colonnes de l'alignement:

- $\nwarrow$ représente une identité ou un remplacement
- $\uparrow$ représente une insertion dans le mot en haut du tableau (mot horizontal)
- $\leftarrow$ représente une suppression dans le mot horizontal (ou une insertion dans le mot vertical)

Exemple 1.12 Dans l'exemple 1.10, il y a six chemins de (m, n) à $(0, 0)$:

	λ	A	A	G	C	T	A	A	G
λ	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
G	2	1	1	1	2	3	4	5	6
G	3	2	2	1	2	3	4	5	5
A	4	3	2	2	2	3	3	4	5
G	5	4	3	2	3	3	4	4	4
G	6	5	4	3	3	4	4	5	4
A	7	6	5	4	4	4	4	4	**5**

Figure 1.6 Exemple de chemins correspondants à des alignements optimaux

Ces six chemins correspondent aux alignements, de coût 5, suivants:

1)
$$\begin{array}{cccccccc} A & A & G & C & T & A & A & G \\ A & G & G & A & G & G & A & - \end{array}$$

2)
$$\begin{array}{cccccccc} A & A & G & C & T & A & A & G & - \\ A & G & G & - & - & A & G & G & A \end{array}$$

3)
$$\begin{array}{cccccccc} A & A & G & C & T & A & A & G & - \\ - & A & G & - & G & A & G & G & A \end{array}$$

4)
$$\begin{array}{cccccccc} A & A & G & C & T & A & A & G & - \\ A & - & G & - & G & A & G & G & A \end{array}$$

5)
$$\begin{array}{cccccccc} A & A & G & C & T & A & A & G & - \\ - & A & G & G & - & A & G & G & A \end{array}$$

6)
$$\begin{array}{cccccccc} A & A & G & C & T & A & A & G & - \\ A & - & G & G & - & A & G & G & A \end{array}$$

Regardons maintenant ce qui se passe si l'on utilise une distance d'édition généralisée sur ce même exemple.

Exemple 1.13 Soit f la fonction de substitution de la figure 1.3. Pour calculer la distance d'édition généralisée D_f entre les deux mêmes séquences que dans l'exemple 1.10, c'est-à-dire les séquences $AAGCTAAG$ et $AGGAGGA$, on commence par initialiser la table en posant, $\forall i$ et $\forall j$, $D_f(i,0) = k * i = 2i$ et $D_f(0,j) = k * j = 2j$. On calcule ensuite la valeur de chacune des cases (i,j) en nous servant des valeurs déjà calculées et de la fonction f:

	λ	A	A	G	C	T	A	A	G
λ	0	2	4	6	8	10	12	14	16
A	2	0	2	4	6	8	10	12	14
G	4	2	1/3	2	4	6	8	10	12
G	6	4	7/3	1/3	7/3	13/3	19/3	25/3	10
A	8	6	4	7/3	4/3	10/3	13/3	19/3	25/3
G	10	8	6	4	10/3	7/3	11/3	14/3	19/3
G	12	10	8	6	15/3	13/3	8/3	12/3	14/3
A	14	12	10	8	21/3	18/3	13/3	8/3	**13/3**

Figure 1.7 Chemins optimaux avec une distance d'édition généralisée

La distance d'édition généralisée D_f entre les deux séquences $AAGCTAAG$ et $AGGAGGA$ est donc de 13/3 et il y a deux alignements réalisant cette distance:

1)
$$\begin{array}{cccccccc} A & A & G & C & T & A & A & G \\ A & G & G & A & - & G & G & A \end{array}$$

2)
$$\begin{array}{cccccccc} A & A & G & C & T & A & A & G \\ A & G & G & - & A & G & G & A \end{array}$$

En comparant les résultats des exemples 1.12 et 1.13, on remarque que les alignements optimaux provenant de la distance d'édition généralisée ont tendance à aligner de longues séquences, composées de nucléotides identiques ou chimiquement semblables. Les alignements optimaux provenant de la distance d'édition, quant à eux, optimisent les identités de nucléotides ce qui brise parfois des séquences similaires biologiquement intéressantes.

1.4 Calcul des occurrences approximatives d'un mot dans un texte

Dans cette section, nous passons enfin au problème central de cette thèse, c'est-à-dire la recherche des occurrences approximatives d'un mot dans un texte. Nous présentons ici la façon dynamique d'aborder le problème.

1.4.1 Le problème

Le problème de la recherche des occurrences approximatives d'un mot dans un texte consiste, étant donné un mot P et un nombre naturel t, à trouver toutes les positions, dans un texte T, où il y a une occurrence de P, ayant au plus t erreurs. Dans les problèmes de recherche en biologie, le mot P est habituellement assez court – quelques centaines de caractères – alors que T peut être très long – des millions de caractères –.

Définition 1.12 On définit

$$D(i,j) = \min_g D(p_1 \dots p_i, t_g \dots t_j),$$

où g varie de 1 à j, comme étant la distance d'édition minimale entre le préfixe $p_1 p_2 \dots p_i$ de longueur i de P et les suffixes du texte T se terminant en position j.

Avec cette définition, si $D(m, j) \leq t$ alors il y a une ou des occurrences approximatives de P, ayant au plus t erreurs, se terminant en position j du texte.

Remarque: Si on utilise une distance d'édition généralisée, on a qu'à remplacer, comme précédemment, l'emploi de la distance D par une distance généralisée D_f.

1.4.2 La relation de récurrence

Pour calculer les occurrences approximatives de P dans T, on se sert de la même relation de récurrence, définit dans la proposition 1.3 pour le problème de l'alignement de deux mots, dans laquelle on modifie légèrement les conditions initiales.

Proposition 1.5 *On a la relation de récurrence suivante:*

$$D(i,j) = \min \begin{cases} D(i-1,j) & + & 1 \\ D(i,j-1) & + & 1 \\ D(i-1,j-1) & + & \delta(i,j) \end{cases},$$

où

$$\delta(i,j) = \begin{cases} 0 & si \;\; x_i = y_j \\ 1 & sinon \end{cases}$$

avec comme conditions initiales que $D(i,0) = i$, $\forall i$ *et* $D(0,j) = 0$, $\forall j$. ■

Remarque: Lorsqu'on emploie une distance d'édition généralisée, on a un coût $f(a,-)$ $= f(-,a) = k$ pour chaque insertion ou suppression et donc les conditions initiales se modifient comme suit: $D_f(i,0) = k * i$ et $D_f(0,j) = 0$. Pour les i et j strictement positifs, on a alors la relation de récurrence suivante:

$$D_f(i,j) = \min \begin{cases} D_f(i-1,j) & + & k \\ D_f(i,j-1) & + & k \\ D_f(i-1,j-1) & + & f(x_i, x_j) \end{cases}$$

1.4.3 Les positions des occurrences approximatives

On veut trouver les occurrences, à t erreurs près, d'un mot P de longueur m, dans un texte T, de longueur n. La solution classique (Sellers, 1980) est obtenue en calculant la matrice des distances $D[0..m, 0..n]$, et en gardant en mémoire les positions j pour lesquelles $D(m, j) \leq t$.

Exemple 1.14 On veut trouver les positions dans le texte $T = ACGTAACGAAT$ où il y a des occurrences, à une erreur près, du mot $P = AAC$. On commence par initialiser la table des distances D en posant $\forall i$, $D(i, 0) = i$ et $\forall j$, $D(0, j) = 0$. On calcule ensuite la valeur de chacune des autres cellules (i, j) en nous servant des valeurs déjà calculées:

	λ	A	C	G	T	A	A	C	G	A	G	G
λ	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	0	1	1	0	1	1
A	2	1	1	2	2	1	0	1	2	1	1	2
C	3	2	**1**	2	3	2	**1**	**0**	**1**	2	2	2

Figure 1.8 Exemple de table des distances pour le problème de la recherche des occurrences approximatives d'un mot dans un texte

Il y a donc une ou des occurrences du mot $P = AAC$, à une erreur près, se terminant en chacune des positions j du texte suivante: 2, 6, 7, 8.

1.4.4 Obtenir les occurrences approximatives

On connaît maintenant toutes les positions d'un texte T où se terminent des occurrences approximatives d'un mot P. On voudrait maintenant connaître ces occurrences. Comme dans le cas de l'alignement de deux mots, il s'agit de garder des pointeurs, pour chaque cellule (i, j), indiquant d'où provient $D(i, j)$.

Proposition 1.6 *Soit j une position où se termine une ou des occurrences approximatives d'un mot $P = p_1 \dots p_m$ dans un texte T. Il y a bijection entre les chemins de (m, j) à $(0, l)$ et les différentes occurrences de P se terminant en cette position.* ■

Exemple 1.15 Reprenons l'exemple 1.14. En position 2 du texte, on a le suffixe AC qu'on peut aligner au mot AAC pour obtenir deux alignements différents de coût 1. Ils correspondent aux deux chemins de (3,2) à (0,0) suivants:

	λ	A	C	G	T	A	A	C	G	A	G	G
λ	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	0	1	1	0	1	1
A	2	1	1	2	2	1	0	1	2	1	1	2
C	3	2	**1**	2	3	2	**1**	**0**	**1**	2	2	2

En position 6 du texte, on a le suffixe AA pour lequel il existe un alignement de coût 1 avec AAC. Cet alignement correspond au chemin suivant de (3,6) à (0,4):

	λ	A	C	G	T	A	A	C	G	A	G	G
λ	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	0	1	1	0	1	1
A	2	1	1	2	2	1	0	1	2	1	1	2
C	3	2	**1**	2	3	2	**1**	**0**	**1**	2	2	2

En position 7 du texte on a le suffixe AAC et donc une occurrence parfaite de P. L'alignement correspond au chemin suivant de (3,7) à (0,4):

	λ	A	C	G	T	A	A	C	G	A	G	G
λ	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	0	1	1	0	1	1
A	2	1	1	2	2	1	0	1	2	1	1	2
C	3	2	**1**	2	3	2	**1**	**0**	**1**	2	2	2

En position 8 du texte, on a le suffixe $AACG$ pour lequel il existe un alignement de coût 1 avec le mot AAC. L'alignement correspond au chemin suivant de (3,8) à (0,4):

	λ	A	C	G	T	A	A	C	G	A	G	G
λ	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	0	1	1	0	1	1
A	2	1	1	2	2	1	0	1	2	1	1	2
C	3	2	**1**	2	3	2	**1**	**0**	**1**	2	2	2

Figure 1.9 Chemins correspondants à des occurrences approximatives du mot $P = AAC$ dans le texte $T = ACGTAACGAGG$

CHAPITRE II

DIFFÉRENTES APPROCHES AU PROBLÈME DE LA RECHERCHE APPROXIMATIVE D'UN MOT DANS UN TEXTE

Dans ce chapitre, nous présentons différents algorithmes classiques pour résoudre le problème de la recherche des occurrences approximatives d'un mot dans un texte. Chacune des sections présente une façon différente d'aborder le problème, que ce soit par un prétraitement du mot à chercher (Ukkonen, 1985), par un calcul de diagonales (Landau-Vishkin, 1988; Landau-Vishkin, 1989; Galil - Park, 1990; Ukkonen - Wood, 1993), par la méthode des 4 Russes (Masek - Paterson, 1980; Wu - Manber - Myers, 1996) ou par une méthode à la Boyer-Moore (Tarhio - Ukkonen, 1990). Les preuves des résultats énoncés dans ces différentes sections ont été supprimées pour ne pas alourdir la présentation de l'idée générale de chacun des algorithmes. Les algorithmes plus récents, utilisant une approche vectorielle seront, quant à eux, présentés au chapitre 3.

2.1 Ne calculer qu'une partie de la table des distances

Nous présentons ici une idée provenant de l'analyse de la table des distances D, pour le problème de la recherche approximative d'un mot dans un texte. Rappelons que dans cette table, $D(i,j)$ est définie comme étant la distance d'édition minimale entre le préfixe $p_1 \dots p_i$, du mot P cherché, et les suffixes du texte T se terminant en position j. Ukkonen (Ukkonen, 1985) a déduit le lemme suivant de la récurrence de la proposition 1.5 permettant de calculer les entrées de la table des distances D:

Lemme 2.1 *Dans la matrice D, on a toujours*

$$D(i,j) = D(i-1,j-1) \quad ou \quad D(i,j) = D(i-1,j-1)+1$$ ■

Ce lemme permet facilement de voir que plusieurs des entrées de la table D sont inutiles pour le calcul des occurrences approximatives d'un mot, à t erreurs près. En effet, si on a une entrée de valeur supérieure à t dans le cellule $(i-1, j-1)$ de D alors, il ne sert à rien de calculer les entrées des cellules (i, j), $(i+1, j)$, $\ldots$, (m, j), où m est la longueur du mot P cherché, puisque ces cellules contiennent alors des entrées de valeur supérieure à t.

Donc, si on garde en mémoire pour une colonne donnée j, la dernière ligne i pour laquelle la valeur de la cellule $(i, j) \leq t$ alors, lors du calcul des cellules de la colonne $j+1$, seulement les cellules des lignes 0 à $i+1$ contiendront des entrées $\leq t$.

Exemple 2.1 Soit $T = GCGTTGCAGGAACG$, un texte et $P = AACG$, un mot. Voici la table des distances D pour le calcul des occurrences approximatives, à une erreur près, de P dans T. Les X indiquent les cellules (i, j) pour lesquelles le calcul de $D(i, j)$ n'est pas nécessaire, étant donné la discussion précédente.

	λ	G	C	G	T	T	G	C	A	G	G	A	A	C	G
λ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1
A	2	2	2	2	2	2	2	2	1	1	2	1	0	1	2
C	3	X	X	X	X	X	X	X	X	2	2	X	1	0	1
G	4	X	X	X	X	X	X	X	X	X	X	X	X	**1**	**0**

Figure 2.1 Un exemple de calcul partiel de la table des distances

Dans cet exemple, P est de longueur 4 et T est de longueur 14. Sur les $4 * 14 = 56$ entrées de la table D, 21 entrées ne sont pas nécessaires au calcul des occurrences approximatives de P dans T. Le lemme 2.1 nous sauve donc ici environ 40% du travail.

Cette idée de ne calculer qu'une partie de D est intéressante dans certains exemples, comme on vient de le voir avec l'exemple 2.1, mais son efficacité varie selon le mot P et

le texte T choisis. Regardons un nouvel exemple de calcul de la table D, où le mot P est le même que dans l'exemple 2.1 mais où le texte T a été modifié:

Exemple 2.2 Les valeurs de 8 cellules seulement sont inutiles au calcul des occurrences de $AACG$ dans $ACGTAACGAGGAAC$.

	λ	A	C	G	T	A	A	C	G	A	G	G	A	A	C
λ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	0	1	1	0	1	1	0	0	1
A	2	1	0	1	2	1	0	1	2	1	1	2	1	0	1
C	3	X	1	1	2	X	1	0	1	2	2	2	2	1	0
G	4	X	X	**1**	2	X	X	**1**	**0**	**1**	2	X	3	X	**1**

Figure 2.2 Un autre exemple, moins intéressant, de calcul partiel de la table D

Plus il y a d'occurrences approximatives de P dans T, plus le nombre de cellules inutiles dans la table D est petit. Le nombre d'erreurs permis, t, fait lui aussi varier le nombre de cellules inutiles pour le calcul des occurrences d'un mot. La complexité en temps de cet algorithme est donc, encore, de $\mathcal{O}(nm)$, dans la plupart des cas.

2.2 Prétraitement du mot P à l'aide d'un automate fini déterministe

Dans son article (Ukkonen, 1985), Ukkonen présente aussi une façon de calculer les positions j du texte où se terminent une occurrence du mot P cherché, en commençant par effectuer un prétraitement du mot P.

Le prétraitement consiste à construire, à partir du mot P cherché et du nombre d'erreurs permis t, un automate fini déterministe, dénoté M_P (Pour un rappel sur les automates finis, voir (Hopcroft - Ullman, 1979)). L'idée est que lorsqu'on donne en entrée à l'automate M_P un texte T, l'automate arrive dans un état final, après la lecture d'une lettre t_j de T, si et seulement si T contient une occurrence approximative de P, ayant

$e \leq t$ erreurs, se terminant en position j du texte. Mais comment construire M_P?

Intuitivement, étant donné un mot P fixé, chaque état de M_P correspond à une colonne pouvant apparaître dans la matrice des distances D quand le texte T varie. Un état de M_P est final si l'élément en position m dans la colonne correspondante de cet état est $\leq t$. Voyons cette construction sur un exemple.

Exemple 2.3 Soit $P = aba$ et $t = 1$. On veut construire l'automate M_P. L'état initial de cet automate est l'état correspondant à la première colonne de la table des distances D, c'est-à-dire 0123. Maintenant, le texte T commence soit par un a, soit par un b. On calcule donc, pour ces deux cas, la prochaine colonne de la table D:

	λ	a
λ	0	0
a	1	0
b	2	1
a	3	2

	λ	b
λ	0	0
a	1	1
b	2	1
a	3	2

Donc, de l'état initial 0123, on se rend à l'état 0012 avec la transition a et à l'état 0112 avec la transition b. Pour ces deux nouveaux états, on calcule les colonnes de D correspondant à la lecture d'un a ou de b et on obtient encore de nouveaux états. On arrête l'algorithme de construction lorsque tous les nouveaux états (ou nouvelles colonnes de D) font déjà partie de l'automate. Comme le nombre d'erreurs permis est $t = 1$ dans cet exemple, les états finaux de l'automate sont les états se terminant par un 1 ou un 0:

Maintenant qu'on a construit l'automate M_P pour le mot $P = aba$ et le nombre d'erreur permis $t = 1$, on peut donner en entrée à M_P n'importe quel texte T sur l'alphabet $\{a, b\}$. L'automate arrivera dans un état final, après la lecture d'une lettre t_j du texte si et seulement si il y a une occurrence approximative de P, ayant au plus 1 erreur, se terminant en position j du texte.

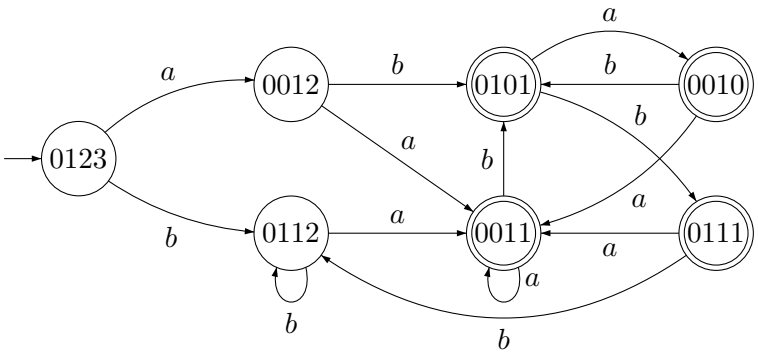


Figure 2.3 Un automate M_P

La complexité en temps de cet algorithme de recherche est donnée par le théorème suivant tiré de l'article (Ukkonen, 1985):

Théorème 2.1 *Étant donné un mot $P = p_1 \dots p_m$ sur un alphabet Σ, et un nombre d'erreurs permis t, il est possible de construire l'automate M_P en temps $\mathcal{O}(m \cdot |\Sigma| \cdot k_P)$, où $k_P = min(3^m, 2^t \cdot |\Sigma|^t \cdot m^{t+1})$. Après la construction de M_P, on trouve toutes les positions des occurrences de P, à t erreurs près, dans T en temps $\mathcal{O}(n)$.* ■

Cet algorithme est efficace si on veut chercher un certain mot P dans plusieurs textes différents construits sur un même alphabet. La taille du mot P et de l'alphabet considéré sont aussi des facteurs importants dans l'efficacité de l'algorithme.

2.3 Prétraitement de P et calcul de diagonales

Dans cette section nous présentons différents algorithmes, (Landau-Vishkin, 1988), (Landau-Vishkin, 1989), (Galil - Park, 1990), (Ukkonen - Wood, 1993), utilisant une idée de calcul de diagonales de la table des distances D. Chacun des algorithmes considérés utilise un prétraitement différent du mot P.

2.3.1 Calcul de diagonales

On peut déduire, à partir du lemme 2.1 une façon plus compacte de représenter l'information contenue dans la table D.

Définition 2.1 Soit D, la table des distances pour le problème de la recherche des occurrences approximatives d'un mot dans un texte. La *diagonale* d de la table comprend l'ensemble des cellules (i, j) telles que $j - i = d$.

Exemple 2.4 Dans la table des distances D suivante, les entrées de la diagonale 2 sont écrites en caractères gras:

	λ	G	C	G	T	T	G	C	A	G	G	A	A	C	G
λ	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	**1**	1	1	1	1	0	1	1	0	0	1	1
A	2	2	2	2	**2**	2	2	2	1	1	2	1	0	1	2
C	3	3	2	3	3	**3**	3	2	2	2	2	2	1	0	1
G	4	3	3	2	3	4	**3**	3	3	2	2	3	2	1	0

Figure 2.4 Une diagonale dans la table des distances

La nouvelle idée de représentation de la table D est de garder en mémoire, pour chacune des diagonales de la table, seulement les positions où il y a une augmentation de la valeur de l'entrée par rapport à la valeur de l'entrée précédente sur cette même diagonale.

Définition 2.2 Pour une diagonale d et un nombre d'erreurs e, on définit $C(e, d)$ comme étant la colonne de plus grand indice j telle que $D(j - d, j) = e$. On a donc que les entrées de valeur e sur la diagonale d se terminent à la colonne $j = C(e, d)$. Si d ne comprend que des valeurs plus petites que e, on pose $C(e, d)$ égale à l'indice j de la dernière colonne présente dans cette diagonale.

Exemple 2.5 Dans l'exemple 2.4, $C(1, 6) = 8$ étant donné que la dernière entrée de valeur 1 sur la diagonale 6 est dans la colonne 8 de la table. On pose $C(1, 10) = 14$ car toutes les entrées de la diagonale 10 sont plus petites que 1 et la dernière colonne de cette diagonale est la colonne 14.

Notons que $C(e,d) - d$ est l'indice de la ligne correspondant à la dernière entrée de valeur e sur la diagonale d.

La définition de $C(e,d)$ implique que le nombre minimal d'erreurs entre $p_1 \dots p_{C(e,d)-d}$ et un suffixe de $t_1 \dots t_{C(e,d)}$ est e et que $t_{C(e,d)+1} \neq p_{C(e,d)-d+1}$. On a donc une occurrence approximative de P en position $m+d$ de T ayant au plus t erreurs, si et seulement si $C(e,d) = m+d$, pour $e \leq t$, et où m est la longueur du mot P.

Donc, si on calcule la table des $C(e,d)$, on aura l'information nécessaire pour le calcul des occurrences approximatives d'un mot dans un texte. Il est facile de calculer la table des $C(e,d)$, lorsqu'on a la table des distances D. On a alors qu'à se servir de la définition 2.2.

Exemple 2.6 L'information de la table des distances de l'exemple 2.4 peut être représentée par la table des $C(e,d)$ de la figure 2.5. Les valeurs de la première ligne de cette table et les valeurs -1 et $-\infty$ sont des valeurs initiales qui seront expliquées à la suite de cet exemple. Pour trouver les positions des occurrences approximatives de P dans T, on a qu'à regarder, dans chaque colonne d, s'il y a une entrée de valeur $m+d$. Dans notre exemple, comme $P = AACG$ on doit regarder, pour chacune des colonnes d, s'il y a des valeurs $4+d$ dans cette colonne. On a la valeur 4+9=13 dans la colonne 9, à la ligne $e = 1$, qui nous indique qu'il y a une occurrence de P, ayant une erreur, se terminant en position 13 du texte. Dans la colonne 10, il y a deux entrées de valeur $4+10 = 14$, une dans la ligne $e = 0$ et une dans la ligne $e = 1$. Cela nous indique qu'il y a une occurrence exacte (ayant 0 erreur) de P se terminant en position 14 du texte.

On voudrait maintenant pouvoir calculer la table des $C(e,d)$ sans avoir à calculer la table des distances D. Pour ce faire, on a besoin des valeurs initiales suivantes.

Pour les diagonales $d \geq 0$, la valeur initiale de la diagonale d est 0, car $D(0,j) = j$, $\forall j$. On assigne donc la valeur $d-1$ à $C(-1,d)$ pour indiquer que les entrées imaginaires de valeur -1 se terminent à la colonne $d-1$. Comme la valeur initiale des diagonales d,

e \ d	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12
-1		$-\infty$	-1	0	1	2	3	4	5	6	7	8	9	10	11
0	$-\infty$	-1	0	1	2	3	4	5	6	8	8	9	**14**	12	
1	-1	0	1	2	3	4	5	6	8	9	9	**13**	**14**		

Figure 2.5 Une table de valeurs $C(e, d)$

pour $-k+1 \leq d \leq -1$, est $|d|$, on assigne -1 à $C(|d|-1, d)$. Finalement, on assigne la valeur $-\infty$ à $C(|d|-2, d)$, pour $-k+1 \leq d \leq -1$.

Définition 2.3 On définit une *C-diagonale c* comme étant l'ensemble des $C(e, d)$ telles que $e + d = c$.

Exemple 2.7 La C-diagonale 3 de l'exemple 2.6 est représentée dans la table suivante par les éléments en caractères gras:

e \ d	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12
-1		$-\infty$	-1	0	1	2	**3**	4	5	6	7	8	9	10	11
0	$-\infty$	-1	0	1	2	**3**	4	5	6	8	8	9	**14**	12	
1	-1	0	1	2	**3**	4	5	6	8	9	9	13	14		

Figure 2.6 Une C-diagonale

On peut calculer la table C, C-diagonale par C-diagonale, en commençant par l'élément en haut d'une diagonale. Pour ce faire, on utilise l'algorithme suivant, tiré d'un article de Galil et Park (Galil - Park, 1990). Cet algorithme est une variation de l'algorithme

d'Ukkonen (Ukkonen, 1983) qui calculait plutôt une table $L(e,d)$, où $L(e,d)$ est la *ligne* d'indice le plus grand, sur la diagonale d qui a comme valeur e. Avec cette méthode, si $L(e,d) = m$ alors il y a une occurrence de P, ayant e erreurs, se terminant en position $L(e,d) + d = m + d$ du texte.

Théorème 2.2 *Si $C(e,d)$ appartient à la C-diagonale c et que toutes les C-diagonales à gauche de c ont déjà été calculées, alors on peut obtenir sa valeur avec l'algorithme suivant:*

1. $j \leftarrow \max(C(e-1,d-1)+1, C(e-1,d)+1, C(e-1,d+1))$
2. **tant que** $t_{j+1} = p_{j-d+1}$ **faire**
3. $\quad j \leftarrow j + 1$
4. **fin tant que**
5. $C(e,d) \leftarrow j$

■

On peut donc calculer une C-diagonale en temps $\mathcal{O}(m)$, étant donné que, dans le pire des cas, m paires (p_i, t_j), telles que $j - i = d$ sont comparées. Comme il y a n C-diagonales, le calcul de la table C se fait en temps $\mathcal{O}(nm)$.

Nous verrons dans la prochaine section que cette idée de travailler avec les diagonales a été développée par plusieurs groupes de chercheurs. Chaque groupe a trouvé une méthode pour calculer la boucle tant que, pour un $C(e,d)$ donné, en temps constant, en faisant un prétraitement du mot P. Le temps de calcul d'une C-diagonale se réduit alors à $\mathcal{O}(t)$, étant donné que chacune de ces diagonales contient t éléments: $C(0,d)$, $C(1,d)$, ..., $C(t,d)$.

Remarque 2.1 Si on travaille avec les $L(e,d)$ plutôt qu'avec les $C(e,d)$, l'algorithme du théorème 2.2 devient:

1. $i \leftarrow \max(L(e-1,d-1)+1, L(e-1,d)+1, L(e-1,d+1))$
2. **tant que** $p_{i+1} = t_{d+i+1}$ **faire**
3. $\quad i \leftarrow i + 1$

4. **fin tant que**
5. $L(e, d) \leftarrow i$

Exemple 2.8 Voici la table des $L(e,d)$ pour l'exemple 2.4:

d

e	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12
-1		-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0	0	0	0	1	0	0	**4**	1	
1	1	1	1	1	1	1	1	1	2	2	1	**4**	**4**		

Figure 2.7 Une L-diagonale

2.3.2 Premier algorithme de Landau-Vishkin

Dans leur algorithme (Landau-Vishkin, 1988), Landau et Vishkin font un prétraitement du mot P qui consiste à calculer une table de dimension $m \times m$, qu'ils dénotent Max_Length. Max_Length $(i,j) = f$ implique que le plus long préfixe commun à $p_{i+1} \dots p_m$ et $p_{j+1} \dots p_m$ est de longueur f. La table est calculée pour les valeurs i et j telles que $0 \leq i,j \leq m-1$.

Exemple 2.9 Soit $P = AACG$ un mot. Voici la table des valeurs de Max_Length pour ce mot:

	0	1	2	3
0	4	1	0	0
1		3	0	0
2			2	0
3				1

Cette table peut être calculée facilement en temps $\mathcal{O}(m^2)$.

Maintenant, pour calculer les occurrences approximatives, à t erreurs près, d'un mot P dans un texte T, voici l'idée de Landau et Vishkin. Leur algorithme procède en

$n - m + t + 1$ itérations. À l'itération i, $0 \leq i \leq n - m + t$, l'algorithme regarde s'il y a une occurrence de P, ayant moins de t erreurs, *commençant* à la position $i+1$ du texte. Une façon simple de faire cela, mais qui nous amène à un algorithme de complexité en temps de $\mathcal{O}(nm^2)$, est la suivante:

1) À l'itération i, on construit la table des distances $D^{(i)}$, de dimension $(m+k+1) \times (m+1)$, découlant de l'alignement de P avec le sous-mot du texte de longueur $m+k$ commençant en position $i+1$. (On se sert de la récurrence pour l'alignement global de deux mots, définie dans la proposition 1.3.)

2) Si $D(m, j) \leq t$, pour au moins un j, $m - t \leq j \leq m + t$, alors on conclut que $t_{i+1} \ldots t_j$ est une occurrence approximative de P, ayant $D(m, j) \leq t$ erreurs.

Pour chaque itération i, calculer la table $D^{(i)}$ prend un temps $\mathcal{O}(m^2)$, car on a $m \times (m+k)$ valeurs à calculer dans chacune des tables. Comme l'algorithme procède en $n-m+k+1$ itérations, la complexité en temps est de $\mathcal{O}(nm^2)$. L'idée est donc de réduire le temps de calcul de l'étape 1 de l'algorithme.

La première idée pour réduire le temps de calcul de l'étape 1 est de se servir des diagonales d'Ukkonen (Voir la section 2.3.1). On peut alors remplacer l'étape 1 précédente par ce qui suit:

1') Calculer pour $D^{(i)}$, les valeurs de $L(e, d)$ pour $0 \leq e \leq t$ et $-t \leq d \leq t$.

Ici, on a qu'à calculer $L(e, d)$ pour les diagonales d, telles que $-t \leq d \leq t$, car on travaille avec la table des distances entre deux mots et, dans cette table, chaque diagonale d commence par la valeur $|d|$. Comme le nombre d'erreurs permis est t, les seules diagonales pertinentes à notre calcul sont celles comprises entre $-t$ et t. On a vu que le calcul d'une L-diagonale se fait en temps $\mathcal{O}(m)$ et donc, en utilisant cette nouvelle méthode, l'étape 1' de l'algorithme a une complexité en temps de $\mathcal{O}(mt)$, car on calcule $2t + 1$ diagonales. La complexité de l'algorithme est donc, dans ce cas, de $\mathcal{O}(mnt)$.

Maintenant, on se sert de notre prétraitement du mot P, pour augmenter l'efficacité de l'algorithme en diminuant le temps de calcul de la boucle **tant que** de l'algorithme présenté dans la remarque 2.1. Cette boucle calcule le plus long préfixe commun à un suffixe du mot P et un suffixe du texte T. On va voir que l'on peut réduire le temps de calcul de ce plus long préfixe en gardant certaines informations concernant des préfixes communs provenant d'une itération antérieure.

Supposons que l'on soit rendu au calcul de l'itération i de l'algorithme. À la fin de l'itération $i-1$, on a calculé, pour chaque position $x = 1, 2, \ldots, i$ du texte, le plus long préfixe de $t_x t_{x+1} \ldots t_n$ qui s'aligne, avec au plus t erreurs, avec un préfixe du mot P. Pour chaque x, notons $t_x t_{x+1} \ldots t_{j(x)}$ ce préfixe. Soit j le maximum de $j(1), j(2), \ldots, j(i)$. Ce j représente donc la position la plus à droite dans le texte qu'on a réussi à aligner dans une itération précédente. Finalement, on dénote r l'itération précédent i où cette position j a été atteinte.

