UE de musique (1 cours de 3h) :

Automates et modèles symboliques (application aux logiciels d'improvisation OMax-ImproteK-Djazz)

Marc Chemillier

Master M2 Atiam (Ircam), 2019-2020

Automates et langages formels

- Définitions générales
  - Mots, langages
  - Aspects algébriques : monoïdes, codes
- Notion d'automate fini
  - Automate fini déterministe (AFD)
  - Langage reconnu par un automate
  - Exemple du jeu de dés

Application aux logiciels d'improvisation

- Historique OMax-ImproteK-Djazz
- Pattern matching
  - Arbre des suffixes

1. Quelques définitions générales

1.1 Mots

Un *mot* u est une suite finie de symboles. La longueur de u notée $|u|$ est le nombre de symboles du mot. Le *mot vide* noté ε est le mot de longueur nulle (unique). L'ensemble des symboles noté Σ est appelé *alphabet*, et l'ensemble des mots sur l'alphabet Σ est noté Σ^*.

Pour deux mots $u = u_1 \ldots u_n$ et $v = v_1 \ldots v_m$, on définit la *concaténation* uv comme le mot obtenu en mettant les lettres de v à la suite de celles de u :

$uv = u_1 \ldots u_n v_1 \ldots v_m$.

Un mot $u \in \Sigma^*$ est *facteur* du mot $w \in \Sigma^*$ s'il existe $v, v' \in \Sigma^*$ tels que $w = vuv'$. Si $v = \varepsilon$, on dit que u est *préfixe*. Si $v' = \varepsilon$, on dit que u est *suffixe*.

Exemple : *abb* est facteur de *babba*, et *ba* est à la fois préfixe et suffixe, mais *aa* n'est pas facteur.

Un mot fini u est *périodique* si $u = x^n$ pour $n \geq 2$. Tout mot non périodique est dit *primitif*.

L'idée fondamentale de ce travail est que la notion de mot permet de représenter le principe de succession d'événements dans une séquence musicale, d'où son intérêt pour la modélisation en informatique musicale.

1.2 Langages

Les sous-ensembles de Σ^* sont appelés des *langages* (c'est-à-dire des ensembles de mots, ou de séquences musicales dans le contexte de l'informatique musicale).

Opérations sur les langages :

• opérations classiques sur les ensembles :

union $\cup$, intersection $\cap$, différence \, complémentaire,

• opérations héritées de la concaténation :

La concaténation de deux langages est définie par :

$L_1L_2 = \{uv, u \in L_1 \text{ et } v \in L_2\}$.

Exemple : $L_1 = \{a, ab\}$, $L_2 = \{c, bc\}$, combien de mots dans L_1L_2 ?

→ $L_1L_2 = \{ac, abc, abbc\}$

La *puissance* d'un langage L est définie inductivement :

$L^0 = \{\varepsilon\}, L^1 = L, L^{n+1} = L^nL$.

L'*étoile* d'un langage L, notée L^*, est :

$L^* = \{\varepsilon\} \cup L \cup L^2 \ldots \cup L^n \cup \ldots$

Ce langage contient un nombre infini de mots, qui sont les répétitions indéfinies de mots de L.

→ il existe deux cas où L^* **n'est pas infini**, lesquels ?

$L = \{\varepsilon\}, L = \varnothing$

→ dans les deux cas : $L^* = \{\varepsilon\}$

Exemple : $L = \{ab\}$, L^* est l'ensemble de tous les mots de la forme ab^n.

1.3 Aspects algébriques : monoïdes

Un *monoïde* est un ensemble muni d'une opération

• associative,

• possédant un élément neutre 1.

U n *sous-monoïde* est un sous-ensemble fermé pour l'opération et contenant l'élément neutre. L'ensemble Σ^* des mots sur l'alphabet Σ est un monoïde pour la concaténation, dont l'élément neutre est le mot vide ε. Ce monoïde est engendré par l'ensemble de ses lettres, c'est-à-dire l'alphabet Σ.

Remarque fondamentale : Soient deux mots $u = u_1 \ldots u_n$ et $v = v_1 \ldots v_m$ de Σ^*. L'égalité $u = v$ équivaut à l'égalité de toutes les lettres de u et v une à une, c'est-à-dire

• $n = m$ (égalité des longueurs)

• $u_i = v_i$ pour tout $i \leq n$

Un monoïde vérifiant cette propriété est appelé *monoïde libre* (tous les monoïdes ne sont pas libres).

Exemples :

• cette propriété n'est pas vraie dans un groupe :

$xx^{-1} = yy^{-1}$ n'implique pas $x = y$

Un *code* est une partie C de Σ^* qui est la base d'un sous-monoïde **libre** de Σ^*, c'est-à-dire

- C engendre le sous-monoïde et

- tout élément du sous-monoïde s'écrit **de manière unique** comme suite d'éléments de C.

