Rapport de Projet de stage

Le caractère limité des automates finis

Brebant Alexandre
Xue Juedong

10 août 2014

Préambule

Durant ce stage, nous avons voulu approfondir nos connaissances dans une branche de l'informatique théorique que nous avons étudiée lors de ce dernier semestre de licence. Ce sont donc les automates et les interrogations qui gravitent autour de cette structure qui nous ont intéressés.

Marc Zeitoun, notre enseignant sur ce domaine au dernier semestre, nous a proposé un sujet traitant d'un problème réputé difficile concernant les automates : déterminer leur caractère limité, ou *limitedness*[1] Il s'agit d'un problème qui a émergé très tôt en raison de ses liens avec une question fondamentale, celle de la hauteur d'étoile, qui se formalise ainsi : peut-on calculer le nombre minimal d'étoiles imbriquées permettant d'exprimer un langage rationnel donné ?

Ce problème, ainsi que la question de *limitedness*, ont fait l'objet de recherches sur une longue période avant d'être tous deux résolus. Les articles majeurs s'étendent de 1982 à 2005 [4, 8, 9, 5], ce qui témoigne de l'intérêt des chercheurs.

En raison du caractère difficile du problème au niveau licence, Marc Zeitoun nous a expliqué qu'il était préférable de travailler sur ce sujet à 2 étudiants, plutôt que l'un d'entre nous s'y consacre et l'autre travaille sur un second sujet. Nous avons travaillé de la façon suivante : le stage s'est déroulé sur 6 semaines, avec une réunion par semaine. Entre ces réunions, nous avons travaillé ensemble, d'une part pour comprendre l'algorithme proposé, puis pour l'implémenter. La rédaction du rapport a commencé en fin de stage et s'est terminée début Août. Même si nous avons travaillé chacun sur les deux aspects, Alexandre Brebant a travaillé plus particulièrement sur la partie théorique, et Juedong Xue sur l'implémentation. Avec l'accord de Marc Zeitoun, nous avons écrit un mémoire un peu plus long que ceux attendus (23 pages au lieu de 15), ce qui est justifié par le fait que nous avons travaillé à 2. L'ensemble, rapport et code, est disponible sous `git` à l'adresse

`https://github.com/Keijy/distance_automata_limitedness`

La présentation de ce mémoire est la suivante : dans une première section, nous expliquons le problème et son intérêt. Dans la seconde section, nous présentons l'algorithme de résolution, essentiellement dû à Simon et Leung. Dans la dernière partie, nous présentons la réalisation logicielle que nous avons écrite.

1. Nous emploierons parfois ce terme anglais dans le rapport, en raison de l'usage courant qui en est fait.

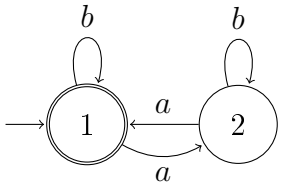
Table des matières

Chapitre 1

Introduction : le problème du caractère limité, ou limitedness

1.1 Les automates finis

Un automate est une machine abstraite qui va accepter ou rejeter des mots écrits sur un alphabet fini. Voici l'exemple d'un automate très basique qui accepte tous les mots sur l'alphabet $\{a, b\}$ qui comportent un nombre pair de a.



Pour représenter graphiquement un automate, on utilise ainsi un graphe orienté dans lequel les sommets du graphe sont les *états* de l'automate et où chaque arrête correspond à une *transition* de l'automate. Une transition permet de changer d'état à la lecture d'une lettre. Un mot est *accepté* par un automate $\mathcal{A}$ s'il existe un chemin dans celui-ci, partant d'un état initial (marqué sur la figure par la flèche entrante dans l'état 1) et allant jusqu'à un état final (marqué par de double cercle dans l'état 1) en lisant les lettres une par une. Un tel chemin est appelé *chemin acceptant* et l'ensemble des mots acceptés par un automate est appelé *Langage* de l'automate $\mathcal{A}$, que l'on note $L(\mathcal{A})$.

Formellement, un automate est défini par 5 ensembles de la façon suivante :

$$\mathcal{A} = \{A, Q, I, F, \delta\} \text{ avec}$$

A : l'alphabet,
Q : l'ensemble des états,
I : l'ensemble des états initiaux,
F : l'ensemble des états finaux,
δ : l'ensemble des transitions.

Nous pouvons donc définir l'automate ci-dessus de cette manière :

$$\mathcal{A} = \left(A = \{a, b\},\ Q = \{1, 2\},\ I = \{1\},\ F = \{1\},\ \delta = \left\{ \begin{array}{lcl} (1, a) & \to & 2 \\ (2, a) & \to & 1 \\ (1, b) & \to & 1 \\ (2, b) & \to & 2 \end{array} \right. \right)$$

La théorie des automates et des langages est un domaine très actif depuis le milieu du 20e siècle et il existe aujourd'hui plusieurs modèles d'automates nés pour répondre à certaines problématiques.

Ainsi les automates permettent de caractériser certains langages, par exemple, les automates finis qui sont le modèle de base des automates, définissent ce qu'on appelle les langages rationnels. Voir par exemple [1] pour une introduction. Mais il existe des extensions à ce modèle, comme par exemple les automates à pile qui permettent de définir les langages algébriques, qui sont grandement utilisés en analyse syntaxique par exemple.

1.2 Les automates à distance

C'est une autre extension des automates finis qui va nous intéresser dans ce rapport : les automates à distance (*distance automata* en anglais). Ce sont des automates finis, possiblement non déterministes, dans lesquels les transitions sont *pondérées*. Cela signifie que l'on associe un coût pour chaque mot accepté par l'automate. Ce coût est calculé de la façon suivante :

- Chaque transition a un coût, qui est un entier naturel.
- Le coût d'un chemin s'obtient en faisant la *somme* des coûts de chacune de ses transitions.
- Le coût d'un mot correspond au *minimum* des coûts de tous ses chemins acceptants dans l'automate.

On appelle ces automates *automates à distance*, ou *automate à coûts*.

Ces automates sont très naturels car les coûts permettent de modéliser facilement de nombreux problèmes d'optimisation de ressources : le coût peut représenter par exemple une énergie nécessaire pour « exécuter » le mot, ou bien un coût financier. Dans ce contexte, il est donc naturel de chercher à ajouter les coûts le long d'un chemin, et de minimiser le coût obtenu sur tous les chemins qui acceptent le mot.