À l'itération r, $r < i$, on a trouvé, par définition de r, un alignement entre $t_{r+1} \ldots t_j$ et un préfixe p de P, ayant au plus t erreurs. Cet alignement induit un alignement, ayant au plus t erreurs, entre $t_{i+1} \ldots t_j$ et un suffixe de p. Cet alignement est constitué d'au plus $t+1$ séquences d'identités séparées par les erreurs. On code l'alignement de la façon suivante:

- Si $t_{p+1} \ldots t_{p+f} = p_{c+1} \ldots p_{c+f}$ et que $t_{p+f+1} \neq p_{c+f+1}$, on dénote cette suite d'identités par le triplet (p, c, f).
- Si dans l'alignement, il y a une erreur en position t_{h+1}, on dénote ce fait par le triplet $(h, 0, 0)$.

L'alignement entre $t_{i+1} \ldots t_j$ et un suffixe de p peut alors être décrit par une séquence d'au plus $2t+2$ triplets. On dénote cette séquence $S_{i,j}$. Voyons ces définitions sur un exemple, tiré de l'article (Landau-Vishkin, 1988).

Exemple 2.10 Soit $t_{17} \ldots t_{30} = abaaacddacdcac$, un sous-mot de T et le préfixe de P,

$p_1 \dots p_{13} = aaaaeddcdcbab$. Si on se trouve à l'itération $i = 20$, que le nombre d'erreurs permis est $t = 5$, que $r = 16$ et $j = 30$, on a l'alignement suivant qui nous provient de l'itération $i = 16$:

1		2	3	4	5	6	7		8	9	10	11	12	13
a		a	a	a	e	d	d		c	d	c	b	a	b
a	b	a	a	a	c	d	d	a	c	d	c		a	c
17	18	19	20	21	22	23	24	25	26	27	28		29	30

L'alignement de $t_{17} \dots t_{30}$ avec $p = p_1 \dots p_{13}$ induit ici un alignement de $t_{21} \dots t_{30}$ avec le suffixe $p_4 \dots p_{13}$ de p. Cet alignement induit peut être représenté par la séquence de triplets, $S_{20,30}$, suivante:

$$\{(20, 3, 1), (21, 0, 0), (22, 5, 2), (24, 0, 0), (25, 7, 3), (28, 11, 1), (29, 0, 0)\}.$$

Le triplet $(22, 5, 2)$ nous dit, par exemple, que $t_{23}t_{24} = p_6p_7$ et $t_{25} \neq p_8$. On dit alors que le triplet $(22, 5, 2)$ couvre les positions 23 et 24 du texte. Le triplet $(24, 0, 0)$ nous révèle, tant qu'à lui, une erreur d'alignement en position 25 du texte. Ce triplet ne couvre que la position 25 du texte.

Le but de toutes ces nouvelles définitions est de calculer de façon efficace la boucle **tant que** de l'algorithme de calcul des $L(e, d)$, qui à l'itération i est:

1. ligne $\leftarrow \max(L(e-1, d-1)+1, L(e-1, d)+1, L(e-1, d+1))$
2. **tant que** $p_{ligne+1} = t_{i+ligne+d+1}$ **faire**
3. ligne $\leftarrow$ ligne + 1
4. **fin tant que**
5. $L(e, d) \leftarrow ligne$

On cherche donc le plus grand w tel que

$$p_{ligne+1} \dots p_{ligne+w} = t_{i+ligne+d+1} \dots t_{i+ligne+d+w}.$$

Donc, pour une itération i, l'étape 1' de l'algorithme devient

1") Calculer pour $D^{(i)}$ les valeurs de $L(e,d)$ pour $0 \leq e \leq t$ et $-t \leq d \leq t$ on se servant des séquences $S_{i,j}$. (Pour les détails de la procédure voir (Landau-Vishkin, 1988).)

La complexité en temps de cet algorithme est de $\mathcal{O}(t^2 n)$. Cela vient du fait que pour chacune des $n - m + t + 1$ itérations i, l'algorithme calcule $\mathcal{O}(t)$ diagonales. Chacune de ces diagonales est calculée en temps $\mathcal{O}(t)$ étant donné que pour le calcul de chaque élément $L(e,d)$ d'une diagonale d, on se sert des triplets de $S_{i,j}$ et il y a, au plus, $2t+2$ de ces triplets.

2.3.3 Algorithme de Galil-Park

L'idée de Galil et Park (Galil - Park, 1990) est de reprendre l'algorithme de Landau et Vishkin (Landau-Vishkin, 1988) et de calculer les C-diagonales (et non les L-diagonales) correspondant à la table des distances D pour le problème de la recherche des occurrences approximatives, ayant au plus t erreurs, de P dans T. Donc ici, on ne calcule plus les diagonales correspondant à une table $D^{(i)}$ pour chacune des itérations i, $0 \leq i \leq n - m + t$, comme Landau et Vishkin le faisaient (voir section précédente). On calcule les C-diagonales d'une seule table, la table $(m+1) \times (n+1)$ des distances correspondant à la recherche approximative de P dans T.

Galil et Park se servent du même prétraitement du mot P que Landau et Vishkin, c'est-à-dire qu'il calcule une table de longueur de préfixes communs à tous les suffixes de P. Il nomme Préfixe(i,j) la longueur du plus long préfixe commun à $p_{i+1} \ldots p_m$ et $p_{j+1} \ldots p_m$. Ils se servent aussi de l'idée de triplets de références (u,v,w), où u est la position de départ dans le texte, v la position finale dans le texte et w la D-diagonale où l'on se trouve. On a le triplet (u,v,w) si $t_u \ldots t_v = p_{u-w} \ldots p_{v-w}$ et $t_{v+1} \neq p_{v-w+1}$.

L'algorithme calcule la table des $C(e,d)$, C-diagonale par C-diagonale. Le calcul des $C(e,d)$ se fait en utilisant la table Préfixe et les triplets de références en temps $\mathcal{O}(t)$. Comme il y a $n - m + k + 1$ C-diagonales à calculer, la complexité en temps de cet algorithme est de $\mathcal{O}(tn)$.

2.3.4 Deuxième algorithme de Landau-Vishkin

Un an après la parution de leur premier algorithme de recherche approximative, Landau et Vishkin ont publié un nouvel algorithme ayant une complexité en temps de $\mathcal{O}(tn)$ (Landau-Vishkin, 1989). Comme nous allons le voir, cet algorithme se sert encore du calcul des L-diagonales à la Ukkonen (Ukkonen, 1983) mais, cette fois-ci, l'algorithme se sert d'un arbre de suffixes pour calculer les $L(e, d)$ (voir remarque 2.1) en temps constant.

Le nouvel algorithme procède en deux étapes:

1. On fait la concaténation du texte T et du mot P cherché et on obtient une séquence $S = t_1 \dots t_n p_1 \dots p_m$. On construit l'arbre des suffixes de $S\$$ en temps $\mathcal{O}(n + m)$.
2. On trouve les occurrences approximatives de P dans T, ayant au plus t erreurs, en calculant les L-diagonales correspondant à la table des distances D pour la recherche approximative de P dans T (voir section 2.3.1).

Pour mieux comprendre l'étape 1 de l'algorithme, on va maintenant définir le concept d'arbre des suffixes et en construire un exemple. Le premier algorithme linéaire pour la construction d'un arbre de suffixe est dû à Weiner, (Weiner, 1973). Pour un algorithme linéaire plus simple voir (Ukkonen, 1995).

Définition 2.4 Soit le mot $s_1 s_2 \dots s_{l-1}$ sur un alphabet A. Supposons que le symbole $\$ \notin A$. On pose $s_l = \$$ et on définit l'*arbre des suffixes* de $S = s_1 \dots s_l$ de la façon suivante:

1. C'est un arbre dans lequel toutes les arêtes sont dirigées dans la direction opposée à la racine et dont le degré des nœuds est 0, si ce nœud est une feuille, et ≥ 2, sinon.
2. Chaque suffixe $S_i = s_i \dots s_l$ du mot S définit une feuille de l'arbre. L'arbre a donc l feuilles.

3. Soit S_i et S_j deux suffixes de S. Supposons que $s_i \dots s_{i+f}$ est leur plus long préfixe commun. On a donc $s_i \dots s_{i+f} = s_j \dots s_{j+f}$ et $s_{i+f+1} \neq s_{j+f+1}$. Alors, $s_i \dots s_{i+f}$ définit un nœud interne de l'arbre.

Exemple 2.11 Si on prend le mot P et le texte T de l'exemple 2.4, c'est-à-dire le mot $P = AACG$ et le texte $T = GCGTTGCAGGAACG$ alors, l'arbre des suffixes de $S = GCGTTGCAGGAACGAACG\$$ est:

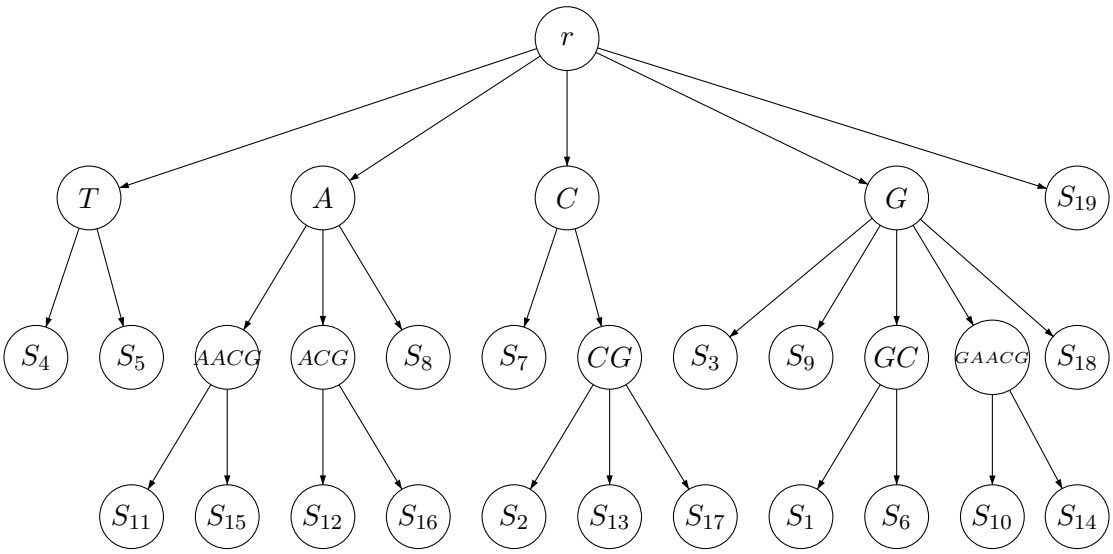


Figure 2.8 Un arbre des suffixes

La nouvelle idée de Landau et Vishkin est de se servir de l'arbre des suffixes pour calculer les $L(e,d)$ en temps $\mathcal{O}(1)$. Rappelons que l'algorithme de calcul des $L(e,d)$ est:

1. $i \leftarrow \max(L(e-1,d-1)+1, L(e-1,d)+1, L(e-1,d+1))$
2. **tant que** $p_{i+1} = t_{d+i+1}$ **faire**
3. $\quad i \leftarrow i+1$
4. **fin tant que**
5. $L(e,d) \leftarrow i$

Après l'instruction 1 de cet algorithme, on a aligné, avec e erreurs, $p_1 \dots p_i$ avec un suffixe du texte T se terminant en position $d + i$. La boucle **tant que** nous permet alors de trouver le plus grand q tel que $p_{i+1} \dots p_{i+q} = t_{i+d+1} \dots t_{i+d+q}$, c'est-à-dire la longueur du plus grand préfixe commun à $p_{i+1} \dots p_m$ et $t_{i+d+1} \dots t_n$.

Définition 2.5 On dénote $LCA(i, d)$ le *plus petit ancêtre commun* des feuilles représentant le suffixe $p_{i+1} \dots p_m$ de P et le suffixe $t_{i+d+1} \dots t_n$ de T.

Avec cette définition, la valeur de q est donnée par **longueur** $(LCA(i, d))$, et l'algorithme de calcul des $L(e, d)$ devient:

1. $i \leftarrow \max(L(e-1, d-1)+1, L(e-1, d)+1, L(e-1, d+1))$
2. $q \leftarrow$ **longueur**$(LCA(i, d))$
3. $L(e, d) \leftarrow i + q$

Le problème de trouver q est donc réduit au problème consistant à trouver le plus petit ancêtre commun de deux feuilles dans un arbre des suffixes. Ce problème peut être résolu en temps constant, après un prétraitement, en temps $\mathcal{O}(n)$ de l'arbre (voir (Harel - Tarjan, 1984) ou (Schieber - Vishkin, 1988)).

La complexité en temps de ce deuxième algorithme de Landau et Vishkin est donc bien de $\mathcal{O}(tn)$, étant donné que l'on doit calculer $\mathcal{O}(n)$ L-diagonales et que chacune de ces L-diagonales comprend t valeurs $L(e, d)$.

2.3.5 Algorithme de Ukkonen-Wood

On veut trouver une façon de calculer la boucle **tant que** du théorème 2.6 en temps constant. On a déjà remarqué que si j_0 et j_1 représentent les valeurs de j avant et après la boucle, $j_1 - j_0$ est égale à la longueur du plus long préfixe commun à $p_{j_0-d+1} \dots p_m$ et $t_{j_0+1} \dots t_n$. On veut donc une façon de calculer rapidement les plus longs préfixes communs à tous les suffixes de P et T, avec un prétraitement de P ou un prétraitement

de P et T. Pour arriver à cela, Ukkonen et Wood (Ukkonen - Wood, 1993) font un prétraitement de P consistant en la construction de l'*automate suffixe*, $S(P)$, de P.

L'automate $S(P)$ reconnaît le langage

$$\Sigma^* \cdot (P_1 + P_2 + \ldots + P_m),$$

où Σ est l'alphabet du texte T considéré et où $P_i = p_i \ldots p_m$. On appelle $S(P)$ l'automate suffixe de P étant donné qu'il reconnaît tous les mots se terminant par un suffixe de P.

Exemple 2.12 L'automate $S(P)$ du mot $P = AACG$ est le suivant:

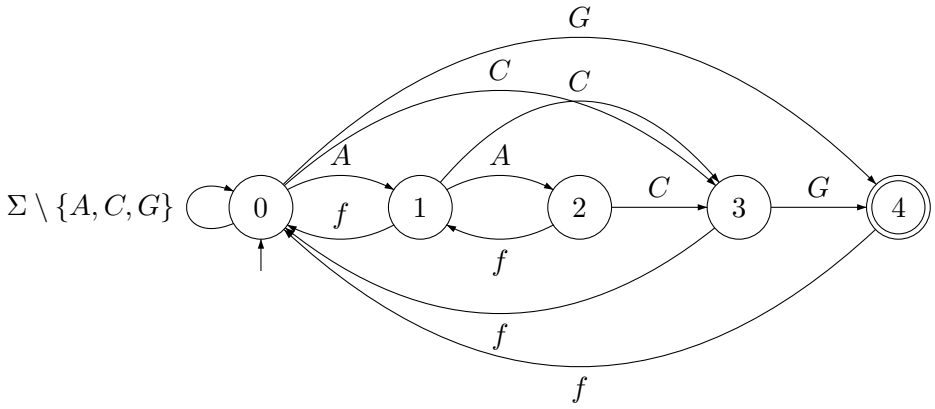


Figure 2.9 Un automate suffixe

On voit que l'automate $S(P)$ n'est pas complet. Il contient seulement les transitions nécessaire à la lecture de tout mot se terminant par un suffixe de P. On prend une transition f (pour *fail*), dans un état e à la lecture d'un symbole s, si la transition $\delta(e, s)$ n'est pas définie. Une transition f est comme une transition λ et donc, en la prenant, on n'avance pas dans le texte T.

Pour des algorithmes de construction de ces automates, voir (Blumer et al., 1985; Crochemore, 1986; Crochemore, 1988). (Il est intéressant de mentionner ici que l'automate suffixe, en plus de reconnaître au moins tous les suffixes de P, est un automate acyclique contenant un nombre linéaire de transitions. Dans un article récent (Allauzen -

Crochemore - Raffinot, 2001), Allauzen, Crochemore et Raffinot ont introduit un nouvel automate, appelé l'oracle des suffixes, ayant les mêmes propriétés que l'automate suffixe mais dont le nombre d'états est minimal, c'est-à-dire comportant $m+1$ états, où m est la longueur du mot P.)

Après la lecture de $t_1 \dots t_j$ l'automate nous donne l'information suivante:

1. La longueur du plus long suffixe de $t_1 \dots t_j$ qui est aussi un préfixe de P_1 ou P_2 ou $\dots$ ou P_m.
2. La position dans P où se termine ce suffixe.

Cette information permet à Ukkonen et Wood de calculer les plus longs préfixes communs à tous les suffixes de P et T en temps constant et on a donc, encore une fois un algorithme ayant une complexité en temps de $\mathcal{O}(tn)$.

2.4 La méthode des 4 Russes

En 1970, quatre mathématiciens russes, Arlazarov, Dinic, Kronrod et Faradzev, publient un algorithme permettant de calculer les entrées d'une matrice de dimension $n \times n$, reliée au calcul de la fermeture transitive d'un graphe orienté sur n sommets, en temps $\mathcal{O}(n^2/\log n)$ (Arlazarov et al., 1970). La méthode utilisée dans leur algorithme consiste à précalculer toutes les matrices correspondant à des problèmes plus petits, c'est-à-dire, dans ce cas-ci, toutes les matrices correspondant au calcul de la fermeture transitive de graphes orientés sur l sommets. Le nombre l est choisi assez petit et habituellement l divise parfaitement n. Ensuite, on subdivise la grande matrice $n \times n$ en sous-matrices de dimension $l \times l$ et on se sert des résultats précalculés pour trouver le résultat final. Cette méthode est communément appelée la *méthode des 4 Russes*.

2.4.1 Algorithme de Masek-Paterson pour l'alignement global

En 1980, Masek et Paterson ont l'idée de se servir de la méthode des 4 Russes pour résoudre le problème de l'alignement global de deux mots en temps $\mathcal{O}(n^2/\log\, n)$ (Masek - Paterson, 1980). Étant donné deux mots de longueur n, ils commencent par choisir un petit nombre k, tel que k divise n. Ils précalculent ensuite toutes les sous-matrices possibles de dimension $(k+1) \times (k+1)$. Le calcul d'une de ces sous-matrices est entièrement déterminé par sa première ligne L, sa première colonne C et deux mots, m_1 et m_2, de longueur k sur l'alphabet Σ considéré. Étant donné L, C, m_1 et m_2 fixés, on garde en mémoire la dernière ligne et la dernière colonne de la sous-matrice des distances correspondante. L'algorithme se termine par le calcul de la matrice des distances D, sous-matrices par sous-matrices, au lieu de cellule par cellule. Mais combien de sous-matrices doit-on précalculer avec cette méthode?

Les lignes et les colonnes d'une table de distances D, entre deux mots de longueur n, comprennent des valeurs entre 0 et n. De plus, deux entrées successives, sur une même ligne ou une même colonne, diffèrent toujours de 0, 1 ou -1. On a donc environ $n \cdot 3^k$ premières lignes ou colonnes possibles dans des matrices de dimension $(k+1) \times (k+1)$. De plus, le nombre de mots de longueur k sur un alphabet Σ est $|\Sigma|^k$. On doit donc précalculer $n^2 \cdot 3^{2k} \cdot |\Sigma|^{2k}$ matrices de dimension $(k+1) \times (k+1)$. Chacune de ces matrice est calculée en temps $\mathcal{O}(k^2)$. C'est long!!! Ce qu'il faut remarquer c'est qu'on calcule beaucoup trop de sous-matrices avec cette idée. Voyons maintenant un exemple, pour visualiser la technique, et parlons ensuite du raffinement que Masek et Paterson ont apporté à leur algorithme pour que le nombre de sous-matrices à calculer demeure raisonnable.

Exemple 2.13 Supposons que l'on veuille calculer la distance d'édition entre les deux séquences $GTCAGG$ et $CATAGT$. Ici la longueur des mots est 6 et on peut prendre, par exemple, $k = 3$. La première étape, que je ne ferai pas ici, consiste à précalculer toutes les sous-matrices de dimension 4×4 possibles sur l'alphabet $\{A, C, G, T\}$ (il y

a $6^2 \cdot 3^6 \cdot 4^6 = 107\ 495\ 424$) et de garder en mémoire la dernière ligne et la dernière colonne de chacune de ces sous-matrices. Maintenant, initialisons la table des distances pour ce problème et subdivisons-la en matrices de dimension 4×4:

	λ	G	T	C	A	G	G
λ	0	1	2	3	4	5	6
C	1						
A	2						
T	3						
A	4						
G	5						
T	6						

On commence par aller chercher la dernière ligne et la dernière colonne de la sous-matrice en haut à gauche, c'est-à-dire la sous-matrice ayant comme première ligne 0123, comme première colonne 0123 et dont les sous-mots à aligner sont GTC et CAT. On obtient ce qui suit:

	λ	G	T	C	A	G	G
λ	0	1	2	3	4	5	6
C	1			2			
A	2			3			
T	3	3	2	3			
A	4						
G	5						
T	6						

On a maintenant l'information nécessaire au calcul des dernières lignes et colonnes de deux nouvelles sous-matrices. La première correspond aux vecteurs 3456, 3233 et aux sous-mots AGG et CAT et la deuxième aux vecteurs 3323 et 3456 et aux sous-mots GTC et AGT:

	λ	G	T	C	A	G	G
λ	0	1	2	3	4	5	6
C	1			2			5
A	2			3			4
T	3	3	2	3	3	3	4
A	4			3			
G	5			4			
T	6	5	4	5			

Finalement, on va chercher l'information correspondant à la dernière sous-matrice, en bas à droite. On obtient alors que la distance d'édition entre $GTCAGG$ et $CATAGT$ est 4:

	λ	G	T	C	A	G	G
λ	0	1	2	3	4	5	6
C	1			2			5
A	2			3			4
T	3	3	2	3	3	3	4
A	4			3			4
G	5			4			4
T	6	5	4	5	5	4	4

L'idée, pour diminuer le nombre de sous-matrices de dimension $(k+1)\times(k+1)$ à calculer, est de travailler avec des vecteurs de différences horizontales $(D(i,j) - D(i,j-1))$ et de différences verticales $(D(i,j) - D(i-1,j))$ au lieu de travailler avec des lignes et des colonnes pouvant apparaître dans la table D.

Exemple 2.14 La sous-matrice suivante de la table des distances de l'exemple 2.13 était représentée par sa première ligne 3456, sa première colonne 3233 ainsi que les sous-mots AGG et CAT:

	C	A	G	G
λ	3	4	5	6
C	2			
A	3			
T	3			

Nous la représenterons maintenant par les mêmes sous-mots AGG et CAT et par le vecteur de différences horizontales 111, associé à la ligne 3456, et le vecteur de différences verticales -110, associé à la colonne 3233.

Étant donné un vecteur de différences horizontales, correspondant à la première ligne d'une sous-matrice de distances de dimension $(k+1) \times (k+1)$, un vecteur de différences verticales, correspondant à la première colonne de la même sous-matrice, et deux mots m_1 et m_2 sur un alphabet Σ, Masek et Paterson donne un algorithme pour le calcul, en temps $\mathcal{O}(k^2)$, du vecteur de différences horizontales, correspondant à la dernière ligne de la sous-matrice, et du vecteur de différences verticales, correspondant à la dernière colonne de la sous-matrice (Pour les détails voir (Masek - Paterson, 1980)).

Comme il y a 3^k vecteur de différences possibles, le nombre de sous-matrices à précalculer est réduit à $3^{2k} \cdot |\Sigma|^{2k}$.

Exemple 2.15 Avec cette nouvelle méthode, dans l'exemple 2.13, on diminue le nombre de sous-matrices à calculer à seulement 2 985 984. C'est encore beaucoup mais c'est vraiment mieux que dans le cas précédent.

2.4.2 Algorithme de Wu, Manber et Myers pour la recherche approximative

Dans un article publié en 1996, Wu, Manber et Myers (Wu - Manber - Myers, 1996) présentent deux algorithmes pour résoudre le problème de la recherche des occurrences approximatives, ayant au plus t erreurs, d'un mot P dans un texte T, employant la

méthode des 4 Russes et ayant une complexité en temps de $\mathcal{O}(nm/\log\, n)$ et $\mathcal{O}(nt/\log\, n)$, respectivement. Nous décrivons brièvement ces algorithmes dans ce qui suit.

Le premier algorithme se base sur une idée d'Ukkonen (Ukkonen, 1985) qui consiste à faire un prétraitement du mot à chercher P en construisant, à partir de celui-ci, un automate déterministe (voir la section 2.2).

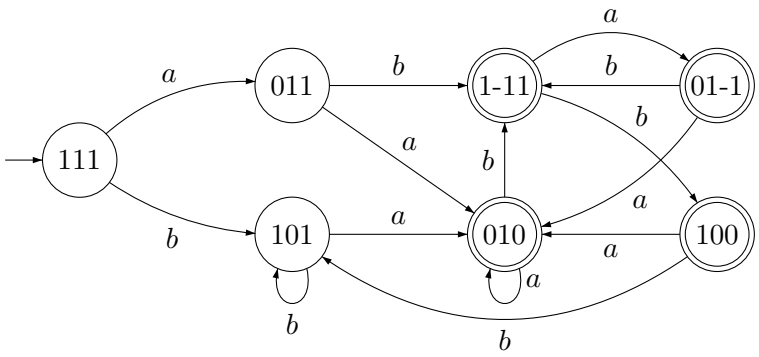
Rappelons que l'automate M_P, construit à partir de P, a comme états toutes les colonnes pouvant possiblement apparaître dans la table des distances D, lorsque le texte T varie. Ukkonen a démontré (Ukkonen, 1985) que le nombre d'états de M_P est borné par 3^m, où m est la longueur du mot P. Ce résultat découle du fait que la différence (verticale ou horizontale) entre deux éléments de la table des distances est toujours comprise entre -1 et 1. Pour expliquer l'idée de Wu, Manber et Myers, commençons par introduire une petite variation sur l'automate M_P.

Définition 2.6 On définit le *vecteur de différences verticales*, $\mathbf{\Delta v_j}$, associé à la colonne j de la table des distances D, pour $1 \leq i \leq m$, par:

$$\mathbf{\Delta v_j}[i] = \Delta v_{i,j} = D(i,j) - D(i-1,j).$$

Une autre façon de représenter M_P est de se servir des vecteurs $\mathbf{\Delta v_j}$, au lieu des colonnes j de la table D, comme états de l'automate.

Exemple 2.16 L'automate de l'exemple 2.3 est maintenant représenté de la façon suivante:



Pour bien comprendre les transitions entre les nouveaux états de M_P, il faut pouvoir calculer $\mathbf{\Delta v_j}$ à partir de $\mathbf{\Delta v_{j-1}}$. Pour ce faire, on doit introduire le concept des différences horizontales d'une colonne.

Définition 2.7 On définit le *vecteur de différences horizontales*, $\mathbf{\Delta h_j}$, associé à la colonne j de la table des distances D, pour $1 \leq i \leq m$, par:

$$\mathbf{\Delta h_j}[i] = \Delta h_{i,j} = D(i,j) - D(i,j-1).$$

Théorème 2.3 *On a la relation de récurrence suivante:*

$$\Delta v_{i,j} = \min \begin{cases} 1 \\ \Delta v_{i,j-1} - \Delta h_{i-1,j} + 1 \\ \delta(i,j) - \Delta h_{i-1,j} \end{cases},$$

où $\delta(i,j) = 0$ *si* $x_i = y_j$ *et* 1 *sinon et où* $v_{i,0} = 1$, *pour* $1 \leq i \leq m$. ■

Maintenant, pour calculer les valeurs du vecteur $\mathbf{\Delta h_j}$, on a le lemme suivant:

Lemme 2.2 *On a que* $\Delta h_{0,j} = 0$, $\forall j$ *et*

$$\Delta h_{i,j} = \Delta h_{i-1,j} + \Delta v_{i,j} - \Delta v_{i,j-1}.$$

■

Exemple 2.17 À l'exemple 2.16, on a construit l'automate M_P à partir du mot $P = aba$. Dans cet automate, on obtient la transition a entre l'état 111 et l'état 011 en exécutant les calculs suivants:

Premièrement, l'état initial de l'automate correspond au vecteur $\mathbf{\Delta v_0}$, c'est-à-dire le vecteur des différences verticales entre les éléments de la colonne 0 de la table D. Cette colonne est 0123 et donc $\mathbf{\Delta} v_0 = 111$. Maintenant, on veut calculer $\mathbf{\Delta v_1}$ si la première

lettre du texte T est a. Pour ce faire, on utilise le théorème 2.3 et le lemme 2.2:

$$\begin{aligned} \Delta v_{1,1} &= \min\{1, \Delta v_{1,0} - \Delta h_{0,1} + 1, \delta(1,1) - \Delta h_{0,1}\} \\ &= \min\{1, 1 - 0 + 1, 0 - 0\} \\ &= 0, \end{aligned}$$

car $\Delta h_{0,1} = 0$ et comme $P_1 = a$, on a que $\delta(1,1) = 0$. Maintenant, on doit calculer $\Delta h_{1,1}$.

$$\begin{aligned} \Delta h_{1,1} &= \Delta h_{0,1} + \Delta v_{1,1} - \Delta v_{1,0} \\ &= 0 + 0 - 1 = -1. \end{aligned}$$

Cette valeur nous permet de calculer la valeur de $\Delta v_{2,1}$ qui est de 1. Finalement, on procède de la même façon, et on obtient que $\Delta v_{3,1} = 1$.

Comme M_P peut contenir 3^m états, sa construction peut prendre un temps exponentiel en m. Pour contourner ce problème, Wu, Manber et Myers emploient la méthode des 4 Russes pour simuler l'automate M_P à l'aide d'une combinaison d'automates plus petits.

Pour ce faire, chaque vecteur $\mathbf{\Delta v_j}$ est divisé en sous-vecteurs, appelés *régions*, de longueur r. On prend habituellement un nombre r assez petit et divisant m.

Définition 2.8 On dénote $\mathbf{\Delta v_j}\langle q\rangle$ la $q^{\text{ième}}$ *région* du vecteur $\mathbf{\Delta v_j}$, c'est-à-dire le sous-vecteur

$$\mathbf{\Delta v_j}\langle q\rangle = \mathbf{\Delta v_j}[(q-1)r + 1 \ldots qr].$$

Chaque région peut donc être un des 3^r vecteurs de différences possibles. L'idée est de construire un automate $M_P\langle q\rangle$, ayant comme états les 3^r vecteurs de différences possibles, de sorte qu'il soit possible de trouver $\mathbf{\Delta v_j}\langle q\rangle$ à partir de $\mathbf{\Delta v_{j-1}}\langle q\rangle$ en se déplaçant dans l'automate pendant la lecture du texte T. Si on se reporte au théorème 2.3, l'état $\mathbf{\Delta v_j}\langle q\rangle$, dans $M_P\langle q\rangle$, dépend de trois facteurs:

1. L'état $\mathbf{\Delta v_{j-1}}\langle q\rangle$.

2. La valeur de $\mathbf{\Delta h_j}[(q-1)r]$. Cette valeur est la dernière différence horizontale calculée dans la région $q-1$ du vecteur $\mathbf{\Delta h_j}$. Cette valeur sera une sortie de l'automate $M_P\langle q-1\rangle$.

3. Les valeurs de $\delta(i,j)$, pour $(q-1)r+1 \leq i \leq qr$. Ces valeurs forment le *vecteur caractéristique*, $\mathbf{t_j}\langle q\rangle$, du symbole t_j, sur la région q du mot P.

Les transitions entre les états sont des couples (vc, s) où vc est un des vecteurs caractéristiques possibles et $s \in \{-1, 0, 1\}$ est la valeur de $\mathbf{\Delta h_j}[(q-1)r]$ provenant de l'automate $M_P\langle q-1\rangle$.

Maintenant, si P est de longueur m et r est la longueur des régions que l'on veut considérer alors, pour simuler l'automate M_P, on prend m/r copies de notre automate universel $M_P\langle q\rangle$, où q est n'importe quelle des régions de longueur r.

