Rapport Projet de stage : Limitedness

Brebant Alexandre
Xue Juedong

2 août 2014

Table des matières

1 Introduction au problème 2

I Mise en place théorique de l'algorithme 5

2 De l'automate aux matrices 6

3 Matrices idempotentes 9

4 Matrices stables 11

II Exécution détaillée du programme 14

5 Réutilisation du programme d'Adrien Boussicault 15

6 Création et manipulation des matrices 17

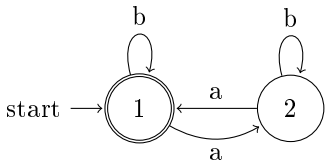
7 Structure de stockage des matrices 20

8 Résolution du problème de limitedness 24

Chapitre 1

Introduction au problème

Durant ce stage, nous avons voulu approfondir nos connaissances dans une branche de l'informatique théorique que nous avons commencé à étudier 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.
Un automate est une machine abstraite qui va accepter ou rejeter des mots dans 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 donc 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, c'est à dire 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 (start) et allant jusqu'à un état final (double cercle) 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*, on le 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

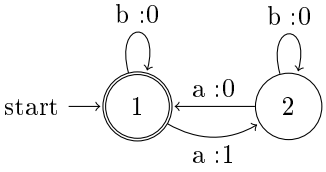
Nous pouvons donc définir l'automate ci-dessus de cette manière :

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

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 ([1]). Mais il existe des extensions à ce modèle, comme par exemple les automates à pile qui permettent de définir les langages algébriques ([1]), qui sont grandement utiliser en analyse syntaxique. Et c'est une autre extension de l'automate fini qui va nous intéresser dans ce rapport : les automates à distance (distance automata).
Ce sont des automates finis, non déterministes, dans lesquels les transitions sont pondérées. On associe donc un coût pour chaque mot accepté par l'automate. Le coût d'un mot correspond au minimum des coûts de tous ses chemins acceptants dans l'automate .

Reprenons l'automate représenté plus haut en y ajoutant des coûts sur les transitions :



Avant d'aller plus loin dans la description de cet automate, nous devons définir une notion élémentaire : le déterminisme. Un automate est déterministe si et seulement si il possède un unique état initial et que pour tout état, il existe au plus, une transition par lettre de l'alphabet.
Cependant, la déterminisation des automates à distance est encore une question ouverte aujourd'hui. On dit alors qu'un automate à distance est déterministe si l'automate sans les poids l'est.

Notre automate à distance est donc 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. On peut observer dans cet exemple que le coût correspondra au nombre de paires de a dans le mot.

Ce type d'automate a été présenté par Hashiguchi dans [3] pour la résolution d'une célèbre problématique, la question de hauteur d'étoile (star height problem) [2]. Mais les automates à distance posent également un autre problème, la limitation 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.
La décidabilité de ce problème n'est pas évidente. En effet il existe des problème similaires dont on a prouvé leur indécidabilité, c'est à dire la preuve qu'il n'existe pas d'algorithme capable de résoudre le problème. Il est par exemple prouvé que la question suivante est indécidable :

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 égale au coût de w dans $\mathcal{A}_2$, $\forall w \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$?

Le travail d'Hashiguchi étant considéré comme difficile à comprendre, d'autres chercheurs, comme Simon [5] ou Leung [4], 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 capable d'implémenter un algorithme permettant la détection de limites dans un automate à distance.

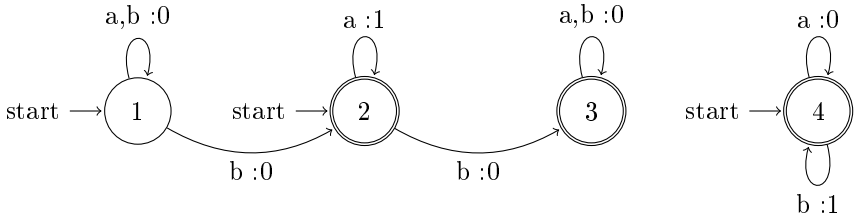
Première partie

Mise en place théorique de l'algorithme

Chapitre 2

De l'automate aux matrices

Le papier dont nous nous sommes le plus servi afin de comprendre et résoudre le problème de limitedness, est [5] 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 utiliseront l'automate suivant en tant que référence pour nos exemples :



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

Cet automate est-il limité ? Plus formellement :

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

La réponse n'est évidemment pas triviale. C'est pourquoi nous allons 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 tout les états de l'automate de la manière suivante :

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

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 fonctionnement 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.
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 :

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

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 $cost(ab) = M(ab)_{(1,2)} = 0$ 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.