Reprenons l'automate représenté plus haut en y ajoutant des coûts sur les transitions, comme indiqué en FIGURE 1.1. Que calcule cet automate ? Comme cet automate accepte les mots qui ont un nombre pair de a, le coût d'un mot non accepté, donc ayant un nombre impair de a, est ∞ par convention. Par ailleurs, le coût d'une lettre b est nul, et seules les lettres a en position impaires coûtent 1. Cet automate calcule donc, pour un mot ayant un nombre pair de a, ce nombre divisé par 2. Par exemple, le coût de *ababbbaab* est 2.

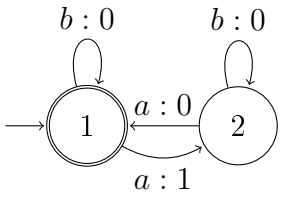


FIGURE 1.1: Un automate à distance

1.3 L'importance du non déterminisme

Avant d'aller plus loin, rappelons une notion élémentaire (vue lors du cours de licence) : le déterminisme. Un automate est *déterministe* si et seulement si il possède un unique état initial et si pour tout état, il existe **au plus** une transition sortant de cet état pour chaque lettre de l'alphabet. Par exemple, l'automate de la FIGURE 1.1 est déterministe.

Pour les automates finis, il est classique que l'on peut déterminiser un automate, c'est-à-dire calculer algorithmiquement un automate déterministe qui reconnaît le même langage. Cependant, la déterminisation des automates à distance est encore une question ouverte aujourd'hui (on dit qu'un automate à distance est déterministe si l'automate sans les poids l'est) : on ne sait pas si on peut détecter quand un automate est équivalent à un automate déterministe, et si c'est le cas, calculer cet automate déterministe équivalent.

L'automate à distance de la FIGURE 1.1 est déterministe. Cela signifie qu'il n'existe qu'un seul chemin acceptant pour tout les mots reconnus par l'automate et donc qu'il n'existe également qu'un seul coût pour ce mot. Comme on l'a dit, dans cet exemple que le coût correspondra au nombre de paires de a dans le mot. Au contraire, l'automate de la FIGURE 2.1 n'est pas déterministe, parce qu'il a 3 états initiaux, et qu'il y a 2 transitions étiquetées b partant de l'état 1.

Sur ce type d'automate, il est plus difficile de calculer la valeur associée à un mot par un automate à distance, puisqu'il faut considérer tous les chemins acceptants pour minimiser le coût. Comme nous allons le voir, le problème du caractère limité, ou *limitedness* que nous allons présenter tire sa difficulté du fait que les automates sont non déterministes.

1.4 Le problème du caractère limité

Les automates à distance ont été introduits par Hashiguchi dans [4] pour la résolution d'une célèbre problématique, la question de hauteur d'étoile (*star height problem*) [3] pour les automates finis classiques. Les automates à distance ont donc été introduits en connexion avec la hauteur d'étoile, et le problème posé sur ces automates est celui du caractère limité des automates à distance (*limitedness problem*).

Le problème de *limitedness* est très simple à comprendre : il est simplement question de savoir si il existe une majoration des coûts pour tous les mots acceptés par un automate à distance. Autrement dit,

Problème de *limitedness* pour un automate à distance

Existe-t-il un entier B tel que tous les mots acceptés par l'automate le sont avec un coût au plus B ?

Il est facile de relier ce problème sur les automates à distance et un problème sur les automates classiques. Comme nous l'avons dit, c'est le cas pour le problème de la hauteur d'étoile (les automates à distance ont été introduits pour cela), mais nous n'avons pas étudié le lien pendant le stage. En revanche, considérons le problème suivant sur les automates classiques :

Soit L un langage reconnu par automate fini. Existe-t-il un entier n tel que $L^* = L^n$?

Ce problème, dit de l'étoile finie, se relie au problème de *limitedness*. Supposons que l'on a un algorithme pour résoudre le problème de limitedness. Alors on peut résoudre le problème de l'étoile finie ainsi : on construit un automate normalisé pour L (un seul état initial q_i qui n'est atteint par aucune transition, et un seul état final q_f dont ne part aucune transition). La construction classique due à Thomson pour construire un automate pour L^* est d'ajouter une transition par le mot vide reliant q_f et q_i. Si on transforme ce dernier automate en un automate à distance, en disant que toutes les transitions ont un poids 0 sauf la transition ajoutée $q_f \xrightarrow{\varepsilon} q_i$ qui a un coût 1, le coût d'un mot w sera exactement le nombre de fois minimal qu'on est obligé d'emprunter cette transition additionnelle pour lire w, c'est-à-dire l'entier minimal n tel que $w \in L^n$. Donc, la réponse au problème de l'étoile finie sur l'automate original est OUI si et seulement si l'automate à distance qu'on a construit est LIMITÉ. Cela motive la recherche d'un algorithme pour tester si un automate est limité.

Malheureusement, en dépit de sa formulation très simple, la décidabilité de ce problème n'est pas évidente, c'est-à-dire qu'il n'est pas clair s'il existe un algorithme pour le résoudre. En effet il existe des problèmes similaires dont on a prouvé l'indécidabilité, c'est-à-dire qu'il n'existe pas d'algorithme capable de résoudre le problème. De façon surprenante, il est par exemple prouvé que la question suivante est indécidable [6] :

Soit $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates à distance distincts. Est ce que le coût de w dans $\mathcal{A}_1$ est égal au coût de w dans $\mathcal{A}_2$, pour tout mot w accepté par $\mathcal{A}_1$ et $\mathcal{A}_2$?

Le travail d'Hashiguchi étant considéré comme difficile à comprendre, d'autres chercheurs, comme Simon [9] ou Leung [7], ont étudié le problème et apporté d'autres solutions sur le sujet.

Notre travail a donc consisté, dans un premier temps, à lire et comprendre les différents travaux de ces chercheurs afin d'être capables d'implémenter un algorithme permettant la détection du caractère limité dans un automate à distance.