Cet algorithme a une complexité en temps de $\mathcal{O}(nm)/\log n$. Si on associe cette idée avec celle des diagonales d'Ukkonen, (Ukkonen, 1985), on obtient un algorithme en $\mathcal{O}(nt)/\log n$, où t est le nombre d'erreurs permis dans les occurrences. Pour les détails de ce dernier algorithme voir (Wu - Manber - Myers, 1996).

2.5 Une approche à la Boyer-Moore

Dans un article datant de 1990, Tarhio et Ukkonen (Tarhio - Ukkonen, 1990) utilisent une approche à la Boyer-Moore (Boyer - Moore, 1977) pour résoudre le problème de la recherche des occurrences approximatives d'un mot dans un texte. Rappelons que l'algorithme de Boyer et Moore permet de trouver les occurrences exactes d'un mot dans un texte en travaillant en deux étapes: une étape de comparaison et une étape de déplacement vers la droite. L'algorithme commence par comparer le mot $P = p_1 \ldots p_m$, de droite à gauche, avec le préfixe de longueur m de T. Dès que l'on trouve une erreur, on arrête et on se déplace le plus possible vers la droite, puis on recommence la comparaison. L'idée de Tarhio et Ukkonen est donc de généraliser ce concept à la recherche des occurrences approximatives de P.

L'algorithme de Tarhio et Ukkonen consiste, premièrement, à trouver et marquer les positions j du texte où il y a très probablement une occurrence approximative de P. On calcule ensuite, dans la table D, seulement les diagonales commençant par les entrées $D(0, j)$ marquées. Pour ce calcul, on suppose que les entrées à l'extérieur des diagonales marquées sont égales à ∞.

Mais comment fait-on pour marquer une position j du texte? Rappelons que les occurrences approximatives, ayant moins de t erreurs, de P dans T peuvent être représentées par des alignements de P avec des sous-mots de T. On a vu au théorème 1.15 que ces alignements correspondent aux chemins de $D(0, j)$ à $D(m, h)$, où $D(m, h) \leq t$. On a alors le lemme suivant:

Lemme 2.3 *Les entrées d'un chemin représentant une occurrence approximative de P dans T, ayant au plus t erreurs, sont contenues dans au plus $t+1$ diagonales successives de D.* ■

Ce résultat vient du fait que lorsqu'on est dans une diagonale d, seules les opérations d'insertion et de suppression nous font changer de diagonale. Comme il y a au plus t de ces opérations dans un chemin représentant une occurrence de P ayant au plus t erreurs, ce chemin doit être compris dans au plus $t + 1$ diagonales.

Définition 2.9 Pour i, $1 \leq i \leq m$, on définit le *t-environnement* de p_i comme étant le mot $C_i = p_{i-t} \ldots p_{i+t}$, où on pose $p_j = \lambda$ pour $j < 1$ et $j > m$.

Lemme 2.4 *Si un chemin représentant une occurrence approximative de P, ayant au plus t erreurs, passe par une cellule sur la diagonale d de D alors, pour au plus t indices i, $1 \leq i \leq m$, t_{d+i} n'apparaît pas dans le t-environnement C_i.* ■

Ce lemme suggère une façon de marquer le texte. Pour une diagonale d, on regarde, pour $i = m, m-1, \ldots, 1$, si t_{d+i} est dans C_i. On arrête lorsque $t+1$ mauvaises colonnes

ont été trouvées. Si le nombre de mauvaises colonnes est $\leq t$, alors on marque les diagonales $d-t, \ldots, d+t$, c'est-à-dire les entrées $D(0, d-t), \ldots, D(0, d+t)$ de la table D.

Pour trouver les mauvaises colonnes rapidement, on peut précalculer, $\forall a \in \Sigma$ et $\forall i$, $1 \leq i \leq m$,

$$\text{Mauvaise}(i, a) = \begin{cases} \text{vrai} & \text{si } a \text{ n'apparaît pas dans le } t\text{-environnement } C_i \\ \text{faux} & \text{sinon} \end{cases}$$

Exemple 2.18 Calculons les valeurs de Mauvaise (i, a), pour le mot $P = AACG$ et l'alphabet $\Sigma = \{A, C, G, T\}$, si la recherche approximative permet au plus $t = 1$ erreur. On commence par trouver les t-environnement C_i pour $i = 1$ à 4:

$$C_1 = p_0 p_1 p_2 = \lambda AA = AA, \quad C_2 = p_1 p_2 p_3 = AAC,$$

$$C_3 = p_2 p_3 p_4 = ACG, \quad C_4 = p_3 p_4 p_5 = CG\lambda = CG,$$

puis on calcule la table des valeurs de Mauvaise(i, a):

$a \setminus j$	1	2	3	4
A	faux	faux	faux	vrai
C	vrai	faux	faux	faux
G	vrai	vrai	faux	faux
T	vrai	vrai	vrai	vrai

Après l'analyse de ce qui se passe autour de la diagonale d, on doit se déplacer vers la droite et recommencer l'analyse pour une autre diagonale. Le déplacement vers la droite ne doit pas être trop grand sinon on risque d'oublier une occurrence approximative de P. Il est évident qu'on peut faire un déplacement d'au moins $t+1$ diagonales et continuer l'analyse à la diagonale $d+t+1$. (À l'étape précédente on a marqué (ou non) les diagonales $d-t$ à $d+t$.) On peut aussi permettre un décalage plus grand en trouvant (avec des techniques à la Boyer-Moore, voir (Tarhio - Ukkonen, 1990)) la première

diagonale $d+h$ pour laquelle au moins un symbole $t_{d+h+m}, t_{d+h+m-1}, \ldots, t_{d+h+m-t}$ est identique au symbole correspondant dans le mot P. On prend alors, le maximum M entre $t+1$ et h et on recommence l'analyse à la diagonale $d+M$. Lorsqu'on arrive à la fin du texte, on a marqué toutes les diagonales pertinentes et on calcule ces diagonales en utilisant la programmation dynamique.

La complexité en temps de l'algorithme est au pire de $\mathcal{O}(nm/t)$, pour l'étape de comparaison et de déplacement. Pour le calcul des occurrences approximatives de P, il s'agit ensuite de calculer les valeurs de la table D pour les diagonales marquées à l'étape précédente. Cette étape requière donc un temps $\mathcal{O}(Nm)$, où N est le nombre de diagonales marquées. Dans le pire des cas, cet algorithme n'est donc pas très efficace, mais en moyenne, lorsque $m > 5$, $n > 5$ et k est petit, Tarhio et Ukkonen (Tarhio - Ukkonen, 1990) ont montré que l'efficacité de cet algorithme est souvent supérieur aux algorithmes présentés dans les sections précédentes.

2.6 Récapitulatif de la complexité en temps des différents algorithmes

Pour terminer ce chapitre, voici un tableau présentant la complexité en temps pour chacun des différents algorithmes. Il est bon de se rappeler que le m représente la longueur du mot cherché; le n, la longueur du texte; le t, le nombre d'erreurs permis et $|\Sigma|$ représente la cardinalité de l'alphabet d'entrée (du mot et/ou du texte). La colonne **Prétraitement** comprend la complexité du prétraitement et la colonne **Complexité** comprend la complexité après le prétraitement, c'est-à-dire ne tient pas compte de la complexité du prétraitement:

Approche	**Prétraitement**	**Complexité**
1. Ne calculer qu'une partie de la table des distances (Ukkonen, 1985) (section 2.1)	-	$\mathcal{O}(nm)$
2. Prétraitement du mot P à l'aide d'un automate (Ukkonen, 1985) (section 2.2)	$\mathcal{O}(m\|\Sigma\|k_p)$ où $k_p = min(3^m, 2^t\|\Sigma\|^t m^{t+1})$	$\mathcal{O}(n)$
3. Calcul de diagonales		
a) (Landau-Vishkin, 1988) (section 2.3.2)	$\mathcal{O}(m^2)$	$\mathcal{O}(t^2 n)$
b) (Galil - Park, 1990) (section 2.3.3)	$\mathcal{O}(m^2)$	$\mathcal{O}(tn)$
c) (Landau-Vishkin, 1989) (section 2.3.4)	$\mathcal{O}(n+m)$	$\mathcal{O}(tn)$
d) (Ukkonen - Wood, 1993) (section 2.3.5)	voir (Crochemore, 1988)	$\mathcal{O}(tn)$
4. La méthode des 4 Russes (section 2.4.2) (Wu - Manber - Myers, 1996)		
a)Premier algorithme	Voir l'article	$\mathcal{O}(nm/logn)$
b)Deuxième algorithme	Voir l'article	$\mathcal{O}(nt/logn)$
5. Une approche à la Boyer-Moore (Tarhio - Ukkonen, 1990) (section 2.5)	$\mathcal{O}(nm/t)$	$\mathcal{O}(Nm)$ N=le nombre de diagonales marquées

Figure 2.10 Tableau comparatif de la complexité des différents algorithmes de recherche approximative

CHAPITRE III

APPROCHES VECTORIELLES AU PROBLÈME DE LA RECHERCHE APPROXIMATIVE D'UN MOT DANS UN TEXTE

Dans ce qui suit, nous verrons deux approches vectorielles au problème de la recherche des occurrences approximatives d'un mot dans un texte. La première approche utilise des vecteurs de bits pour coder l'ensemble des états d'un automate non-déterministe, (Wu - Manber, 1992; Baeza-Yates - Gonnet, 1996). Une autre approche, développée par Myers (Myers, 1999), utilise des vecteurs de bits pour coder les séquences d'entrée et de sortie d'un automate déterministe. L'exploitation du parallélisme des opérations sur les vecteurs de bits améliore grandement le temps de calcul de ces algorithmes par rapport aux algorithmes présentés dans le chapitre précédent. Encore une fois, les preuves des théorèmes présentés dans ce chapitre ont été omises.

3.1 Opérations vectorielles, notations et algorithmes vectoriels

Nous introduisons ici la notation vectorielle qui sera utilisée dans ce qui suit. Nous utilisons la notation vectorielle standard, pour les opérations vectorielles usuelles, que nous étendons pour répondre à nos besoins en la matière.

Soit $\boldsymbol{x}= x_1 \dots x_m$ et $\boldsymbol{y}= y_1 \dots y_m$ deux vecteurs d'entiers.

Définition 3.1 L'équation $\boldsymbol{x} + \boldsymbol{y}$ ou $\boldsymbol{x} + \boldsymbol{y}$ (mod d) dénote une *addition vectorielle* habituelle, exécutée terme à terme.

Définition 3.2 Si $\boldsymbol{x}$ et $\boldsymbol{y}$ sont des vecteurs ne comprenant que des valeurs booléennes,

alors ils sont appelés *vecteurs de bits* et

$$\boldsymbol{x} \vee \boldsymbol{y}, \quad \boldsymbol{x} \wedge \boldsymbol{y}, \quad \neg \boldsymbol{x},$$

dénotent les opérations logiques de *disjonction, conjonction et négation*, dans lesquelles 0 remplace la valeur de vérité faux et 1, la valeur vrai. On peut aussi additionner des vecteurs de bits,

$$\boldsymbol{x} +_{\boldsymbol{b}} \boldsymbol{y},$$

en utilisant l'*addition binaire* avec retenue, exécutée de gauche à droite, en laissant tomber la dernière retenue si nécessaire.

Exemple 3.1 Voici la table de calculs pour les opérations logiques $\boldsymbol{x} \vee \boldsymbol{y}$, $\boldsymbol{x} \wedge \boldsymbol{y}$ et $\neg \boldsymbol{x}$:

$\boldsymbol{x}$	$\boldsymbol{y}$	$\boldsymbol{x} \vee \boldsymbol{y}$	$\boldsymbol{x} \wedge \boldsymbol{y}$	$\neg \boldsymbol{x}$
1	1	1	1	0
1	0	1	0	0
0	1	1	0	1
0	0	0	0	1

Exemple 3.2 Voici un exemple d'addition binaire. Le 1 entre parenthèse représente la dernière retenue qu'on oublie dans le calcul:

$$\begin{array}{r} 100111 \\ 110101 \\ \hline 001001(1) \end{array}$$

Nous allons aussi avoir besoin d'une opération vectorielle permettant le déplacement des éléments d'un vecteur d'une position vers la droite.

Définition 3.3 Soit $\boldsymbol{x} = x_1 \ldots x_m$.

$$\uparrow_a \boldsymbol{x} = a x_1 \ldots x_{m-1}$$

est appelé le *déplacement vers la droite* de $\boldsymbol{x}$. Les valeurs du vecteur $\boldsymbol{x}$ ont été déplacées d'une position vers la droite et la première valeur du vecteur devient a.

Exemple 3.3 Soit $\boldsymbol{x}$= 10000001, alors on a que $\uparrow_0 \boldsymbol{x} = 01000000$.

Pour pouvoir représenter un vecteur quelconque en vecteurs de bits, il nous faut maintenant introduire le concept de vecteur caractéristique.

Définition 3.4 Soit $\boldsymbol{x}$ un vecteur et a un symbole dans ce vecteur. Le *vecteur caractéristique* de a, noté $\boldsymbol{a}$, est le vecteur comportant des 1 aux positions dans $\boldsymbol{x}$ où le caractère a apparaît, et des 0 partout ailleurs.

Exemple 3.4 Soit $\boldsymbol{x}$= *abbacdaaba*, alors on a les vecteurs caractéristiques suivants:

$$\boldsymbol{a} = 1001001101, \qquad \boldsymbol{c} = 0000100000,$$

$$\boldsymbol{b} = 0110000010, \qquad \boldsymbol{d} = 0000010000.$$

Nous généralisons aussi la notation vectorielle à des prédicats et des termes arbitraires.

Exemple 3.5 $(\boldsymbol{x < y})$,$(\boldsymbol{F(x,y) = k})$ et $(\boldsymbol{x \in S})$ équivalent aux vecteurs booléens suivants:

$$(\boldsymbol{x < y}) = (x_1 < y_1, \ldots, x_m < y_m),$$

$$(\boldsymbol{F(x,y) = k}) = (F(x_1, y_1) = k, \ldots, F(x_m, y_m) = k),$$

$$(\boldsymbol{x \in S}) = (x_1 \in S, \ldots, x_m \in S).$$

Pour simplifier la notation, nous allons aussi écrire des propositions sous forme vectorielle.

Exemple 3.6 La proposition suivante:

$\forall i \in \{0, \ldots, m\}$, si $V_{(k-1)i} = \min(X_i, k-1)$ alors $V_{ki} = \begin{cases} V_{(k-1)i} + 1 & \text{si } X_i \geq k \\ V_{(k-1)i} & \text{sinon} \end{cases}$

s'écrit, sous forme vectorielle, de la façon simple suivante:

$$\text{Si } \boldsymbol{V_{k-1} = \min(X, k-1)} \text{ alors } \boldsymbol{V_k = V_{k-1} + (X \geq k).}$$

On a maintenant tous les outils nécessaires à l'introduction du concept d'algorithmes vectoriels.

Définition 3.5 Un *algorithme vectoriel* est un algorithme qui trouve un vecteur de sortie en appliquant un nombre d'opérations sur le vecteur d'entrée, indépendant de la longueur du vecteur.

Les algorithmes vectoriels peuvent donc être implantés en parallèles et/ou en employant les opérations sur les vecteurs de bits disponibles dans les processeurs ce qui augmente grandement leur efficacité. Dans ce qui suit, nous présentons différentes approches vectorielles à la recherche des occurrences approximatives d'un mot dans un texte. Nous utiliserons, dans la présentation de ces algorithmes, la notation vectorielle que nous venons d'introduire, plutôt que les notations particulières à chaque groupe d'auteurs, dans le but d'uniformiser le travail.

3.2 Algorithme de Wu et Manber

En 1992, Baeza-Yates et Gonnet (Baeza-Yates - Gonnet, 1992) présentent un algorithme vectoriel pour la recherche des occurrences exactes d'un mot dans un texte. Dans leur article, ils présentent aussi une version modifiée de leur algorithme permettant la recherche des occurrences approximatives d'un mot, ayant au plus t substitutions. (L'article ne couvre donc pas les insertions et suppressions de symboles.) Wu et Manber (Wu - Manber, 1992) se sont basés sur ces idées et ont développé un algorithme vectoriel pour le problème général de recherche des occurrences approximatives, c'est-à-dire le problème

où l'on permet des insertions, des suppressions et des substitutions de caractères. Dans ce qui suit, nous commençons par présenter l'idée de Baeza-Yates et Gonnet pour la recherche exacte d'un mot et nous donnons ensuite la généralisation de Wu et Manber.

3.2.1 Recherche exacte

L'idée de Baeza-Yates et Gonnet (Baeza-Yates - Gonnet, 1992) pour la recherche des occurrences exactes d'un mot $P = p_1 \dots p_m$ dans un texte $T = t_1 \dots t_n$ est de travailler avec des vecteurs de bits de longueur m, où m est la longueur du mot cherché. Pour chaque position j dans le texte T, on définit un vecteur de bits $\boldsymbol{R_j}$ qui contient l'information sur tous les alignements exacts de préfixes du mot P se terminant en position j du texte. Plus précisément, on a la définition suivante:

Définition 3.6 Le vecteur $\boldsymbol{R_j}$ est défini de la façon suivante, pour $1 \leq i \leq m$:

$$\boldsymbol{R_j}[i] = \begin{cases} 1 & \text{si le préfixe de longueur } i \text{ de } P \text{ est identique} \\ & \text{au suffixe de longueur } i \text{ de } t_1 \dots t_j \\ 0 & \text{sinon} \end{cases}$$

L'idée est que lorsqu'on lit la prochaine lettre du texte, t_{j+1}, on doit déterminer si cette lettre étend l'un des alignements exacts de l'étape précédente. Donc, si on avait $\boldsymbol{R_j}[i] = 1$ et que $p_{i+1} = t_{j+1}$, on a que $\boldsymbol{R_{j+1}}[i+1] = 1$. Si $\boldsymbol{R_j}[i] = 0$ alors, par définition $p_1 \dots p_i \neq t_{j-i+1} \dots t_j$ et donc, même si $p_{i+1} = t_{j+1}$, on n'a pas une occurrence exacte de $p_1 \dots p_{i+1}$ dans le texte se terminant en position $j+1$ et donc, dans ce cas, $\boldsymbol{R_{j+1}}[i+1] = 0$. Si $t_{j+1} = p_1$ alors on a $\boldsymbol{R_{j+1}}[1] = 1$. Finalement, à chaque position j du texte, si $\boldsymbol{R_j}[m] = 1$ alors il y a une occurrence exacte de P se terminant en position j du texte , et donc commençant en position $j - m + 1$ du texte.

On peut calculer le vecteur $\boldsymbol{R_{j+1}}$, à partir du vecteur $\boldsymbol{R_j}$, avec la récurrence suivante:

Définition 3.7 Soit $\boldsymbol{R_0}$ le vecteur de bits initial défini par $\boldsymbol{R_0}[i] = 0$, $\forall i,\ 1 \leq i \leq m$. Si on suppose que $\boldsymbol{R_j}[0] = 1$, $\forall j,\ 0 \leq j \leq n$, alors on a la transition suivante entre $\boldsymbol{R_j}$

et $\boldsymbol{R_{j+1}}$:

$$\boldsymbol{R_{j+1}}[i] = \begin{cases} 1 & \text{si } \boldsymbol{R_j}[i-1] = 1 \text{ et } p_i = t_{j+1} \\ 0 & \text{sinon} \end{cases}$$

Cette transition peut être calculée très simplement, de façon vectorielle.

Lemme 3.1 *On a que*

$$\boldsymbol{R_{j+1}} = \uparrow_1 \boldsymbol{R_j} \wedge \boldsymbol{t_{j+1}},$$

où $\boldsymbol{t_{j+1}}$ *représente le vecteur caractéristique de la lettre* t_{j+1} *dans le mot cherché* P. ■

Le lemme 3.1 nous permet donc de calculer les occurrences exactes d'un mot $P = p_1 \dots p_m$ dans un texte $T = t_1 \dots t_n$, en temps $\mathcal{O}(n)$, en calculant, de façon vectorielle, n vecteurs $\boldsymbol{R_j}$, $1 \leq j \leq n$. Si la longueur m du mot P est plus petite que la longueur d'un mot machine, alors la calcul de $\boldsymbol{R_{j+1}}$ à partir de $\boldsymbol{R_j}$ requière seulement deux opérations sur les vecteurs de bits (un déplacement vers la droite et une conjonction) et se fait donc en temps constant.

Exemple 3.7 Calculons les occurrences exactes du mot $P = TTA$ dans le texte $T = ACGTTACGTAAT$. Pour cela, on doit commencer par calculer les vecteurs caractéristiques de chacune des lettres, apparaissant dans le texte T, par rapport au mot P:

$$\boldsymbol{A} = 001, \qquad \boldsymbol{C} = 000, \qquad \boldsymbol{G} = 000, \qquad \boldsymbol{T} = 110.$$

Le vecteur $\boldsymbol{R_0}$= 000, par définition. Maintenant, on se sert du lemme 3.1 pour calculer les valeurs des vecteurs $\boldsymbol{R_j}$, $1 \leq j \leq n = 12$:

$$\boldsymbol{R_1} = \uparrow_1 \boldsymbol{R_0} \wedge \boldsymbol{A} = 100 \wedge 001 = 000,$$

$$\boldsymbol{R_2} = \uparrow_1 \boldsymbol{R_1} \wedge \boldsymbol{C} = 100 \wedge 000 = 000,$$

$$\vdots$$

On obtient la table de vecteurs suivantes, où $\boldsymbol{R_j}$ est sous le caractère t_j de T:

	A	C	G	T	T	A	C	G	T	A	A	T
T	0	0	0	1	1	0	0	0	1	0	0	1
T	0	0	0	0	1	0	0	0	0	0	0	0
A	0	0	0	0	0	**1**	0	0	0	0	0	0

Il y a une occurrence exacte de $P = p_1 \dots p_m$ dans T en position j si et seulement si $\boldsymbol{R_j}[m] = 1$. Ici, la seule occurrence exacte de $P = ATT$ dans le texte T se termine en position $j = 6$ du texte.

3.2.2 Recherche approximative

L'idée de Wu et Manber (Wu - Manber, 1992) est de généraliser l'algorithme de recherche exacte présenté dans la section précédente.

Pour chaque position j du texte, on calcule toujours le vecteur $\boldsymbol{R_j}$ de la même façon que précédemment, mais on calcule aussi t nouveaux vecteurs, où t est le nombre d'erreurs permis, $\boldsymbol{R_j^1}$, $\boldsymbol{R_j^2}$, ..., $\boldsymbol{R_j^t}$. Le vecteur $\boldsymbol{R_j^d}$ contient l'information sur les occurrences de préfixes de P, contenant au plus d erreurs, se terminant en position j du texte.

On veut pouvoir calculer le vecteur $\boldsymbol{R_{j+1}^d}$ à partir du vecteur $\boldsymbol{R_j^d}$. Pour ce faire, utilisons la remarque suivante:

Remarque 3.1 Il y a quatre façons différentes d'obtenir une occurrence de $p_1 \dots p_i$, contenant au plus d erreurs, se terminant en t_{j+1}:

1) Identité: il y a une occurrence de $p_1 \dots p_{i-1}$, contenant au plus d erreurs, se terminant en position j du texte et $t_{j+1} = p_i$.

2) Substitution: il y a une occurrence de $p_1 \dots p_{i-1}$, contenant au plus $d-1$ erreurs, se terminant en position j du texte et $t_{j+1} \neq p_i$.

3) Suppression de p_i: il y a une occurrence de $p_1 \dots p_{i-1}$, contenant au plus $d-1$ erreurs, se terminant en position $j+1$ du texte.

4) Insertion de t_{j+1}: il y a une occurrence de $p_1 \dots p_i$, contenant au plus $d-1$ erreurs, se terminant en position j du texte.

Dénotons les vecteurs $\boldsymbol{R_j}$ du calcul des occurrences exactes par $\boldsymbol{R_j^0}$. On a alors le résultat suivant:

Lemme 3.2 *Si* $\boldsymbol{R_0^d} = \overbrace{11\dots1}^{d}\overbrace{00\dots0}^{m-d}$, *alors*

$$\boldsymbol{R_{j+1}^d} = (\uparrow_1 \boldsymbol{R_j^d} \wedge \boldsymbol{t_{j+1}}) \vee \uparrow_1 (\boldsymbol{R_j^{d-1}} \vee \boldsymbol{R_{j+1}^{d-1}}) \vee \boldsymbol{R_j^{d-1}}$$

Preuve: Le lemme est seulement une traduction de la remarque 3.1 en vecteurs. En effet, on a

$$\boldsymbol{R_{j+1}^d} = \underbrace{(\uparrow_1 \boldsymbol{R_j^d} \wedge \boldsymbol{t_{j+1}})}_{1)} \vee \underbrace{(\uparrow_1 \boldsymbol{R_j^{d-1}})}_{2)} \vee \underbrace{(\uparrow_1 \boldsymbol{R_{j+1}^{d-1}})}_{3)} \vee \underbrace{\boldsymbol{R_j^{d-1}}}_{4)},$$

ce qui nous donne le résultat voulu. ■

Le calcul de chacun des vecteur $\boldsymbol{R_j^d}$, $1 \leq j \leq n$, requiert seulement 6 opérations sur des vecteurs de bits (2 déplacements vers la droite, 3 disjonctions et 1 conjonction) et peut donc être calculé en temps $\mathcal{O}(n)$. Comme d varie entre 0 et t, le calcul des occurrences approximatives de P, ayant au plus t erreurs, se fait en temps $\mathcal{O}(tn)$, si la longueur du mot P est plus petite que la longueur d'un mot machine.

Exemple 3.8 Reprenons l'exemple 3.7 et calculons cette fois les occurrences de P dans T ayant au plus 1 erreur. Les vecteurs $\boldsymbol{R_j^0}$ sont donnés par la table calculée à l'exemple 3.7. Pour calculer les valeurs des vecteurs $\boldsymbol{R_j^1}$, on se sert du lemme 3.2 et du vecteur initial $\boldsymbol{R_0^1}$= 100 et on obtient les vecteurs suivants:

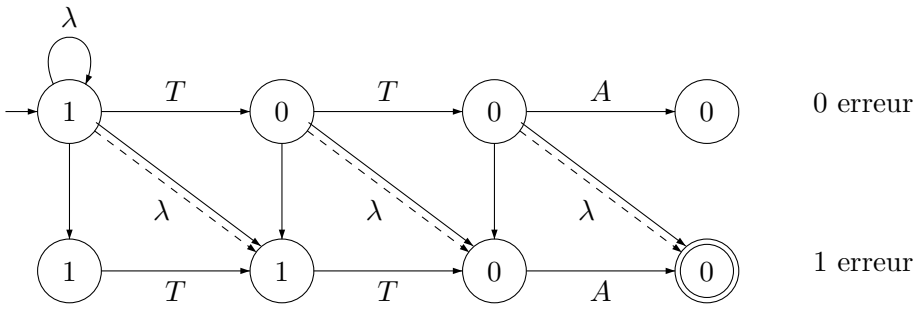
	A	C	G	T	T	A	C	G	T	A	A	T
T	1	1	1	1	1	1	1	1	1	1	1	1
T	0	0	0	1	1	1	0	0	1	1	0	1
A	0	0	0	0	**1**	**1**	**1**	0	0	**1**	**1**	0

On a donc des occurrences de $P = TTA$, ayant au plus 1 erreur, se terminant aux positions 5,6,7,10 et 11 du texte $T = ACGTTACGTAAT$. En fait, toutes ces occurrences contiennent exactement une erreur sauf celle se terminant en position 6 car, pour cette position, on a aussi que $\boldsymbol{R}_{\mathbf{6}}^{\mathbf{0}}[3] = 1$.

3.3 Algorithme de Baeza-Yates et Gonnet

En 1996, Baeza-Yates et Gonnet présentent un algorithme basé sur la simulation d'un automate fini non-déterministe, diagonale par diagonale (Baeza-Yates - Gonnet, 1996). Simuler l'automate par diagonales permet de trouver les nouveaux états actifs en parallèle, en utilisant des vecteurs de bits. Voyons, sur un exemple, comment leur algorithme fonctionne.

Exemple 3.9 Supposons que l'on veuille calculer les occurrences, ayant au plus 1 erreur, de $P = TTA$ dans le texte $T = ACGTTACGTAAT$, comme nous l'avons fait à l'exemple 3.8. On commence par construire l'automate non-déterministe suivant à partir de P:



Chaque ligne de l'automate dénote le nombre d'erreurs d'alignement et chaque colonne j équivaut à l'alignement de $p_1 \dots p_j$ avec un sous-mot du texte T. À chaque itération, l'automate change d'états, c'est-à-dire que son sous-ensemble d'états actifs est mis à jour. Si après la lecture de t_k l'état final est atteint alors il y a une occurrence de P, ayant au plus 1 erreur, se terminant en position k du texte. (Si on veut connaître exactement le nombre d'erreurs de l'occurrence, on prend comme états finaux tous les

états de la colonne m de l'automate. Si un ou plusieurs de ces états finaux sont actifs, après la lecture de t_k, on prend le plus petit indice i de ces états finaux actifs et on a une occurrence de P, ayant i erreurs, en position k du texte.)

Dans l'automate précédent, les transitions horizontales représentent l'identité entre un caractère du texte et une lettre de P. Les transitions verticales représentent l'insertion d'un caractère dans le mot P. Pour les transitions diagonales, les lignes solides représentent le remplacement d'un caractère par un autre et les lignes pointillées indiquent la suppression d'un caractère de P. Ces transitions sont des transitions λ puisqu'en les suivant on efface une lettre de P, sans avancer dans le texte T. La transition λ sur l'état initial permet de commencer l'alignement de P à n'importe quel endroit dans le texte T.

En général, l'automate a $(m+1)(t+1)$ états. On assigne le couple (i,j) à l'état situé à l'intersection de la ligne i et de la colonne j, pour $0 \leq i \leq t$ et $0 \leq j \leq m$. Au départ, les états actifs de la ligne i sont les états des colonnes de 0 à i pour indiquer le fait qu'on peut commencer par supprimer les i premiers caractères du mot P avant de continuer l'alignement. Les états actifs sont étiquetés par des 1 dans l'automate (Voir l'exemple 3.9).

Définition 3.8 Soit $\boldsymbol{A_k}$ la matrice booléenne, ou matrice de bits, correspondant aux états actifs de notre automate non-déterministe après la lecture du préfixe $t_1 \ldots t_k$ de T, définie par

$$A_k(i,j) = \begin{cases} 1 & \text{si l'état } (i,j) \text{ est actif, après la lecture de } t_1 \ldots t_k \\ 0 & \text{sinon} \end{cases} .$$

Exemple 3.10 La matrice de bits $\boldsymbol{A_0}$ correspondant aux états actifs, au départ, de l'automate de l'exemple 3.9 est la matrice suivante:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

À la lecture d'un nouveau caractère du texte, t_k, les valeurs de $\boldsymbol{A_k}$ peuvent être calculées à partir des valeurs de $\boldsymbol{A_{k-1}}$.

Théorème 3.1 *On a la relation suivante entre les matrices* $\boldsymbol{A_{k-1}}$ *et* $\boldsymbol{A_k}$*, pour* $1 \leq i \leq t$ *et* $1 \leq j \leq m$*:*

$$\begin{aligned} A_k[i,j] &= (A_{k-1}[i,j-1] \wedge id(k,j)) \vee A_{k-1}[i-1,j] \vee A_{k-1}[i-1,j-1] \\ &\quad \vee A_k[i-1,j-1], \end{aligned}$$

où $id(k,j) = 1$ *si* $t_k = p_j$ *et 0, sinon. Lorsque* $i = 0$*, alors*

$$A_k[0,j] = A_{k-1}[0,j-1] \wedge \delta(k,j).$$

Pour $j = 0$*, on a rien à calculer puisque les états de cette colonne sont toujours actifs.* ■

L'idée de Baeza-Yates et Gonnet est de calculer la matrice $\boldsymbol{A_k}$, diagonale par diagonale, à partir des valeurs de la matrice $\boldsymbol{A_{k-1}}$. En fait, ils ne calculent pas toute la matrice $\boldsymbol{A_k}$ mais seulement les éléments correspondants aux diagonales de longueur $t+1$ de l'automate, où t est le nombre d'erreurs permis dans la recherche, car seuls les états de ces diagonales peuvent influencer l'activation de l'état final (m,t). (Ce sont les diagonales commençant aux colonnes 0 à $m-t$.)