Chapitre 3

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 limite, 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 fixe
- ω qui correspond à un coût instable, qui va croître
- ∞ qui a toujours la même signification, à savoir correspondre aux coûts impossibles à payer

Et on a donc $0 < 1 < \omega < \infty$, 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, le semi-anneau tropical que nous nommeront également $\mathcal{R}$.

On remarque tout de fois 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 $\mathcal{R}$. Il nous est donc impossible de détecter les coûts instables, en effet certaines positions contenant la valeur 1 vont rester à inchangées après une multiplication alors que le coût réel va croître d'une certaine valeur pour chaque lecture du mot représenté par la matrice.

Reprenons par exemple notre matrice $M(a)$:

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

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

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

On remarque que $M(aa) = M(a)$. Ce qui implique que $M(aaaa...) = M(a)$ et donc que $M(aaaa...)_{(2,2)} = M(a)_{(2,2)} = 1$.
Or on peut observer sur l'automate 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.

Pour distinguer ce genre de coût de ceux qui ne causent pas ce problème de croissance infinie, il faut donc ajouter un traitement supplémentaire aux matrices dont la multiplication avec elles-même dans $\mathcal{R}$ donne la même matrice. On appelle ces matrices des matrices idempotentes.

Le dernier chapitre de cette partie théorique va donc consister à la détection des coûts instables dans les matrices idempotentes.

Chapitre 4

Matrices stables

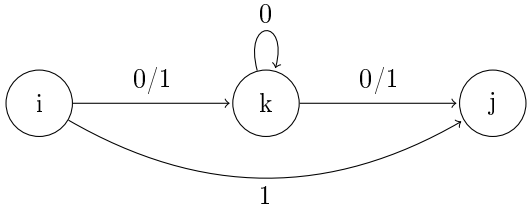
Nous avons donc vu que lorsqu'une matrice est idempotente, il y a un risque que certaines positions voient leur coût exploser à chaque lecture du mot de cette matrice. Il s'avère donc nécessaire de représenter l'itération d'un mot d'une matrice.
Au lieu d'écrire $M(aaaaa...)$ comme nous l'avons fait dans le chapitre précédent, nous allons introduire un nouveau symbole afin de simplifier la notation et écrire ce genre de matrices de la façon suivante : $M(a^{\#})$.

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'itération.

Lemme : *Soit w un mot tel que $M(w)$ 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 matricielle, il suffit d'appliquer le lemme 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 :

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

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 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, 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'impotence, il faudra calculer la matrice dièse correspondante.

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 le chapitre 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 correspondant à cette matrice sera celui qui causera le coût non limité.

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.

Une fois cette partie théorique assimilée, ils 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 fait pour, à la fois être efficace mais également conserver une certaine clarté dans le code pour que cette partie théorique soit visible au sein des fonctions.

Deuxième partie

Exécution détaillée du programme

Chapitre 5

Réutilisation du programme 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. 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 commencer 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 associé 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 ajouté 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 dans le chapitre 7.