Chapitre 2

Un algorithme pour tester le caractère limité d'un automate

2.1 Encodage de l'automate par des matrices

Le papier dont nous nous sommes le plus servi afin de comprendre et résoudre le problème de limitedness est [9], mais il nous a d'abord fallu comprendre la complexité du problème. Pour simplifier les calculs de coûts, il fallait utiliser des structures mathématiques pour représenter la lecture d'une lettre ou d'un mot. Les matrices permettent de créer des « tables de coût » pour la lecture d'une lettre sur les différents états. Pour la suite du rapport nous utiliserons l'automate suivant en tant que référence pour nos exemples :

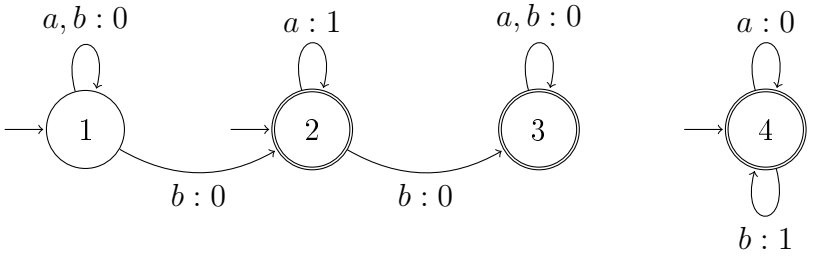


FIGURE 2.1: Automate $\mathcal{A}_{ref}$

Cet automate est-il limité ? Plus formellement :

$$\exists ? B \in \mathbb{N} \text{ tel que } \forall w \in L(\mathcal{A}_{ref}),\ cost_{\mathcal{A}_{ref}}(w) \leq B$$

où $cost_{\mathcal{A}_{ref}}(w)$ désigne le coût du mot w dans l'automate $\mathcal{A}_{ref}$. La réponse n'est pas triviale. Pour y répondre « à la main », on peut essayer de comprendre ce que cet automate calcule. Par exemple, le mot $a^n b a^n$ a un coût 1, en partant de l'état 4. Le mot $(ab)^n$ a un coût 0 : on part de l'état 1 et on le quitte sur le dernier b.

Plus généralement, un mot sur $\{a, b\}$ peut s'écrire $a^{n_0} b a^{n_1} b \cdots b a^{n_{k-1}} b a^{n_k}$ avec $k \geqslant 0$ et $n_i \geqslant 0$. Le cas $k = 0$ correspond à celui où il n'y a pas de b, et le cas $n_i = 0$ correspond au

fait qu'il n'y a pas de a entre le i-ème b et le $(i+1)$-ème b. Si on commence par l'état 4, on calcule le nombre de b, c'est-à-dire k. Si on part par l'état 2, on calcule la taille du premier bloc de a, c'est-à-dire n_0. Si on part par l'état 1, on utilise le non-déterminisme sur la lettre b pour quitter cet état juste avant le plus petit bloc a^{n_i}, pour $i \geqslant 1$, et on calcule ainsi la taille du plus petit bloc de a, à partir du bloc a^{n_1}. Au final, on calcule donc $\min(k, n_0, n_1, \ldots, n_k)$. Pour produire des mots avec des coûts arbitrairement grands, il faut donc que k soit grand et que tous les blocs de a soient grands. C'est le cas par exemple pour le mot $a^n(ba^n)^n$, qui a un coût n.

La difficulté de résoudre ce problème de façon générique conduit à utiliser des matrices pour représenter les différents coûts pour une lecture de lettres. La matrice $M(a)$ va par exemple contenir le coût de la lecture de la lettre a depuis tous les états de l'automate de la manière suivante :

$$M_{i,j} = \begin{cases} n\ , & \text{s'il existe une transition } (i, a:n) \to j \\ \infty, & \text{sinon.} \end{cases}$$

Pour $\mathcal{A}_{ref}$, on a donc par exemple :

$$M(a) = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$
$$\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}$$

et

$$M(b) = \begin{pmatrix} 0 & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 \end{pmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$
$$\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}$$

Ces matrices sont donc définies sur $\{\mathbb{N} \cup \infty\}$, nous les appellerons *matrices de coûts*. Leur principe est très simple : la numérotation de chaque ligne et de chaque colonne correspond aux états de l'automate. Par exemple, dans la matrice de coûts pour la lettre b, on a $M(b)_{(2,3)} = 0$. Cela signifie que le coût pour aller de l'état 2 à l'état 3 en lisant un b est de 0, ce qu'on peut vérifier sur l'automate.

Avant d'aller plus loin il est nécessaire de définir la structure algébrique qui est au cœur de l'algorithme de résolution du problème de limitedness, les semi-anneaux.
Un semi-anneau est formé d'un ensemble E muni de deux lois associatives, $\oplus$ et $\otimes$, et d'un

élément neutre pour chaque loi. De plus, la première loi est commutative et la seconde est distributive sur la première. On a également les règles habituelles pour les éléments neutres.

Exemple : Les entiers naturels, munis de l'addition et de la multiplication, forment un semi-anneau : $\{\mathbb{N},\ +,\ *\}$. Les éléments neutres sont 0 pour l'addition et 1 pour la multiplication. Ces deux lois sont bien associatives, l'addition est commutative et la multiplication est distributive sur l'addition. On remarque que la multiplication est également commutative, on dit que $\mathbb{N}$ est un semi-anneau commutatif.

Cette structure va nous servir pour définir la multiplication de deux matrices de coûts. Le premier semi-anneau que nous allons définir est celui formé par l'ensemble $\{\mathbb{N} \cup \infty\}$ tel que :

— $a \oplus b = min(a,\ b)$, avec élément neutre ∞
— $a \otimes b = a + b$, avec élément neutre 0

Nous avons maintenant la possibilité de multiplier deux matrices de coûts entre elles afin d'obtenir la matrice correspondant à la concaténation des mots des matrices. On a donc :

$$M(ab) = M(a) \times M(b) = \begin{pmatrix} 0 & 0 & \infty & \infty \\ \infty & \infty & 1 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 \end{pmatrix}$$

On constate que $M(ab)$ encode les coûts du mot ab pour aller d'un état à un autre : le coût pour aller de i à j est $M(ab)_{i,j}$. Il n'est pas difficile de voir que c'est le cas pour tout mot.

Il est ainsi possible d'obtenir les matrices de coûts pour n'importe quel mots composé de lettres de l'alphabet de l'automate. Et pour obtenir le coût du mot, il faut prendre le minimum des coûts situés dans l'intersection des états initiaux en ligne et des états finaux en colonne. Nous représentons les coûts compris dans cette intersection en rouge.
On peut donc dire que $cost_{\mathcal{A}_{ref}}(ab) = M(ab)_{(1,2)} = 0$ et il s'obtient en partant de l'état 1 vers l'état 2.

Cela ne nous permet cependant pas de déterminer si l'automate est limité ou pas. En effet deux facteurs rendent le nombre de matrices infini : l'ensemble de définition des matrices $\{\mathbb{N} \cup \infty\}$ et le nombre de mots acceptés par l'automate qui est lui aussi infini.