À cause des transitions λ, on a que si un état (i,j) est actif à un moment de la simulation alors, tous les états $(i+d, j+d)$, $d > 0$, appartenant à la même diagonale sont aussi actifs à ce même moment. On peut donc représenter chacune des diagonales j, $0 \leq j \leq m-t$, par un nombre $D_k[j]$ qui représente la ligne du premier état actif de la diagonale j, après la lecture de $t_1 \ldots t_k$. Si aucun état de la diagonale j est actif, alors on pose $D_k[j] = t+1$. On peut donc représenter la simulation de l'automate, à tout moment de la lecture de T, par $m-t+1$ nombres compris entre 0 et $t+1$.

Exemple 3.11 Au départ, l'automate de l'exemple 3.9 peut être représenté par le fait que le premier état actif sur la diagonale 0 est sur la ligne 0, c'est-à-dire $D_0[0] = 0$.

Comme aucun des états n'est actif, au départ, dans les diagonales 1 et 2 on a que $D_0[1] = 2$ et $D_0[2] = 2$.

On peut calculer les valeurs de $D_k[j]$, $0 \leq j \leq m - t$, à partir des valeurs $D_{k-1}[j]$ avec les équations suivantes, dérivées du théorème 3.1.

Théorème 3.2 *On a que*

$$\begin{aligned} D_k[0] &= 0 \\ D_k[j] &= \min(D_{k-1}[j] + 1, D_{k-1}[j+1] + 1, g(D_{k-1}[j-1], t_k)), \end{aligned}$$

où $g(D_j, c)$ *est défini comme*

$$g(D_k[j], c) = \min(\{t+1\} \cup \{i \mid i \geq D_k[j] \wedge p_{i+j} = c\}).$$

■

Avec ces $D_k[j]$, on a une occurrence de P, ayant au plus t erreurs, se terminant en position k du texte si $D_k[m-t] \leq t$.

Pour pouvoir calculer tous les $D_k[j]$ en parallèle, Baeza-Yates et Gonnet donnent une représentation de chaque valeur $D_k[j]$ par un vecteur de bits de longueur $t+1$:

$$\boldsymbol{D_k[j]} = \overbrace{0\ldots0}^{t+1-D_k[j]}\ \overbrace{1\ldots1}^{D_k[j]}.$$

Ils représentent ensuite l'état de la simulation de l'automate, après la lecture de $t_1 \ldots t_k$ par un seul grand vecteur de bits, comprenant les valeurs de D_j pour $1 \leq j \leq m - t$:

$$\boldsymbol{D_k} = 0\boldsymbol{D_k[1]}0\boldsymbol{D_k[2]}0\ldots0\boldsymbol{D_k[m-t]}.$$

L'implantation en C permettant de calculer $\boldsymbol{D_k}$ à partir de $\boldsymbol{D_{k-1}}$ est présenté dans leur article (Baeza-Yates - Gonnet, 1996). Avec cette notation en vecteurs de bits, on a une occurrence de P, ayant au plus t erreurs, se terminant en position k du texte si $\boldsymbol{D_k} \wedge (00^{t+1})^{m-t-1}010^t$ est le vecteur nul.

3.4 Algorithme de Myers

L'idée de Myers (Myers, 1999) repose sur le fait qu'on obtient le même résultat en calculant les différences horizontales et verticales entre les éléments de la table des distances D, pour le problème de la recherche des occurrences d'un mot $P = p_1 \dots p_m$, dans un texte $T = t_1 \dots t_n$, ayant au plus t erreurs, qu'en calculant la table D elle-même.

Définition 3.9 On définit $\Delta v_{i,j}$ comme étant la *différence verticale* entre les éléments de la table des distances situés en les positions (i, j) et $(i-1, j)$:

$$\Delta v_{i,j} = D[i, j] - D[i-1, j].$$

De la même façon, on définit $\Delta h_{i,j}$ comme étant la *différence horizontale* entre les éléments de la table des distances situés en les positions (i, j) et $(i, j-1)$:

$$\Delta h_{i,j} = D[i, j] - D[i, j-1].$$

Pour connaître les valeurs des éléments de la colonne j de la table des distances D, il suffit de connaître le vecteur de différences verticales associé à cette colonne, c'est-à-dire le vecteur $\boldsymbol{\Delta v_j} = \Delta v_{1,j} \dots \Delta v_{m,j}$, étant donné que $D(0, j) = 0$, $\forall j$.

Myers remplace donc le problème consistant à calculer la table des distances D par le problème consistant à calculer les vecteurs $\boldsymbol{\Delta v_j}$, pour $0 \leq j \leq n$. Pour trouver les positions des occurrences de P dans T on sait que, lorsqu'on travaille avec la table D, il y a une occurrence de P se terminant en position j de T, ayant t erreurs, si $D(m, j) = t$. Pour que le travail avec les vecteurs $\boldsymbol{\Delta v_j}$ soit efficace, il faut être aussi capable de calculer facilement les valeurs de $D(m, j)$ avec cette méthode. Pour ce faire, on maintient un score, $score_j$, lors du calcul de la table des $\boldsymbol{\Delta v_j}$, tel que $score_j = D(m, j)$. Une façon simple de maintenir le score est de poser comme valeur initiale $score_0 = m$, car $D[m, 0] = m$. Ensuite, on a, par définition que

$$score_j = score_{j-1} + \Delta h_{m,j}.$$

Nous verrons, dans ce qui suit, que lorsqu'on possède les valeurs des différences verticales, les différences horizontales sont faciles à calculer et donc le maintient du score ne demandera pas beaucoup de travail.

Le but de l'algorithme de Myers est de calculer, si $m < w$, où w est la longueur d'un mot machine, les valeurs successives de $\boldsymbol{\Delta v_j}$, pour $0 \leq j \leq n$, en temps constant, en utilisant des opérations sur les vecteurs de bits.

La première étape est donc de trouver une représentation, par des vecteurs de bits, pour les vecteurs de différences $\boldsymbol{\Delta v_j}$, dont les éléments sont dans l'ensemble $\{-1, 0, 1\}$. Myers se sert de 2 vecteurs de bits pour représenter $\boldsymbol{\Delta v_j}$: $\boldsymbol{Pv_j}$ et $\boldsymbol{Mv_j}$ qui sont, respectivement, les vecteurs caractéristiques de la valeur 1 et -1 dans le vecteur $\boldsymbol{\Delta v_j}$. Le vecteur caractéristique de la valeur 0 dans $\boldsymbol{\Delta v_j}$ est alors donné par $\boldsymbol{\neg(Pv_j \vee Mv_j)}$.

Exemple 3.12 Voici la table des distances pour le problème des occurrences approximatives de $P = AACG$ dans $T = GCGTTGCAGGAACG$:

	λ	G	C	G	T	T	G	C	A	G	G	A	A	C	G
λ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1
A	2	2	2	2	2	2	2	2	1	1	2	1	0	1	2
C	3	3	2	3	3	3	3	2	2	2	2	2	1	0	1
G	4	3	3	2	3	4	3	3	3	2	2	3	2	1	0

La table des vecteurs $\boldsymbol{\Delta v_j}$ associés à ce problème est:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
1	1	1	1	1	1	1	1	0	1	1	0	0	1	1
1	1	1	1	1	1	1	1	1	0	1	1	0	0	1
1	1	0	1	1	1	1	0	1	1	0	1	1	-1	-1
1	0	1	-1	0	1	0	1	1	0	0	1	1	1	-1

Le vecteur $\boldsymbol{\Delta v_{13}}$= 10−11 est représenté par deux vecteurs de bits. Le vecteur $\boldsymbol{Pv_{13}}$ est le vecteur caractéristique de la valeur 1 dans $\boldsymbol{\Delta v_{13}}$ et donc

$$\boldsymbol{Pv_{13}} = 1001,$$

et le vecteur $\boldsymbol{Mv_{13}}$ qui est le vecteur caractéristique de la valeur -1 dans $\boldsymbol{\Delta v_{13}}$:

$$\boldsymbol{Mv_{13}} = 0010.$$

Le vecteur caractéristique de la valeur 0 dans $\boldsymbol{\Delta v_{13}}$ est

$$\neg(\boldsymbol{Pv_{13}} \vee \boldsymbol{Mv_{13}}) = \neg(1001 \vee 0010) = 0100.$$

Maintenant, regardons comment on peut calculer la colonne $\boldsymbol{\Delta v_j}$ à partir de la colonne $\boldsymbol{\Delta v_{j-1}}$. Pour ce faire, considérons la cellule suivante, où $Eq_{i,j} = 1$ si $p_i = t_j$ et 0 sinon:

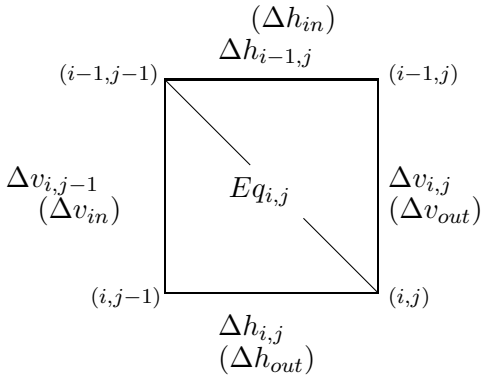


Figure 3.1 Différences horizontales et verticales d'une cellule

Théorème 3.3 *On a que*

$$\Delta v_{i,j} = \min\{-Eq_{i,j}, \Delta v_{i,j-1}, \Delta h_{i-1,j}\} + (1 - \Delta h_{i-1,j})$$

et que

$$\Delta h_{i,j} = \min\{-Eq_{i,j}, \Delta v_{i,j-1}, \Delta h_{i-1,j}\} + (1 - \Delta v_{i,j-1}).$$

Preuve: On a qu'à se servir de la définition de $\Delta v_{i,j}$ et de la récurrence sur les $D[i,j]$. (Voir proposition 1.5) ■

On peut voir $\Delta v_{i,j-1}$, $\Delta h_{i-1,j}$ et $Eq_{i,j}$ comme étant les entrées d'une cellule et $\Delta v_{i,j}$ et $\Delta h_{i,j}$ comme étant les sorties de la cellule. (Voir figure 3.1)

Comme Δv_{in} et Δh_{in} peuvent prendre seulement les valeurs -1, 0 ou 1 et que $Eq = 0$ ou 1, il n'y a que 18 entrées différentes possibles pour une cellule. L'étude des différents cas d'entrée nous donne le théorème suivant:

Théorème 3.4 *On a, pour les différences verticales, les formules suivantes:*

$$\begin{aligned} Xv &= Eq \vee Mv_{in} \\ Pv_{out} &= Mh_{in} \vee \neg(Xv \vee Ph_{in}) \\ Mv_{out} &= Ph_{in} \wedge Xv \end{aligned} \tag{3.1}$$

De même, pour les différences horizontales, on a les formules symétriques suivantes:

$$\begin{aligned} Xh &= Eq \vee Mh_{in} \\ Ph_{out} &= Mv_{in} \vee \neg(Xh \vee Pv_{in}) \\ Mh_{out} &= Pv_{in} \wedge Xh \end{aligned} \tag{3.2}$$

■

Toutes ces formules logiques peuvent être calculées facilement mais, pour ce faire, on doit connaître les valeurs de $Eq_{i,j}$ pour chacune des cellules (i,j). On va précalculer, pour chaque lettre a de l'alphabet considéré, le vecteur caractéristique, $\boldsymbol{a}$, de cette lettre par rapport au mot P cherché. Le vecteur $\boldsymbol{Eq_j}$ est alors, tout simplement, le vecteur caractéristique $\boldsymbol{t_j}$ de la lettre t_j du texte T.

On veut calculer $score_j$ et les vecteurs de bits $\boldsymbol{Pv_j}$ et $\boldsymbol{Mv_j}$, qui encodent $\boldsymbol{\Delta v_j}$, étant donné l'information provenant de la colonne $j-1$ et la lettre t_j du texte. Au départ,

on a que

$$\begin{aligned} \boldsymbol{Pv_0}(i) &= 1, \quad \forall i \\ \boldsymbol{Mv_0}(i) &= 0, \quad \forall i \\ score_0 &= m. \end{aligned}$$

Cela vient du fait que la première colonne de la table des distances est $012 \ldots m$. Pour trouver $\boldsymbol{\Delta v_j}$, on procède alors comme suit:

1. Premièrement, les différences verticales de la colonne $j-1$, $\boldsymbol{\Delta v_{j-1}}$, sont utilisées pour calculer les différences horizontales en bas de leur cellule respective, en utilisant les formules 3.2. On a donc que

$$\begin{aligned} \boldsymbol{Ph_j}(i) &= \boldsymbol{Mv_{j-1}}(i) \vee \neg(\boldsymbol{Xh_j}(i) \vee \boldsymbol{Pv_{j-1}}(i)) \\ \boldsymbol{Mh_j}(i) &= \boldsymbol{Pv_{j-1}}(i) \wedge \boldsymbol{Xh_j}(i) \end{aligned}$$

2. On modifie le score de la façon suivante:

$$score_j = score_{j-1} + \begin{cases} 1 & \text{si } \boldsymbol{Ph_j}(m) = 1 \\ -1 & \text{si } \boldsymbol{Mh_j}(m) = 1 \\ 0 & \text{sinon} \end{cases}$$

3. On utilise alors chacune des différences horizontales, calculées à l'étape 1, dans la cellule au-dessous pour calculer les différences verticales de la colonne j, en utilisant les formules 3.1. Cela nous donne les équations suivantes:

$$\begin{aligned} \boldsymbol{Ph_j}(0) &= \boldsymbol{Mv_j}(0) = 0 \\ \boldsymbol{Pv_j}(i) &= \boldsymbol{Mh_j}(i-1) \vee \neg(\boldsymbol{Xv_j}(i) \vee \boldsymbol{Ph_j}(i-1)) \\ \boldsymbol{Mv_j}(i) &= \boldsymbol{Ph_j}(i-1) \wedge \boldsymbol{Xv_j}(i) \end{aligned}$$

Si on connaît la valeur des vecteurs $\boldsymbol{Xh_j}$ et $\boldsymbol{Xv_j}$, toutes les formules précédentes peuvent être calculées en temps constant, en utilisant des opérations sur les vecteurs de bits. Il faut maintenant voir comment on peut aussi arriver à calculer les vecteurs $\boldsymbol{Xh_j}$ et $\boldsymbol{Xv_j}$ en temps constant. De la définition de ces vecteurs on a que

Théorème 3.5

$$\begin{aligned} \boldsymbol{Xv_j}(i) &= \boldsymbol{t_j}(i) \vee \boldsymbol{Mv_{j-1}}(i) \\ \boldsymbol{Xh_j}(i) &= \boldsymbol{t_j}(i) \vee \boldsymbol{Mh_j}(i-1) \end{aligned}$$

■

Lors du calcul des différences verticales de la colonne j, calculer $\boldsymbol{Xv_j}$ ne pose pas de problème étant donné qu'on a précalculé tous les vecteurs caractéristiques $\boldsymbol{t_j}$ et que le vecteur $\boldsymbol{Mv_{j-1}}$ est connu de l'étape de calcul précédente. Par contre, pour calculer le vecteur $\boldsymbol{Xh_j}$, il nous faut la valeur du vecteur $\boldsymbol{Mh_j}$ qui, à son tour, requiert la valeur du vecteur $\boldsymbol{Xh_j}$...

En utilisant l'addition binaire avec retenue et la disjonction exclusive, dénoté par $\oplus$, Myers dérive la formule suivante, permettant un calcul vectoriel du vecteur $\mathbf{Xh_j}$ en temps constant.

$$\boldsymbol{Xh_j} = (((\boldsymbol{t_j} \wedge \boldsymbol{Pv_{j-1}}) +_{\boldsymbol{b}} \boldsymbol{Pv_{j-1}}) \oplus \boldsymbol{Pv_{j-1}}) \vee \boldsymbol{t_j} \ .$$

Donc, lorsque la longueur du mot P cherché est plus petite que la longueur d'un mot machine, Myers a développé un algorithme vectoriel permettant le calcul des occurrences approximatives de P dans T en temps $\mathcal{O}(n)$. Le pseudocode, en C, de l'algorithme est disponible dans son article (Myers, 1999).

Dans le prochain chapitre, nous allons généraliser les idées de Myers en développant un algorithme vectoriel, pour le problème de la recherche approximative, utilisant une distance d'édition généralisée.

CHAPITRE IV

AUTOMATES RÉSOLUBLES ET ALGORITHMES VECTORIELS

Dans ce chapitre, nous allons définir une nouvelle classe d'automates de Moore, les automates *résolubles*, pour laquelle on peut déduire automatiquement un algorithme vectoriel à partir de la table de transition de l'automate. Nous allons ensuite appliquer nos résultats à la construction d'algorithmes vectoriels efficaces. Nous en déduirons un algorithme vectoriel pour le problème de la recherche des occurrences approximatives d'un mot dans un texte, lorsque la distance utilisée est une distance d'édition généralisée.

4.1 Introduction

Étant donné un automate fini déterministe et une séquence d'entrée $e_1 e_2 \dots e_m$, on s'intéresse à la sortie $r_1 r_2 \dots r_m$ des états visités par l'automate lors de la lecture de l'entrée. Étant donné que l'exécution d'une transition est une opération qui requiert un temps constant, la séquence de sortie est obtenue en temps $\mathcal{O}(m)$.

Une façon d'améliorer le temps de calcul est d'exploiter le parallélisme des opérations vectorielles et/ou des opérations sur les vecteurs de bits. Par exemple, on a vu que dans (Wu - Manber, 1992) et (Baeza-Yates - Gonnet, 1996), des vecteurs de bits sont utilisés pour coder l'ensemble des états d'un automate non-déterministe. Une autre approche, développée par Myers (Myers, 1999), utilise des vecteurs de bits pour coder les séquences d'entrée et de sortie d'un automate. C'est cette technique que nous avons utilisée pour développer notre algorithme de recherche. Dans ce qui suit, nous verrons la théorie nécessaire à la compréhension de notre algorithme vectoriel.

4.2 Automates de Moore et algorithmes vectoriels

Dans cette section nous définissons une nouvelle classe d'automates, les *automates résolubles* pour laquelle on peut construire des algorithmes vectoriels. Nous travaillerons avec des automates finis déterministes, sans état de sortie. En fait, la sortie, étant donné un caractère d'entrée, sera uniquement déterminée par l'état où on arrive après la lecture de ce caractère. Ce genre d'automate est appelé *automate de Moore* (Hopcroft - Ullman, 1979).

Exemple 4.1 Voici un automate de Moore:

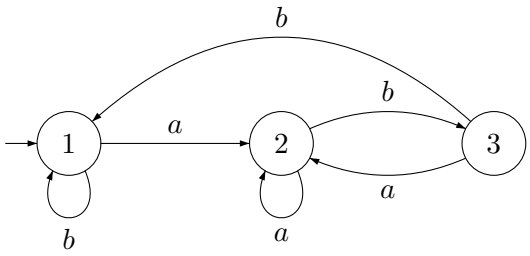


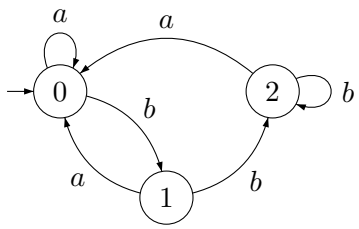
Figure 4.1 Un automate de Moore

Étant donné la séquence d'entrée $\boldsymbol{e} = aababba$, l'automate de la Figure 4.1 produira la séquence de sortie 2232312, qui est la suite des états visités par l'automate lors de la lecture de la séquence d'entrée.

4.2.1 Un exemple simple

Voyons maintenant, sur un exemple simple, comment associer un algorithme vectoriel à un automate de Moore.

Considérons l'automate de Moore suivant:



Soit $\boldsymbol{e} = e_1 \dots e_m$, le vecteur d'entrée de l'automate et $\boldsymbol{r} = r_1 \dots r_m$, le vecteur de sortie. On a les calculs vectoriels *directs* suivants, où $\boldsymbol{(r = k)}$ représente le vecteur booléen "l'automate est dans l'état k":

$$\boldsymbol{(r = 0)} = \boldsymbol{a}, \tag{4.1}$$

$$\boldsymbol{(r = 1)} = \boldsymbol{\uparrow_1(r = 0) \wedge b}, \tag{4.2}$$

$$\boldsymbol{(r = 2)} = \boldsymbol{\neg((r = 0) \vee (r = 1))}, \tag{4.3}$$

L'équation 4.1 nous dit simplement que l'état suivant la lecture d'un a dans l'entrée, peu importe où l'on se trouve dans l'automate, sera toujours l'état 0. La seule transition se rendant dans l'état 1 de l'automate est une transition d'étiquette b partant de l'état 0. Ce fait implique l'équation 4.2. Finalement, on est dans l'état 2 de l'automate si on n'est ni dans l'état 0, ni dans l'état 1, d'où l'équation 4.3.

Exemple 4.2 Voici un exemple d'utilisation de ces trois équations: supposons que la séquence d'entrée donnée à l'automate est $\boldsymbol{e} = babba$. Alors les vecteurs caractéristiques des lettres a et b sont:

$$\boldsymbol{a} = 01001$$

$$\boldsymbol{b} = 10110$$

L'équation 4.1 nous dit que la sortie est 0 si et seulement si l'entrée est a, alors:

$$\boldsymbol{(r = 0)} = \boldsymbol{a} = 01001$$

L'équation 4.2 nous dit que la sortie est 1 si le caractère d'entrée est b, et l'état précédent

0:

$$\begin{aligned}(\boldsymbol{r=1}) &= \uparrow_1 (\boldsymbol{r=0}) \wedge \boldsymbol{b} \\ &= 10100 \wedge 10110 \\ &= 10100\end{aligned}$$

Dans tous les autres cas, l'état de sortie est 2, et on a:

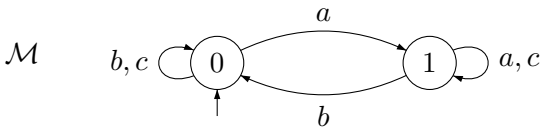
$$\begin{aligned}(\boldsymbol{r=2}) &= \neg(01001 \vee 10100) \\ &= 00010\end{aligned}$$

Si on suppose que les opérations vectorielles sont exécutées en parallèle alors, peu importe la longueur de notre vecteur d'entrée, le vecteur de sortie peut être calculé avec 6 opérations vectorielles: 2 assignations, une conjonction, une disjonction, un déplacement vers la droite et une négation.

Dans l'exemple simple de cette section, l'état de sortie dépend d'au plus 2 caractères de l'entrée. En général, les états de sortie d'un automate vont dépendre d'événements arbitrairement loin mais, dans quelques cas intéressants, il sera encore possible de réduire le calcul du vecteur de sortie à un calcul direct d'opérations sur les vecteurs de bits. C'est le sujet de la prochaine section.

4.2.2 Une mémoire des événements passés

L'exemple le plus simple d'un calcul influencé par des événements passés est donné par l'automate $\mathcal{M}$ suivant:



Avec cet automate, le fait que le caractère d'entrée c donne un 0 ou un 1 dans le vecteur de sortie peut dépendre d'un événement s'étant produit très longtemps avant dans le vecteur d'entrée, comme par exemple dans le cas des séquences d'entrées ac^n et bc^n.

Dans ce cas particulier, heureusement, le lemme suivant nous aidera à trouver facilement le vecteur de sortie de l'automate $\mathcal{M}$.

Lemme 4.1 Le lemme de l'addition*: Le vecteur de sortie* $\boldsymbol{r}$ *de l'automate* $\mathcal{M}$ *peut être décrit par les deux vecteurs suivants:*

$$(\boldsymbol{r = 0}) \;=\; \boldsymbol{b} \vee [\boldsymbol{c} \wedge (\neg \boldsymbol{b} +_{\boldsymbol{b}} \neg(\boldsymbol{b} \vee \boldsymbol{c}))] \tag{4.4}$$

$$(\boldsymbol{r = 1}) \;=\; \boldsymbol{a} \vee [\boldsymbol{c} \wedge \neg(\boldsymbol{a} +_{\boldsymbol{b}} (\boldsymbol{a} \vee \boldsymbol{c}))]. \tag{4.5}$$

Preuve: Comme Myers le note dans (Myers, 1999), l'automate $\mathcal{M}$ est semblable à l'automate de Moore classique pour l'addition binaire:

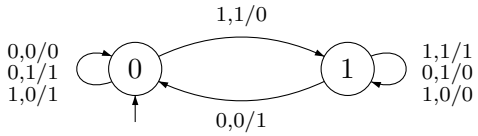


Figure 4.2 Automate d'addition binaire

Dans cet automate, l'état i signifie que la retenue est i, l'élément à gauche du symbole '/' dans une transition représente les bits des 2 vecteurs additionnés en une position fixée, et l'élément à droite du symbole '/' est la somme des deux bits avec la retenue courante.

On atteint l'état 1 dans les deux cas suivants:

1. Les deux bits à additionner sont 1.

2. Les deux bits à additionner sont différents, et leur somme (avec retenue) est 0.

Donc, si deux vecteurs de bits $\boldsymbol{x_1}$ et $\boldsymbol{x_2}$ sont additionnés de gauche à droite utilisant l'addition binaire avec retenue, la retenue est 1 lorsque

$$(\boldsymbol{x_1} \wedge \boldsymbol{x_2}) \vee [((\neg \boldsymbol{x_1} \wedge \boldsymbol{x_2}) \vee (\boldsymbol{x_1} \wedge \neg \boldsymbol{x_2})) \wedge \neg(\boldsymbol{x_1} +_{\boldsymbol{b}} \boldsymbol{x_2})].$$

(Cette équation est seulement la traduction des conditions 1) et 2) précédentes.)

Maintenant, si on définit $\boldsymbol{x_1} = \boldsymbol{a}$ et $\boldsymbol{x_2} = \boldsymbol{a} \vee \boldsymbol{c}$, alors

$$\begin{aligned} \boldsymbol{a} &= \boldsymbol{x_1} \wedge \boldsymbol{x_2} \\ \boldsymbol{b} &= \neg \boldsymbol{x_1} \wedge \neg \boldsymbol{x_2} \\ \boldsymbol{c} &= \neg \boldsymbol{x_1} \wedge \boldsymbol{x_2} \end{aligned}$$

Avec ces identités, l'automate $\mathcal{M}$ devient un sous-automate de l'automate d'addition binaire. Comme la formule $\boldsymbol{x_1} \wedge \neg \boldsymbol{x_2}$ est toujours fausse, on obtient, par substitution, la formule pour $\boldsymbol{(r = 1)}$. La formule pour $\boldsymbol{(r = 0)}$ est obtenue de façon similaire en définissant $\boldsymbol{x_1} = \neg \boldsymbol{b}$ et $\boldsymbol{x_2} = \neg(\boldsymbol{b} \vee \boldsymbol{c})$. ■

En fait, l'équation 4.5 du lemme de l'addition nous dit que l'on est dans l'état 1 de l'automate $\mathcal{M}$ si on a un a dans le vecteur d'entrée ou un c, et que ce c est tel que toutes les entrées le précédant, jusqu'à une occurrence de a, sont aussi des c.

4.2.3 Automates résolubles ⇒ algorithmes vectoriels

Dans cette section, nous établirons une condition suffisante à l'existence d'un algorithme vectoriel pour un automate en dégageant une classe d'automates pour laquelle un algorithme vectoriel peut être automatiquement déduit de la table de transition d'un automate.

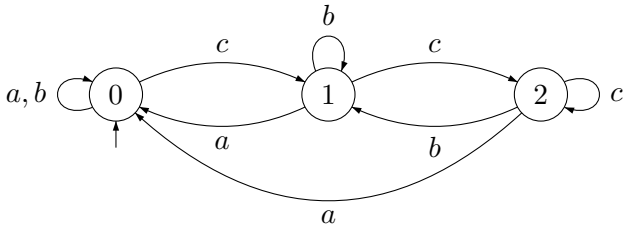
Définition 4.1 Soit $\mathcal{A} = (\Sigma, Q, \sigma, i, F)$ un automate complet déterministe. Un état s de $\mathcal{A}$ est dit *résoluble* si et seulement si:

$$\forall x \in \Sigma,\ [\exists s' \neq s \text{ tel que } \sigma(s', x) = s] \Rightarrow [\forall q \in Q, \sigma(q, x) = s].$$

L'événement x est alors appelé un *annulateur* car la fonction de transition de l'automate, restreinte à cet événement, est constante.

L'ensemble des annulateurs de l'état résoluble s est appelé *l'ensemble indicateur* de s et est noté I_s.

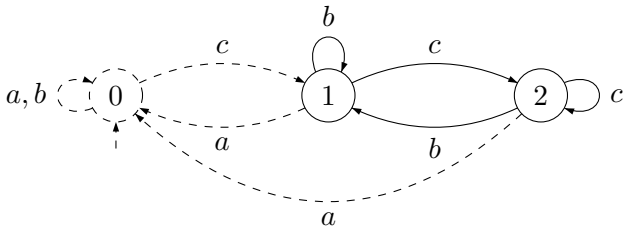
Exemple 4.3 Soit l'automate $\mathcal{A}$ suivant:



Dans cet automate, l'état 0 est résoluble et $I_0 = \{a\}$.

Un état résoluble s peut être *enlevé* d'un automate dans le sens suivant. Soit I_s l'ensemble indicateur de l'état s et $\mathcal{A} \setminus \{s\}$ l'automate obtenu de $\mathcal{A}$ en lui enlevant l'état s et ses transitions adjacentes. Comme s est résoluble, $\mathcal{A} \setminus \{s\}$ est aussi un automate complet sur l'alphabet $\Sigma \setminus I_s$. En effet, on a que $\sigma(r, e) \neq s$ si $e \notin I_s$.

Exemple 4.4 Reprenons l'automate $\mathcal{A}$ de l'exemple précédent:

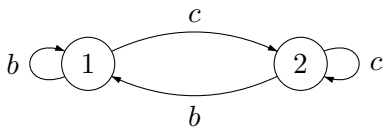


Si on enlève l'état 0 et ses transitions adjacentes, on voit que l'automate $\mathcal{A} \setminus \{0\}$, en lignes pleines, est complet sur l'alphabet $\Sigma \setminus \{a\} = \{b, c\}$.

Définition 4.2 Un automate $\mathcal{A}$ est *résoluble* s'il possède seulement un état, ou s'il a un état résoluble s et que l'automate $\mathcal{A} \setminus \{s\}$ est résoluble.

Lorsqu'un automate $\mathcal{A}$, à d états, est résoluble, il y a un ordre, non-unique, induit sur ces états commençant par le premier état résoluble, puis le suivant et ainsi de suite. On peut alors renommer les états de $\mathcal{A}$ et supposer que $Q = \{0, 1, \ldots, d-1\}$.

Exemple 4.5 L'automate $\mathcal{A}$ de l'exemple précédent est résoluble puisqu'il possède l'état résoluble 0 et que l'automate $\mathcal{A} \setminus \{0\}$:



est clairement résoluble.

On va maintenant montrer que lorsqu'un état s est résoluble, on peut facilement trouver les positions du vecteur de sortie de l'automate correspondant à cet état, c'est-à-dire le vecteur $(\boldsymbol{r = s})$, avec l'aide du lemme de l'addition 4.1.

Pour ce faire, définissons la fonction $Ind(s, \boldsymbol{X}, \boldsymbol{Y})$ de la façon suivante:

Définition 4.3

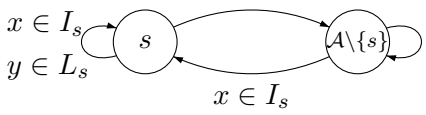
$$Ind(s, \boldsymbol{X}, \boldsymbol{Y}) = \begin{cases} \boldsymbol{X} \vee [\boldsymbol{Y} \wedge (\neg \boldsymbol{X} +_{\boldsymbol{b}} \neg(\boldsymbol{X} \vee \boldsymbol{Y}))] \text{ si } s \text{ est l'état initial} \\ \boldsymbol{X} \vee [\boldsymbol{Y} \wedge \neg(\boldsymbol{X} +_{\boldsymbol{b}} (\boldsymbol{X} \vee \boldsymbol{Y}))] \text{ sinon.} \end{cases}$$

Cette fonction comprend les 2 cas du lemme de l'addition 4.1, au sens suivant. Considérons $\boldsymbol{X}$ comme étant le vecteur caractéristique des événements entrant dans l'état s et $\boldsymbol{Y}$, comme étant celui des événements bouclant sur s mais n'appartenant pas à $\boldsymbol{X}$. La définition découle alors tout simplement des 2 formules trouvées, pour le cas où s est initial ou non, dans le lemme de l'addition 4.1.