Informatiquement, cela signifie qu'on peut « décoder » les éléments de C^*.

Exemple : message $m = abcabcabcabc$ se décode sur l'ensemble $C_1 = \{ab, bc, ca\}$

→ $m = (ab)(ca)(bc)(ab)(ca)(bc)$

→ qu'est-ce que ça donne avec la correspondance $ab = C$, $bc = U$, $ca = O$?

$m = COUCOU$

mais il ne se décode pas sur l'ensemble $C_2 = \{a, ab, c, bc\}$

→ $m = (ab)(c)abcabcabc$ ou $m = (a)(bc)abcabcabc$? → ambiguïté

• Attention : Le code Morse (Samuel Morse, 1791-1872) n'est pas un code dans le sens ci-dessus.

On a T = « — », E = « • », et N = « —• »,

donc on a TE = N.

Le décodage en Morse se fait grâce à la présence de séparateurs entre les lettres.

INTERNATIONAL MORSE CODE

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to five dots.

A	·−	U	··−
B	−···	V	···−
C	−·−·	W	·−−
D	−··	X	−··−
E	·	Y	−·−−
F	··−·	Z	−−··
G	−−·		
H	····		
I	··		
J	·−−−	1	·−−−−
K	−·−	2	··−−−
L	·−··	3	···−−
M	−−	4	····−
N	−·	5	·····
O	−−−	6	−····
P	·−−·	7	−−···
Q	−−·−	8	−−−··
R	·−·	9	−−−−·
S	···	0	−−−−−
T	−		

Un *morphisme* de monoïde est une application φ telle que $\varphi(xy) = \varphi(x)\varphi(y)$ et $\varphi(1) = 1$.

Toute application d'un alphabet Σ dans un monoïde quelconque se prolonge dans Σ^* en un unique morphisme de monoïdes.

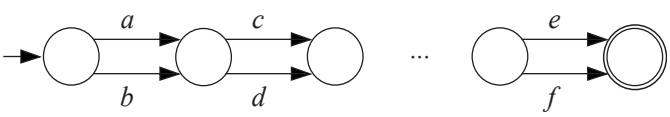
→ Il en résulte que pour définir un morphisme sur Σ^*, il suffit de définir l'image de toutes ses lettres (de la même manière que dans un espace vectoriel, on définit un morphisme en donnant l'image de tous les vecteurs de la base).

2. Notion d'automate fini

2.1 Définition des automates finis déterministes (AFD)

Le jeu de dés est un automate :

- on part d'un état initial
- on passe d'un état au suivant par des transitions : choix de la mesure *a* ou *b*, *c* ou *d*, etc.



Définition. *Un automate fini déterministe AFD sur un alphabet* Σ *est la donnée d'un n-uplet* (Q, δ, i, F) *où* :

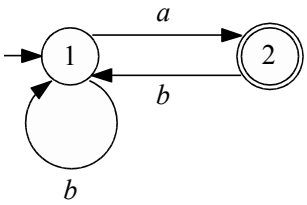
• Q *est un ensemble fini d'états*,

• δ *est une fonction de transition de* $Q \times \Sigma$ *dans* Q,

• i *est un état particulier de* Q *dit initial*,

• F *est une partie de* Q *d'états dits finals*.

Exemple :



$Q = \{1, 2\}$,

$i = 1$, état noté avec une petite flèche entrante,

$F = \{2\}$, état noté avec deux cercles.

$\delta : Q \times \Sigma \rightarrow Q$

$(1, a) \rightarrow 2$

$(1, b) \rightarrow 1$

$(2, b) \rightarrow 1$

Pour décrire un automate, il est commode d'utiliser une table de transitions :

	1	2
a	2	
b	1	1

2.2 Langage reconnu par un AFD

Le calcul de l'automate consiste à suivre des flèches, en partant de l'état initial et en s'arrêtant dans un état final. Le mot correspondant à ce calcul est la suite des étiquettes des flèches.

Définition. *Le langage reconnu (ou accepté) par un automate AFD est l'ensemble des mots qui correspondent à un calcul de l'automate partant d'un état initial et s'arrêtant dans un état final.*

Exemples :

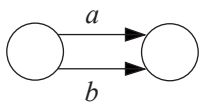
- distributeur de café : les pièces introduites sont les symboles de l'alphabet, l'état terminal est atteint quand le montant est supérieur au montant demandé,

- mécanisme contrôlant le code d'accès d'une porte (digicode) : les chiffres tapés sont les symboles, l'état terminal est celui qui déclenche l'ouverture,

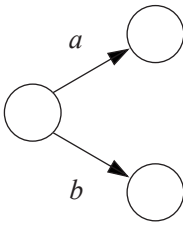
2.3 Exemple du jeu de dés

Dans l'automate du jeu de dés, toutes les flèches partant d'un état vont **vers le même état**, donc les lettres sont interchangeables : ce qui les suit est identique.