2.2 Matrices idempotentes

Dans ce chapitre, nous allons faire évoluer nos matrices afin de rendre leur nombre limité. Tout d'abord, il est nécessaire de modifier le semi-anneau qui est défini sur un ensemble infini. Dans le problème de *limitedness*, la valeur du coût en elle-même n'a pas vraiment d'utilité. Ce qui nous intéresse, c'est de savoir si cette valeur est fixe ou si elle va croître avec la longueur du mot. Si elle croît, ce signifie qu'il existe une boucle dans l'automate dont le

poids est strictement supérieur à 0.
Nous allons donc définir un ensemble de 4 valeurs qui suffiront à représenter l'intégralité des coûts possibles :

— 0 qui correspond au coût gratuit,
— 1 qui correspond à un coût potentiellement grand mais borné,
— ω qui correspond à un coût instable, qui va croître,
— ∞ qui a toujours la même signification, à savoir correspondre aux coûts impossibles à payer (transitions inexistantes).

On a donc naturellement $0 < 1 < \omega < \infty$. Il reste maintenant à définir les deux lois associatives du nouveau semi-anneau. La première est identique car nous souhaitons toujours obtenir le minimum parmi tous les chemins : $a \oplus b = min(a,\ b)$. Mais l'addition de coûts ne nous intéresse plus, nous voulons maintenant connaître le plus gros coût parmi toutes les transitions du chemin de lecture d'un mot. On prend donc $a \otimes b = \max(a,\ b)$.
Nous formons ainsi le semi-anneau qui est le sujet principal de l'article de Simon [9], le semi-anneau tropical que nous nommeront également $\mathcal{R}$.

On remarque toutefois qu'à partir des matrices de base dont on dispose et de ces deux nouvelles lois, il est impossible d'obtenir la valeur ω par simple multiplication matricielle dans $\mathcal{R}$. Autrement dit, détecter les coûts instables ne peut pas se faire par simple multiplication des matrices de base qui ne contiennent pas ω. En effet, certaines positions contenant la valeur 1 vont rester à inchangées après multiplication, alors que le coût réel va croître d'une certaine valeur. Si on itère un mot w en w^n, qui correspond à élever sa matrice à la puissance n, il est possible qu'une position 1 dans la matrice abstraite $M_{abs}(w)^n$, sur $\mathcal{R}$, soit en réalité non bornée dans la matrice concrète $M(w)^n$. Nous voulons détecter les cas où ce coût réel croît ainsi.

Reprenons par exemple notre matrice $M(a)$ et considérons-la comme matrice abstraite, sur $\mathcal{R}$. Dans cet exemple, pour différencier matrice dans $\mathbb{N} \cup \{\infty\}$ et dans $\mathcal{R}$, nous notons $M_{abs}(w)$ une matrice abstraite.

$$M_{abs}(a) = M(a) \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

Calculons maintenant $M(aa)$ dans $\mathcal{R}$:

$$M_{abs}(aa) = M_{abs}(a) \times M_{abs}(a) = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

On remarque que $M_{abs}(aa) = M_{abs}(a)$, alors qu'on n'a pas $M(aa) = M(a)$. Ceci implique que $M_{abs}(aaaa...) = M_{abs}(a)$ et donc que $M_{abs}(aaaa...)_{(2,2)} = M_{abs}(a)_{(2,2)} = 1$.

Or, on sait que la lecture répétée de la lettre a en bouclant sur l'état 2 va causer un coût croissant dépendant du nombre de a lus, autrement dit $M(a^n)_{2,2} = n$. Ce coût non borné doit être détecté, et il ne l'est pas par simple multiplication dans $\mathcal{R}$. Une telle position, $(2, 2)$, est alors dite instable.

Pour distinguer ce genre de coût de ceux qui ne causent pas ce problème de croissance non bornée, il faut ajouter un traitement supplémentaire aux matrices dont la multiplication avec elles-mêmes dans $\mathcal{R}$ donne la même matrice, c'est-à-dire telles que, dans $\mathcal{R}$, on ait

$$M = M^2.$$

On appelle ces matrices des matrices idempotentes.

La dernière partie de ce chapitre théorique va donc consister à la détection des coûts instables dans les matrices idempotentes.

2.3 Matrices stables

Nous avons donc vu que lorsqu'une matrice est idempotente, il y a un risque que certaines positions voient leur coût exploser à chaque itération de la matrice concrète correspondante, obtenue en interprétant les coûts dans $\mathbb{N} \cup \infty$. Il s'avère donc nécessaire de représenter l'itération d'un mot d'une matrice.

L'idée des positions stables est la suivante : si on a une matrice abstraite idempotente, on veut savoir si son itération *concrète* (c'est-à-dire en interprétant les coefficients dans $\mathbb{N} \cup \infty$ et non dans $\mathcal{R}$) va faire croître de façon non bornée les valeurs à certaines positions. Les positions *instables* seront celles qui conduisent à de telles valeurs non bornées. L'idée de Simon [9] est

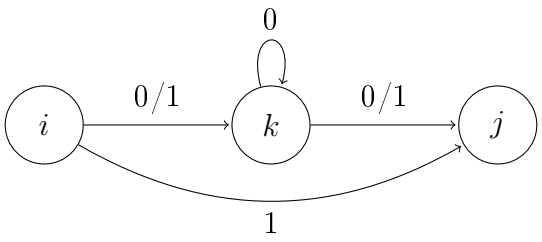
1. d'obtenir un algorithme pour détecter les positions instables,
2. d'implémenter une version abstraite de l'itération, dans $\mathcal{R}$: l'itération d'une matrice idempotente M est $M^{\#}$, obtenue de M en remplaçant toutes les positions instables (originellement des 1 dans M) par la valeur ω.

Soit w un mot, tel que $M(w)$ idempotente. La matrice $M(w^{\#})$ va donc contenir les coûts dans $\mathcal{R}$ d'un nombre quelconque de lectures successives du mot w.
Dans ces matrices que nous appellerons simplement matrices dièse, il reste donc à déterminer la valeur de chaque position. Dans son article, Simon défini le terme « ancrée » pour définir les positions dont le coût est stable ou va se stabiliser au bout d'un nombre fini d'itérations.

Lemme 2.3.1. *Soit w un mot tel que $M(w)$ soit idempotente dans $\mathcal{R}$. Si $M(w)_{(i,j)} = 0$ alors la position (i, j) est ancrée. De plus, la position (i, j) est ancrée si il existe $k \in [1,\ n]$ qui respecte les conditions suivantes :*

- $M(w)_{(k,k)} = 0$
- $M(w)_{(i,j)} = \max(M(w)_{(i,k)},\ M(w)_{(k,j)}) = 1$

Graphiquement, cela se schématise de la façon suivante :



— $M(w)_{(i,j)} = \max(M(w)_{(i,k)},\ M(w)_{(k,j)}) = 1$ signifie qu'on peut atteindre l'état k en payant un coût fixe puis rejoindre en j avec un coût également fixe.
— $M(w)_{(k,k)} = 0$ correspond à une boucle de coût 0 sur l'état k. Elle va permettre d'absorber les lectures successives de w.