Proposition 4.1 *Si s est un état résoluble, et si L_s est l'ensemble des événements bouclant sur l'état s mais n'appartenant pas à l'ensemble I_s, alors*

$$\boldsymbol{(r = s)} = Ind(s, \boldsymbol{e \in I_s}, \boldsymbol{e \in L_s}).$$

Preuve: Si s est un état résoluble dans un automate $\mathcal{A}$, alors, par définition de résolubilité, on peut représenter l'automate $\mathcal{A}$ de la façon compacte suivante:



L'automate $\mathcal{A}$ sera donc dans l'état s dans les deux cas suivants:

1. On est dans un état quelconque de $\mathcal{A}$, on lit l'événement x et cet événement est un annulateur de l'état s, c'est-à-dire que $x \in I_s$.

2. On est dans l'état s et on lit un événement bouclant y, tel que $y \notin I_s$, c'est-à-dire que $y \in L_s$.

De ce point de vue, il y a donc une équivalence entre l'automate $\mathcal{A}$ et l'automate $\mathcal{M}$ du lemme de l'addition 4.1 et donc, si s est l'état initial, la formule pour $(\boldsymbol{r = s})$ est la formule pour l'état 0 de l'automate $\mathcal{M}$, dans laquelle on substitue le vecteur $\boldsymbol{b}$ par le vecteur $\boldsymbol{e \in I_s}$, et le vecteur $\boldsymbol{c}$ par le vecteur $\boldsymbol{e \in L_s}$. Si l'état s n'est pas initial, on utilise la formule pour l'état 1, dans laquelle on substitue le vecteur $\boldsymbol{a}$ par le vecteur $\boldsymbol{e \in I_s}$, et le vecteur $\boldsymbol{c}$ par le vecteur $\boldsymbol{e \in L_s}$. ■

Théorème 4.1 *Si un automate $\mathcal{A}$ est résoluble, alors il existe un algorithme vectoriel pour $\mathcal{A}$.*

Preuve: Soit $Q = \{0, 1, \ldots, d-1\}$ un ordre sur les états de $\mathcal{A}$ tel que l'état k est résoluble dans l'automate $\mathcal{A} \setminus \{0, 1, \ldots, k-1\}$, et soit i l'état initial de $\mathcal{A}$.

Étant donné une séquence d'entrée $\boldsymbol{e} = e_1 e_2 \ldots e_m$, nous allons trouver le vecteur de sortie $\boldsymbol{r} = r_1 r_2 \ldots r_m$ en calculant récursivement les vecteurs de bits $(\boldsymbol{r = k})$ pour k dans $\{0, 1, \ldots, d-1\}$.

Le cas $k = 0$ est le plus simple. En effet, comme l'état 0 est résoluble dans $\mathcal{A}$ alors, par la proposition 4.1:

$$(\boldsymbol{r = 0}) = Ind(0, \boldsymbol{e \in I_0}, \boldsymbol{e \in L_0}).$$

Supposons maintenant que le vecteur $(\boldsymbol{r < k})$ est connu. Pour calculer le vecteur $(\boldsymbol{r = k})$, on doit considérer deux cas: soit l'état précédent s est plus petit que k, soit il est plus grand.

Comme $(\boldsymbol{r < k})$ est connu, on connaît toutes les positions dans le vecteur $\boldsymbol{r}$ pour lesquelles l'état précédent s est plus petit que k, ces positions peuvent être calculées par l'expression $(\uparrow_{\boldsymbol{i}} \boldsymbol{r < k})$. On peut maintenant appliquer la table de transition σ de l'automate en ces positions et regarder si le résultat est k. Pour résumer, on a que l'état précédent est plus petit que k et le résultat est k si et seulement si:

$$(\uparrow_{\boldsymbol{i}} \boldsymbol{r < k}) \wedge (\boldsymbol{\sigma}(\uparrow_{\boldsymbol{i}} \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k}). \tag{4.6}$$

Maintenant, si $s \geq k$, comme l'état k est résoluble dans $\mathcal{A} \setminus \{0, 1, \ldots, k-1\}$, on peut définir l'ensemble indicateur I_k de l'état k:

$$\text{Si } e \in I_k \text{ alors } \forall r \in \{k, \ldots, d-1\} \text{ on a que } \sigma(r, e) = k.$$

Par l'hypothèse de résolubilité, on ne peut atteindre l'état k d'un état $s > k$ que par ces événements, et on boucle si $s = k$. Donc, le vecteur suivant couvre au moins toutes les possibilités d'atteindre k d'un état $s > k$:

$$(\boldsymbol{r \geq k}) \wedge (\boldsymbol{e \in I_k}). \tag{4.7}$$

En combinant les équations 4.6 et 4.7, on obtient le vecteur suivant qui couvre au moins toutes les possibilités d'atteindre k, en partant d'un état différent de k:

$$\boldsymbol{N_k} = [(\uparrow_{\boldsymbol{i}} \boldsymbol{r < k}) \wedge (\boldsymbol{\sigma}(\uparrow_{\boldsymbol{i}} \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k})] \vee [(\boldsymbol{r \geq k}) \wedge (\boldsymbol{e \in I_k})]. \tag{4.8}$$

Finalement, si L_k est l'ensemble des événements bouclant sur k mais n'appartenant pas à l'ensemble I_k, on a:

$$(\boldsymbol{r = k}) = Ind(k, \boldsymbol{N_k}, \boldsymbol{e \in L_k}).$$

■

Voici une présentation en pseudo-code de l'algorithme présenté dans le théorème précédent. Commençons par définir le vecteur $\boldsymbol{K}$ comme étant $(\boldsymbol{r < k})$, et notons que, comme $(\uparrow_{\boldsymbol{i}} \boldsymbol{r < k}) = \uparrow_{\boldsymbol{i<k}} (\boldsymbol{r < k})$, la valeur de $(\uparrow_{\boldsymbol{i}} \boldsymbol{r < k})$ peut être obtenue par le vecteur $\uparrow_{\boldsymbol{i<k}} \boldsymbol{K}$.

On commence par calculer $(\boldsymbol{r = 0})$, puis on initialise $\boldsymbol{K}$ avec cette valeur:

$$\begin{aligned} (\boldsymbol{r = 0}) &\leftarrow Ind(0, \boldsymbol{e \in I_0}, \boldsymbol{e \in L_0}) \\ \boldsymbol{K} &\leftarrow (\boldsymbol{r = 0}) \end{aligned}$$

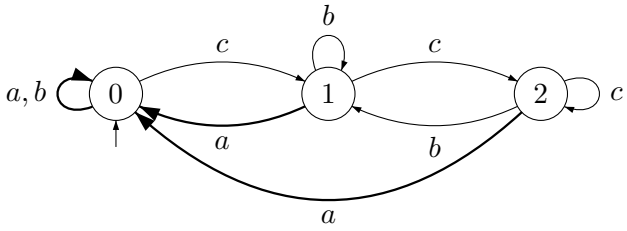
Maintenant, pour les valeurs successives de k dans $\{1, \ldots, d-2\}$, on exécute les 3 instructions suivantes:

$$\begin{aligned} \boldsymbol{N} &\leftarrow [(\uparrow_{\boldsymbol{i<k}} \boldsymbol{K}) \wedge (\boldsymbol{\sigma}(\uparrow_{\boldsymbol{i}} \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k})] \vee [\neg \boldsymbol{K} \wedge (\boldsymbol{e \in I_k})] \\ (\boldsymbol{r = k}) &\leftarrow Ind(k, \boldsymbol{N}, \boldsymbol{e \in L_k}) \\ \boldsymbol{K} &\leftarrow \boldsymbol{K} \vee (\boldsymbol{r = k}) \end{aligned}$$

La valeur de $(\boldsymbol{r = d-1})$ est finalement calculée en prenant la négation du dernier vecteur $\boldsymbol{K}$. Voyons maintenant un exemple concret de l'exécution de l'algorithme.

Exemple 4.6 Soit $\mathcal{A}$ l'automate de l'exemple 4.3 et $\boldsymbol{e} = abbcbaccb$ une séquence d'entrée. (Remarque: Les états de l'automate $\mathcal{A}$ sont déjà dans l'ordre de résolubilité.)

La première étape consiste à trouver les positions de la séquence de sortie où il y a des 0. Pour ce faire on regarde toutes les transitions entrant dans l'état 0



c'est-à-dire qu'on calcule le vecteur

$$(\boldsymbol{r = 0}) \leftarrow Ind(0, \boldsymbol{e \in I_0}, \boldsymbol{e \in L_0})$$

Ici, $I_0 = \{a\}$, $L_0 = \{b\}$ et donc $\boldsymbol{e \in I_0}$ est le vecteur caractéristique $\boldsymbol{a}$ de l'événement a et $\boldsymbol{e \in L_0}$ est le vecteur caractéristique $\boldsymbol{b}$ de l'événement b:

$$\begin{aligned} \boldsymbol{e \in I_0} &= \boldsymbol{a} = 100001000, \\ \boldsymbol{e \in L_0} &= \boldsymbol{b} = 011010001 \end{aligned}$$

Comme 0 est l'état initial de l'automate, on a, par la définition 4.3 que

$$Ind(0, \boldsymbol{e} \in \boldsymbol{I_0}, \boldsymbol{e} \in \boldsymbol{L_0}) = Ind(0, \boldsymbol{a}, \boldsymbol{b}) \quad = \quad \boldsymbol{a} \vee [\boldsymbol{b} \wedge (\neg \boldsymbol{a} +_{\mathbf{b}} \neg(\boldsymbol{a} \vee \boldsymbol{b}))]$$

Après calcul, on obtient finalement que $(\boldsymbol{r} = \mathbf{0}) = 111001000$, ce qui nous donne les éléments suivants du vecteur de sortie:

$\boldsymbol{e}$	a	b	b	c	b	a	c	c	b
$\boldsymbol{r}$	0	0	0			0			

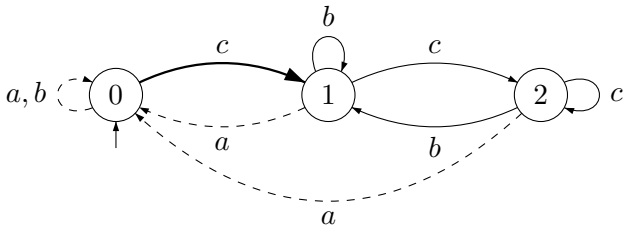
L'étape d'initialisation de l'algorithme est maintenant terminée et on a les affectations suivantes:

$$\begin{aligned} (\boldsymbol{r} = \mathbf{0}) &\leftarrow 111001000 \\ \boldsymbol{K} &\leftarrow (\boldsymbol{r} = \mathbf{0}) \end{aligned}$$

Maintenant, pour calculer les positions du vecteur de sortie où il y a des 1, il faut commencer par calculer le vecteur suivant:

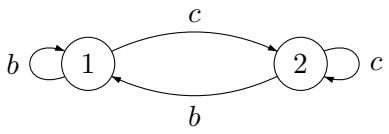
$$\boldsymbol{N} \quad \leftarrow \quad [(\uparrow_{\boldsymbol{i}<\mathbf{1}} \boldsymbol{K}) \wedge (\boldsymbol{\sigma}(\uparrow_{\boldsymbol{i}} \boldsymbol{r}, \boldsymbol{e}) = \mathbf{1})] \vee [\neg \boldsymbol{K} \wedge (\boldsymbol{e} \in \boldsymbol{I_1})]$$

La partie $[(\uparrow_{\boldsymbol{i}<\mathbf{1}} \boldsymbol{K}) \wedge (\boldsymbol{\sigma}(\uparrow_{\boldsymbol{i}} \boldsymbol{r}, \boldsymbol{e}) = \mathbf{1})]$ du vecteur représente les positions où, partant d'un état plus petit que 1 (et donc de l'état 0), une transition nous amène dans l'état 1. On peut trouver facilement ces positions, en nous servant de notre automate, ou de sa table de transition, et de notre table d'entrée-sortie:



$\boldsymbol{e}$	a	b	b	$\boldsymbol{c}$	b	a	$\boldsymbol{c}$	c	b
$\boldsymbol{r}$	0	0	**0**	1		**0**	1		

Maintenant, comme on a regardé toutes les transitions entrant et sortant de l'état 0, on peut enlever cet état de l'automate et regarder ce qu'il reste:



Dans cet automate, chaque fois qu'il y a un b dans le vecteur d'entrée, la sortie est 1. C'est ce que nous fait calculer la deuxième partie du vecteur $\boldsymbol{N}$, c'est-à-dire la partie $[\neg \boldsymbol{K} \wedge (\boldsymbol{e} \in \boldsymbol{I_1})]$. On obtient donc les nouveaux éléments suivants du vecteur de sortie:

$\boldsymbol{e}$	a	b	b	c	b	a	c	c	b
$\boldsymbol{r}$	0	0	0	1	**1**	0	1		**1**

Comme L_1 est un ensemble vide, calculer le vecteur $(\boldsymbol{r = 1})$ revient à calculer le vecteur $\boldsymbol{N}$. On a donc

$$\begin{aligned} \boldsymbol{N} &\leftarrow 000100100 \wedge 000010001 = 000110101 \\ (\boldsymbol{r = 1}) &\leftarrow Ind(k, \boldsymbol{N}, \emptyset) = \boldsymbol{N} \\ \boldsymbol{K} &\leftarrow \boldsymbol{K} \vee (\boldsymbol{r = 1}) = 111111101 \end{aligned}$$

Le dernier vecteur à calculer est $(\boldsymbol{r = 2})$ et il est égale à la négation du dernier vecteur $\boldsymbol{K}$ calculé:

$$(\boldsymbol{r = 2}) \leftarrow \neg \boldsymbol{K} = 000000010$$

Notre vecteur de sortie est donc

$\boldsymbol{e}$	a	b	b	c	b	a	c	c	b
$\boldsymbol{r}$	0	0	0	1	1	0	1	2	1

ce qui met fin à notre exemple.

Il est clair que si on laisse de côté le test $\boldsymbol{\sigma(\uparrow_i r, e) = k}$, l'algorithme requière $\mathcal{O}(d)$ étapes élémentaires, étant donné que tous les vecteurs de la forme $(\boldsymbol{e \in S})$ peuvent être précalculés. Dans la prochaine section, dans le cas particulier de la recherche approximative d'un mot dans un texte, nous donnerons une définition simple et récursive pour le test $\boldsymbol{\sigma(\uparrow_i r, e) = k}$. Cette définition utilise des propriétés arithmétiques de la table de transition particulière de ce problème qui ne peuvent être généralisées. Le cas général sera traité au chapitre 5.

4.3 Algorithme vectoriel pour la recherche approximative d'un mot dans un texte

Dans cette section, nous verrons un cas particulier d'application de l'algorithme de la section précédente à la bioinformatique. En fait, nous allons donner un algorithme vectoriel pour résoudre le problème de recherche approximative d'un mot dans un texte, lorsque la distance utilisée est une distance d'édition généralisée.

4.3.1 Rappel du problème

On rappelle que ce problème consiste, étant donné un mot $P = p_1 \dots p_m$, et un texte $T = t_1 \dots t_n$, à trouver toutes les occurrences approximatives de P dans T. Formellement, on veut trouver toutes les positions j dans T telles que, étant donné un nombre d'erreurs permis $t \geq 0$, on a $\min_g D(P, T[g, j]) \leq t$. Dans les problèmes de recherche en biologie, le mot P est habituellement assez court – quelques centaines de caractères – alors que T peut être très long.

Rappelons aussi que la solution classique (Sellers, 1980) est obtenue en calculant la matrice des distances $D[0..m, 0..n]$, avec la relation de récurrence:

$$D[i,j] = \min \begin{cases} D[i-1, j-1] + \delta(p_i, t_j) \\ D[i, j-1] + c \\ D[i-1, j] + c \end{cases} \tag{4.9}$$

et les conditions initiales $D[0, j] = 0$ et $D[i, 0] = ic$.

Les valeurs successives de $D[m, j]$ peuvent alors être comparées à t, donnant les positions j du texte où il y une occurrence approximative de P.

Exemple 4.7 Supposons que l'on veuille calculer les occurrences approximatives, à une erreur près, du mot $TATA$ dans le texte $ACGTAATAGC \dots$ avec la distance d'édition

habituelle. La matrice suivante est la matrice des distances du problème:

		A	C	G	T	A	A	T	A	G	C	...
	0	0	0	0	0	0	0	0	0	0	0	...
T	1	1	1	1	0	1	1	0	1	1	1	...
A	2	1	2	2	1	0	1	1	0	1	...	...
T	3	2	2	3	2	1	1	1	1	1	...	...
Ligne m A	4	3	3	3	3	2	**1**	2	**1**	2	...	...
							↑		↑			

Dans cette table, on voit qu'à deux occasions, le mot P est à une distance 1 de sous-suites du texte – la sous-suite TAA, en position 6, et les sous-suites ATA, $AATA$, et $TAATA$, en position 8.

Le calcul peut être exécuté colonne par colonne requérant un temps $\mathcal{O}(nm)$ et un espace $\mathcal{O}(m)$. Dans le but de faire mieux, commençons par mentionner un lemme qui borne la valeur absolue des différences verticales et horizontales entre deux valeurs du tableau:

Lemme 4.2 $|D[i,j] - D[i-1,j]| \leq c$ *et* $|D[i,j] - D[i,j-1]| \leq c$.

Preuve: Ces relations sont clairement vérifiées pour la première ligne et la première colonne de la matrice, étant donné que $D[0,j] = 0$ et $D[i,0] = ic$.

Supposons maintenant qu'elles sont vérifiées pour les paires $[i-1,j]$ et $[i,j-1]$. On va montrer qu'elles seront alors aussi vérifiées pour la paire $[i,j]$. Par la récurrence 4.9 on a que

$$D[i,j] - D[i-1,j] = \min \begin{cases} D[i-1,j-1] - D[i-1,j] + \delta(p_i, t_j) \\ D[i,j-1] - D[i-1,j] + c \\ c \end{cases}$$

Donc, $D[i,j] - D[i-1,j] \leq c$. On doit maintenant montrer que $D[i,j] - D[i-1,j] \geq -c$. Par hypothèse, $D[i-1,j-1] - D[i-1,j] \geq -c$ et, comme $\delta(p_i, t_j) \geq 0$, on a certainement que $D[i-1,j-1] - D[i-1,j] + \delta(p_i, t_j) \geq -c$.

Finalement, comme $D[i, j-1] - D[i-1, j] + c$ peut être écrit de la façon suivante:

$$(D[i, j-1] - D[i-1, j-1]) + (D[i-1, j-1] - D[i-1, j]) + c,$$

et que chacun des termes entre parenthèses est plus grand que $-c$, alors leur somme est plus grande que $-c$.

Un argument similaire permet de démontrer que $|D[i, j] - D[i, j-1]| \leq c$. ■

Comme les différences verticales et horizontales sont bornées, on peut coder le calcul de la table des distances par un automate qui, nous le verrons dans ce qui suit, est résoluble.

4.3.2 Calculer les distances avec un automate

Comme c'est le cas pour tous les algorithmes vectoriels considérés au chapitre précédent, notre algorithme utilisera un automate pour résoudre le problème des occurrences approximatives d'un mot P dans un texte T. Par contre, contrairement aux automates des algorithmes présentés précédemment, qui dépendaient du mot P cherché, notre automate dépendra seulement de la distance d'édition généralisée utilisée.

Dans ce qui suit, les notations employées sont une généralisation, à une distance d'édition généralisée, des notations introduites par Myers dans le cas de la distance d'édition (voir la section 3.4).

Avec les notations de la récurrence 4.9, définissons:

$$\begin{aligned} \Delta v_{i,j} &= c - (D[i, j] - D[i-1, j]) \\ \Delta h_{i,j} &= D[i, j] - D[i, j-1] + c \end{aligned}$$

Du lemme 4.2, on a que $\Delta v_{i,j}$ et $\Delta h_{i,j}$ font partie de l'intervalle $\{0, \ldots, 2c\}$, et si les valeurs successives de $\Delta h_{m,j}$ sont connues, alors la valeur du *score*, $D[m, j]$, peut être calculée par la récurrence suivante, $D[m, j] = D[m, j-1] + \Delta h_{m,j} - c$, avec condition initiale $D[m, 0] = mc$.

Avec quelques manipulations arithmétiques, on peut transformer l'équation 4.9 en:

$$\Delta h_{i,j} = \min \begin{cases} \Delta v_{i,j-1} + \delta(p_i, t_j) \\ \Delta v_{i,j-1} + \Delta h_{i-1,j} \\ 2c \end{cases} \tag{4.10}$$

avec les conditions initiales suivantes: $\Delta h_{0,j} = c$ et $\Delta v_{i,0} = 0$.

On peut donc définir un automate $\mathcal{B}$ qui calculera la colonne $\boldsymbol{\Delta h_j} = \Delta h_{1,j} \dots \Delta h_{m,j}$ étant donné la séquence de couples suivante:

$$\boldsymbol{(\Delta v_{j-1}, \delta(p, t_j))} = ((\Delta v_{1,j-1}, \delta(p_1, t_j)) \dots (\Delta v_{m,j-1}, \delta(p_m, t_j)).$$

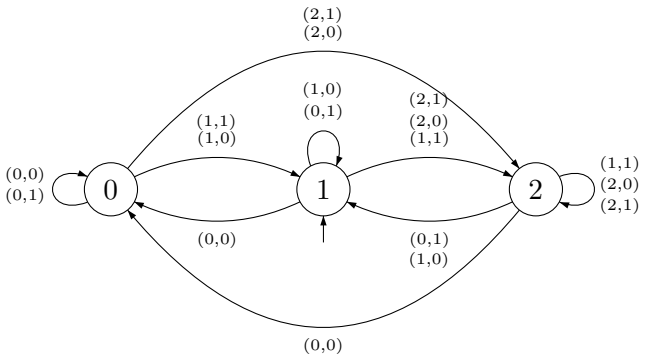
Les états de l'automate $\mathcal{B}$ sont $\{0, \dots, 2c\}$, l'état initial est c, et la fonction de transition σ de $\mathcal{B}$ est donné par:

$$\sigma(s, (\Delta v, \delta)) = \min \begin{cases} \Delta v + \delta \\ \Delta v + s \\ 2c. \end{cases}$$

pour un événement $(\Delta v, \delta)$ dans le produit cartésien $\{0, \dots, 2c\} \times \{0, \dots, 2c-1\}$.

Donc, si on connaît une colonne de la table des distances D, l'automate $\mathcal{B}$ nous donne l'information nécessaire à la connaissance de la colonne suivante de la table.

Exemple 4.8 L'automate $\mathcal{B}$ suivant est au coeur du calcul des positions des occurrences approximatives d'un mot P dans un texte T, lorsque la distance utilisée est la distance d'édition:



Cet automate a été construit de la façon suivante. Ici, la distance utilisée est la distance d'édition et donc $c = 1$. L'automate est donc constitué de trois états, les états 0, 1 et 2, et l'état initial est 1. Comme $c = 1$, on a que $\Delta v \in \{0, 1, 2\}$ et $\delta \in \{0, 1\}$. En utilisant le fait que la fonction de transition σ de $\mathcal{B}$ est définie, dans le cas de la distance d'édition, par

$$\sigma(s, (\Delta v, \delta)) = \min \begin{cases} \Delta v + \delta \\ \Delta v + s \\ 2. \end{cases}$$

on obtient la table de transition suivante, d'où l'automate:

	(0,0)	(0,1)	(1,0)	(1,1)	(2,0)	(2,1)
0	0	0	1	1	2	2
1	0	1	1	2	2	2
2	0	1	1	2	2	2

Il est bien de mentionner ici que, lorsqu'on lit l'article de Myers (Myers, 1999) entre les lignes, on se rend compte que l'efficacité de son algorithme repose sur le fait qu'il utilise un automate résoluble qui est, en fait, l'automate résoluble de l'exemple 4.8.

L'algorithme de Myers est donc une variante de l'algorithme général suivant:

Théorème 4.2 *L'automate $\mathcal{B}$ pour le problème de recherche approximative avec distance d'édition généralisée est résoluble.*

Preuve: On va montrer que, pour $k \in \{0, \ldots, 2c\}$, k est résoluble dans $\mathcal{B}_{k-1} = \mathcal{B} \setminus \{0, \ldots, k-1\}$, avec l'ensemble d'événements $(\Delta v, \delta)$ tels que $\Delta v + \delta = k$.

Premièrement notons que, comme $\Delta v + \delta < k$ implique que $\sigma(s, (\Delta v, \delta)) < k$, les seuls événements appartenant à l'automate $\mathcal{B}_{k-1}$ sont ceux satisfaisant l'équation $\Delta v + \delta \geq k$. Si $\Delta v + \delta = k$, alors $\min(\Delta v + \delta, \Delta v + s, 2c) = k$, étant donné que $s \geq k$ pour les états dans $\mathcal{B}_{k-1}$. Si $\Delta v + \delta > k$ et $s > k$, alors le minimum est certainement plus grand que k, et $\sigma(s, (\Delta v, \delta)) > k$. ■

4.3.3 Notre algorithme vectoriel

Comme l'automate $\mathcal{B}$ est résoluble, le théorème 4.1 peut être utilisé pour produire un algorithme vectoriel lui correspondant. Rappelons que la partie principale de l'algorithme contient trois instructions, où i est l'état initial et $\boldsymbol{K}$ est le vecteur $\boldsymbol{(r < k)}$:

$$\begin{aligned} \boldsymbol{N} &\leftarrow [(\uparrow_{\boldsymbol{i<k}} \boldsymbol{K}) \wedge (\boldsymbol{\sigma}(\uparrow_{\boldsymbol{i}} \boldsymbol{r}, \boldsymbol{e}) = \boldsymbol{k})] \vee [\neg \boldsymbol{K} \wedge (\boldsymbol{e} \in \boldsymbol{I_k})] \\ \boldsymbol{(r = k)} &\leftarrow Ind(k, \boldsymbol{N}, \boldsymbol{e} \in \boldsymbol{L_k}) \\ \boldsymbol{K} &\leftarrow \boldsymbol{K} \vee \boldsymbol{(r = k)} \end{aligned}$$

Dans le but d'adapter cet algorithme à l'automate $\mathcal{B}$, on doit trouver de bonnes expressions pour les formules $\boldsymbol{(\sigma(\uparrow_i r, e) = k)}$, $\boldsymbol{(e \in I_k)}$, et $\boldsymbol{(e \in L_k)}$.

La preuve du théorème 4.2 nous donne une façon simple de reconnaître l'ensemble des éléments de I_k. En effet, comme l'état k est résoluble avec l'ensemble des événements $(\Delta v, \delta)$ tels que $\Delta v + \delta = k$, alors, sous l'hypothèse que $\boldsymbol{(r \geq k)}$, on a:

$$\boldsymbol{(e \in I_k)} = \boldsymbol{(\Delta v + \delta = k)}.$$

Les événements bouclant sont aussi facilement détectables, étant donné que si $k = s < 2c$, alors soit $\Delta v = 0$ ou $\Delta v + \delta = k$. Comme les événements de la forme $\Delta v + \delta = k$ sont dans I_k, les seuls événements bouclant n'appartenant pas à l'ensemble I_k sont ceux pour lesquels $\Delta v = 0$. Maintenant, si l'état précédent est k et que l'événement courant est tel que $\Delta v = 0$ alors, par définition de la fonction de transition σ de l'automate, soit on va boucler sur k (lorsque $\delta \geq k$) ou on va se rendre dans un état plus petit que k (lorsque $\delta < k$). On peut donc calculer le vecteur $\boldsymbol{(r = k)}$ de la façon suivante:

$$\boldsymbol{(r = k)} \leftarrow Ind(k, \boldsymbol{N}, \boldsymbol{(\Delta v = 0)}) \wedge \neg \boldsymbol{K},$$

où on a remplacé le vecteur $\boldsymbol{e \in L_k}$ de la formule précédente par $\boldsymbol{(\Delta v = 0)}$ et où on ajouté la conjonction avec $\neg \boldsymbol{K}$, pour ne garder des événements $(\Delta v, \delta)$ tels que $\Delta v = 0$, que les événements bouclant sur k.

Finalement, dans le but de calculer les valeurs de $\boldsymbol{(\sigma(\uparrow_c r, e) = k)}$, on définit le vecteur

suivant:

$$\boldsymbol{V_k = \Delta v + min(\uparrow_c r, k)}$$

Clairement, $\boldsymbol{V_0 = \Delta v}$ et on a la relation récursive suivante entre $\boldsymbol{V_k}$ et $\boldsymbol{V_{k-1}}$:

Proposition 4.2

$$\boldsymbol{V_k = V_{k-1} + \neg(\uparrow_c r < k)}$$

Preuve: Commençons par démontrer qu'on a l'identité suivante, dans laquelle nous additionnons des nombres et des valeurs de vérité:

$$min(a,k) \quad = \quad min(a,k-1) + (a \geq k) \tag{4.11}$$

En effet, supposons que $k > a$, alors $min(a,k) = a = min(a,k-1) + 0$. De façon équivalente, si $k \leq a$ alors, $min(a,k) = k = min(a,k-1) + 1$. Maintenant, par définition, on a que

$$\boldsymbol{V_k = \Delta v + min(\uparrow_c r, k)}.$$

En utilisant l'équation 4.11, on obtient que

$$\begin{aligned} \boldsymbol{V_k} &= \boldsymbol{\Delta v + min(\uparrow_c r, k-1) + (\uparrow_c r \geq k)} \\ &= \boldsymbol{V_{k-1} + \neg(\uparrow_c r < k)} \end{aligned}$$

■

La prochaine proposition montre que l'on peut remplacer le calcul des vecteurs $\boldsymbol{N}$ de l'algorithme avec le calcul d'une expression qui contient seulement $\boldsymbol{V_k}$ et d'autres vecteurs connus. Notons que le vecteur $\boldsymbol{N}$ peut être écrit comme

$$\boldsymbol{N = [(\uparrow_c r < k) \wedge (r = k)] \vee [(r \geq k) \wedge (\Delta v + \delta = k)]}$$

en utilisant l'identités vectorielle $\boldsymbol{(\sigma(\uparrow_i r, e) = k) = (r = k)}$ et le fait que $\boldsymbol{(e \in I_k) = (\Delta v + \delta = k)}$, si $\boldsymbol{(r \geq k)}$.

Proposition 4.3 *Le vecteur*

$$\boldsymbol{N} = [(\uparrow_{\boldsymbol{c}} \boldsymbol{r} < \boldsymbol{k}) \wedge (\boldsymbol{r} = \boldsymbol{k})] \vee [(\boldsymbol{r} \geq \boldsymbol{k}) \wedge (\boldsymbol{\Delta v} + \boldsymbol{\delta} = \boldsymbol{k})]$$

est égal au vecteur suivant:

$$[((\uparrow_{\boldsymbol{c}} \boldsymbol{r} < \boldsymbol{k}) \wedge (\boldsymbol{V_k} = \boldsymbol{k})) \vee (\boldsymbol{\Delta v} + \boldsymbol{\delta} = \boldsymbol{k})] \wedge (\boldsymbol{r} \geq \boldsymbol{k}).$$

Preuve: La preuve est élémentaire, mais plus facile à suivre si l'on se rappelle les définitions de $\boldsymbol{r}$ et de $\boldsymbol{V_k}$ composantes par composantes:

$$r_i = \min(\Delta v_i + \delta_i, \Delta v_i + r_{i-1}, 2c)$$

$$V_{ki} = \Delta v_i + \min(r_{i-1}, k)$$

Supposons que $r_{i-1} < k$, et que $r_i = k$, on va montrer que cela implique que $V_{ki} = k$ ou que $\Delta v_i + \delta_i = k$. En fait, si $\Delta v_i + \delta_i \neq k$, alors comme par définition $r_i \leq \Delta v_i + \delta_i$, on doit avoir que $\Delta v_i + \delta_i > k$, ce qui implique que $r_i = \Delta v_i + r_{i-1}$. Comme $r_{i-1} < k$, $V_{ki} = \Delta v_i + r_{i-1} = r_i = k$.