Il n'y a pas de bifurcation.



jeu de dés



bifurcation

Problème : Peut-on reconnaître qu'un langage est reconnaissable par un automate de type « jeu de dés » ?

Langage $L = \{abc, bbb, abb, bbc\}$

Ce langage a-t-il une structure de « jeu de dés » avec des lettres interchangeables indépendamment de ce qui suit et ce qui précède ?

→ oui il se factorise :

$L = \{abc, bbb, abb, bbc\} = \{a, b\}\{b\}\{c, b\}$

Remarque : si on enlève bbc, $L' = \{abc, bbb, abb\}$ ne se factorise plus

→ $\{a, b\}\{b\}\{c\} = \{abc, bbc\} \neq L'$

3. Historique OMax-ImproteK-Djazz

1999 deux types de travaux :

- Gérard Assayag, Shlomo Dubnov : simulation stylistique hors temps (algorithme de Lempel-Ziev) → voir exemple de l'*Offrande musicale*
- Marc Chemillier : grammaire de substitutions harmoniques, Carlos Agon : implémentation temps réel des substitutions dans OpenMusic (Lisp)

2001 **création d'OMax** (Marc Chemillier, Gérard Assayag) = improvisation par recombinaison de fragments musicaux pré-existants ou captés à la volée qui sont concaténés, implémentation d'un automate pour la recombinaison appelé **oracle des facteurs** (dans OpenMusic en Lisp), mode free (**sans pulsation**), patch en Max/MSP pour la gestion en temps réel

2009-2010 **création d'ImproteK** (Marc Chemillier, Jérôme Nika) = reprise d'OMax avec **pulsation et grille**, développé par Jérôme Nika dans sa thèse, **algorithme de pattern matching (Morris & Pratt)** pour suivre une grille

- **lecture MIDI avec tempo adaptable (Antescofo)**
- matérialisation de la grille (gridconnect)
- transformations bouclage et accélération

2015 **création de Djazz** (Marc Chemillier) = variante d'ImproteK adaptée à l'utilisation en concert (**battue manuelle de la pulsation**), assemblage des **versions MIDI et audio**, pattern matching basé sur **l'arbre des suffixes et l'algorithme du plus court chemin** (Solon Pissis, Lorraine Ayad)

4. Pattern matching

4.1 Arbre des suffixes

On a une grille x et un dictionnaire y constitué d'un solo étiqueté par une autre grille. On cherche à factoriser $x = y_1 y_2 \ldots y_k$ de telle sorte que les $y_1, y_2 \ldots, y_k$ apparaissent dans y, et de plus que la décomposition de x soit minimale (k minimal).

→ le but est de concaténer les fragments de solos des $y_1, y_2 \ldots, y_k$ pour fabriquer un solo sur la grille x

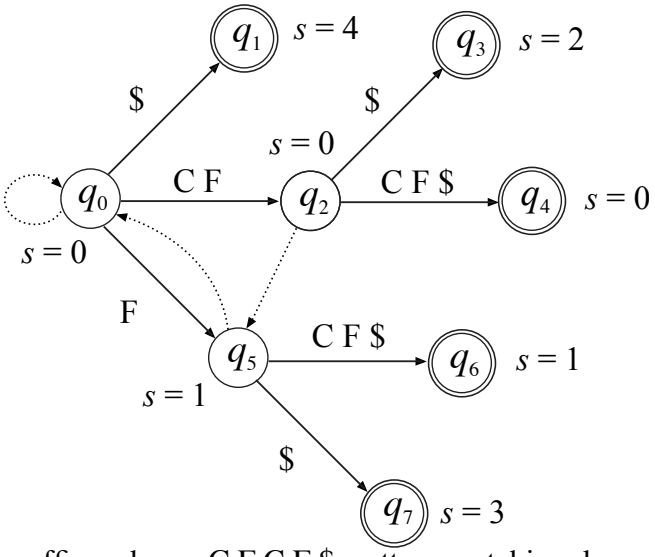
→ on calcule **l'arbre des suffixes du dictionnaire y**

Exemple : pattern matching de la grille x = C C F F par rapport à l'arbre des suffixes de y = C F C F \$