Cette configuration permet donc une lecture multiple d'un mot sans que le coût ne grandisse avec le nombre de lectures. D'un point de vue matriciel, il suffit d'appliquer le lemme 2.3.1 pour chaque position de la matrice pour laquelle nous voulons obtenir sa « version dièse ». Si une position n'est pas ancrée, on dit qu'elle est instable. On a donc :

$$M(w^{\#})_{(i,j)} = \begin{cases} \omega\ , & \text{si } M(w)_{(i,j)} = 1 \text{ et } (i,j) \text{ est instable} \\ M(w)_{(i,j)}, & \text{sinon.} \end{cases}$$

Revenons maintenant sur notre automate $\mathcal{A}_{ref}$. Nous avons vu que sa matrice $M(a)$ était idempotente et que par conséquent, il était nécessaire de déterminer sa stabilité.

$$M(a) = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

On a $M(a)_{(2,2)} = 1$ et on remarque qu'on peut prendre $k = 1$ ou $k = 3$ ou $k = 4$ car on a $M(a)_{(1,1)} = M(a)_{(3,3)} = M(a)_{(4,4)} = 0$.

Cependant, la deuxième condition du lemme 2.3.1 n'est vérifiée pour aucun de ces k :

$$\forall k \in \{1,\ 3,\ 4\}, \quad M(a)_{(i,j)} \neq \max(M(a)_{(i,k)},\ M(a)_{(k,j)})$$

La position $(2,2)$ est donc instable, et on peut alors déterminer la matrice $M(a^{\#})$:

$$M(a^{\#}) = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & \omega & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

Nous sommes maintenant en possession de l'intégralité des éléments qui vont nous permettre de déterminer si un automate à distance est limité ou non. Comme nous l'avons vu, $\mathcal{R}$ est un semi-anneau défini sur un ensemble de 4 valeurs. Le principe de l'algorithme sera donc le calcul de l'intégralité des matrices de l'automate pour chacune de ses lettres, puis en cas d'idempotence, il faudra calculer la matrice dièse correspondante. Comme $\mathcal{R}$ est fini, on est sûr que ce calcul va se terminer.

Une fois l'intégralité des matrices calculée, il ne reste qu'à regarder les valeurs situées dans l'intersection des états initiaux en ligne et des états finaux en colonne comme nous l'avons expliqué dans la section 2.2. Si le minimum dans chacune de ces matrices est 0 ou 1, alors l'automate sera limité. Mais si l'une (ou plusieurs) d'entre elles a pour minimum ω, alors l'automate ne sera pas limité et le mot abstrait correspondant à cette matrice sera celui qui causera le coût non limité. Pour obtenir une suite de mots concrets faisant exploser le coût, il suffira de remplacer chaque # par un entier k arbitrairement grand.

Le nombre de matrices calculé augmente très rapidement avec le nombre d'états puisque celui ci défini n, la taille des matrices. Le nombre de matrices possibles est donc égal à 4^{n^2}. Pour $\mathcal{A}_{ref}$ par exemple, on a $n = 4$ et donc possiblement $4^{4^2} = 4\ 294\ 967\ 296$ matrices. Bien sûr, ceci est une borne supérieure et en général, on espère que l'algorithme n'aura pas à toutes les énumérer.

Une fois cette partie théorique assimilée, il nous a donc fallu mettre en place des structures permettant le stockage important de matrices ainsi qu'un accès efficace à ces dernières. Nous allons donc dans cette deuxième partie décrire l'implémentation des différentes fonctions et structures que nous avons mis en place ainsi que les choix que nous avons faits pour, à la fois être efficaces mais également conserver une certaine clarté dans le code pour que cette partie théorique soit visible au sein des fonctions.

Chapitre 3

Un programme implémentant l'algorithme

3.1 Réutilisation de la bibliothèque d'Adrien Boussicault

Pour l'implémentation de l'algorithme, nous avons choisi de reprendre le code que nous avions utilisé lors du projet d'informatique théorique 2, écrit par Adrien Boussicault sous licence libre. Ce code comprenait une implémentation des automates de base, quelques fonctions de manipulation, ainsi qu'un module de table de hachage permettant le stockage de structures.

Nous avons donc commencé par adapter le code existant afin que celui-ci puisse gérer les automates à distance. Voici les structures telles que nous les utilisons dans le programme :

```
struct _Automate {
        Ensemble * etats;
        Ensemble * alphabet;
        Table* transitions;
        Ensemble * initiaux;
        Ensemble * finaux;
};

typedef struct _Cle {
        int origine;
        int lettre;
        int cout;
} Cle;
```

Le nom et le type des variables permettent très facilement de comprendre chaque élément de ces structures.

Un automate est donc composé de quatre ensembles, `etats`, `alphabet`, `initiaux` et `finaux` et d'une table de hachage contenant les transitions. Les clés de cette table sont décrites dans la structure `Cle` qui contient trois entiers explicites, `origine`, `lettre` et `cout`.

Dans la table `transitions`, chaque clé est donc associée à un ensemble qui correspond aux états destination de la transition décrite par les valeurs de la clé. La variable `cout` de la structure `Cle` est la seule variable de structure que nous avons ajoutée par rapport au code d'origine. Ensuite, il a fallu modifier toutes les fonctions de modification des automates pour qu'elles prennent en charge la gestion des coûts des transitions.

Nous avons également ajouté un accesseur pour cette variable et modifier l'affichage des automate pour que celui-ci soit plus clair.

Voici notre automate référence $\mathcal{A}_{ref}$ affiché grâce à la fonction

```
void print_automate( const Automate * automate ) :
```

```
- Etats :       {0, 1, 2, 3, }
- Initiaux :    {0, 1, 3, }
- Finaux :      {1, 2, 3, }
- Alphabet :    {a, b, }
- Transitions :
{
         {0} --> (a : 0) --> {0, }
         {0} --> (b : 0) --> {0, 1, }
         {1} --> (a : 1) --> {1, }
         {1} --> (b : 0) --> {2, }
         {2} --> (a : 0) --> {2, }
         {2} --> (b : 0) --> {2, }
         {3} --> (a : 0) --> {3, }
         {3} --> (b : 1) --> {3, }

}
```

On distingue aisément les différents éléments de la structure et l'affichage est très similaire à l'affichage formel que nous avons présenté dans l'introduction. La structure `Mautotmate` ainsi que les fonctions la concernant sont également des ajouts de notre part, nous aborderons leur description plus loin.