Si $r_{i-1} < k$ et $V_{ki} = k$, alors $V_{ki} = \Delta v_i + r_{i-1} = k$. Comme $r_i \leq \Delta v_i + r_{i-1}$, alors $r_i \geq k$ implique que $r_i = k$. ■

En mettant tous ces morceaux ensembles, on obtient l'algorithme:

$$\begin{aligned} (\boldsymbol{r} = \boldsymbol{0}) &\leftarrow Ind(0, \boldsymbol{\Delta v} + \boldsymbol{\delta} = \boldsymbol{0}, \boldsymbol{\Delta v} = \boldsymbol{0}) \\ \boldsymbol{K} &\leftarrow (\boldsymbol{r} = \boldsymbol{0}) \\ \boldsymbol{V} &\leftarrow \boldsymbol{\Delta v} \end{aligned}$$

Pour les valeurs successives de k dans $\{1, \ldots, 2c-1\}$, on exécute les 4 instructions suivantes:

$$\begin{aligned} \boldsymbol{N} &\leftarrow [((\uparrow_{\boldsymbol{c}<\boldsymbol{k}} \boldsymbol{K}) \wedge (\boldsymbol{V} = \boldsymbol{k})) \vee (\boldsymbol{\Delta v} + \boldsymbol{\delta} = \boldsymbol{k})] \wedge \neg \boldsymbol{K} \\ (\boldsymbol{r} = \boldsymbol{k}) &\leftarrow Ind(k, \boldsymbol{N}, \boldsymbol{\Delta v} = \boldsymbol{0}) \wedge \neg \boldsymbol{K} \\ \boldsymbol{K} &\leftarrow \boldsymbol{K} \vee (\boldsymbol{r} = \boldsymbol{k}) \\ \boldsymbol{V} &\leftarrow \boldsymbol{V} + \neg(\uparrow_{\boldsymbol{c}<\boldsymbol{k}} \boldsymbol{K}). \end{aligned}$$

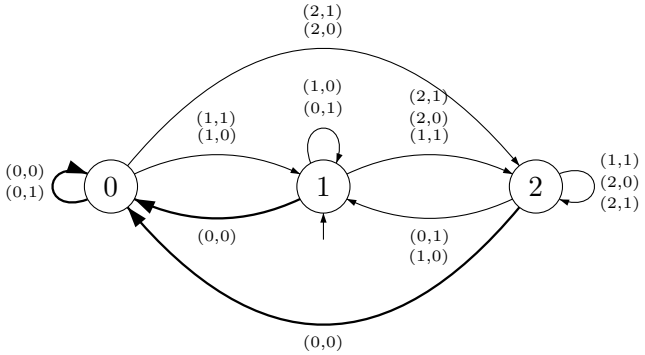
Dans le but de compléter la présentation d'un algorithme vectoriel pour le calcul de $\boldsymbol{\Delta v_j}$, on utilise la relation $\Delta v_{i,j} = \Delta h_{i-1,j} + \Delta v_{i,j-1} - \Delta h_{i,j}$ qui nous donne l'équation vectorielle suivante:

$$\boldsymbol{\Delta v_j} = \uparrow_c \boldsymbol{\Delta h_j} + \boldsymbol{\Delta v_{j-1}} - \boldsymbol{\Delta h_j}.$$

Exemple 4.9 Voyons comment fonctionne notre algorithme en calculant, comme dans l'exemple 4.7, les occurrences approximatives, à une erreur près, de $TATA$ dans le texte $ACGTAATAGC...$, quand la distance utilisée est la distance d'édition. Rappelons qu'on a une occurrence de P, avec une erreur ou moins, se terminant en position j de T si et seulement si $D[m, j] \leq 1$, où m est la longueur du mot P. Donc ici, à toutes les positions j telles que $D[4, j] \leq 1$, on a une occurrence de $P = TATA$. Au départ, $D[4, 0] = 4$ et notre algorithme nous permettra de calculer les valeurs successives de $D[4, j]$, pour j variant de 1 à n.

Au départ, comme la colonne 0 de la table des distances est la colonne 01234, le vecteur de différences verticales correspondant à cette colonne est $\boldsymbol{\Delta v_0} = 0000$. La première lettre du texte T étant A, on a $\boldsymbol{\delta}(A, TATA) = 1010$, ce qui nous donne le vecteur $\boldsymbol{\Delta v + \delta} = 1010$. Pour trouver $\boldsymbol{\Delta h_1}$, on se sert maintenant de notre automate résoluble décrit dans l'exemple 4.8.

La première étape consiste à trouver les positions de la séquence de sortie, c'est-à-dire les positions dans le vecteur $\boldsymbol{\Delta h_1}$, où il y a des 0. Pour ce faire on regarde toutes les transitions entrant dans l'état 0



c'est-à-dire qu'on calcule le vecteur

$$(\boldsymbol{r = 0}) \quad \leftarrow \quad Ind(0, \boldsymbol{\Delta v + \delta = 0}, \boldsymbol{\Delta v = 0})$$

Ici, $\boldsymbol{\Delta v + \delta} = 1010$ et donc le vecteur $\boldsymbol{\Delta v + \delta = 0}$ est le vecteur 0101. De la même façon, comme $\boldsymbol{\Delta v} = 0000$, on a que le vecteur $\boldsymbol{\Delta v = 0}$ est le vecteur 1111. Comme 0 n'est pas l'état initial de l'automate, on a, par la définition 4.3 que

$$Ind(0, \boldsymbol{\Delta v + \delta = 0}, \boldsymbol{\Delta v = 0}) \quad = \quad Ind(0, 0101, 1111) \quad = \quad 0111.$$

Donc $\boldsymbol{(r = 0)} = 0111$, ce qui nous donne les éléments suivants du vecteur de sortie:

$\boldsymbol{e}$	$(0,1)$	$(0,0)$	$(0,1)$	$(0,0)$
$\boldsymbol{r = \Delta h_1}$		0	0	0

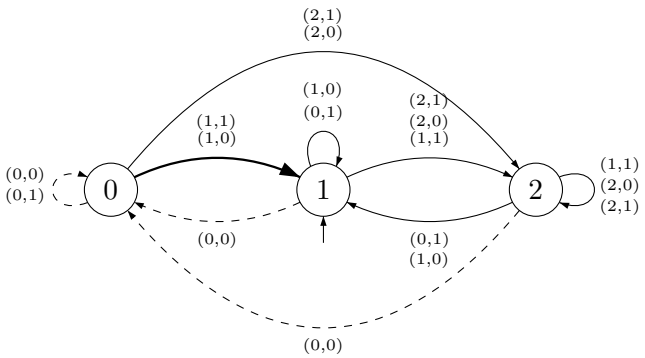
L'étape d'initialisation de l'algorithme est maintenant terminée et on a les affectations suivantes:

$$\begin{aligned} \boldsymbol{(r = 0)} &\leftarrow 0111 \\ \boldsymbol{K} &\leftarrow \boldsymbol{(r = 0)} \\ \boldsymbol{V} &\leftarrow 0000 \end{aligned}$$

Maintenant, pour calculer les positions du vecteur de sortie où il y a des 1, il faut commencer par calculer le vecteur suivant:

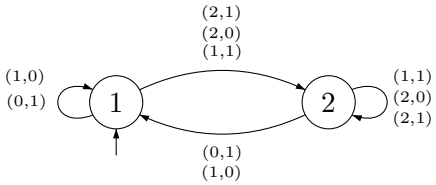
$$\boldsymbol{N} \quad \leftarrow \quad [((\uparrow_{\boldsymbol{1<k}} \boldsymbol{K}) \wedge \boldsymbol{(V = k)}) \vee \boldsymbol{(\Delta v + \delta = k)}] \wedge \neg \boldsymbol{K}$$

La partie $\boldsymbol{(\uparrow_{1<k} K) \wedge (V = k)}$ du vecteur représente les positions où, partant d'un état plus petit que 1, une transition nous amène dans l'état 1:



$\boldsymbol{e}$	$(0,1)$	$(0,0)$	$(0,1)$	$(0,0)$
$\boldsymbol{r = \Delta h_1}$		0	0	0

On se rend compte que cela n'arrive jamais dans cet exemple particulier. Maintenant, comme on a regardé toutes les transitions entrant et sortant de l'état 0, on peut enlever cet état de l'automate et regarder ce qu'il reste:



Dans cet automate, chaque fois qu'il y a un couple $(\Delta v, \delta)$, tel que $\Delta v + \delta = 1$ dans le vecteur d'entrée, la sortie est 1. C'est ce que nous fait calculer la deuxième partie du vecteur $\boldsymbol{N}$, c'est-à-dire la partie $\boldsymbol{(\Delta v + \delta = k) \wedge \neg K}$. Cela nous donne, dans ce cas, le dernier élément de notre vecteur de sortie:

$\boldsymbol{e}$	$(0,1)$	$(0,0)$	$(0,1)$	$(0,0)$
$\boldsymbol{r = \Delta h_1}$	**1**	0	0	0

On vient de calculer l'affectation suivante:

$$\boldsymbol{N} \leftarrow 0000 \wedge 1000 = 1000.$$

Maintenant, comme 1 est l'état initial de l'automate, on obtient par la définition 4.3 que

$$\begin{aligned} \boldsymbol{(r = 1)} &= Ind(1, \boldsymbol{N}, \boldsymbol{\Delta v = 0}) \wedge \boldsymbol{\neg K} \\ &= Ind(1, 1000, 1111) \wedge 1000 \\ &= 1111 \wedge 1000 \\ &= 1000, \end{aligned}$$

et

$$\begin{aligned} \boldsymbol{K} &\leftarrow \boldsymbol{K} \vee \boldsymbol{(r = k)} \\ &\leftarrow 0111 \vee 1000 \\ &\leftarrow 1111. \end{aligned}$$

Le dernier vecteur à calculer est $(\boldsymbol{r = 2})$ et il est égale à la négation du dernier vecteur $\boldsymbol{K}$ calculé:

$$(\boldsymbol{r = 2}) \leftarrow \neg \boldsymbol{K} = 0000.$$

On a donc trouvé que la première colonne de différence horizontales, $\boldsymbol{\Delta} h_1$, est 1000.

Il est bien de remarquer ici que même si le calcul d'une colonne de différences horizontales semble demander beaucoup de travail, il ne demande qu'en fait $\mathcal{O}(c)$ opérations vectorielles, peu importe la longueur m du mot cherché P. En effet, le calcul de chaque vecteur $(\boldsymbol{r = k})$ demande un nombre constant d'opérations vectorielles et nous devons calculer $2c$ de ces vecteurs.

Maintenant, comme $\Delta h_{4,1} = 0$, on a que

$$\begin{aligned} D[4,1] &= D[4,0] + \Delta h_{4,1} - 1 \\ &= 4 + 0 - 1 \\ &= 3. \end{aligned}$$

On n'a donc pas d'occurrences de $TATA$, contenant une erreur ou moins, se terminant en position 1 du texte. Pour trouver $\boldsymbol{\Delta h_2}$, on a besoin du vecteur $\boldsymbol{\Delta v_1}$, qu'on trouve avec l'équation vectorielle suivante:

$$\begin{aligned} \boldsymbol{\Delta v_1} &= \uparrow_1 \boldsymbol{\Delta h_1} + \boldsymbol{\Delta v_0} - \boldsymbol{\Delta h_1} \\ &= 1100 + 0000 - 1000 \\ &= 0100. \end{aligned}$$

On calculerait de la même façon $\boldsymbol{\Delta h_2}$ en utilisant notre automate $\mathcal{B}$ sur le vecteur d'entrée $(\boldsymbol{\Delta v, \delta})$, où $\boldsymbol{\Delta v} = 0100$ et $\boldsymbol{\delta} = 1111$, car la deuxième lettre du texte T est C et que $\boldsymbol{\delta}(C, TATA) = 1111$.

4.3.4 Mesure de complexité

Il reste à éclaircir certains points pour obtenir une bonne implantation avec vecteurs de bits de l'algorithme et sa mesure de complexité. Les procédures essentielles à cette implantation sont celles pour l'assignation, la comparaison et l'arithmétique modulo

$2c+1$, avec retenue. Toutes ces opérations peuvent être raisonnablement implantées en temps $\mathcal{O}(\log c)$ en utilisant les opérations sur les bits.

La plupart des opérations de l'algorithme vectoriel pour l'automate $\mathcal{B}$ sont des opérations habituelles sur les vecteurs de bits. L'implantation de $(\mathbf{\Delta v} = \mathbf{0})$ et $(\mathbf{V_k} = \mathbf{k})$ est directe, si la comparaison modulo $2c+1$ a été préalablement implantée. La somme $\mathbf{\Delta v} + \delta$ peut être implantée comme une somme bornée, c'est-à-dire:

$$\mathbf{\Delta v} + \delta = \begin{cases} 2c \text{ si } \mathbf{\Delta v} + \delta (mod \quad 2c+1) \text{ nous donne une retenue} \\ \mathbf{\Delta v} + \delta (mod \quad 2c+1) \text{ sinon} \end{cases}$$

Donc, comme le nombre d'opérations de notre algorithme de la section 4.4.3 est $\mathcal{O}(c)$, la complexité en temps d'une implantation de notre algorithme, avec vecteurs de bits, est de $\mathcal{O}(c \log c)$ opérations vectorielles. Cette borne est très bonne en biologie, étant donné que la valeur de c est toujours petite dans les applications.

CHAPITRE V

DÉCOMPOSITION EN CASCADE D'AUTOMATES

La classe des automates apériodiques a, depuis de nombreuses années, été très étudiée en théorie des automates finis. Dans ce chapitre, nous démontrons que l'on peut associer un algorithme vectoriel à tout automate apériodique. La construction de cet algorithme vectoriel repose sur la décomposition en cascade de Krohn-Rhodes (Krohn - Rhodes, 1965) et nous démontrons que l'algorithme comprend souvent un nombre exponentiel (en le nombre de transitions de l'automate) d'opérations vectorielles. D'un autre côté, nous montrons que la décomposition en cascade des automates résolubles est particulièrement simple et que l'algorithme sur les vecteurs de bits qui en découle est de complexité linéaire en le nombre de transitions de l'automate.

5.1 Introduction

On s'intéresse à la question de savoir quels sont les automates, finis et déterministes, pour lesquels il est possible de trouver le vecteur de sortie $r_1 \dots r_m$, des états visités lors de la lecture d'un vecteur d'entrée $e_1 \dots e_m$, en utilisant un nombre constant d'opérations vectorielles. Au chapitre 4, nous avons identifié une classe d'automates, les automates résolubles, pour laquelle nous avons démontré l'existence d'algorithmes vectoriels utilisant un nombre borné d'opérations vectorielles. Dans ce chapitre, nous allons étendre cette classe aux automates *apériodiques* en étudiant les liens entre la décomposition en cascade d'un automate apériodique et les algorithmes vectoriels. Nous verrons que ces algorithmes vectoriels, découlant de la décomposition en cascade d'un automate apériodique, utilisent, dans le pire des cas, un nombre exponentiel (en le nombre de transitions de l'automate) d'opérations vectorielles. Par contre, nous allons montrer que les automates résolubles admettent, avec cette méthode, des algorithmes bit-vectoriels

dont la complexité est *linéaire* en le nombre de transitions de l'automate.

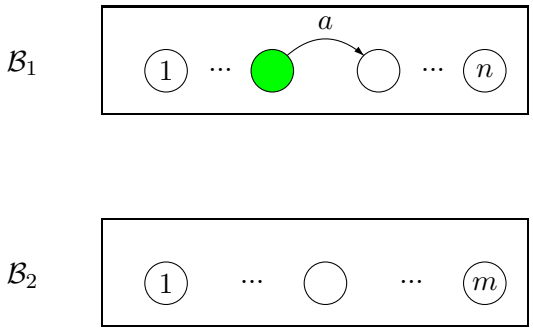
5.2 Construction de cascades d'automates

Dans cette section, nous présentons le concept de décomposition en cascade d'un automate. Nous discutons ensuite de la complexité de cette construction.

5.2.1 Définition de la décomposition en cascade

Le *produit en cascade* d'automates (Zeiger, 1967), et son équivalent algébrique, le *produit en couronne* de semi-groupes, ont des définitions assez complexes. Dans ce qui suit, nous allons d'abord donner l'idée générale de la construction en termes simples. Nous formaliserons par la suite.

Considérons deux automates: l'automate $\mathcal{B}_1$, avec n états, et l'automate $\mathcal{B}_2$, avec m états, que l'on va représenter par le diagramme générique suivant, où a représente une transition arbitraire de $\mathcal{B}_1$ et où les transitions de $\mathcal{B}_2$ sont pour l'instant non définies.



Dans le but de définir le *produit en cascade* $\mathcal{B}_1 \circ \mathcal{B}_2$, on attache à chacun des états de $\mathcal{B}_1$ une copie de l'automate $\mathcal{B}_2$, ayant m états, et dont la fonction de transition peut varier de copie en copie.

La machine, c'est-à-dire le produit $\mathcal{B}_1 \circ \mathcal{B}_2$, opère suivant le protocole suivant. L'automate $\mathcal{B}_1$ est dans un état courant –en vert – et chacune des copies de l'automate $\mathcal{B}_2$ est dans un même état courant – aussi en vert. Cette paire d'états représente *l'état global* de la machine.

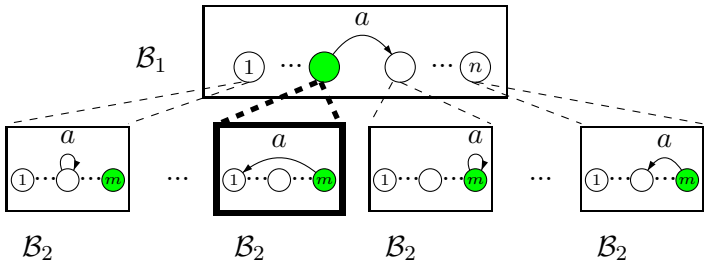


Figure 5.1 Un état global du produit en cascade

Si la prochaine entrée est a, l'automate $\mathcal{B}_1$ réagit de façon habituelle en suivant la transition a. L'automate $\mathcal{B}_2$ réagit, quant à lui, en fonction de la copie attachée à l'état courant de $\mathcal{B}_1$. En assumant que l'état global de la machine est celui de la Figure 5.1 alors, après la lecture de l'entrée a, l'état global du produit sera:

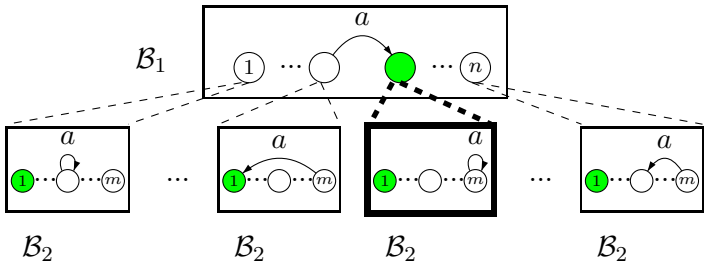


Figure 5.2 Le nouvel état global après la transition a

Il est clair que l'on peut décrire le fonctionnement du produit avec un automate ayant $n \times m$ états, étant donné que pour chaque état global (q_1, q_2), et chacune des lettres de l'alphabet d'entrée, il y a un unique état global correspondant. Cette construction peut aussi être itérée, permettant le produit en cascade de n automates. Plus formellement, on a la définition suivante du produit en cascades, tirée de l'article de Maler et Pnueli (Maler - Pnueli, 1994). Ces auteurs ont réécrit plusieurs définitions et théorèmes concernant les semi-groupes en terme d'automates. Le traitement étant intéressant, nous l'emploierons dans ce qui suit.

Définition 5.1 Soit $\mathcal{B}_i = (Q_1 \times \cdots \times Q_{i-1} \times \Sigma, Q_i, \delta_i)$ une famille d'automates ayant

comme alphabets d'entrée les produits $Q_1 \times \cdots \times Q_{i-1} \times \Sigma$ et ayant δ_i comme fonctions de transition, possiblement partielles. Le *produit en cascade* $\mathcal{C} = (\Sigma, Q, \delta') = \mathcal{B}_1 \circ \mathcal{B}_2 \circ \ldots \circ \mathcal{B}_k$ est un automate tel que:

1. $Q = Q_1 \times \ldots \times Q_k$
2. La fonction de transition globale, δ', est évaluée coordonnée par coordonnée par

$$\delta'(q_1 \ldots q_k, \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, (q_1, \sigma)), \ldots, \delta_k(q_k, (q_1 \ldots q_{k-1}, \sigma)))$$

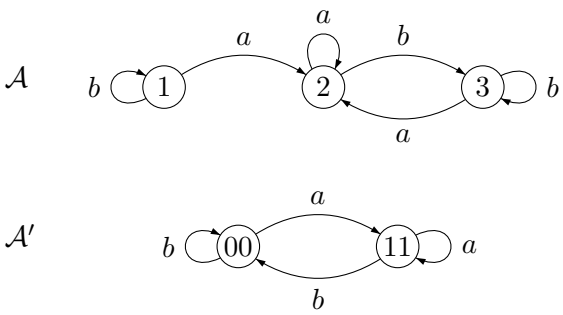
Pour pouvoir définir le concept de décomposition en cascade d'un automate, on doit d'abord définir ce qu'est un homomorphisme d'automates.

Définition 5.2 Une fonction surjective $\varphi : Q \to Q'$ est un *homomorphisme d'automates*, de $\mathcal{A} = (\Sigma, Q, \delta)$ à $\mathcal{A}' = (\Sigma, Q', \delta')$, si elle satisfait l'égalité suivante, $\forall q \in Q$ et $\forall \sigma \in \Sigma$:

$$\varphi(\delta(q, \sigma)) = \delta'(\varphi(q), \sigma).$$

On dit que l'homomorphisme d'automates est *partiel* lorsqu'il n'est défini que sur un sous-ensemble de Q.

Exemple 5.1 Soient $\mathcal{A}$ et $\mathcal{A}'$ les automates suivants sur l'alphabet $\Sigma = \{a, b\}$:



L'homomorphisme $\varphi : Q \setminus \{1\} \to Q'$ définit par $\varphi(2) = 11$ et $\varphi(3) = 00$, est un homomorphisme partiel d'automates car

$$\varphi(\delta(2, a)) = \varphi(2) = 11 = \delta'(11, a) = \delta'(\varphi(2), a),$$

$$\varphi(\delta(2,b)) = \varphi(3) = 00 = \delta'(11,b) = \delta'(\varphi(2),b),$$

$$\varphi(\delta(3,a)) = \varphi(2) = 11 = \delta'(00,a) = \delta'(\varphi(3),a),$$

$$\varphi(\delta(3,b)) = \varphi(3) = 00 = \delta'(00,b) = \delta'(\varphi(3),b),$$

et que lorsque $q = 1$, la fonction φ n'est pas définie.

La *décomposition en cascade* d'un automate $\mathcal{A}$ peut alors être définie comme suit:

Définition 5.3 Soit $\mathcal{A}$ un automate. Une *décomposition en cascade* de $\mathcal{A}$ est donnée par un produit en cascade $\mathcal{C} = \mathcal{B}_1 \circ \mathcal{B}_2 \circ \ldots \circ \mathcal{B}_k$, $k > 1$, et un homomorphisme – possiblement partiel– φ de $\mathcal{C}$ à $\mathcal{A}$.

Exemple 5.2 Soit l'automate suivant:

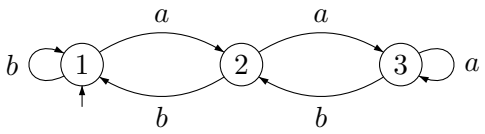


Figure 5.3 Un compteur borné

Cet automate admet la décomposition suivante avec l'homomorphisme

$$\varphi(0,0) = 1 \quad \varphi(0,1) = \varphi(1,0) = 2 \quad \varphi(1,1) = 3.$$

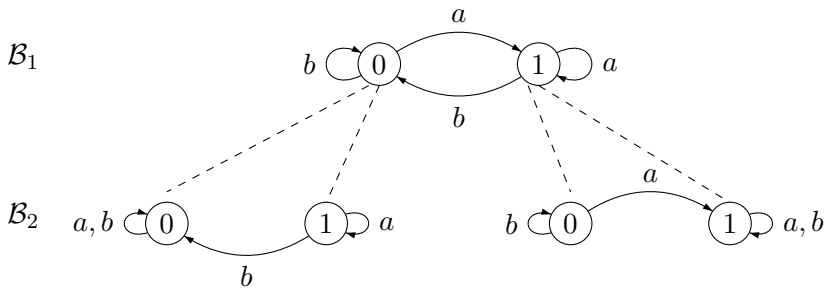


Figure 5.4 Une décomposition en cascade

Dans cet exemple, le comportement global $\mathcal{C}$ du produit en cascade, et l'homomorphisme φ, peuvent être illustrés par le diagramme suivant:

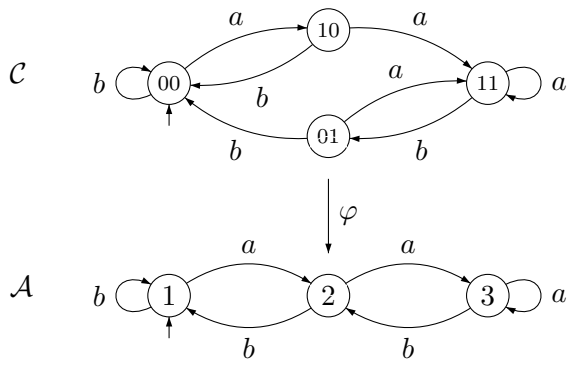


Figure 5.5 L'homomorphisme de $\mathcal{C} = \mathcal{B}_1 \circ \mathcal{B}_2$ à $\mathcal{A}$

Un théorème de Krohn et Rhodes, (Krohn - Rhodes, 1965), dit que tout automate admet une décomposition en cascades dont les composantes sont très simples, et dont la nature reflète les propriétés algébriques du langage reconnu par l'automate. Mais avant d'énoncer ce théorème, il nous faut quelques définitions.

Définition 5.4 Soit $\mathcal{A}$ un automate, on définit trois types particuliers de transitions dans $\mathcal{A}$:

1. Un événement a de l'alphabet d'entrée Σ de $\mathcal{A}$ est appelé un *reset* si la fonction induite par a sur les états de $\mathcal{A}$ est constante:

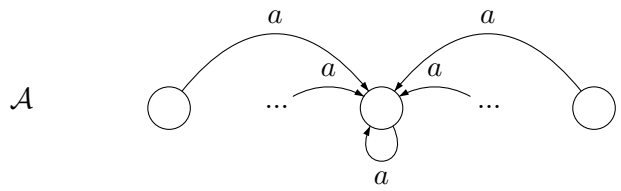


Figure 5.6 Un reset

2. Un événement a de l'alphabet d'entrée Σ de $\mathcal{A}$ est appelé une *identité* si la fonction induite par a sur les états de $\mathcal{A}$ est la fonction identité:

Figure 5.7 Une identité

3. Un événement a de l'alphabet d'entrée Σ de $\mathcal{A}$ est appelé une *permutation* si la fonction induite par a sur les états de $\mathcal{A}$ permute l'ensemble des états sur lesquels a est définie:

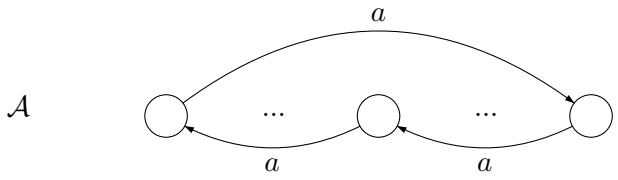


Figure 5.8 Une permutation

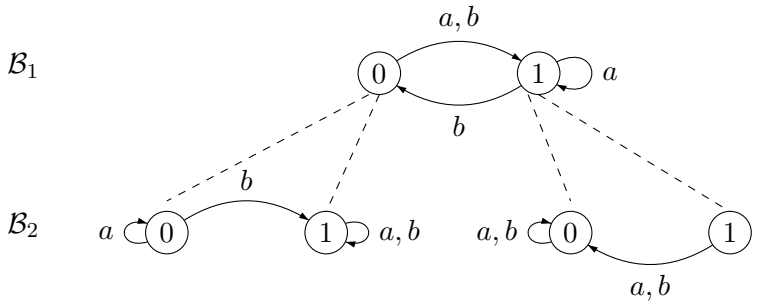
Définition 5.5 Un automate *reset* est un automate dans lequel toutes les lettres $a \in \Sigma$ induisent des resets ou des identités. Un automate *permutation-reset* est un automate dans lequel chaque lettre a de Σ induit soit une permutation, soit un reset sur les états sur lesquels a est définie.

Théorème 5.1 (Krohn-Rhodes, 1965) *Tout automate $\mathcal{A}$ possède une décomposition en cascade $\mathcal{C} = \mathcal{B}_1 \circ \mathcal{B}_2 \circ \ldots \circ \mathcal{B}_k$, où chaque automate $\mathcal{B}_i$ est un automate permutation-reset.* ■

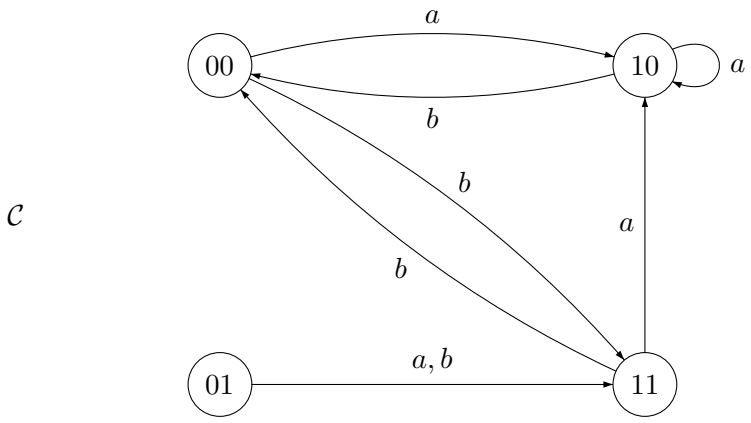
5.2.2 Problèmes reliés à la décomposition

Le problème avec le théorème 5.1 est que sa preuve ne donne pas de façon simple pour construire une décomposition en cascade d'un automate. Bien sûr, à partir d'une cascade d'automates, on peut facilement trouver un automate $\mathcal{A}$ dont cette cascade peut provenir. Voyons cela sur un exemple.

Exemple 5.3 Étant donné la cascade suivante:



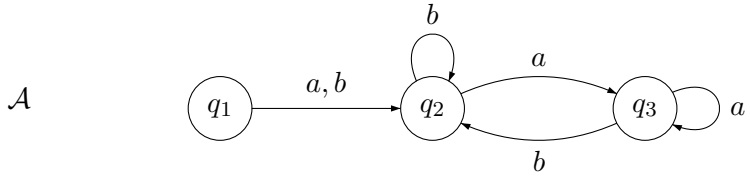
Le produit en cascade des automates $\mathcal{B}_1$ et $\mathcal{B}_2$ est l'automate $\mathcal{C} = \mathcal{B}_1 \circ \mathcal{B}_2$ suivant:



qu'on obtient en utilisant la définition 5.1 de la fonction de transition globale. Soit φ l'homomorphisme:

$$\varphi(01) = q_1, \quad \varphi(00) = \varphi(11) = q_2, \quad \varphi(10) = q_3.$$

Avec cet homomorphisme, on obtient l'automate $\mathcal{A}$ suivant, tel que $(\mathcal{C}, \varphi)$ est une décomposition en cascade de $\mathcal{A}$:



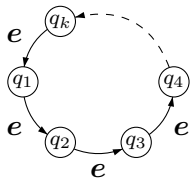
Le problème est qu'à partir d'un automate $\mathcal{A}$, il est rarement facile de trouver une décomposition en cascade de $\mathcal{A}$. Maler et Pnueli (Maler - Pnueli, 1994) proposent un algorithme exponentiel pour construire une décomposition en cascade d'un automate. Cette construction passe par plusieurs constructions intermédiaires et a une taille exponentielle en le nombre d'états de l'automate. Dans la prochaine section, nous parlerons de la spécialisation du théorème de Krohn et Rhodes aux automates apériodiques et nous montrerons comment associer un algorithme vectoriel à une décomposition en cascade.

5.3 Algorithmes vectoriels associés aux automates apériodiques

Dans cette section, nous verrons que l'on peut associer un algorithme vectoriel à tout automate apériodique. Mais commençons par définir le concept d'automates apériodiques.

5.3.1 Définitions et exemples

Soit $\mathcal{A}$ un automate. En général, un mot $\boldsymbol{e}$ induit une *permutation non-triviale* des états de l'automate $\mathcal{A}$ s'il existe une séquence de $k > 1$ états de $\mathcal{A}$ qui sont reliés cycliquement par la lecture de $\boldsymbol{e}$.



Définition 5.6 Un automate est *apériodique* si aucun mot n'induit une permutation non-triviale des états.

Exemple 5.4 L'automate $\mathcal{A}$ suivant est apériodique:

Cela vient du fait que que la fonction induite par la transition a est la fonction constante 2, et la fonction induite par b est la fonction constante 1. Il est clair que dans cet automate, aucun mot n'induit une permutation non-triviale des états.