Chapitre 6

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 vu dans la première partie. Voici tout d'abord la structure `Matrice` telle que nous l'avons implémenté :

```
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 alloué. 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 défini 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 implémenté 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 à 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 concaténation 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 3. 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 énoncé dans le chapitre 4. 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` calcul et renvoie la matrice dièse d'une matrice m dans $\mathcal{R}$. Comme vu dans le chapitre 4, il suffit de tester la stabilité de la 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.

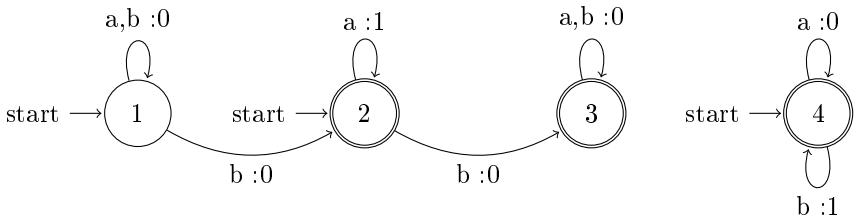
Grâce à ce module, nous avons maintenant les éléments nécessaires pour manipuler nos matrices définies dans $\mathcal{R}$. Cependant, nous avons vu qu'un très grand nombre de ces matrices allait être calculé. C'est pourquoi il nous a fallu trouver un moyen d'effectuer tous ces calculs de la manière la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

Chapitre 7

Structure de stockage des matrices

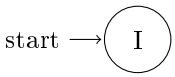
Le nombre de matrices calculé étant potentiellement très grand, il nous a donc fallu réfléchir à une méthode permettant un calcul optimisé de l'intégralité des matrice de l'automate. C'est à dire qu'il faut organiser les matrices de sorte qu'on sache quelles sont celles qui reste à calculer et éviter de calculer une même matrice plusieurs fois.

La structure que nous avons choisis 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 qui contient 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 $\mathcal{A}_{ref}$:

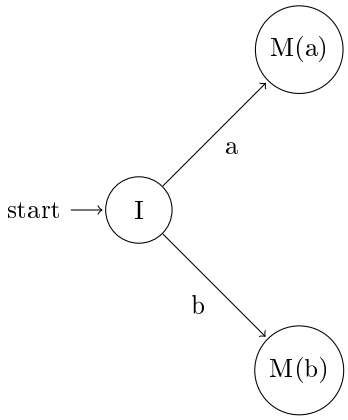


Automate $\mathcal{A}_{ref}$

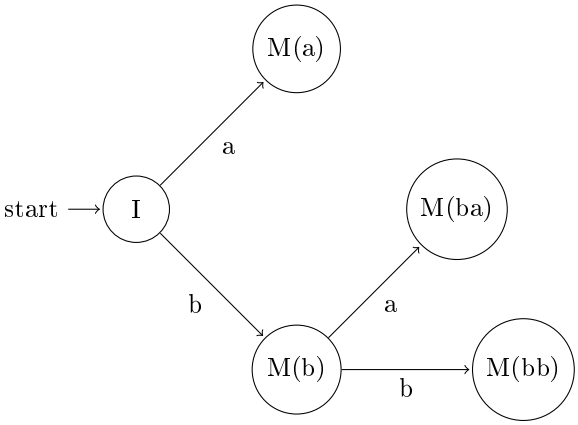
Dans un premier temps, nous allons créer un état initial qui sera le départ de tout calcul matriciel.



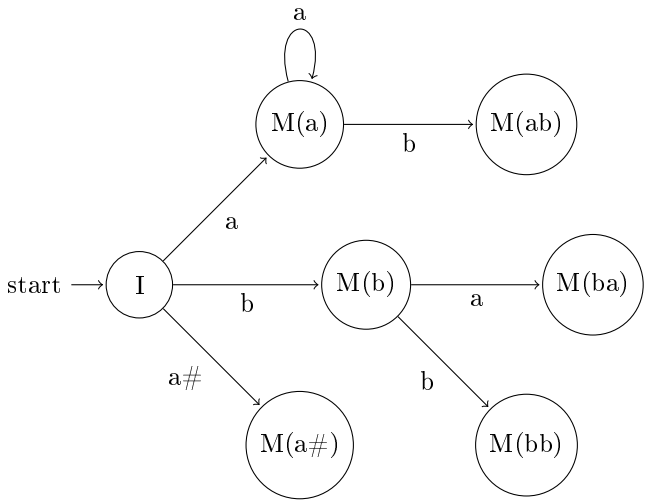
L'alphabet sera le même que l'automate sur lequel on travail, ici $\{a,\ b\}$. Depuis l'état initial, nous allons créer une transition pour chaque lettre de cet alphabet. La destination de ces transitions sera la matrice correspondant à la lettre de la transition. Pour $\mathcal{A}_{ref}$ cela nous donne 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 $(M(x_1),\ l) \to M(x_2)$ 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 3 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 4, 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 :



On itère ce processus jusqu'à ce que le mautomate soit complet. Celui-ci grandi 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 rajouté dans *"automate.c"* et qui est au coeur 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 table de hachage : `t_matrices_etats` qui stock 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 `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 la, il est nécessaire d'appeler à nouveau la fonction précédente pour compléter le mautomate. On alterne ainsi les appels de ces deux 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.

Chapitre 8

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. Seule la table `t_matrices_etats` est utilisée dans la structure mautomate. En effet seule les valeurs contenues dans les matrices nous interessent. Nous utilisons l'automate pour déterminer l'intersection des états initiaux et finaux 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 accompagné du mot qui la caractérise, est celle qui possède le coût illimité.
Nous arrêtons à cette première matrice mais il en existe possiblement d'autre. 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 le graphe.

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 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)
```

Nous avons grandement approfondi nos connaissances dans le domaine des automates et dans la réalisation d'un rapport poussé.
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. Lors de la seconde partie, l'implémentation de l'algorithme, nous avons du réfléchir aux structures qui seraient le plus efficaces pour effectuer le travail demandé. Ensuite il ne reste qu'à appliquer la partie théorique dans les fonctions.

Bibliographie

[1] J. Berstel, D. Beauquier, and P. Chrétienne. *Elements d'algorithmique*. 2005.

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

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

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

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