3.2 Création et manipulation des matrices

Comme nous l'avons vu dans la première partie, le cœur de l'algorithme de résolution du problème de limitedness repose sur la manipulation de matrices représentant les différents coûts de l'automate. Nous avons donc ajouté un module nommé `Matrice.c` contenant la structure et les fonctions nécessaires aux manipulations vues dans la première partie. Voici tout d'abord la structure `Matrice` telle que nous l'avons implémentée :

```
struct _Matrice {
  int** tab;
  tree mot;
  int taille;
};
```

On construit ensuite la matrice avec la fonction `creer_matrice(int taille)`. Toutes les matrices sont de la forme carrée $n \times n$, avec n le nombre d'états de l'automate. C'est pourquoi la taille de la matrice est contenue dans une seule variable. La matrice contient également le mot correspondant à la lecture de la matrice ainsi qu'un tableau à deux dimensions du type `int`. On alloue alors la mémoire pour la structure de la matrice puis pour le tableau.

Pour la destruction, on libère les structures dans l'ordre inverse avec lequel on les a allouées. Sont également présents trois accesseurs pour chacune des variables de la structure, ainsi qu'une fonction d'affichage prenant en compte ω et ∞ que nous avons définis respectivement à 2 et 3, comme variables globales, dans `Matrice.h`.

On observe que dans notre structure `Matrice` la variable `mot` est de type `tree`, c'est une structure que nous avons implémentée dans `arbresynthaxique.c` afin de manipuler facilement et efficacement les mots d'une matrice.
En effet, nous avions débuté notre code par une simple structure `char` pour contenir les mots, mais celle-ci s'est révélée bien peu efficace lorsque qu'il a fallu calculer une matrice dièse ou simplement pour la multiplication de deux matrices.

Avec cette structure `tree`, nous représentons les mots sous forme d'arbres avec trois types de sommets différents :

- Une lettre de l'alphabet de l'automate qui est un sommet avec aucun fils.
- Le dièse qui est un sommet avec un seul fils, ce qui permet efficacement de transformer une expression x en $x^{\#}$.
- Le point qui correspond à la concaténation de deux expressions et qui a donc deux fils.

Ainsi on peut manipuler nos mots sans avoir à faire des manipulations fastidieuses et coûteuses de chaînes de caractères. Cette structure nous permet donc d'effectuer des concaténations et de mettre une matrice sous sa forme *« dièse »* en temps constant $O(1)$.

Revenons maintenant à notre module `Matrice.c` afin de passer en revue les fonctions principales pour la manipulation de matrices.

`Matrice multiplication_in_MnR(Matrice m1,Matrice m2)`

Cette fonction effectue la multiplication matricielle dans l'ensemble $\mathcal{R}$ que nous avons défini dans le chapitre 2. Les matrices `m1` et `m2` sont forcément de même taille puisqu'elles concernent le même automate.

On rappelle que les lois de $\mathcal{R}$ sont $a \oplus b = min(a,\ b)$ et $a \otimes b = max(a,\ b)$. On crée donc une nouvelle matrice m3 qu'on initialise à -1 pour chacune de ses positions, ce qui va nous permettre d'extraire la valeur maximum d'une des deux matrices lors de la multiplication.

`int est_idempotent(Matrice m)`

Le test d'idempotence est très simple puisque comme nous l'avons vu dans la première partie, il suffit de multiplier la matrice avec elle-même, puis de tester l'égalité.

`int stable(int i,int j,Matrice m)`

Cette fonction résulte directement du lemme 2.3.1. En effet, elle se contente d'effectuer, pour chaque position, tous les tests nécessaires afin que les conditions du lemme soient respectées.

`Matrice creer_matrice_diese(Matrice m)`

Comme son nom l'indique, la fonction `creer_mattice_diese` calcule et renvoie la matrice dièse d'une matrice m dans $\mathcal{R}$. Comme vu en section 2.3, il suffit de tester la stabilité de chaque position si $tab[i][j] = 1$. Si la position n'est pas stable, on remplace sa valeur par ω sinon, on la laisse à 1.

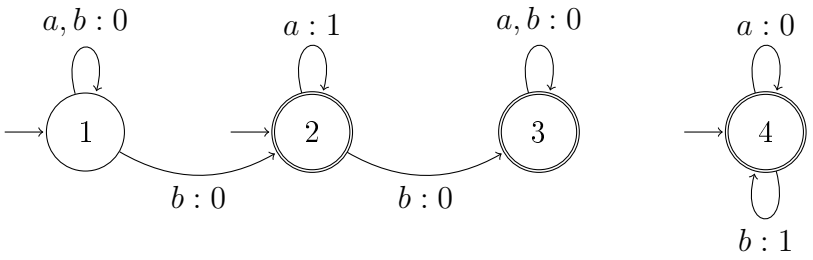
Grâce à ce module, nous avons maintenant les éléments nécessaires pour manipuler nos matrices définies dans $\mathcal{R}$. Cependant, nous avons vu qu'un très grand nombre de ces matrices allait être calculé. C'est pourquoi il nous a fallu trouver un moyen d'effectuer tous ces calculs de la manière la plus efficace possible.

3.3 Structure de stockage des matrices

Le nombre de matrices calculées étant potentiellement très grand, il nous a donc fallu réfléchir à une méthode permettant un calcul optimisé de l'intégralité des matrice de l'automate. C'est-à-dire qu'il faut organiser les matrice de sorte qu'on sache quelles sont celles qui restent à calculer et éviter de calculer une même matrice plusieurs fois. Au passage, nous avons donc besoin de tester si une matrice a déjà été calculée.

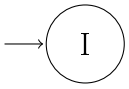
La structure que nous avons choisie n'est autre qu'un automate... Son fonctionnement n'est pas évident mais cette nouvelle structure va simplifier grandement l'algorithme, notamment la condition d'arrêt qui signifie que toute les matrices ont bien été calculées.