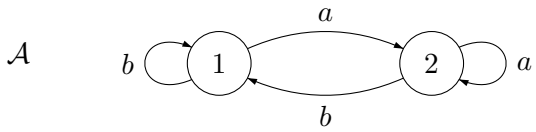


Figure 5.9 Un automate apériodique

Exemple 5.5 L'automate $\mathcal{B}$ suivant n'est pas apériodique:

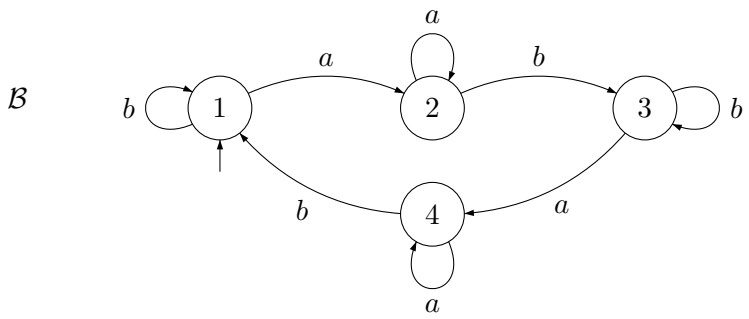
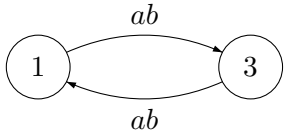


Figure 5.10 Un automate non-apériodique

En effet, le mot $\mathbf{e} = ab$ induit la permutation non-triviale suivante des états de $\mathcal{B}$:

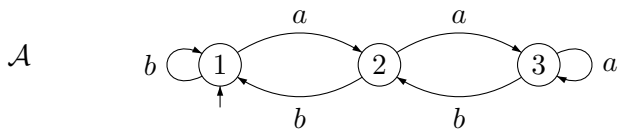


Théorème 5.2 (Krohn - Rhodes, 1965) *Tout automate apériodique admet une décomposition en cascade ne contenant que des automates resets binaires (à 2 états).* ■

5.3.2 Les décompositions en cascade sont des algorithmes vectoriels

Le théorème 5.2 nous donne le lien entre les automates apériodiques et les algorithmes vectoriels. En effet, étant donné la décomposition en cascade $(\mathcal{C}, \varphi)$ d'un automate apériodique $\mathcal{A}$, il est facile de produire une caractérisation logique des états de sorties de $\mathcal{A}$. Voyons cela sur un exemple.

Exemple 5.6 Regardons la décomposition en cascade de l'automate suivant:



Cet automate admet la décomposition en cascade de la Figure 5.11 avec l'homomorphisme

$$\varphi(0,0) = 1 \quad \varphi(0,1) = \varphi(1,0) = 2 \quad \varphi(1,1) = 3.$$

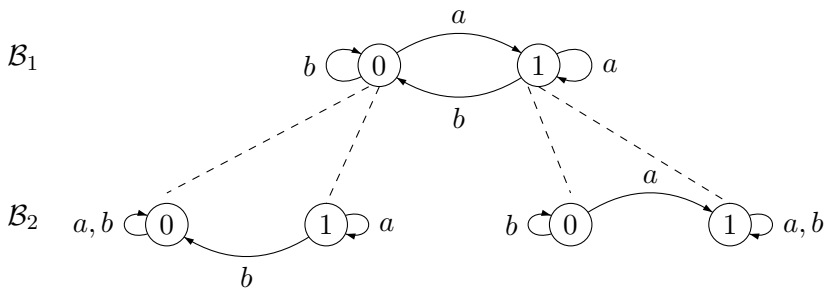


Figure 5.11 Une décomposition en cascade de $\mathcal{A}$

Comme $\varphi(0,0) = 1$ et que $\varphi(x,y) \neq 1$ si $(x,y) \neq (0,0)$, on a que

$\mathcal{A}$ est dans l'état 1 ssi

$\mathcal{B}_1$ est dans l'état 0 $\wedge$ $\mathcal{B}_2$ est dans l'état 0

En général, puisque l'homomorphisme φ est surjectif, on a que chaque état s de l'automate $\mathcal{A}$ possède une image inverse, $\varphi^{-1}(s)$, constituée d'un ou plusieurs état global de la décomposition en cascade. Comme chacun de ces états peut s'écrire sous la forme $\mathcal{B}_1$ est dans l'état s_1 et ... et $\mathcal{B}_n$ est dans l'état s_n, on a le résultat suivant:

Corollaire 5.1 *Pour tout automate apériodique $\mathcal{A}$, la proposition "$\mathcal{A}$ est dans l'état s" est équivalente à une disjonction de propositions de la forme:*

$$\mathcal{B}_1 \textit{ est dans l'état } s_1 \wedge ... \wedge \mathcal{B}_n \textit{ est dans l'état } s_n,$$

où chaque $\mathcal{B}_i$ est un automate reset binaire. ■

Le corollaire 5.1 implique que le problème de transformer un calcul sur un automate apériodique en un algorithme sur les vecteurs de bits se réduit au problème de trouver des algorithmes vectoriels pour les automates resets binaires.

Un automate reset binaire a une structure très simple que nous représentons par l'automate générique suivant.

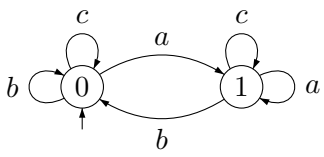


Figure 5.12 Un automate reset binaire

Les lettres a et b sont des resets des états 1 et 0, respectivement. La lettre c induit une identité.

Supposons que l'état 0 est l'état initial et définissons L_q comme étant l'ensemble des mots non-vides se terminant dans l'état q. On a alors la caractérisation suivant de L_0.

$$L_0 \quad = \quad \{\boldsymbol{e}|\ \boldsymbol{e} \text{ est } c \dots c, \text{ ou il y a un } b \text{ dans } \boldsymbol{e} \text{ et aucun } a \text{ depuis le dernier } b\} \quad (5.1)$$

Étant donné un mot $\boldsymbol{e}$, considérons ses vecteurs caractéristiques $\boldsymbol{a}$ et $\boldsymbol{b}$. Une reformulation du lemme de l'addition (lemme 4.1) fait le lien entre l'appartenance à L_0 et des opérations sur les vecteurs de bits:

Lemme 5.1 (Le lemme de l'addition, version 2) *Le mot $\boldsymbol{e} \in L_0$ si et seulement si le dernier bit de*

$$\boldsymbol{b} \vee (\neg \boldsymbol{a} \wedge (\boldsymbol{a} +_{\boldsymbol{b}} \neg \boldsymbol{b}))$$

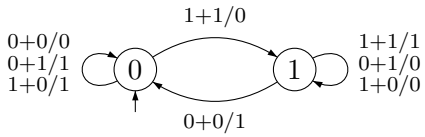
est 1.

Preuve (version 2): Supposons que $\boldsymbol{e}$ est dans L_0. Si $\boldsymbol{e}$ est de la forme $c \dots c$, alors $\boldsymbol{a} = \boldsymbol{b} = 0 \dots 0$, et on obtient facilement que $\neg \boldsymbol{a} \wedge (\boldsymbol{a} +_{\boldsymbol{b}} \neg \boldsymbol{b}) = 1 \dots 1$.

Maintenant, supposons qu'il y a un b dans $\boldsymbol{e}$ et aucune occurrence de a depuis la dernière occurrence de b; supposons aussi que la dernière lettre de $\boldsymbol{e}$ n'est pas b, étant donné que, dans ce cas, la première partie de la formule rend la proposition toujours vraie. Alors, $\boldsymbol{e}$ peut s'écrire comme $\boldsymbol{y}bc\ldots c$ pour un certain mot $\boldsymbol{y}$. On peut alors calculer partiellement l'expression $\neg\boldsymbol{a} \wedge (\boldsymbol{a} +_{\boldsymbol{b}} \neg\boldsymbol{b})$ comme suit.

$$\begin{aligned}
\boldsymbol{a} &= ? \; 0 \; 0 \; \ldots \; 0 \\
\neg\boldsymbol{b} &= ? \; 0 \; 1 \; \ldots \; 1 \\
\boldsymbol{a} +_{\boldsymbol{b}} \neg\boldsymbol{b} &= ? \; ? \; 1 \; \ldots \; 1 \\
\neg\boldsymbol{a} \wedge(\boldsymbol{a} +_{\boldsymbol{b}} \neg\boldsymbol{b}) &= ? \; ? \; 1 \; \ldots \; 1
\end{aligned}$$

D'un autre côté, supposons que le dernier bit de $\boldsymbol{b} \vee (\neg\boldsymbol{a} \wedge(\boldsymbol{a} +_{\boldsymbol{b}} \neg\boldsymbol{b}))$ est 1. Alors, on a soit que le dernier bit du vecteur $\boldsymbol{b}$ est 1, et dans ce cas $\boldsymbol{e}$ est certainement dans L_0, ou que le dernier bit de $\neg\boldsymbol{a} \wedge (\boldsymbol{a} +_{\boldsymbol{b}} \neg\boldsymbol{b})$ est 1. Dans ce cas, on peut assumer que la dernière lettre de $\boldsymbol{e}$ est c, et que le dernier bit de la somme binaire $\boldsymbol{a} +_{\boldsymbol{b}} \neg\boldsymbol{b}$ est 1, ce qui correspond à la somme $0 + 1 = 1$. Dans l'automate d'addition binaire,

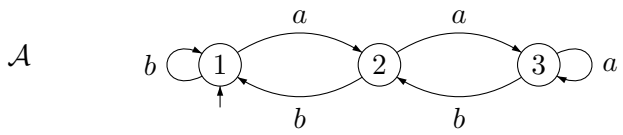


on a que $0 + 1 = 1$ s'il n'y a pas d'occurrence de $1 + 1$ depuis la dernière occurrence de $0 + 0$. Cela ce traduit, dans notre cas, littéralement par "*il n'y a eu aucune occurrence de a depuis la dernière occurrence de b*". ■

Maintenant, le corollaire 5.1 et le lemme 5.1 impliquent le théorème suivant.

Théorème 5.3 *Les états de sortie d'un automate apériodique peuvent toujours être calculés par un algorithme bit-vectoriel.* ■

Exemple 5.7 Reprenons l'automate de l'exemple 5.2:



Le Figure 5.4 donne une décomposition en cascade de l'automate $\mathcal{A}$. On présente ici cette décomposition dans une forme un peu différente mais plus standard: les deux copies de l'automate $\mathcal{B}_2$ ont été confondues, et les transitions dans $\mathcal{B}_2$ ont maintenant un préfixe, correspondant à l'état de $\mathcal{B}_1$ auquel la copie était accrochée.

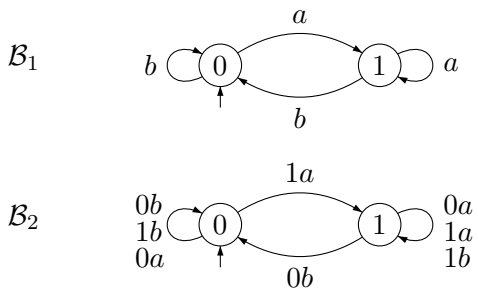


Figure 5.13 Une façon compacte de représenter une décomposition en cascade

Par exemple, dans la Figure 5.13, la transition $0b$ représente la proposition: *l'automate* $\mathcal{B}_1$ *était dans l'état* 0 *et l'événement courant est* b.

Nous avons déjà mentionné, à l'exemple 5.6, que l'automate $\mathcal{A}$ est dans l'état 1 si et seulement si la cascade $\mathcal{B}_1 \circ \mathcal{B}_2$ est dans l'état global $(0, 0)$.

En utilisant le lemme de l'addition (lemme 5.1) on a que l'automate $\mathcal{B}_1$ est dans l'état 0 si et seulement si:

$$\boldsymbol{b} \vee(\neg \boldsymbol{a} \wedge(\boldsymbol{a} +_{\boldsymbol{b}} \neg \boldsymbol{b})), \tag{5.2}$$

et l'automate $\mathcal{B}_2$ est dans l'état 0 si et seulement si:

$$\mathbf{0}\boldsymbol{b} \vee(\neg \mathbf{1}\boldsymbol{a} \wedge(\mathbf{1}\boldsymbol{a} +_{\boldsymbol{b}} \neg \mathbf{0}\boldsymbol{b})). \tag{5.3}$$

Comme le vecteur $\boldsymbol{b}$ implique le vecteur $\neg \boldsymbol{a}$, l'équation 5.2 peut être remplacée par $\boldsymbol{b}$.

Dans l'équation 5.3, on a les deux propositions suivantes:

$\mathbf{0}\boldsymbol{b}$: $\mathcal{B}_1$ était dans l'état 0 et la transition courante est b

$\mathbf{1}\boldsymbol{a}$: $\mathcal{B}_1$ était dans l'état 1 et la transition courante est a

qui se traduisent, respectivement, comme suit:

1) $(\uparrow_1 \boldsymbol{b}) \wedge \boldsymbol{b}$

2) $\neg(\uparrow_1 \boldsymbol{b}) \wedge \boldsymbol{a}$.

En utilisant l'équivalence propositionnelle $\neg\ (\uparrow_1 \boldsymbol{b}) \Leftrightarrow (\uparrow_0 \neg\boldsymbol{b})$, la deuxième proposition devient $\uparrow_0 \boldsymbol{a}\ \wedge\ \boldsymbol{a}$ et on peut réécrire la formule 5.3 comme suit:

$$(\uparrow_1 \boldsymbol{b}\ \wedge\ \boldsymbol{b}) \vee (\neg(\uparrow_0 \boldsymbol{a}\ \wedge\ \boldsymbol{a}) \wedge ((\uparrow_0 \boldsymbol{a}\ \wedge\ \boldsymbol{a}) +_{\boldsymbol{b}} \neg(\uparrow_1 \boldsymbol{b}\ \wedge\ \boldsymbol{b}))).$$

Comme le vecteur $\boldsymbol{b}$ implique $\neg(\uparrow_0 \boldsymbol{a}\ \wedge\ \boldsymbol{a})$, l'équation précédente est équivalente à l'équation suivante:

$$(\uparrow_1 \boldsymbol{b}\ \wedge\ \boldsymbol{b}) \vee (\boldsymbol{b} \wedge ((\uparrow_0 \boldsymbol{a}\ \wedge\ \boldsymbol{a}) +_{\boldsymbol{b}} \neg(\uparrow_1 \boldsymbol{b}\ \wedge\ \boldsymbol{b}))). \tag{5.4}$$

Finalement, en utilisant l'équivalence logique $(p \wedge q) \vee (q \wedge r) \Leftrightarrow q \wedge (p \vee r)$, l'équation 5.4 devient:

$$\boldsymbol{b} \wedge (\uparrow_1 \boldsymbol{b} \vee ((\uparrow_0 \boldsymbol{a}\ \wedge\ \boldsymbol{a}) +_{\boldsymbol{b}} \neg(\uparrow_1 \boldsymbol{b}\ \wedge\ \boldsymbol{b})). \tag{5.5}$$

Donc, étant donné un vecteur d'entrée $\boldsymbol{e}$, on a que l'état de sortie de l'automate $\mathcal{A}$ est 1 si et seulement si l'équation 5.5 est vérifiée et cette équation ne demande que 9 opérations sur les vecteurs de bits; 3 déplacements vers la droite, 3 conjonctions, une négation, une disjonction et une addition binaire.

5.3.3 Lien avec la logique temporelle et complexité

La construction de la section 5.3.2 est, malheureusement, possiblement exponentielle en le nombre de transitions de l'automate. Cela vient du fait que, dans plusieurs cas, certains états de l'automate $\mathcal{A}$ doivent être encodés par un nombre exponentiel de configurations dans la décomposition en cascade de l'automate. Voyons cela sur un exemple.

Exemple 5.8 Soit l'automate suivant:

$\mathcal{A}$

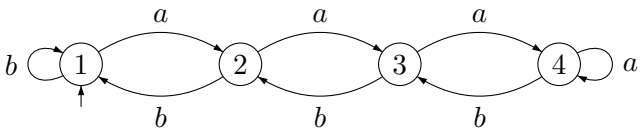
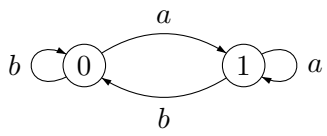


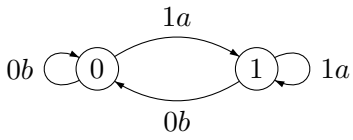
Figure 5.14 Un compteur borné

Cet automate admet la décomposition suivante avec l'homomorphisme $\varphi(000) = 1$, $\varphi(010) = \varphi(100) = \varphi(001) = 2$, $\varphi(110) = \varphi(011) = \varphi(101) = 3$ et $\varphi(111) = 4$. Dans la décomposition seulement les transitions induisant des resets ont été dessinées, les identités ont été omises:

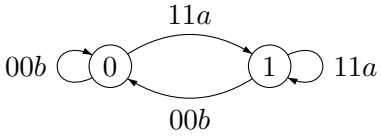
$\mathcal{B}_1$



$\mathcal{B}_2$



$\mathcal{B}_3$



Le comportement global $\mathcal{C}$ du produit en cascade est illustré dans le diagramme suivant:

Dans cet exemple, l'état 2 de l'automate $\mathcal{A}$ correspond aux états globaux 100, 010, 001 de la cascade $\mathcal{C}$ et l'état 3 correspond aux états globaux 110, 101 et 011. Ce nombre exponentiel de configurations pour certains des états de $\mathcal{A}$ implique que la longueur des expressions logiques qui encodent les langages reconnus par ces états peut être exponentielle en le nombre de transitions de $\mathcal{A}$.

Plus généralement, Maler et Pnueli (Maler - Pnueli, 1994) ont démontré que pour un compteur borné $\mathcal{A}$ à n états (et donc ayant $2n$ transitions), la plus petite décomposition en cascade pour $\mathcal{A}$ est une décomposition $\mathcal{C} = \mathcal{B}_1 \ldots \mathcal{B}_n$ ayant $\mathcal{O}(2^n)$ transitions.

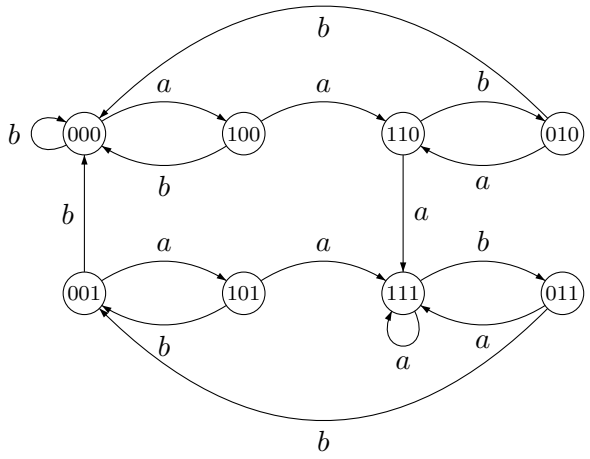


Figure 5.15 $\mathcal{C} = \mathcal{B}_1 \circ \mathcal{B}_2 \circ \mathcal{B}_3$

Une autre façon d'étudier la complexité de nos algorithmes est de faire le lien entre les automates resets et la logique temporelle du passé (Pin, 1997).

Dans (Maler - Pnueli, 1994), Maler et Pnueli définissent les formules logiques suivantes:

Définition 5.7 Soit $\mathcal{A} = (\Sigma, Q, \delta, i, F)$ un automate reset et soit $\Sigma' \subseteq \Sigma$, le sous-ensemble des éléments de l'alphabet d'entrée qui induisent des resets dans $\mathcal{A}$. Pour chaque état $q \in Q$, on définit les formules logiques du passé suivantes:

$$in_q = \bigvee_{\{\sigma \in \Sigma' \,:\, \sigma \text{ entre dans } q\}} \sigma \ ,$$

$$out_q = \bigvee_{\{\sigma \in \Sigma' \,:\, \sigma \text{ sort de } q\}} \sigma \ .$$

Maler et Pnueli démontrent alors la proposition suivante:

Proposition 5.1 *L'ensemble L_q des mots finis dont la lecture se termine à l'état q est exactement l'ensemble des mots satisfaisant la formule logique suivante:*

$$\Phi_q = (\neg out_q)\mathcal{S}(in_q),$$

où l'expression $x\mathcal{S}y$ se traduit par "x depuis y" ($\mathcal{S}$ = since). ■

(On pourraît traduire l'expression $(\neg out_q)\mathcal{S}(in_q)$ par "Il n'y a pas eu de transitions sortant de l'état q depuis le dernier reset vers q".)

La propositon 5.1 implique que

Corollaire 5.2 *Tout langage L_q peut être exprimé par une formule logique du passé de longueur $\mathcal{O}(|\Sigma|)$.* ■

Ce corollaire implique, avec un peu de travail de logique du passé (voir (Maler - Pnueli, 1990)), le corollaire suivant:

Corollaire 5.3 *Tout langage accepté par un automate apériodique à n états peut être exprimé par une formule logique du passé de longueur $\mathcal{O}(2^{nlogn})$.* ■

La deuxième version du lemme de l'addition 5.1 nous donne une reformulation de l'appartenance à un ensemble L_q en termes d'opérations vectorielles. Les résultats négatifs du corollaire 5.3 s'appliquent donc aussi, malheureusement, à nos algorithmes vectoriels.

5.4 Le cas des automates résolubles

Dans cette section, nous allons voir que si $\mathcal{A}$ est un automate résoluble alors on peut construire une décomposition en cascade particulièrement simple de $\mathcal{A}$ qui correspond à un algorithme vectoriel linéaire en le nombre de transitions de $\mathcal{A}$. Tout au long de cette section, nous allons supposer que $\mathcal{A}$ est un automate résoluble avec n états, dont la fonction de transition est δ. Rappelons qu'un automate résoluble (voir la définition 4.3) est un automate pour lequel il existe un étiquetage des états de 1 à n tel que, pour chaque transition b,

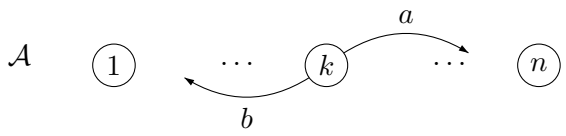
$$\delta(k,b) < k \text{ implique que } \forall k' \geq \delta(k,b), \quad \delta(k',b) = \delta(k,b).$$

Si on pense aux états de $\mathcal{A}$ comme étant les sorties d'un calcul, la propriété ci-dessus signifie que, si la sortie diminue, alors sa valeur ne dépend que de l'entrée et non de l'état courant de l'automate.

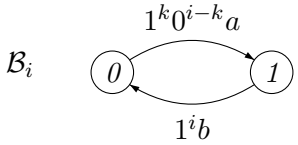
5.4.1 Décomposition en cascades d'un automate résoluble

Dans cette section nous voyons que si un automate est résoluble, alors on peut construire une décomposition en cascade simple pour cet automate. Nous verrons aussi le résultat inverse à savoir que cette décomposition, construite dans ce qui suit, provient nécessairement d'un automate résoluble.

Théorème 5.4 *Soit $\mathcal{A} = (\Sigma, Q, \delta, i, F)$ un automate résoluble à n états. On va supposer que les états sont dans l'ordre de résolubilité et que $Q = \{1, \ldots, n\}$:*



L'automate $\mathcal{A}$ admet une décomposition en cascade de n automates binaires resets $\mathcal{C} = \mathcal{B}_0 \circ \ldots \circ \mathcal{B}_{n-1}$, avec l'homomorphisme[1] *$\varphi(1^k 0^{n-k}) = k$, où chaque $\mathcal{B}_i$ est de la forme:*



avec un reset $1^k 0^{i-k} a$, vers l'état 1, pour chaque transition a et état k dans l'automate $\mathcal{A}$ tel que:

$$\delta(k, a) > i \geq k,$$

et un reset $1^i b$, vers l'état 0, pour chaque transition b et état k dans l'automate $\mathcal{A}$ tel

[1] L'homomorphisme φ implique que l'état global de la cascade correspondant au fait que les k premiers automates $\mathcal{B}_i$ sont dans l'état 1, et que les suivants sont dans l'état 0 est l'image inverse de l'état k de l'automate.

que:

$$\delta(k, b) \leq i < k.$$

En d'autres mots, les transitions de type a, qui *augmentent* la valeur de l'état de sortie dans $\mathcal{A}$, induisent des resets vers l'état 1 dans $\mathcal{B}_i$, et les transitions de type b, qui *diminuent* la valeur de la sortie induisent des resets vers l'état 0.

Remarque: L'automate $\mathcal{B}_0$ de la décomposition restera toujours dans l'état 1 lors de la simulation de l'automate $\mathcal{A}$ par sa décomposition en cascade. $\mathcal{B}_0$ a donc aucun reset.

Pour démontrer le théorème 5.4 nous devons premièrement démontrer que les automates $\mathcal{B}_i$, apparaissant dans la décomposition en cascade d'un automate résoluble, sont bien des automates resets. Nous devons ensuite voir que φ est bien un homomorphisme d'automates de $\mathcal{C}$ vers $\mathcal{A}$. C'est ce que nous allons faire dans le reste de cette section, après avoir donné un exemple de décomposition en cascade d'un automate résoluble.

Exemple 5.9 Voici un automate résoluble $\mathcal{A}$ et sa décomposition en cascade:

Dans la cascade, les identités ont été omises dans les automates $\mathcal{B}_1$ et $\mathcal{B}_2$, pour ne pas alourdir le dessin et les resets dessinés d'une certaine couleur proviennent des transitions de l'automate ayant la même couleur.

Les propriétés élémentaires suivantes de la construction peuvent être facilement vérifiées:

Propriété 1 Une transition a qui augmente l'état de sortie, définie à partir de l'état k induit un reset vers l'état 1 dans chacun des automates de $\mathcal{B}_k$ à $\mathcal{B}_{\delta(k,a)-1}$.

Propriété 2 Une transition b vers l'état $\delta(k, b)$, qui diminue la valeur de l'état de sortie, induit, par résolubilité, un reset vers l'état 0, étiqueté par $1^i b$, dans chacun des automates $\mathcal{B}_i$, pour i de $\delta(k, b)$ à $n - 1$.

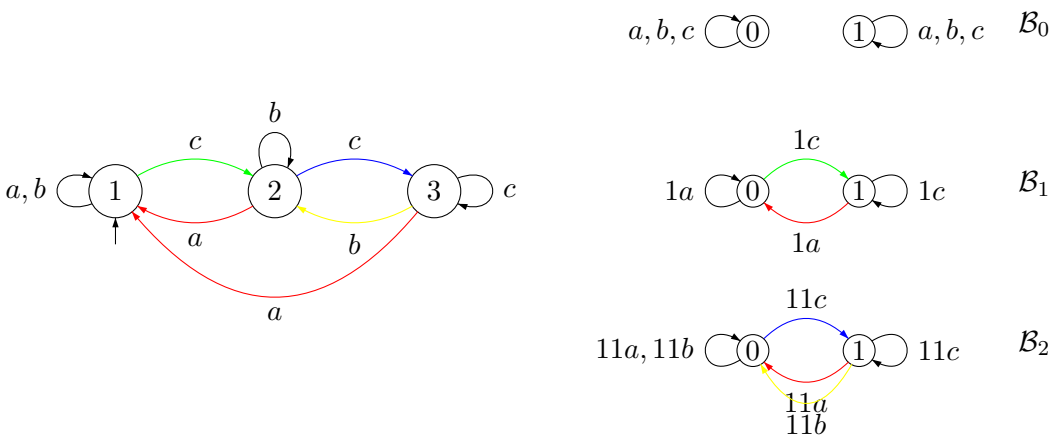
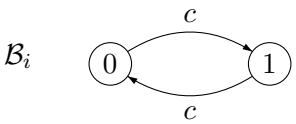


Figure 5.16 Un automate résoluble et sa décomposition en cascade

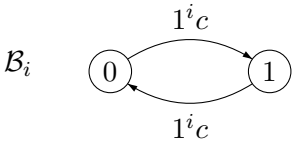
Démontrons maintenant une partie du théorème 5.4, à savoir que les automates $\mathcal{B}_i$, de la décomposition en cascade d'un automate résoluble décrite dans le théorème, sont des resets.

Lemme 5.2 *Pour chaque i, $0 \leq i \leq n-1$, $\mathcal{B}_i$ est un automate reset.*

Preuve: Dans le but de montrer que $\mathcal{B}_i$ est un automate reset, on doit montrer qu'il n'y a aucune transition de la forme



dans les automates $\mathcal{B}_i$. Par construction, une telle transition peut seulement être de la forme suivante:



La transition $1^i c$ de l'état 0 à 1 implique qu'on a une transition c partant de l'état i de $\mathcal{A}$ et se rendant dans un état strictement plus grand que i, c'est-à-dire qu'on a $\delta(i,c) > i$.

La transition $1^i c$ de l'état 1 à 0 implique, quant à elle, qu'il existe un état $j > i$ qui est envoyé sur un état $\leq i$ par une transition c, c'est-à-dire que $\delta(j,c) \leq i$.

L'hypothèse de résolubilité implique alors que

$$\forall k \geq \delta(j,c), \qquad \delta(k,c) = \delta(j,c).$$

Donc, en particulier, comme $i \geq \delta(j,c)$, on devrait avoir que $\delta(i,c) = \delta(j,c) \leq i$ ce qui contredit le fait que $\delta(i,c) > i$. ■

Maintenant, pour démontrer que la fonction φ du théorème 5.4 est un homomorphisme d'automates, on doit commencer par démontrer le lemme suivant:

Lemme 5.3 *Soit* $\delta'(q_0 \dots q_{n-1}, \sigma) = (\delta_0(q_0,\sigma), \dots, \delta_{n-1}(q_{n-1}, (q_0 \dots q_{n-2}, \sigma)))$ *la fonction de transition de la cascade* $\mathcal{C} = \mathcal{B}_0 \circ \dots \circ \mathcal{B}_{n-1}$, *alors*

$$\delta(k,c) = j \text{ } ssi \text{ } \delta'(1^k 0^{n-k}, c) = 1^j 0^{n-j}.$$

Preuve: On va considérer trois cas:

1. $k < j$

   Dans ce cas, si $\delta(k,c) = j$, on a, partant de l'état k, une transition augmentant la valeur de l'état de sortie et par la propriété 1, c induit un reset vers l'état 1 dans chacun des automates de $\mathcal{B}_k$ à $\mathcal{B}_{j-1}$ de la cascade.

   Alors, $\delta'(1^k 0^{n-k}, c) = 1^j 0^{n-j}$. Réciproquement, si $\delta'(1^k 0^{n-k}, c) = 1^j 0^{n-j}$ alors les resets de la figure 5.17 sont définis et, par la propriété 1, ils ne peuvent être définis que si $\delta(k,c) = j$.

2. $k > j$

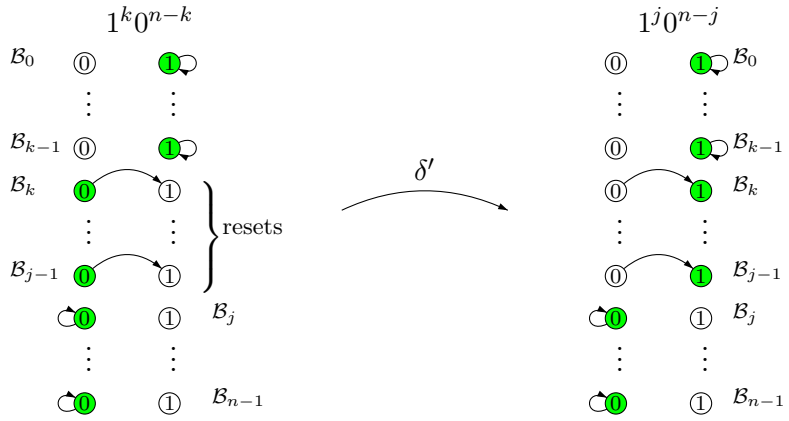


Figure 5.17 Cas $k < j$

Si $\delta(k, c) = j$, on a une transition partant de l'état k vers un état plus petit et alors, par la propriété 2, la transition c induit un reset vers l'état 0 dans chaque automate de $\mathcal{B}_j$ à $\mathcal{B}_{n-1}$ de la cascade:

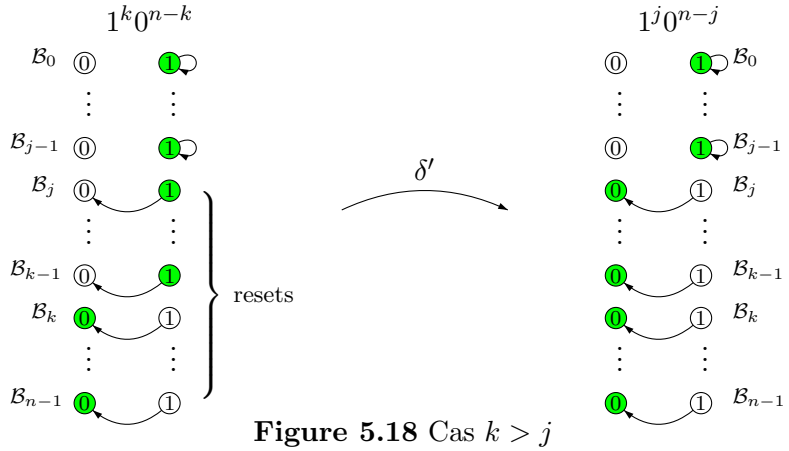


Figure 5.18 Cas $k > j$

Alors, on a bien que $\delta'(1^k0^{n-k}, c) = 1^j0^{n-j}$. Réciproquement, si $\delta'(1^k0^{n-k}, c) = 1^j0^{n-j}$, on a des resets de 1 à 0 dans chacun des automates de $\mathcal{B}_j$ à $\mathcal{B}_{k-1}$, au moins. Alors, la propriété 2 implique que $\delta(k, c) = j$.