1) \$ est un symbole additionnel qui n'apparaît qu'à la fin de y
2) compactage des étiquettes des transitions pour supprimer les états **sans bifurcation**
3) état terminaux s sont associés à la position dans y à partir de laquelle on lit le suffixe
4) le *lien suffixiel* de l'état d'arrivée p du mot aw, $a \in \Sigma$, $w \in \Sigma^*$ est l'état d'arrivée q du mot w

   → si on a réussi à lire aw dans y, on sait qu'on peut lire w et on peut aller directement à l'état correspondant sans relire w

   (par convention le lien suffixiel de l'état initial est lui-même)



Arbre des suffixes de y = C F C F \$, pattern matching de x = C C F F

- on lit C, puis on se trouve entre q_0 et q_2, et on ne peut pas lire un autre C donc stop

→ les états terminaux accessibles depuis q_2 donnent les positions dans y où on peut lire C, $s = 0$ et $s = 2$

- on suit le lien suffixiel de q_0 qui reste en q_0, et on lit C F qui amène à q_2

→ les états terminaux accessibles depuis q_2 donnent les positions dans y où on peut lire C F, $s = 0$ et $s = 2$

- on suit le lien suffixiel de q_2 qui va en q_5 (on a lu un F), mais on ne peut pas lire F F

→ les états terminaux accessibles depuis q_5 donnent les positions dans y où on peut lire F, $s = 1$ et $s = 3$

- on suit le lien suffixiel de q_5 qui va en q_0, on lit un nouveau F qui va en q_5

→ les états terminaux accessibles depuis q_5 donnent les positions dans y où on lit F, $s = 1$ et $s = 3$

On en déduit un tableau donnant pour chaque position dans x la longueur du plus long facteur apparaissant dans y :

LCP = [1, 2, 1, 1]

Cela correspondant à la décomposition : x = (C)(C F)(F)

avec pour chaque facteur les occurrences dans y :

C y = (**C**) F (**C**) F \$

C F y = (**C F**)(**C F**) \$

F y = C (**F**) C (**F**) \$

Dans le dictionnaire y les séquences d'accords sont associées à des solos. Grâce aux positions dans y où apparaissent les facteurs (C), (C F), et (F), on peut calculer une improvisation sur la grille x en concaténant les fragments de solos correspondants.

Remarque : On a vu que l'arbre des suffixes de y est utilisé pour lire les **facteurs de *y***. Il suffit pour cela d'arrêter la lecture dans un état quelconque (et non dans un état final).

On aurait pu imaginer de construire un automate reconnaissant les suffixes de y beaucoup plus simple que l'arbre des suffixes. Il suffisait de considérer **le mot rétrograde $\tilde{y}$** de y (c'est-à-dire le mot y lu à l'envers de droite à gauche) et de construire l'automate reconnaissant les préfixes de $\tilde{y}$. Cet automate est en effet très simple avec un état initial, une flèche pour chaque enchainement de lettre dans $\tilde{y}$ et tous les états terminaux. Mais pour pouvoir reconnaîtres les facteurs de $\tilde{y}$ (et non seulement les préfixes), il aurait fallu rendre tous les états initiaux. L'automate ne serait plus *un automate fini déterministe (AFD)*, il deviendrait **non déterministe**. Concrètement, il ne serait plus utilisable pour faire un parcours, car il faudrait tester tous les états initiaux (à moins de passer par une procédure de déterminisation assez complexe).

Vidéo d'exemples montrés en cours

- Site web: http://digitaljazz.fr
- Chaîne Youtube: http://bit.ly/YoutubeDjazzChannel
- SoundCloud: http://bit.ly/SoundCloudDjazz
- Facebook: http://facebook.fr/improtekjazz
- Twitter: https://twitter.com/DigitalJazz1
- Instagram: https://www.instagram.com/digitaljazz/

Références bibliographiques

- ouvrages et articles généraux

Journal of Mathematics and Music, special issue on "Music and combinatorics on words", Volume 12, 2018 - Issue 3

- Srečko Brlek, Marc Chemillier & Christophe Reutenauer, Music and combinatorics on words: a historical survey, 125-133 (pdf)
- Lorraine A.K. Ayad, Marc Chemillier & Solon P. Pissis, Creating improvisations on chord progressions using suffix trees, 233-247

Lothaire M., *Combinatorics on Words*, Encyclopedia of Mathematics, Vol. 17, Addison-Wesley, 1983 (réédité Cambridge University Press, 1997) (biblio).

Berstel Jean, *Transductions and context-free languages*, Teubner, 1979 (biblio).

Berstel Jean, *Automates et grammaires*, Cours de Licence d'informatique, Université de Marne-la-vallée, 2004-05 (biblio).

Chemillier M., Structure et méthode algébriques en informatique musicale, Thèse Université Paris 7, LITP, 1990.