Nous avons appelé cet automate particulier un *mautomate* car c'est simplement un automate dont les états sont des matrices. Mais avant de présenter le code de cette structure, nous allons expliquer comment celle-ci est pensée. Pour cela nous allons reprendre notre automate $\mathcal{A}_{ref}$:

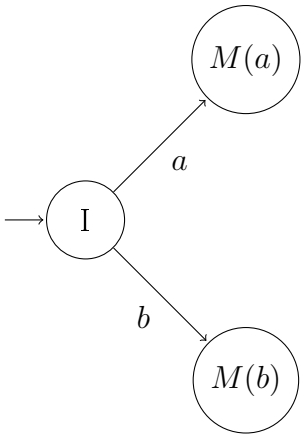


Automate $\mathcal{A}_{ref}$

Dans un premier temps, nous allons créer un état initial qui sera le départ de tout calcul matriciel.

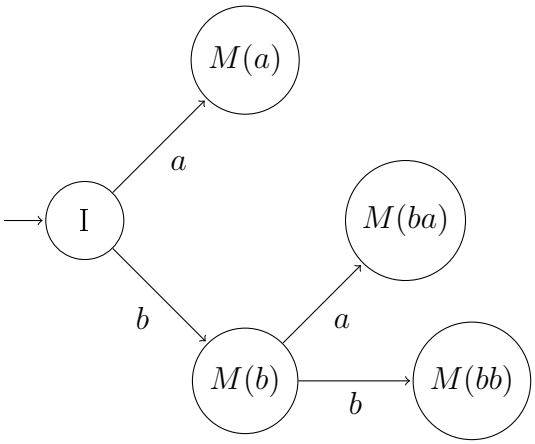


L'alphabet de départ sera le même que l'automate sur lequel on travaille, ici $\{a,\ b\}$. Depuis l'état initial, nous allons créer une transition pour chaque lettre de cet alphabet. La destination de ces transitions sera la matrice correspondant à la lettre de la transition. Pour $\mathcal{A}_{ref}$ cela nous donne l'automate suivant :

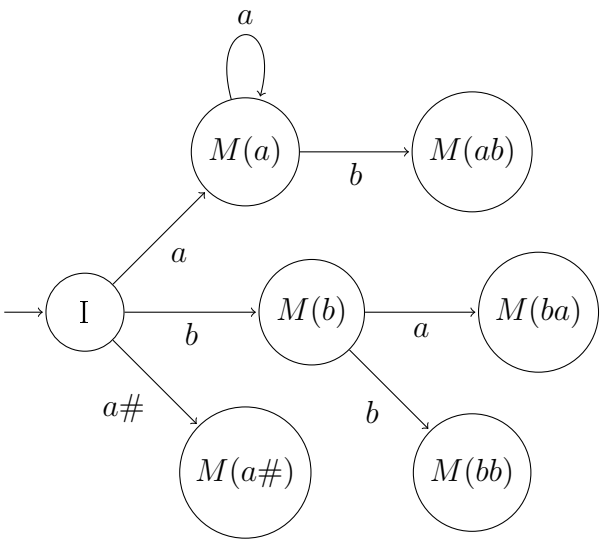


Maintenant, le but va être de rendre l'automate complet. Pour tout état $M(x_1)$, $M(x_2)$ et pour toute lettre l de l'alphabet, Nous aurons donc des transitions de la forme $(M(x_1),\ l) \rightarrow M(x_2)$ avec $M(x_2) = M(x_1) \times M(l)$

Voyons ce que cela donne pour l'état $M(b)$:



Pour $M(a)$ le traitement n'est pas le même. En effet nous avons vu dans le chapitre 2 que la matrice $M(a)$ était idempotente. Cela signifie tout d'abord qu'il va y avoir une boucle sur l'état $M(a)$ pour la lettre a. Mais cela signifie également, comme nous l'avons vu dans le chapitre 2, que nous allons devoir calculer la matrice $M(a^{\#})$. Deux cas sont possibles : si $M(a^{\#}) = M(a)$ alors la boucle suffit. Mais si ce n'est pas le cas, il fait alors traiter $a^{\#}$ comme une nouvelle lettre de l'automate et recompléter en conséquence. Ce dernier cas s'applique ici, puisqu'on a vu que en section 2.2 que la position $(2, 2)$ n'est pas stable pour la matrice $M(a)$, et donc que $M(a^{\#}) \neq M(a)$:



On itère ce processus jusqu'à ce que le mautomate soit complet, en repartant de cet alphabet à 3 lettres, $a, b, a^{\#}$. Il est possible qu'on crée des lettres correspondant à des imbrications de #, comme $a^{\#}(ba^{\#})^{\#}$. On a même vu précédemment que pour cet automate, c'est ce type de mot qui donnera le caractère non limité.

Le mautomate grandit très rapidement à cause des matrices idempotentes qui ont une chance de rajouter une lettre dans l'alphabet. C'est donc cette structure que nous avons ajoutée dans `automate.c` et qui est au cœur de l'algorithme de résolution du problème de limitedness.

La structure `Mautomate` est composée de trois éléments comme suit :

```
struct _Matrice_Automate{
  Automate * automate;
  Table* t_matrices_etats;
  Table* t_matrices_lettres;
};
```

Il y a donc un automate et deux tables de hachage : `t_matrices_etats` qui stocke l'intégralité des matrices calculées et dont la clé correspond à l'état de l'automate qui la contient. La seconde table de hachage, `t_matrices_lettres`, stock également des matrices mais seulement celles qui correspondent à une lettre et la clé est donc cette même lettre. Cette table permet d'accéder facilement aux matrices de base lors du calcul de nouvelles matrices pour chaque transition.

Ainsi, n'avons plus qu'à créer un mautomate à partir de l'automate, puis analyser chacune des matrices afin de détecter si l'une d'elle cause un coût ω, c'est-à-dire un coup non-borné.

Notre fonction de création, `creer_mautomate(Automate* a)`, alloue la mémoire nécessaire à la structure et effectue également chacune des étapes pour obtenir le mautomate complet à la sortie de la fonction.

Nous appelons donc `completer_automate_des_matrices(Mautomate* a)` après l'allocation. Elle fait toutes les multiplications matricielles pour l'alphabet de l'automate et ce, sans s'occuper des matrices idempotentes et donc des dièses. En effet, c'est la fonction `void calculer_diese_mautotmate(Mautomate * ma)` qui effectue ce travail, c'est la seconde fonction appliquée lors de la création.

Celle-ci parcourt l'intégralité du mautomate à la recherche de matrices idempotentes afin de calculer les matrices dièses correspondantes. Comme nous l'avons vu ce calcul peut générer une nouvelle lettre dans l'alphabet du mautomate. C'est pourquoi dans ce cas là, il est nécessaire d'appeler à nouveau la fonction précédente pour compléter le mautomate. On alterne ainsi les appels de ces fonction jusqu'à ce que l'automate soit complet, avec l'intégralité des matrices de transition.