3. $j = k$

Dans ce cas, si $\delta(k, c) = k$, la transition c induit seulement des identités dans la cascade, ce qui implique que $\delta'(1^k0^{n-k}, c) = 1^k0^{n-k}$. Réciproquement, si $\delta'(1^k0^{n-k}, c) = 1^k0^{n-k}$, cela signifie que, pour la transition c, aucun reset n'est

défini dans a cascade et alors, on doit avoir que $\delta(k, c) = k$, car toutes les autres possibilités induisent au moins un reset dans la cascade. ■

En utilisant le lemme précédent, on a le finalement le résultat suivant:

Preuve du théorème 5.4: Le lemme 5.3 implique que le sous-automate de $\mathcal{C}$ généré par l'ensemble des états de la forme $1^k0^{n-k}, k \geq 1$, est isomorphe à $\mathcal{A}$ et donc $\mathcal{C}$ est une décomposition en cascade pour $\mathcal{A}$. ■

Pour terminer cette section, on voudrait démontrer que si on a une décomposition de la forme décrite dans le théorème 5.4 alors cette décomposition provient nécessairement d'un automate résoluble.

Définition 5.8 Soit $\mathcal{C} = \mathcal{B}_0 \circ \cdots \circ \mathcal{B}_{n-1}$ une cascade ne comprennant que des automates $\mathcal{B}_i$ binaires resets. Si cette cascade a la propriété de rester toujours dans ses états de la forme 1^k0^{n-k}, pour $1 \leq k \leq n$, lors de la simulation de l'automate $\mathcal{A}$ dont elle provient, alors on dit que la cascade est *jolie*.

Lemme 5.4 *Si on a une jolie cascade $\mathcal{C} = \mathcal{B}_0 \circ \ldots \circ \mathcal{B}_{n-1}$ et une fonction partielle φ telle que $\varphi(1^k0^{n-k}) = k$, pour $k = 1 \ldots n$, alors $(\mathcal{C}, \varphi)$ est la décomposition en cascade d'un automate résoluble ayant n états $\{1, \ldots, n\}$.*

Preuve: On va montrer que si un automate $\mathcal{A}$ n'est pas résoluble alors il est impossible de construire une jolie cascade à partir de $\mathcal{A}$. Si $\mathcal{A}$ n'est pas résoluble alors il existe un certain k pour lequel l'état k n'est pas résoluble dans $\mathcal{A} \setminus \{1, \ldots k-1\}$. Alors, il existe une transition a d'un état s à un état k, où $s > k$, et une transition a d'un état e_1 à un état e_2, où e_1 et $e_2 > k$:

Maintenant, supposons qu'il existe une jolie cascade pour $\mathcal{A}$. Alors, pour rester dans les états de la forme 1^k0^{n-k}, la transition a de l'état s à l'état $k < s$ dans $\mathcal{A}$ implique les transitions suivantes dans la cascade:

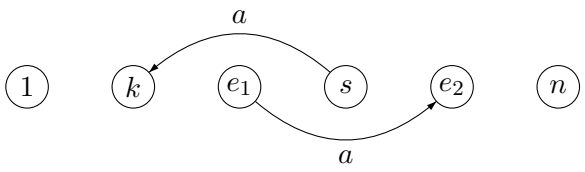


Figure 5.19 Un automate non-résoluble $\mathcal{A}$

1. On boucle sur l'état 1 dans les automates $\mathcal{B}_i$, pour i tel que $0 \leq i < k$, avec une transition de la forme $1^i a$.

2. On a un reset $1^i a$ de l'état 1 à l'état 0 dans les automate $\mathcal{B}_i$, pour i tel que $k \leq i < s$.

Maintenant, la transition de l'état e_1 à l'état e_2 dans $\mathcal{A}$, où $e_1 < e_2$ et $e_1, e_2 > k$ implique, comme la cascade est jolie, les transitions suivantes dans la cascade:

3. On boucle sur l'état 1 dans les automates $\mathcal{B}_i$, pour i tel que $0 \leq i < e_1$, avec une transition de la forme $1^i a$.

4. On a un reset $1^{e_1}0^{i-e_1}a$ de l'état 0 à l'état 1 dans les automate $\mathcal{B}_i$, pour i tel que $e_1 \leq i < e_2$.

Les transitions 2. et 3. sont alors contradictoires dans les automates $\mathcal{B}_i$ pour i tel que $k \leq i < e_1$, et donc notre cascade ne peut être jolie. ■

5.4.2 Un algorithme bit-vectoriel linéaire

La structure simple des automates $\mathcal{B}_i$, de la décomposition d'un automate résoluble, va nous permettre de construire un algorithme bit-vectoriel linéaire pour $\mathcal{A}$ en utilisant le lemme 5.1. Considérons les propositions logiques suivantes:

P_i : L'automate $\mathcal{A}$ va dans l'état i.
Q_i : L'automate $\mathcal{B}_i$ va dans l'état 0.

Pour i dans $[1..n-1]$, le théorème 5.4 implique que P_i est équivalent à $Q_i \wedge \neg Q_{i-1}$, et P_n est simplement $\neg Q_{n-1}$. Alors, si nous connaissons la valeur des Q_i, on peut calculer la sortie de $\mathcal{A}$ en $\mathcal{O}(n)$ étapes.

Pour un automate $\mathcal{B}_i$ dans le produit en cascade, on commence par former la disjonction de tous ses resets. On utilise les notations suivantes:

$$\begin{aligned} a_i &= \bigvee_{\delta(k,a)>i\geq k} 1^k 0^{i-k} a \\ b_i &= \bigvee_{\delta(k,b)\leq i<k} 1^i b \end{aligned}$$

Du lemme de l'addition, on a alors la formule suivante:

$$\boldsymbol{Q_i = b_i \vee(\neg a_i \wedge(a_i +_b \neg b_i))}.$$

La partie un peu plus difficile est que, puisque les disjonctions se font sur des paires d'états et de transitions, calculer chacun des vecteurs $\boldsymbol{a_i}$ et $\boldsymbol{b_i}$ peut demander $\mathcal{O}(mn)$ étapes, où m est la cardinalité de l'alphabet d'entrée pour $\mathcal{A}$. Alors, le temps total du calcul serait de $\mathcal{O}(mn^2)$.

Heureusement, il existe des formules récursives pour calculer les $\boldsymbol{a_i}$ et les $\boldsymbol{b_i}$, qui réduisent le temps total du calcul à $\mathcal{O}(mn)$. Ici, I représente l'état initial de l'automate $\mathcal{A}$.

Lemme 5.5 *Les vecteurs $\boldsymbol{a_i}$ et $\boldsymbol{b_i}$, pour $i \geq 1$ sont donnés par:*

$$\boldsymbol{a_i} = (\neg \boldsymbol{Q'_{i-1}} \wedge(\bigvee_{\delta(i,a)>i} \boldsymbol{a}))\vee(\boldsymbol{a_{i-1}} \wedge \boldsymbol{Q'_{i-1}} \wedge \neg(\bigvee_{\delta(k,a)=i>k} (\uparrow_{i=k} \boldsymbol{P_k} \wedge \boldsymbol{a})))$$

$$\boldsymbol{b_i} = (\neg \boldsymbol{Q'_{i-1}} \wedge(\boldsymbol{b_{i-1}} \vee \bigvee_{\delta(k,b)=i<k} \boldsymbol{b}))$$

où $\boldsymbol{Q'_{i-1}} = \uparrow_{i>I} \boldsymbol{Q_{i-1}}$.

Preuve: La formule pour la disjonction, a_i, des resets de la forme $1^k 0^{i-k} a$ dans l'automate $\mathcal{B}_i$, peut être séparée en deux, dépendamment si $k = i$ ou non:

$$a_i = \left(\bigvee_{\delta(i,a)>i} 1^i a\right) \vee \left(\bigvee_{\delta(k,a)>i>k} 1^k 0^{i-k} a\right).$$

Les transitions de la forme $1^i a$ signifient que l'état précédent de l'automate $\mathcal{A}$ est au moins i, et alors l'état précédent de $\mathcal{B}_{i-1}$ doit être 1. On doit donc avoir que, $\neg(\uparrow_{i>I} \boldsymbol{Q_{i-1}})$ est vrai, où la valeur booléenne $i > I$ est là pour que tout se passe bien avec l'état initial. (En effet, au départ de la lecture de l'entrée, on est dans l'état initial I et si $i > I$ alors B_{i-1} est dans l'état 0.) La première partie de la disjonction devient alors

$$(\neg \boldsymbol{Q'_{i-1}} \quad \wedge \quad \left(\bigvee_{\delta(i,a)>i} \boldsymbol{a} \right)).$$

Dans la deuxième partie de la disjonction, les transitions de la forme $1^k 0^{i-k} a$, avec $k < i$, signifient que l'état précédent est strictement plus petit que i, ce qui est équivalent à la formule $(\uparrow_{i>I} \boldsymbol{Q_{i-1}})$, et toutes les transitions qui étaient dans $\boldsymbol{a_{i-1}}$ sont aussi dans $\boldsymbol{a_i}$ sauf celles pour lesquelles $\delta(k, a) = i$. Maitenant, la formule pour la disjonction des resets de la forme $1^i b$ dans l'automate $\mathcal{B}_i$ est la suivante:

$$b_i \quad = \quad \bigvee_{\delta(k,b)\leq i<k} 1^i b.$$

Les transitions de la forme $1^i b$ signifient que l'état précédent de l'automate $\mathcal{A}$ est au moins i, et alors l'état précédent de $\mathcal{B}_{i-1}$ doit être 1. On doit donc avoir que, $\neg(\uparrow_{i>I} \boldsymbol{Q_{i-1}})$ est vrai. De plus, toutes les transitions qui étaient dans $\boldsymbol{b_{i-1}}$ sont aussi dans $\boldsymbol{b_i}$ sauf celles pour lesquelles $\delta(k, b) = i < k$. Ces faits impliquent la formule suivante pour $\boldsymbol{b_i}$:

$$\boldsymbol{b_i} \quad = \quad (\neg \boldsymbol{Q'_{i-1}} \wedge (\boldsymbol{b_{i-1}} \vee \bigvee_{\delta(k,b)=i<k} \boldsymbol{b}))$$

■

Soit $\mathcal{A}$ un automate résoluble d'états $\{1, \ldots, n\}$. Pour le calcul de la sortie $\boldsymbol{r} = r_1 \ldots r_m$, des états visités par $\mathcal{A}$ lors de la lecture d'un vecteur d'entrée $\boldsymbol{e} = e_1 \ldots e_m$, on obtient finalement l'algorithme linéaire suivant:

On commence par trouver les positions du vecteur de sortie correspondantes au premier état résoluble de l'automate, c'est-à-dire l'état 1 en nous servant de la proposition 4.1:

$$\boldsymbol{(r = 1)} \quad = \quad \boldsymbol{P_1} = Ind(1, \boldsymbol{e \in I_1}, \boldsymbol{e \in L_1}).$$

Maintenant, pour i de 2 à $n-1$, on calcule

$$\boldsymbol{a_i} \leftarrow (\neg \boldsymbol{Q'_{i-1}} \wedge (\bigvee_{\delta(i,a)>i} \boldsymbol{a})) \vee (\boldsymbol{a_{i-1}} \wedge \boldsymbol{Q'_{i-1}} \wedge \neg (\bigvee_{\delta(k,a)=i>k} (\uparrow_{i=k} \boldsymbol{P_k} \wedge \boldsymbol{a})))$$

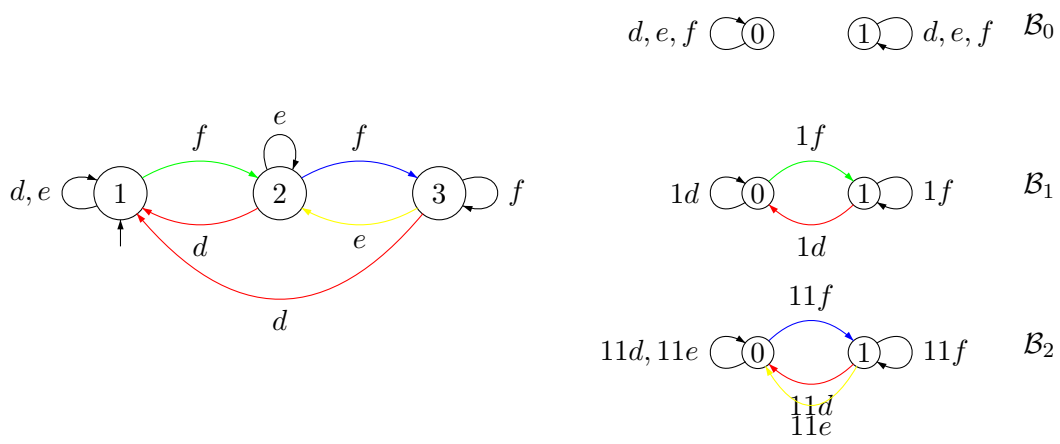
$$\boldsymbol{b_i} \leftarrow (\neg \boldsymbol{Q'_{i-1}} \wedge (\boldsymbol{b_{i-1}} \vee \bigvee_{\delta(k,b)=i<k} \boldsymbol{b}))$$

$$\boldsymbol{Q_i} \leftarrow \boldsymbol{b_i} \vee (\neg \boldsymbol{a_i} \wedge (\boldsymbol{a_i} +_{\boldsymbol{b}} \neg \boldsymbol{b_i}))$$

$$\boldsymbol{(r=i)} \leftarrow \boldsymbol{Q_i} \wedge \neg \boldsymbol{Q_{i-1}}$$

Finalement, le vecteur $\boldsymbol{P_n} = \boldsymbol{(r=n)}$ est donné tout simplement par $\neg \boldsymbol{Q_{n-1}}$.

Exemple 5.10 Reprenons l'automate résoluble $\mathcal{A}$ de la figure 5.16 et sa décomposition en cascade:



Supposons que l'entrée est $\boldsymbol{e} = deefedffe$. On commence par calculer les positions du vecteur de sortie correspondantes à l'état 1 de l'automate $\mathcal{A}$:

$$\begin{aligned}(\boldsymbol{r=1}) &= Ind(1, \boldsymbol{e} \in \boldsymbol{I_1}, \boldsymbol{e} \in \boldsymbol{L_1}) \\ &= \boldsymbol{d} \vee [\boldsymbol{e} \wedge (\neg \boldsymbol{d} +_{\boldsymbol{b}} \neg(\boldsymbol{d} \vee \boldsymbol{e}))] \\ &= 111001000.\end{aligned}$$

Maintenant, pour trouver le vecteur $\boldsymbol{P_2} = \boldsymbol{(r=2)}$, on doit faire les calculs vectoriels suivants:

$$\boldsymbol{a_2} = (\neg \boldsymbol{Q_1'} \wedge (\bigvee_{\delta(2,a)>2} \boldsymbol{a})) \vee (\boldsymbol{a_1} \wedge \boldsymbol{Q_1'} \wedge \neg(\bigvee_{\delta(k,a)=2>k} (\uparrow_{2=k} \boldsymbol{P_k} \wedge \boldsymbol{a})))$$

$$\begin{aligned}&= (000011011 \wedge \boldsymbol{f}) \vee (\boldsymbol{f} \wedge 111100100 \wedge \neg(\uparrow_0 \boldsymbol{P_1} \wedge \boldsymbol{f})) \\ &= 000000010\end{aligned}$$

$$
\begin{aligned}
\boldsymbol{b_2} &= (\neg \boldsymbol{Q'_1} \wedge (\boldsymbol{b_1} \vee \bigvee_{\delta(k,b)=2<k} \boldsymbol{b})) \\
&= 000011011 \wedge (\boldsymbol{d} \vee \boldsymbol{e}) \\
&= 000011001 \\
\boldsymbol{Q_2} &= \boldsymbol{b_2} \vee (\neg \boldsymbol{a_2} \wedge (\boldsymbol{a_2} +_{\boldsymbol{b}} \neg \boldsymbol{b_2})) \\
&= 111111101 \\
\boldsymbol{(r=2)} &= \boldsymbol{Q_2} \wedge \neg \boldsymbol{Q_1} \\
&= 000110101.
\end{aligned}
$$

Finalement, le vecteur $\boldsymbol{P_3} = \boldsymbol{(r=3)}$ est donné tout simplement par $\neg \boldsymbol{Q_2} = 000000010$.

5.4.3 Analyse de complexité

On veut étudier le temps de calcul des vecteurs $\boldsymbol{a_i}$ et $\boldsymbol{b_i}$ du lemme 5.5. Par définition, a_i est l'ensemble des resets, de l'état 0 vers l'état 1, de l'automate $\mathcal{B}_i$. a_i comprend donc un reset de la forme $1^k 0^{i-k} a$ pour chaque transition a d'un état $\leq i$ vers un état plus grand que i. On a donc possiblement $\mathcal{O}(mn)$ resets de ce type. De manière équivalente, on remarque que b_i comprend un reset de la forme $1^i b$, de l'état 1 à l'état 0, chaque fois qu'on a une transition b d'un état plus grand que i vers un état $\leq i$.

Ce qu'il faut noter est que dans les formules du lemme 5.5 pour $\boldsymbol{a_i}$ et $\boldsymbol{b_i}$ une transition vers un état plus grand $\delta(k,a)$ dans l'automate $\mathcal{A}$ génère seulement deux termes, un dans $\mathbf{a_k}$, et un dans $\mathbf{a}_{\delta(\mathbf{k},\mathbf{a})}$. Chaque transition vers un état plus petit $\delta(k,b)$ génère seulement un terme dans $\mathbf{b}_{\delta(\mathbf{k},\mathbf{b})}$. Alors, le temps de calcul total est bien de $\mathcal{O}(mn)$, même si l'effort de calcul pour certains des $\mathbf{a_i}$ peut aussi être de $\mathcal{O}(mn)$.

CONCLUSION

Nous avons établi que si un automate est apériodique alors il existe un algorithme bit-vectoriel pour cet automate résolvant le problème du cheminement. Ce problème consiste, étant donné une séquence d'entrée, à trouver la suite des états visités par l'automate lors de la lecture de l'entrée. Nous avons vu que, malheureusement, ces algorithmes, découlant de la décomposition en cascade de l'automate, sont souvent exponentiels en le nombre de transitions de l'automate. D'un autre côté, nous avons démontré que, pour une sous-classe des automates apériodiques, les automates résolubles, il existe des algorithmes bit-vectoriels qui sont linéaires en le nombre de transitions de l'automate. Cela nous amène à nous poser la question suivante. Est-ce que la classe des automates résolubles est la seule sous-classe des automates apériodiques pour laquelle il existe des algorithmes vectoriels linéaires? L'étude d'autres décompositions d'automates, par exemple la décomposition bilatérale, nous donnera certainement une réponse, du moins partielle, à cette question.

Une autre approche serait l'étude des propriétés algébriques des automates résolubles dans le but de dégager les spécificités algébriques de cette sous-classe des automates apériodiques qui la rendent particulière. Il est intéressant de mentionner ici qu'Olivier Serre, au cours de son DEA sous la direction de Anca Muscholl et de Jean-Éric Pin, a étudié les langages reconnus par les automates résolubles et a démontré, à l'aide de cette étude, que si on possède un algorithme vectoriel pour un automate, alors cet automate est nécessairement apériodique. De plus, il a démontré qu'il est facile de construire un algorithme vectoriel pour la sous-classe des automates apériodiques composée des automates dont le langage est un langage de niveau 3/2 de la hiérarchie de Straubing-Thérien (Pin, 1997). (Soit A, un alphabet. Les langages de niveau 3/2 sur A^* sont les unions finis de langages de la forme $A_0^* a_1 A_1^* a_2 \dots a_k A_k^*$, où $a_i \in A$ et où $A_i \subseteq A$.)

Dans un autre ordre d'idée, nous avons ensuite démontré que des automates résolubles sont au coeur de la résolution du problème de la recherche des occurrences approximatives d'un mot dans un texte. Nous en avons déduit un algorithme vectoriel linéaire pour résoudre ce problème avec l'utilisation d'une distance d'édition généralisée. Même si la distance d'édition généralisée est mieux adaptée aux impératifs biologiques que la distance d'édition (qui était la distance utilisée auparavant dans la plupart des algorithmes de recherche) les biologistes préfèrent utiliser une notion de similarité pour comparer leurs séquences. De plus, les biologistes ont besoin d'un algorithme permettant de travailler avec une notion qui permet l'insertion et la suppression de longues suites de caractères, pour ainsi tenir compte, par exemple, de mutations survenues en cours d'évolution. L'adaptation de nos méthodes à la recherche approximative utilisant une notion de similarité et permettant l'insertion et la suppression de suite de caractères est donc, très certainement, une avenue intéressante d'expansion pour nos recherches.

Une autre généralisation possible de notre travail est la recherche de toutes les occurrences approximatives de motifs (ou sous-arbres) dans un arbre. Ce problème est crucial en biologie étant donné que la structure secondaire de l'ARN est topologiquement équivalente à celle d'un arbre ordonné. Comme nous savons que des séquences d'ARN différentes peuvent produire, en se repliant, des structures secondaires similaires, il est important de pouvoir comparer ces structures de façon à comprendre les fonctions relatives aux différents ARN.

Finalement, plusieurs domaines de la bioinformatique utilisent une approche de programmation dynamique pour résoudre leurs problèmes. Pour ne donner qu'un exemple, un problème qui est d'actualité présentement est la recherche de facteurs de transcription à l'aide d'empreintes d'ADN. Intégrer notre approche vectorielle à ces algorithmes dynamiques augmenterait grandement leur efficacité.

Bibliographie

Aho, A. « Algorithms for Finding Patterns in Strings », *in Handbook of Theoretical Computer Science*, Vol. A, Elsevier, pp. 255–300, 1990.

Altschul, S.F., Gish, W., Miller, W., Myers, E.W. et Lipman, D.J. « Basic local alignment search tool. (BLAST) », *Journal of Molecular Biology*, 215-3, pp. 403–410, 1990.

Arbib, M.A. « Algebraic Theory of Machines, Languages, and Semigroups », *Academic Press*, 1968.

Allauzen, C., Crochemore, M. et Raffinot, M. « Efficient experimental string matching by weak factor recognition », *Combinatorial Pattern Matching 2001*, LNCS 2089, pp. 51–72, 2001.

Arlazarov, V.L., Dinic, E.A., Kronrod, M.A. et Faradzev, I.A. « On economic construction of the transitive closure of a directed graph », *Dokl. Akad. Nauk. SSSR*, 194, pp. 487–488 1970 (en Russe). Tranduction anglaise dans *Soviet Math. Dokl.*, 11, pp. 1209–1210, 1975.

Baeza-Yates, R.A. et Gonnet, G.H. « A New Approach to Text Searching », *Communications of the ACM*, 35, pp. 74–82, 1992.

Baeza-Yates, R.A. et Gonnet, G.H. « A Faster Algorithm for Approximate String Matching », *Proceedings of the 7th Symposium on Combinatorial Pattern Matching*, LNCS 1075, pp. 1–23, 1996.

Bergeron, A. et Hamel, S. « Fast Implementation of Automata Computations », *Implementation and Application of Automata*, LNCS 2088, pp. 47–56, 2001.

Bergeron, A. et Hamel, S. « Vector Algorithms for Approximate String Matching », *International Journal of Foundations of Computer Science*, 13-1, pp.53–66, 2002.

Bergeron, A. et Hamel, S. « Cascade Decompositions are Bit-Vector Algorithms », *CIAA01, International Conference on Automata Computation*, À paraître dans Lecture Note in Computer Science, 15 pages, 2002.

Bergeron, F., Labelle, G. et Leroux, P. « Combinatorial Species and Tree-like Structures », *Encyclopedia of Mathematics and its Applications*, 1997.

Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T. et Seiferas, J. « The smallest automaton recognizing the subwords of a text. », *Theoritical Computer Science*, 40(1), pp. 31–55, 1985.

Boyer, R. et Moore, S. « A fast string searching algorithm », *Communications of the ACM*, 20, pp. 762–772, 1977.

Calude, C.S., Salomaa, K. et Yu, S. « Metric Lexical Analysis », *Automata Implementation*, LNCS 2214, pp. 48–59, 1999.

Collins and al. « Initial sequencing and analysis of the human genome », *Nature*, 409, pp. 860–921, 2001.

Crochemore, M. « Transducers and repetitions », *Theoritical Computer Science*, 45, pp. 63–86, 1986.

Crochemore, M. « String matching with constraints », *Proc. MFCS'88*, LNCS 324, pp. 44–58, 1988.

Crochemore, M., Hancart, C. et Lecroq, T. « Algorithmique du texte », *Vuibert*, 347 pages, 2001.

Dayhoff, M.O. « Survey of new data and computer methods of analysis », *Atlas of protein sequence and structure*, vol.5 suppl. 3, National Biomedical Research Foundation, Georgetown University, Washington D.C., 1978.

Galil, Z. et Park, K. « An Improved Algorithm for Approximate String Matching », *SIAM J. Comput.*, 19-6, pp. 989–999, 1990.

Gonnick, L. et Wheelis, M. « The cartoon guide to genetics », *Harper Perennial*, 1991.

Gusfield, D. « Algorithms on Strings, Trees and Sequences », *Cambridge University Press*, 1997.

Harel, D. et Tarjan, R.E. « Fast algorithms for finding nearest common ancestors », *SIAM J. Comput.*, 13 no.2, pp. 338–355, 1984.

Hennikoff, S. et Hennikoff, J.G. « Amino acid substitution matrices from protein blocks », *Proc. Natl. Acad. Sci.*, 89, pp. 10 915–10 919, 1992.

Holub, J. et Melichar, B. « Implementations of Nondeterministic Finite Automata for Approximate Pattern Matching », *in Automata Implementation*, LNCS 1660, Springer-Verlag, 1999.

Hopcroft, J.E. et Ullman, J.D. « Introduction to Automata Theory, Languages, and Computation », *Addison-Wesley Publishing Company*, 1979.

Kashyap, R.L. et Oommen, B.J. « Spelling correction using probabilistic methods », *Pattern Recognition Letters 2*, 3, pp. 147–154, 1984.

Krohn, K. et Rhodes, J. « Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines », *Trans. Am. Math. Soc.*, 116, pp. 450–464, 1965.

Landau, G.M. et Vishkin, U. « Fast String Matching with k Differences », *Journal of Computer and System Sciences*, 37, pp. 63–78, 1988.

Landau, G.M. et Vishkin, U. « Fast parallel and serial approximate string matching », *J. Algorithms*, 10, pp. 157-169, 1989.

Levenshtein « Binary codes capable of correcting insertions and reversals », *Sov. Phys. Dokl.*, 10, pp. 707–710, 1966.

Li, W. et Graur, D. « Fundamentals of molecular evolution », *Sinauer Associate*, Sunderland, Massachusetts, 1991.

Maler, O. et Pnueli, A. « Tight Bounds on the Complexity of Cascaded Decomposition Theorem », *31st Annual Symposium on Foundations of Computer Science IEEE, volume II*, pp. 672–682, 1990.

Maler, O. et Pnueli, A. « On the Cascaded Decomposition of Automata, its Complexity and its Application to Logic », *document non publié*, 48 pages, 1994.

Masek, W.J. et Paterson, M.S. « A Faster Algorithm Computing String Edit Distance », *Journal of Computer and System Sciences*, 20, pp. 18–31, 1980.

Mateescu, A., Salomaa, A., Salomaa, S. et Yu, S. « Lexical analysis with a simple finite-fuzzy-automaton model », *Journal of Universal Computer Science*, 1-5, pp. 292–311, 1995.

McNaughton R. et Papert, S. « Counter-Free Automata », *The M.I.T. Press*, 1971.

Mount, D.W. « Bioinformatics: Sequence and Genome Analysis », *Cold Spring Harbor Laboratory Press*, 2001.

Myers, G. « A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming », *Journal of the ACM*, 46-3, pp. 395–415, 1999.

Needleman, S.B. et Wunsch, C.D. « A general method applicable to the search for similarities in the amino acid sequence of two proteins », *Journal of Molecular Biology*, 48, pp. 443–453, 1970.

Pin, J.E. « Syntactic Semigroups », *in Handbook of Formal Languages*, Vol.1, Springer, pp. 697–738, 1997.

Stanley, R.P. « Enumerative Combinatorics, Volume 1 », *Cambridge University Press*, 1997.

Schieber, B. et Vishkin, U. « On finding lowest common ancestors: simplifications and parallelization », *SIAM J.Compt*, 17, pp. 1253–1262, 1988.

Sellers, P.H. « An algorithm for the distance between two finite sequences », *Journal of Combinatorial Theory*, 16, pp. 253–258, 1974.

Sellers, P.H. « The Theory and Computation of Evolutionary Distances: Pattern Recognition », *Journal of Algorithms*, 1, pp. 359–373, 1980.

Serre, O. « Implémentations d'automates finis par des opérations vectorielles », *rapport de DEA, non publié*, 73 pages, 2001.

Smith, T.F. et Waterman, M.S. « Identification of common molecular subsequences », *Journal of Molecular Biology*, 147, pp. 195–197, 1981.

Smith, T.F. et Waterman, M.S. « Comparison of biosequences », *Adv. in Applied Mathematics*, 2, pp. 482–489, 1981.

Stern, J. « Complexity of some Problems from the Theory of Automata », *Information and Control*, 66, pp. 163–176, 1985.

Straubing H. « Finite Automata, Formal Logic and Circuit Complexity », *Birkhäuser*, 1994.

Strauss, B.S. « Book review: DNA repair and mutagenesis », *Science*, 270, pp. 1511–1513, 1995.

Tarhio, J. et Ukkonen, E. « Boyer-Moore approach to approximate string matching », *Proc. SWAT'90*, LNCS 447, pp. 348–359, 1990.

Ukkonen, E. « On Approximate String Matching », *Proc. Int. Conf. Found. Comp. Theor.*, LNCS 158, Springer-Verlag, pp. 487–495, 1983.

Ukkonen, E. « Finding Approximate Patterns in Strings », *Journal of Algorithms*, 6, pp. 132–137, 1985.

Ukkonen, E. « On-Line Construction of Suffix Trees », *Algorithmica*, 14, pp. 249–260, 1995.

Ukkonen, E. et Wood, D. « Approximate String Matching with Suffix Automata », *Algorithmica*, 10, pp. 353–364, 1993.

Venter, J.C. and al. « The Sequence of the Human Genome », *Science*, 291, pp. 1304–1351, 2001.

Wagner, R.A. and Fischer M.J. « The String-to-String Correction Problem », *Journal of the Association for Computing Machinery*, 21, pp. 168–173, 1974.

Weiner, P. « Linear pattern matching algorithm », *Proceedings 14th IEEE Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

Wu, S. et Manber, U. « Fast Text Searching Allowing Errors », *Communications of the*

ACM, 35, pp. 83–91, 1992.

Wu, S., Manber, U. et Myers, G. « A Subquadratic Algorithm for Approximate Limited Expression Matching », *Algorithmica*, 15, pp. 50–67, 1996.

Zeiger, H.P. « Cascade Synthesis of Finite-State Machines », *Information and Control*, 10, pp. 419–433, 1967.