Une fois créé, il ne nous reste plus qu'a analyser les matrices afin de déterminer si l'automate de départ est limité ou non.

3.4 Résolution du problème de limitedness

La fonction `est_limite(Automate* a, Mautomate * ma)` a besoin de l'automate ainsi que de son mautomate pour déterminer si celui-ci est limité ou pas. La table `t_matrices_etats` est la seule utilisée dans la structure mautomate. En effet seules les valeurs contenues dans les matrices nous interessent. Nous utilisons l'automate pour déterminer l'intervalle des états initiaux servant dans l'analyse des matrices comme nous l'avons vu dans le chapitre 2. Si la valeur minimale d'une seule de ces matrice est ω ou ∞, alors l'automate n'est pas limité, et cette matrice accompagnée du mot qui la caractérise, est celle qui possède le coût illimité.

Nous arrêtons à cette première matrice mais il en existe possiblement d'autres. Grâce à notre structure mautomate, il serait aisé d'implémenter une fonction répertoriant l'intégralité des matrices causant un coût *« non-payable »*.

Le fichier test, contenant la fonction `main` de notre programme effectue simplement un test avec notre automate $\mathcal{A}_{ref}$. Nous créons cet automate étape par étape (transitions, états initiaux et finaux) en utilisant les fonctions implémentées par Adrien Boussicault, puis nous affichons textuellement l'automate.

Nous avons également intégré un **affichage graphique** via `graphviz`. C'est un outil qui permet depuis un fichier texte `.gv`, de créer une image, du format souhaité, de l'automate

décrit par le texte. Nous avons donc écrit une fonction `creergraphe(Automate* a)` qui traduit les informations de la structure `Automate` en texte interprétable par `graphviz`.

C'est le fichier auto.gv qui contient toutes ces informations. Nous créons ensuite l'image en appelant la commande de graphviz `dot -Tpng auto.gv -o auto.png`, on obtient le graphe *auto.png* qu'on ouvre avec la commande `xdg-open` qui va utiliser le visionneur d'image par défaut .

L'étape suivante est la plus longue, la création du mautomate. C'est pourquoi nous affichons le temps de calcul pour cette opération. Une fois le mautomate obtenu, la fonction `est_limite(Automate* a, Mautomate * ma)` effectue l'analyse des matrices en temps linéaire puisqu'il s'agit d'un seul parcourt d'une table de hachage.
Si l'automate n'est pas limité, nous l'affichons puis le programme s'arrête. Si ce n'est pas le cas, nous demandons si l'utilisateur souhaite afficher l'automate du mautomate. Celui-ci étant possiblement très gros, un affichage n'est pas forcément pertinent. Nous demandons ensuite s'il ne veut pas afficher juste les matrices, et combien il veut en afficher.

Pour notre automate $\mathcal{A}_{ref}$, notre algorithme met environ 6 secondes à calculer l'intégralité des matrices. Nous avons également corrigé plusieurs problèmes de fuite mémoire et nous avons réussi à obtenir un programme qui se termine sans aucune fuite mémoire :

```
==3769==
==3769== HEAP SUMMARY:
==3769==     in use at exit: 0 bytes in 0 blocks
==3769==   total heap usage: 197,901 allocs, 197,901
        frees, 7,434,504 bytes allocated
==3769==
==3769== All heap blocks were freed -- no leaks are
        possible
==3769==
==3769== For counts of detected and suppressed errors,
        rerun with: -v
==3769== ERROR SUMMARY: 0 errors from 0 contexts
        (suppressed: 1 from 1)
```

Le stage s'est donc déroulé en deux étapes, la première partie, a consisté à comprendre la profondeur et la complexité du problème ainsi que sa solution. Le domaine des automates étant vaste, nous avons dû nous heurter à la frustration de découvrir de nombreux sujets liés au nôtre sans pouvoir les approfondir par manque de temps.

Nous avons par exemple commencé le stage par un autre papier de Simon [2], qui concerne la *factorization forest* qui est utilisée pour la preuve de la résolution du problème de *limitedness* mais qui a aussi d'autres applications dans des sujets plus éloignés. Nous pouvons également citer le problème de la hauteur d'étoile, mentionné à de nombreuses reprises lors de ce rapport. Cependant, le caractère limité des automates à distance était également un problème complexe et nous n'aurions pas pu le traiter en si peu de temps si nous nous étions égarés sur d'autres sujets.
Lors de la seconde partie, l'implémentation de l'algorithme, nous avons du réfléchir aux structures qui seraient les plus efficaces pour effectuer le travail demandé. Notre travail se concentra donc sur l'élaboration de la structure `Mautomate` et de la manipulation de matrices. Ensuite il ne restait plus qu'à appliquer la partie théorique dans les fonctions.

Durant ce stage, même si nous avons effleuré le domaine de la recherche en informatique théorique, cela nous a permis d'approfondir nos connaissances sur le sujet dans la réalisation d'un rapport poussé. Nous avons grandement apprécié le sujet du stage et son déroulement. Celui-ci a même un impact sur nos choix d'études pour le futur.

Bibliographie

[1] J. Berstel, D. Beauquier, and P. Chrétienne. *Elements d'algorithmique*. Masson, 2005. Disponible sous `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf`.

[2] M. Bojańczyk. Factorization forests. In V. Diekert and D. Nowotka, editors, *Proc. of DLT'09*, number 5583 in Lect. Notes. Comp. Sci., pages 1–17. Springer, 2009.

[3] L. Eggan. Transition graphs and the star-height of regular events. *Michigan Mathematical Journal*, 10 :385–397, 1963.

[4] K. Hashiguchi. Limitedness theorem on finite automata with distance functions. *J. Comput. Syst. Sci.*, 5201 :233–244, 1982.

[5] D. Kirsten. Distance desert automata and the star height problem. *RAIRO Inform. Théor. Appl.*, 39(3) :455–509, 2005.

[6] D. Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. *Int. J. of Algebra and Comput.*, 4(3) :405–425, 1994.

[7] H. Leung and V. Podolskiy. The limitedness problem on distance automata : Hashiguchi's method revisited. *Theoret. Comput. Sci.*, 310(1-3) :147–158, 2004.

[8] I. Simon. Factorization forests of finite height. *Theoret. Comput. Sci.*, 72(1) :65–94, 1990.

[9] I. Simon. On semigroups of matrices over the tropical semiring. *RAIRO Inform. Théor. Appl.*, 28(3-4) :277–294, 1994.